Part II Sorting and Order Statistics
Part II Sorting and Order Statistics

- Chapter 6. Heapsort, the use of priority queue (skipped)
- Chapter 7. Quicksort, probabilistic analysis, randomized algorithms
- Chapter 8. Sorting in linear time, lower bounds
- Chapter 9. Medians and order statistics
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue
Chapter 6. Heapsort and the use of priority queue
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

- key(parent) ≥ key(leftChild), key(rightChild);

![Diagram of a heap structure and its representation in an array]
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

- key(parent) \geq key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree.
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

- key(parent) ≥ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree;
- can be stored in arrays (indexes begin with 0),
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

- `key(parent) \geq key(leftChild), key(rightChild)`;
- Relationships are modeled with a complete binary tree.
- Can be stored in arrays (indexes begin with 0),
  \[ index(leftChild) = 2 \times index(parent) + 1 \]
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

- key(parent) ≥ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree
- can be stored in arrays (indexes begin with 0),
  \[ \text{index(leftChild)} = 2 \times \text{index(parent)} + 1 \]
  \[ \text{index(rightChild)} = 2 \times \text{index(parent)} + 2 \]
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap** \((A)\)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap**(A)
- **Max-Heapify**(A, i)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap** \( (A) \)
- **Max-Heapify** \( (A, i) \)
- **HeapSort** \( (A) \)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap** \((A)\)
- **Max-Heapify** \((A, i)\)
- **HeapSort** \((A)\)

heaps as priority queues
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap**(A)
- **Max-Heapify**(A, i)
- **HeapSort**(A)

Heaps as priority queues

- **Heap-Maximum**(A)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap**($A$)
- **Max-Heapify**($A$, $i$)
- **HeapSort**($A$)

Heaps as priority queues

- **Heap-Maximum**($A$)
- **Heap-Extract-Max**($A$)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- **Build-Max-Heap** \((A)\)
- **Max-Heapify** \((A, i)\)
- **HeapSort** \((A)\)

Heaps as priority queues

- **Heap-Maximum** \((A)\)
- **Heap-Extract-Max** \((A)\)
- **Heap-Increase-Key** \((A, I, key)\)
Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- \textbf{Build-Max-Heap}(A)
- \textbf{Max-Heapify}(A, i)
- \textbf{HeapSort}(A)

heaps as priority queues

- \textbf{Heap-Maximum}(A)
- \textbf{Heap-Extract-Max}(A)
- \textbf{Heap-Increase-Key}(A, I, \textit{key})
- \textbf{Max-Heap-Insert}(A, \textit{key})
Chapter 6. Heapsort

Algorithm \textsc{HeapSort}(A)
Chapter 6. Heapsort

Algorithm \texttt{HeapSort}(A)

1. \texttt{Build-Max-Heap}(A)
Algorithm \textsc{HeapSort}(A)

1. \textsc{Build-Max-Heap}(A)
2. for $i = \text{length}[A] - 1$ downto 1 \{ indexes begin from 0\}
Chapter 6. Heapsort

Algorithm \textbf{HeapSort}(A)

1. \textbf{Build-Max-Heap}(A)
2. \textbf{for } i = length[A] − 1 \textbf{ downto } 1 \quad \{ \text{indexes begin from 0}\}
3. exchange A[0] \leftrightarrow A[i]
Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. for \( i = \text{length}[A] - 1 \) downto 1 \{ indexes begin from 0\}
3. exchange \( A[0] \leftrightarrow A[i] \)
4. \( \text{heapsize}[A] = \text{heapsize}[A] - 1 \)
Chapter 6. Heapsort

Algorithm \texttt{HeapSort}(A)

1. \texttt{Build-Max-Heap}(A)
2. \texttt{for} \( i = \text{length}[A] - 1 \) \texttt{downto} 1 \quad \{ \text{indexes begin from 0}\}
3. exchange \( A[0] \leftrightarrow A[i] \)
4. \texttt{heapsize}[A] = \texttt{heapsize}[A] - 1
5. \texttt{Max-Heapify}(A, 0)
Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. for \( i = \text{length}[A] - 1 \) downto 1 \{ indexes begin from 0\}
3. exchange \( A[0] \leftrightarrow A[i] \)
4.  \( \text{heapsize}[A] = \text{heapsize}[A] - 1 \)
5.  Max-Heapify(A, 0)

\[
T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0), \text{ where } n = |A|
\]
Chapter 6. Heapsort

Algorithm **HeapSort**(\(A\))

1. Build-Max-Heap\((A)\)
2. for \(i = \text{length}[A] - 1\) downto 1 \{ indexes begin from 0\}
3. exchange \(A[0] \leftrightarrow A[i]\)
4. heapsize\([A]\) = heapsize\([A]\) - 1
5. Max-Heapify\((A, 0)\)

\[ T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0) \], where \(n = |A|\)

Subroutine **Build-Max-Heap**\((A)\)
Chapter 6. Heapsort

Algorithm HeapSort\((A)\)

1. \texttt{Build-Max-Heap}(\(A\))
2. \texttt{for} \(i = \text{length}[A] - 1\) \texttt{downto} 1 \{ indexes begin from 0\}
3. exchange \(A[0] \leftrightarrow A[i]\)
4. \texttt{heapsize}[A] = \texttt{heapsize}[A] - 1
5. \texttt{Max-Heapify}(A, 0)

\[ T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0), \text{ where } n = |A| \]

Subroutine Build-Max-Heap\((A)\)

1. \texttt{heapsize}[A] = \texttt{length}[A]
Chapter 6. Heapsort

Algorithm **HeapSort**($A$)

1. **Build-Max-Heap**($A$)
2. for $i = \text{length}[A] - 1$ downto 1 { indexes begin from 0 }
3. exchange $A[0] \leftrightarrow A[i]$
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. **Max-Heapify**($A, 0$)

$$T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0), \text{ where } n = |A|$$

Subroutine **Build-Max-Heap**($A$)

1. $\text{heapsize}[A] = \text{length}[A]$
2. for $i = \lceil \frac{1}{2} \text{length}[A] \rceil - 1$ downto 0 { indexes begin from 0 }
Chapter 6. Heapsort

Algorithm HeapSort($A$)

1. Build-Max-Heap($A$)
2. for $i = \text{length}[A] - 1$ downto 1 \{ indexes begin from 0 \}
3. exchange $A[0] \longleftrightarrow A[i]$
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. Max-Heapify($A, 0$)

$T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0)$, where $n = |A|$

Subroutine Build-Max-Heap($A$)

1. $\text{heapsize}[A] = \text{length}[A]$
2. for $i = \lceil \frac{1}{2} \text{length}[A] \rceil - 1$ downto 0 \{ indexes begin from 0 \}
3. Max-Heapify($A, i$)
Chapter 6. Heapsort

Algorithm \textsc{HeapSort}(A)

1. \textbf{Build-Max-Heap}(A)
2. \textbf{for} \( i = \text{length}[A] - 1 \) \textbf{downto} 1 \{ indexes begin from 0\}
3. exchange \( A[0] \leftarrow A[i] \)
4. \( \text{heapsize}[A] = \text{heapsize}[A] - 1 \)
5. \textsc{Max-Heapify}(A, 0)

\[ T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)T_{MH}(n, 0), \text{ where } n = |A| \]

Subroutine \textbf{Build-Max-Heap}(A)

1. \( \text{heapsize}[A] = \text{length}[A] \)
2. \textbf{for} \( i = \lceil \frac{1}{2} \text{length}[A] \rceil - 1 \) \textbf{downto} 0 \{ indexes begin from 0\}
3. \textsc{Max-Heapify}(A, i)

\[ T_{BMH}(n) = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} c_2 T_{MH}(n, i) \]
Chapter 6. Heapsort

Subroutine \textsc{Max-Heapify}(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. \( \text{if } (l \leq \text{heapsize}[A]) \) and \( A[l] > A[i] \)
4. \( \text{then } \text{largest} = l \)
5. \( \text{else } \text{largest} = i \)
6. \( \text{if } (r \leq \text{heapsize}[A]) \) and \( A[r] > A[\text{largest}] \)
7. \( \text{then } \text{largest} = r \)
8. \( \text{if } \text{largest} \neq i \)
9. \( \text{then } \text{exchange } A[i] \leftrightarrow A[\text{largest}] \)
10. \text{Max-Heapify}(A, \text{largest})

\( T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \)

Because \( T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1) \), or \( = c_3 + T_{MH}(n, 2i + 2) \)

\( T_{MH}(n, i) \leq c_4 \log_2 n \), for all \( i = 0, 1, \ldots, n - 1 \). (Prove it!)

\( T_{BMH}(n) = \lfloor n/2 \rfloor - 1 \sum_{i=0}^{\lfloor n/2 \rfloor} c_2 T_{MH}(n, i) \leq c_4 n^2 \log_2 n \)

\( T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1) c_2 T_{MH}(n, 0) \leq c_4 n^2 \log_2 n + (n - 1) c_2 c_4 \log_2 n = O(n \log n) \)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

1. $l = 2 \times i + 1$
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. if \( l \leq \text{heapsize}[A] \) and \( A[l] > A[i] \)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if ($l \leq \text{heapsize}[A]$) and ($A[l] > A[i]$)
4. then $\text{largest} = l$

$T_{MH}(n,i) \leq c_3 + T_{MH}(n, 2i + 1)$
Because $T_{MH}(n,i) = c_3 + T_{MH}(n, 2i + 1)$, or $= c_3 + T_{MH}(n, 2i + 2)$

$T_{MH}(n,i) \leq c_4 \log_2 n$, for all $i = 0, 1, \ldots, n - 1$. (Prove it!)

$T_{BMH}(n) = \lfloor n/2 \rfloor - \sum_{i=0}^{n-1} c_2 T_{MH}(n,i)$

$T_{BS}(n) = c_1 + T_{BMH}(n) + (n - 1) c_2 T_{MH}(n, 0)$

$\leq c_4 n^2 \log_2 n + (n - 1) c_2 c_4 \log_2 n = O(n \log n)$
Chapter 6. Heapsort

Subroutine \textsc{Max-Heapify}(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. \textbf{if} \ (l \leq \text{heapsize}[A]) \textbf{and} \ (A[l] > A[i])
4. \textbf{then} \ largest = l
5. \textbf{else} \ largest = i
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if ($l \leq \text{heapsize}[A]$) and ($A[l] > A[i]$) then
4.     largest = $l$
5. else
6.     largest = $i$
7. if ($r \leq \text{heapsize}[A]$) and ($A[r] > A[\text{largest}]$) then
8.     exchange $A[i] \leftrightarrow A[\text{largest}]
9. Max-HEAPIFY($A, \text{largest}$)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if ($l \leq \text{heapsize}[A]$) and ($A[l] > A[i]$)
4. then $\text{largest} = l$
5. else $\text{largest} = i$
6. if ($r \leq \text{heapsize}[A]$) and ($A[r] > A[\text{largest}]$)
7. then $\text{largest} = r$

Because $T_{\text{MH}}(n, i) \leq c_3 + T_{\text{MH}}(n, 2i + 1)$, or $= c_3 + T_{\text{MH}}(n, 2i + 2)$, $T_{\text{MH}}(n, i) \leq c_4 \log_2 n$, for all $i = 0, 1, \ldots, n - 1$. (Prove it!)

$T_{\text{BMH}}(n) = \lfloor \frac{n}{2} \rfloor - 1 \sum_{i=0}^{n} c_2 T_{\text{MH}}(n, i) \leq c_4 n^2 \log_2 n$

$T_{\text{HS}}(n) = c_1 + T_{\text{BMH}}(n) + (n - 1) c_2 T_{\text{MH}}(n, 0) \leq c_4 n^2 \log_2 n + (n - 1) c_2 c_4 \log_2 n = O(n \log n)$
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if $(l \leq \text{heapsize}[A])$ and $(A[l] > A[i])$
   then largest = $l$
4. else largest = $i$
5. if $(r \leq \text{heapsize}[A])$ and $(A[r] > A[\text{largest}])$
   then largest = $r$
6. if largest $\neq i$

Chapter 6. Heapsort

Subroutine MAX-HEAPIFY\((A, i)\)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. \textbf{if} \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\) \textbf{then} largest = \( l \)
4. \textbf{else} largest = \( i \)
5. \textbf{if} \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])\) \textbf{then} largest = \( r \)
6. \textbf{if} largest \( \neq i \) \textbf{then} exchange \( A[i] \leftrightarrow A[\text{largest}] \)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \[ l = 2 \times i + 1 \]
2. \[ r = 2 \times i + 2 \]
3. \[ \text{if } (l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i]) \]
   \[ \text{then } largest = l \]
4. \[ \text{else } largest = i \]
5. \[ \text{if } (r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}]) \]
   \[ \text{then } largest = r \]
6. \[ \text{if } largest \neq i \]
   \[ \text{then exchange } A[i] \leftrightarrow A[\text{largest}] \]
7. \[ \text{MAX-HEAPIFY}(A, largest) \]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. if \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\)
4. then \( \text{largest} = l \)
5. else \( \text{largest} = i \)
6. if \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])\)
7. then \( \text{largest} = r \)
8. if \( \text{largest} \neq i \)
9. then exchange \( A[i] \leftrightarrow A[\text{largest}] \)
10. MAX-HEAPIFY(A, largest)

\[ T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \] Because \( T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1) \), or
\[ = c_3 + T_{MH}(n, 2i + 2) \]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY$(A, i)$

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if $(l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])$
4. then largest $= l$
5. else largest $= i$
6. if $(r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])$
7. then largest $= r$
8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY$(A, \text{largest})$

$T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1)$  Because $T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1)$, or $= c_3 + T_{MH}(n, 2i + 2)$

$T_{MH}(n, i) \leq c_4 \log_2 n$, for all $i = 0, 1, \ldots, n - 1$. (Prove it!)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. if \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\)
4. then \(\text{largest} = l\)
5. else \(\text{largest} = i\)
6. if \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])\)
7. then \(\text{largest} = r\)
8. if \(\text{largest} \neq i\)
9. then exchange \(A[i] \leftrightarrow A[\text{largest}]\)
10. MAX-HEAPIFY(A, largest)

\[ T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \quad \text{Because } T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1), \text{ or } c_3 + T_{MH}(n, 2i + 2) \]

\[ T_{MH}(n, i) \leq c_4 \log_2 n, \text{ for all } i = 0, 1, \ldots, n - 1. \] (Prove it!)

\[ T_{BMH}(n) = \sum_{i=0}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} c_2 T_{MH}(n, i) \]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1.  \( l = 2 \times i + 1 \)
2.  \( r = 2 \times i + 2 \)
3.  if \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\) then largest = l
4.  else largest = i
5.  if \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[largest])\) then largest = r
6.  if largest ≠ i then exchange \( A[i] \leftrightarrow A[largest] \)
7.  MAX-HEAPIFY(A, largest)

\[ T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \quad \text{Because } T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1), \text{ or } \]
\[ = c_3 + T_{MH}(n, 2i + 2) \]

\[ T_{MH}(n, i) \leq c_4 \log_2 n, \text{ for all } i = 0, 1, \ldots, n - 1. \quad \text{(Prove it!)} \]

\[ T_{BMH}(n) = \sum_{i=0}^{\lfloor n/2 \rfloor - 1} c_2 T_{MH}(n, i) \leq c_4 \frac{n}{2} \log_2 n \]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

1. $l = 2 \times i + 1$
2. $r = 2 \times i + 2$
3. if ($l \leq \text{heapsize}[A]$) and ($A[l] > A[i]$)
4.    then largest = $l$
5. else largest = $i$
6. if ($r \leq \text{heapsize}[A]$) and ($A[r] > A[\text{largest}]$)
7.    then largest = $r$
8. if largest $\neq i$
9.    then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)

\[ T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \quad \text{Because } T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1), \text{ or } = c_3 + T_{MH}(n, 2i + 2) \]

\[ T_{MH}(n, i) \leq c_4 \log_2 n, \text{ for all } i = 0, 1, \ldots, n - 1. \quad \text{(Prove it!)} \]

\[ T_{BMH}(n) = \sum_{i=0}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} c_2 T_{MH}(n, i) \leq c_4 \frac{n}{2} \log_2 n \]

\[ T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)c_2 T_{MH}(n, 0) \]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \[ l = 2 \times i + 1 \]
2. \[ r = 2 \times i + 2 \]
3. if \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\) then largest = l
4. else largest = i
5. if \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])\) then largest = r
6. if largest \(\neq i\) then exchange \(A[i] \leftrightarrow A[\text{largest}]\)
7. MAX-HEAPIFY(A, largest)

\[
T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \quad \text{Because } T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1), \text{ or } \\
\quad = c_3 + T_{MH}(n, 2i + 2)
\]

\[
T_{MH}(n, i) \leq c_4 \log_2 n, \text{ for all } i = 0, 1, \ldots, n - 1. \quad \text{(Prove it!)}
\]

\[
T_{BMH}(n) = \sum_{i=0}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} c_2 T_{MH}(n, i) \leq c_4 \frac{n}{2} \log_2 n
\]

\[
T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)c_2 T_{MH}(n, 0) \leq c_4 \frac{n}{2} \log_2 n + (n - 1)c_2 c_4 \log_2 n
\]
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \( l = 2 \times i + 1 \)
2. \( r = 2 \times i + 2 \)
3. \( \text{if } (l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i]) \)
4. \( \text{then largest } = l \)
5. \( \text{else largest } = i \)
6. \( \text{if } (r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}]) \)
7. \( \text{then largest } = r \)
8. \( \text{if largest } \neq i \)
9. \( \text{then exchange } A[i] \leftrightarrow A[\text{largest}] \)
10. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)

\[ T_{MH}(n, i) \leq c_3 + T_{MH}(n, 2i + 1) \quad \text{Because } T_{MH}(n, i) = c_3 + T_{MH}(n, 2i + 1), \text{ or } \\
T_{MH}(n, i) \leq c_4 \log_2 n, \text{ for all } i = 0, 1, \ldots, n - 1. \quad \text{(Prove it!)} \]

\[ T_{BMH}(n) = \sum_{i=0}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} c_2 T_{MH}(n, i) \leq c_4 \frac{n}{2} \log_2 n \]

\[ T_{HS}(n) = c_1 + T_{BMH}(n) + (n - 1)c_2 T_{MH}(n, 0) \leq c_4 \frac{n}{2} \log_2 n + (n - 1)c_2 c_4 \log_2 n = O(n \log n) \]
Chapter 6. Heapsort

Operations on heaps:

Function Heap-Maximum (A) obtain the maximum:
1. return (A[1])

Function Heap-Extract-Max (A) obtain and remove the maximum:
1. if heapsize[A] < 1 then return "heap underflow"
2. max = A[1]
4. heapsize[A] = heapsize[A] - 1
5. Max-Heapify (A, 1)
6. return (max)

Function Heap-Increase-Key (A, i, key) replace a key with a larger value:
1. if key < A[i] then return "new key is smaller than current key"
2. A[i] = key
3. while i > 1 and A[PARENT[i]] < A[i]
5. i = PARENT[i]

Function Max-Heap-Insert (A, key) insert a new key to heap:
1. heapsize[A] = heapsize[A] + 1
2. A[heapsize[A]] = −∞
3. Heap-Increase-Key (A, heapsize[A], key)
Chapter 6. Heapsort

Operations on heaps:

Function $\text{Heap-Maximum}(A)$
1. return $(A[1])$
Chapter 6. Heapsort

Operations on heaps:

Function **Heap-Maximum**(*A*)
1. **return** (*A[1]*)

Function **Heap-Extract-Max**(*A*)
1.  **if** *heapsize[A] < 1*
2.  **then return** ("heap underflow")
3.  *max = A[1]*
6.  **Max-Heapify**(*A*, 1)
7.  **return** (*max*)
Chapter 6. Heapsort

Operations on heaps:

Function `HEAP-MAXIMUM(A)`
1. return \( A[1] \)

Function `HEAP-EXTRACT-MAX(A)`
1. if `heapsize[A] < 1`
2. then return ("heap underflow")
3. `max = A[1]`
5. `heapsize[A] = heapsize[A] - 1`
6. `MAX-HEAPIFY(A, 1)`
7. return (\( max \))

Function `HEAP-INCREASE-KEY(A, i, key)`
1. if `key < A[i]`
2. then return ("new key is smaller than current key")
3. `A[i] = key`
4. while `i > 1` and `A[\text{PARENT}[i]] < A[i]`
5. exchange \( A[i] \leftarrow A[\text{PARENT}[i]] \)
6. `i = \text{PARENT}[i]`
Chapter 6. Heapsort

Operations on heaps:

Function **HEAP-MAXIMUM**(A)
1. `return (A[1])` obtain the maximum

Function **HEAP-EXTRACT-MAX**(A)
1.  
   `if heapsize[A] < 1`
2.  
   `then return ("heap underflow")`
3.  
   `max = A[1]`
4.  
5.  
   `heapsize[A] = heapsize[A] - 1`
6.  
   `MAX-HEAPIFY(A, 1)`
7.  
   `return (max)` obtain and remove the maximum

Function **HEAP-INCREASE-KEY**(A, i, key)
1.  
   `if key < A[i]`
2.  
   `then return ("new key is smaller than current key")`
3.  
   `A[i] = key`
4.  
   `while i > 1 and A[PARENT[i]] < A[i]`
5.  
6.  
   `i = PARENT[i]` replace a key with a larger value

Function **MAX-HEAP-INSERT**(A, key)
1.  
   `heapsize[A] = heapsize[A] + 1`
2.  
   `A[heapsize[A]] = −∞`
3.  
   `HEAP-INCREASE-KEY(A, heapsize[A], key)` insert a new key to heap
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

• divide: re-organize list $A[p, r]$ into two sublists $A[p, q - 1]$ and $A[q + 1, r]$ based on pivot $A[q]$, such that
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q - 1]$ and $A[q + 1, r]$ based on pivot $A[q]$, such that
  
  (a) $A[i] \leq A[q]$ for all $i = p, \cdots, q - 1$
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

• divide: re-organize list \( A[p, r] \) into two sublists \( A[p, q - 1] \) and \( A[q + 1, r] \) based on pivot \( A[q] \), such that
  
  (a) \( A[i] \leq A[q] \) for all \( i = p, \ldots, q - 1 \)
  
  (b) \( A[i] \geq A[q] \) for all \( i = q + 1, \ldots, r \)
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q - 1]$ and $A[q + 1, r]$ based on pivot $A[q]$, such that
  
  (a) $A[i] \leq A[q]$ for all $i = p, \cdots, q - 1$
  
  (b) $A[i] \geq A[q]$ for all $i = q + 1, \cdots, r$

Chapter 7. Quicksort and Randomized algorithms

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

- Assume $A[q]$ partitions list $A,p,r$ evenly, then $T(n) \leq 2T(n/2) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 20% vs 80%, then $T(n) \leq T(0.2n) + T(0.8n) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 1% vs 99%, then $T(n) \leq T(0.01n) + T(0.99n) + cn = O(n \log_2 n)$

How can we identify such a pivot?
Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT \((A, p, r)\)
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT \((A, p, r)\)

1. \textbf{if} \( p < r \)
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm **QUICKSORT** \((A, p, r)\)

1. **if** \(p < r\)
2. **then** \(q = \text{PARTITION}(A, p, r)\)
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT \((A, p, r)\)
1. \textbf{if} \(p < r\)
2. \textbf{then} \(q = \text{PARTITION}(A, p, r)\)
3. \text{QUICKSORT} \((A, p, q - 1)\)

How the pivot \(A[q]\) is identified is crucial to the performance of Quicksort.

- Assume \(A[q]\) partitions list \(A, p, r\) evenly, then \(T(n) \leq 2T(n/2) + cn = O(n \log_2 n)\)
- Assume \(A[q]\) partitions the list 20% vs 80%, then \(T(n) \leq T(n/5) + T(4n/5) + cn = O(n \log_2 n)\)
- Assume \(A[q]\) partitions the list 1% vs 99%, then \(T(n) \leq T(n/100) + T(99n/100) + cn = O(n \log_2 n)\)

How can we identify such a pivot?
Chapter 7. Quicksort and Randomized algorithms

Algorithm \textsc{QuickSort} (A, p, r)
1. if \( p < r \)
2. then \( q = \text{Partition}(A, p, r) \)
3. \textsc{QuickSort} (A, p, q − 1)
4. \textsc{QuickSort} (A, q + 1, r)

How the pivot \( A[q] \) is identified is crucial to the performance of Quicksort.

- Assume \( A[q] \) partitions list \( A,p,r \) evenly, then
  \[ T(n) \leq 2T(n/2) + cn = O(n \log_2 n) \]

- Assume \( A[q] \) partitions the list 20% vs 80%, then
  \[ T(n) \leq T(5n/2) + T(4n/5) + cn = O(n \log_2 n) \]

- Assume \( A[q] \) partitions the list 1% vs 99%, then
  \[ T(n) \leq T(n/100) + T(99n/100) + cn = O(n \log_2 n) \]
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm QuickSort \((A, p, r)\)

1. \textbf{if} \(p < r\)
2. \textbf{then} \(q = \text{Partition}(A, p, r)\)
3. QuickSort \((A, p, q - 1)\)
4. QuickSort \((A, q + 1, r)\)

How the pivot \(A[q]\) is identified is crucial to the performance of Quicksort.
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT \((A, p, r)\)

1. if \(p < r\)
2. then \(q = \text{Partition}(A, p, r)\)
3. QUICKSORT \((A, p, q - 1)\)
4. QUICKSORT \((A, q + 1, r)\)

How the pivot \(A[q]\) is identified is crucial to the performance of Quicksort.

- Assume \(A[q]\) partitions list \(A, p, r\) evenly, then \(T(n) \leq 2T(n/2) + cn = O(n \log_2 n)\)
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm \textsc{QuickSort} \((A, p, r)\)
1. \textbf{if} \(p < r\)
2. \textbf{then} \(q = \textsc{Partition}(A, p, r)\)
3. \textsc{QuickSort} \((A, p, q - 1)\)
4. \textsc{QuickSort} \((A, q + 1, r)\)

How the pivot \(A[q]\) is identified is crucial to the performance of Quicksort.

- Assume \(A[q]\) partitions list \(A, p, r\) evenly, then
  \[ T(n) \leq 2T(n/2) + cn = O(n \log_2 n) \]

- Assume \(A[q]\) partitions the list 20\% vs 80\%, then
  \[ T(n) \leq T(\frac{n}{5}) + T(\frac{4n}{5}) + cn = O(n \log_2 n) \]
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT \((A, p, r)\)

1. if \(p < r\)
2. then \(q = \text{Partition}(A, p, r)\)
3. QUICKSORT \((A, p, q - 1)\)
4. QUICKSORT \((A, q + 1, r)\)

How the pivot \(A[q]\) is identified is crucial to the performance of Quicksort.

- Assume \(A[q]\) partitions list \(A, p, r\) evenly, then
  \(T(n) \leq 2T(n/2) + cn = O(n \log_2 n)\)

- Assume \(A[q]\) partitions the list 20% vs 80%, then
  \(T(n) \leq T(\frac{n}{5}) + T(\frac{4n}{5}) + cn = O(n \log_2 n)\)

- Assume \(A[q]\) partitions the list 1% vs 99%, then
  \(T(n) \leq T(\frac{n}{100}) + T(\frac{99n}{100}) + cn = O(n \log_2 n)\)
Chapter 7. Quicksort

Chapter 7. Quicksort and Randomized algorithms

Algorithm \textsc{QuickSort} \((A, p, r)\)

1. \textbf{if} \( p < r \)
2. \textbf{then} \( q = \textsc{Partition}(A, p, r) \)
3. \textsc{QuickSort} \((A, p, q - 1)\)
4. \textsc{QuickSort} \((A, q + 1, r)\)

How the pivot \( A[q] \) is identified is crucial to the performance of Quicksort.

- Assume \( A[q] \) partitions list \( A, p, r \) evenly, then
  \[ T(n) \leq 2T(n/2) + cn = O(n \log_2 n) \]

- Assume \( A[q] \) partitions the list 20\% vs 80\%, then
  \[ T(n) \leq T(n/5) + T(4n/5) + cn = O(n \log_2 n) \]

- Assume \( A[q] \) partitions the list 1\% vs 99\%, then
  \[ T(n) \leq T(n/100) + T(99n/100) + cn = O(n \log_2 n) \]

How can we identify such a pivot?
Chapter 7. Quicksort

![Quicksort Process Diagram](image)
Chapter 7. Quicksort

PARTITION($A, p, r$)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j \leftarrow p$ to $r - 1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i + 1$
6. exchange $A[i] \leftrightarrow A[j]$
7. exchange $A[i + 1] \leftrightarrow A[r]$
8. return $i + 1$
Partition may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$. 
Partition may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$. 
Partition may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.
- however, chances for skewed cases like above are very small.
Chapter 7. Quicksort

**Partition** may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.

- however, chances for skewed cases like above are very small.

- that is, the cases other than the skewed ones occur much more often.
Partition may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.
- however, chances for skewed cases like above are very small.
- that is, the cases other than the skewed ones occur much more often.

So the idea of Quicksort may work well on a majority of data.
Chapter 7. Quicksort

Assume that the equal likely chance for every number to be in the last position, what is the chance to partition the list into

\[ x\% \text{ vs } (100 - x)\% \]

fragments, for \(10 \leq x \leq 90\)?
Chapter 7. Quicksort

Assume that the equal likely chance for every number to be in the last position, what is the chance to partition the list into

\[ x\% \text{ vs } (100 - x)\% \]

fragments, for \(10 \leq x \leq 90\)?

The chance is = 80%
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ l_0: \quad cn \]

\[ cn \]

\[ cn \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ \begin{array}{ccc}
    l_0: & & cn \\
    l_1: & cn/10 & 9cn/10 \\
\end{array} \]

\[ \begin{array}{c}
    cn \\
    cn \\
\end{array} \]
What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ \begin{align*}
  l_0: & \quad cn & & \quad cn \\
  l_1: & \quad cn/10 & \quad 9cn/10 & \quad cn & \quad 9cn/10 \\
  l_2: & \quad cn/10^2 & \quad 9cn/10^2 & \quad 9cn/10^2 & \quad 9^2cn/10^2 & \quad cn
\end{align*} \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
  &l_0: & & cn & & cn \\
  &l_1: & & cn/10 & & 9cn/10 & & cn \\
  &l_2: & & cn/10^2 & & 9cn/10^2 & & 9^2cn/10^2 & & cn \\
  & & & & \ldots\ldots \\
\end{align*}
\]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ l_0: \quad cn \]
\[ l_1: \quad cn/10 \quad 9cn/10 \]
\[ l_2: \quad cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \]

\[ l_h: \quad cn/10^h \quad \cdots \cdots \quad c9^h n/10^h \]

where \( c' = c/\log_{10}9 \).
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
    l_0: & \quad cn & \quad cn \\
    l_1: & \quad cn/10 & \quad 9cn/10 & \quad cn \\
    l_2: & \quad cn/10^2 & \quad 9cn/10^2 & \quad 9^2cn/10^2 & \quad cn \\
    \vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots \\
    l_h: & \quad cn/10^h & \quad \cdots & \quad c9^h n/10^h & \quad cn \\
\end{align*}
\]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ l_0: \quad cn \]
\[ l_1: \quad cn/10 \quad 9cn/10 \]
\[ l_2: \quad cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \]
\[ l_h: \quad cn/10^h \quad \cdots \cdots \quad c9^h n/10^h \]
\[ l_k: \quad \cdots \cdots \quad c9^k n/10^k \leq cn \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
l_0: & \quad cn \\
l_1: & \quad cn/10 \quad 9cn/10 \\
l_2: & \quad cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \\
\vdots & \quad \vdots \\
l_h: & \quad cn/10^h \quad \cdots \quad c9^hn/10^h \quad \cdots \quad cn \\
l_k: & \quad \cdots \quad \cdots \quad c9^kn/10^k \quad \leq cn
\end{align*}
\]

where \( (\frac{1}{10})^hn = 1 \), i.e., \( h = \log_{10} n \)
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
  l_0 &: \quad cn \\
  l_1 &: \quad cn/10 \quad 9cn/10 \\
  l_2 &: \quad cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \\
  \vdots & \quad \vdots \\
  l_h &: \quad cn/10^h \quad \cdots \quad c9^hn/10^h \\
  l_k &: \quad \cdots \quad \cdots \quad c9^kn/10^k \leq cn
\end{align*}
\]

where \((\frac{1}{10})^h n = 1\), i.e., \(h = \log_{10} n\)

\((\frac{9}{10})^k n = 1\), i.e., \(k = \log_{\frac{10}{9}} n\)
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ l_0: \quad \text{cn} \]
\[ l_1: \quad \frac{cn}{10} \quad \frac{9cn}{10} \quad \frac{9cn}{10} \quad \frac{9^2cn}{10^2} \quad \text{cn} \]
\[ l_2: \quad \frac{cn}{10^2} \quad \frac{9cn}{10^2} \quad \frac{9cn}{10^2} \quad \frac{9^2cn}{10^2} \quad \text{cn} \]
\[ \vdots \]
\[ l_h: \quad \frac{cn}{10^h} \quad \frac{9^h n}{10^h} \quad \frac{9^h n}{10^h} \quad \text{cn} \]
\[ \vdots \]
\[ l_k: \quad \frac{c9^k n}{10^k} \quad \leq \text{cn} \]

where \((\frac{1}{10})^h n = 1\), i.e., \( h = \log_{10} n \)
\((\frac{9}{10})^k n = 1\), i.e., \( k = \log_{\frac{9}{10}} n \)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\begin{align*}
\text{l}_0: & \quad cn \\
\text{l}_1: & \quad cn/10 \quad 9cn/10 \\
\text{l}_2: & \quad cn/10^2 \quad 9cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \\
\text{l}_h: & \quad cn/10^h \\
\text{l}_k: & \quad \ldots \\
\text{l}_h: & \quad c9^h n/10^h \\
\text{l}_k: & \quad \ldots \\
\text{l}_k: & \quad \leq cn \\
\end{align*}

where \( \left(\frac{1}{10}\right)^h n = 1 \), i.e., \( h = \log_{10} n \)

\( \left(\frac{9}{10}\right)^k n = 1 \), i.e., \( k = \log_{\frac{10}{9}} n \)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n \]

\[ T(n) \leq cn \log_{\frac{10}{9}} n \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{aligned}
l_0: & \quad cn \\
l_1: & \quad cn/10 \\
l_2: & \quad cn/10^2 \quad 9cn/10^2 \\
l_h: & \quad cn/10^h \\
l_k: & \quad \ldots \quad c9^h n/10^h \\
\end{aligned}
\]

where \((\frac{1}{10})^h n = 1\), i.e., \(h = \log_{10} n\)

\((\frac{9}{10})^k n = 1\), i.e., \(k = \log_{\frac{9}{10}} n\)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{9}{10}} n \]

\[ T(n) \leq cn \log_{\frac{9}{10}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
  l_0: & \quad cn \\
  l_1: & \quad cn/10 \quad \quad \quad 9cn/10 \\
  l_2: & \quad cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \\
  l_h: & \quad cn/10^h \quad \cdots \\
  l_k: & \quad \cdots \\
\end{align*}
\]

where \((\frac{1}{10})^h n = 1\), i.e., \(h = \log_{10} n\)

\((\frac{9}{10})^k n = 1\), i.e., \(k = \log_{\frac{10}{9}} n\)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n \]

\[ T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c'n \log_2 n \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[ l_0: \quad cn \]
\[ l_1: \quad cn/10 \]
\[ l_2: \quad cn/10^2 \quad 9cn/10^2 \quad 9cn/10^2 \quad 9^2cn/10^2 \]
\[ l_h: \quad cn/10^h \quad \ldots \]
\[ l_k: \quad \ldots \]

where \((\frac{1}{10})^h n = 1\), i.e., \(h = \log_{10} n\)
\((\frac{9}{10})^k n = 1\), i.e., \(k = \log_{\frac{9}{10}} n\)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{9}{10}} n \]

\[ T(n) \leq cn \log_{\frac{9}{10}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c' n \log_2 n = O(n \log_2 n) \]
Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

\[ T(n) \leq T(n/10) + T(9n/10) + cn \]

Using the recursive-tree method (in the book notation), we have

\[
\begin{align*}
  l_0: & & cn \\
  l_1: & & cn/10 \\
  l_2: & & cn/10^2 \\
  l_h: & & cn/10^h \\
  l_k: & & \cdots \\
\end{align*}
\]

\[
\begin{align*}
  l_1: & & cn/10 \\
  l_2: & & 9cn/10^2 \\
  l_h: & & 9^{h}cn/10^h \\
  l_k: & & \cdots \\
\end{align*}
\]

\[
\begin{align*}
  l_2: & & 9cn/10 \\
  l_h: & & 9^{h}cn/10^h \\
  l_k: & & \cdots \\
\end{align*}
\]

\[
\begin{align*}
  l_0: & & \cdots \\
  l_1: & & \cdots \\
  l_2: & & \cdots \\
  l_h: & & \cdots \\
  l_k: & & \cdots \\
\end{align*}
\]

where \((\frac{1}{10})^{h}n = 1\), i.e., \(h = \log_{10} n\)

\((\frac{9}{10})^{k}n = 1\), i.e., \(k = \log_{\frac{10}{9}} n\)

\[ cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n \]

\[ T(n) \leq cn \log_{\frac{9}{10}} n = cn \frac{\log_{2} n}{\log_{2} \frac{10}{9}} = c' n \log_{2} n = O(n \log_{2} n) \]

where \(c' = c/ \log_{2} \frac{10}{9}\)
Chapter 7. Quicksort

Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.
Chapter 7. Quicksort

Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm Randomized-Partition \((A, p, r)\)
1. \(i = \text{random}(p, r)\)
2. exchange \(A[r] \leftrightarrow A[i]\)
3. return (Partition\((A, p, r)\))
Instead of analyzing Quicksort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm **Randomized-Partition** \((A, p, r)\)

1. \(i = \text{random}(p, r)\)
2. exchange \(A[r] \leftrightarrow A[i]\)
3. **return** \((\text{Partition}(A, p, r))\)

Algorithm **Randomized Quicksort** \((A, p, r)\)
Chapter 7. Quicksort

Instead of analyzing \textsc{QuickSort} (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm \textbf{Randomized-Partition}\((A, p, r)\)
1. \(i = \text{random}(p, r)\)
2. exchange \(A[r] \leftrightarrow A[i]\)
3. \textbf{return} \((\text{Partition}\((A, p, r)\))\)

Algorithm \textbf{Randomized QuickSort}\((A, p, r)\)
1. \textbf{if} \(p < r\)
Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm **RANDOMIZED-PARTITION** \((A, p, r)\)
1. \(i = \text{random}(p, r)\)
2. exchange \(A[r] \leftrightarrow A[i]\)
3. return \((\text{PARTITION}(A, p, r))\)

Algorithm **RANDOMIZED QUICKSORT** \((A, p, r)\)
1. \(\text{if } p < r\)
2. \(\text{then } q = \text{RANDOMIZED-PARTITION}(A, p, r)\)
Instead of analyzing \textsc{QuickSort} (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

**Algorithm** \textsc{Randomized-Partition}(\textit{A, p, r})
1. \( i = \text{random}(p, r) \)
2. exchange \( A[r] \leftarrow A[i] \)
3. \textbf{return} \( (\textsc{Partition}(A, p, r)) \)

Algorithm \textsc{Randomized QuickSort}\ (\textit{A, p, r})
1. \textbf{if} \( p < r \)
2. \textbf{then} \( q = \textsc{Randomized-Partition}(A, p, r) \)
3. \textbf{Randomized QuickSort} \( (A, p, q - 1) \)
Chapter 7. Quicksort

Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm \textbf{Randomized-Partition}(A, p, r)
1. \( i = \text{random}(p, r) \)
2. exchange \( A[r] \leftarrow A[i] \)
3. \textbf{return} (\textbf{Partition}(A, p, r))

Algorithm \textbf{Randomized QuickSort} (A, p, r)
1. \textbf{if} \( p < r \)
2. \textbf{then} \( q = \text{Randomized-Partition}(A, p, r) \)
3. \textbf{randomized QuickSort} (A, p, q − 1)
4. \textbf{randomized QuickSort} (A, q + 1, r)
Chapter 7. Quicksort

Up to this point, you should have known:

1. the details of QuickSort algorithm, especially Partition;
2. Why it runs $O(n \log n)$ on uniformly distributed data, intuitively;
3. the connection between
   (a) requiring prob distribution in the input data;
   (b) randomized algorithms;
Chapter 7. Quicksort

Analysis of \texttt{RANDOMIZED-QUICKSORT}
Analysis of **Randomized-QuickSort**

- count the expected number of comparisons between $x_i$ and $x_j$;
Chapter 7. Quicksort

Analysis of **RANDOMIZED-QUICKSORT**

- count the expected number of comparisons between $x_i$ and $x_j$;

**Observation 1:** $x_i$ is compared with $x_j$ only when either is a pivot;
Chapter 7. Quicksort

Analysis of Randomized-QuickSort

• count the expected number of comparisons between \( x_i \) and \( x_j \);

Observation 1: \( x_i \) is compared with \( x_j \) only when either is a pivot;

Observation 2: \( x_i \) is compared with \( x_j \) at most once;
Analysis of Randomized-QuickSort

- count the expected number of comparisons between $x_i$ and $x_j$;

**Observation 1:** $x_i$ is compared with $x_j$ only when either is a pivot;

**Observation 2:** $x_i$ is compared with $x_j$ at most once;

- define random variable $X_{i,j} \in \{0, 1\}$, such that
Chapter 7. Quicksort

Analysis of **Randomized-QuickSort**

- count the expected number of comparisons between $x_i$ and $x_j$;

**Observation 1**: $x_i$ is compared with $x_j$ only when either is a pivot;

**Observation 2**: $x_i$ is compared with $x_j$ at most once;

- define random variable $X_{i,j} \in \{0, 1\}$, such that

  \[ X_{i,j} = 1 \text{ iff a comparison between } x_i \text{ and } x_j \text{ occurs} \]
Chapter 7. Quicksort

Analysis of **Randomized-QuickSort**

- count the expected number of comparisons between \( x_i \) and \( x_j \);

**Observation 1**: \( x_i \) is compared with \( x_j \) only when either is a pivot;

**Observation 2**: \( x_i \) is compared with \( x_j \) at most once;

- define random variable \( X_{i,j} \in \{0, 1\} \), such that

  \[
  X_{i,j} = 1 \text{ iff a comparison between } x_i \text{ and } x_j \text{ occurs}
  \]

- let \( X = \sum_{i<j} X_{i,j} \), total number of comparisons
Chapter 7. Quicksort

Analysis of **Randomized-QuickSort**

- count the expected number of comparisons between $x_i$ and $x_j$;

**Observation 1**: $x_i$ is compared with $x_j$ only when either is a pivot;

**Observation 2**: $x_i$ is compared with $x_j$ at most once;

- define random variable $X_{i,j} \in \{0, 1\}$, such that

  $$X_{i,j} = 1 \text{ iff a comparison between } x_i \text{ and } x_j \text{ occurs}$$

- let $X = \sum_{i<j} X_{i,j}$, total number of comparisons

- the **expected** number of comparisons is

  $$E(X) = E(\sum_{i<j} X_{i,j})$$
Chapter 7. Quicksort

Analysis of **Randomized-QuickSort**

- count the expected number of comparisons between $x_i$ and $x_j$;

**Observation 1**: $x_i$ is compared with $x_j$ only when either is a pivot;

**Observation 2**: $x_i$ is compared with $x_j$ at most once;

- define random variable $X_{i,j} \in \{0, 1\}$, such that

\[ X_{i,j} = 1 \text{ iff a comparison between } x_i \text{ and } x_j \text{ occurs} \]

- let $X = \sum_{i<j} X_{i,j}$, total number of comparisons

- the **expected** number of comparisons is

\[ E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \]
Chapter 7. Quicksort

Analysis of \textsc{Randomized-QuickSort}

- count the expected number of comparisons between $x_i$ and $x_j$;

\underline{Observation 1:} $x_i$ is compared with $x_j$ only when either is a pivot;

\underline{Observation 2:} $x_i$ is compared with $x_j$ at most once;

- define random variable $X_{i,j} \in \{0, 1\}$, such that
  \[
  X_{i,j} = 1 \text{ iff a comparison between $x_i$ and $x_j$ occurs}
  \]

- let $X = \sum_{i<j} X_{i,j}$, total number of comparisons

- the expected number of comparisons is

  \[
  E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} P(X_{i,j} = 1)
  \]

by linearity of expectations.
Analysis of RANDOMIZED-QUICKSORT (cont.)

\[ E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]
Analysis of \texttt{RANDOMIZED-QUICKSORT} (cont.)

\[ E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\( X_{i,j} = 1 \), i.e., comparison between \( x_i \) and \( x_j \) occurs only when
Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT (cont.)

\[ E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\( X_{i,j} = 1 \), i.e., comparison between \( x_i \) and \( x_j \) occurs only when

1. \( x_i, x_j \) are in the same sublist \( L \);
Chapter 7. Quicksort

Analysis of **RANDOMIZED-QUICKSORT** (cont.)

\[ E(X) = E\left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\( X_{i,j} = 1 \), i.e., comparison between \( x_i \) and \( x_j \) occurs only when

1. \( x_i, x_j \) are in the same sublist \( L \);
2. either is chosen to be the pivot;
Analysis of $\text{RANDOMIZED-QUICKSORT}$ (cont.)

$$E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between $x_i$ and $x_j$ occurs only when

1. $x_i, x_j$ are in the same sublist $L$;
2. either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2 \frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?
Chapter 7. Quicksort

Analysis of **RANDOMIZED-QUICKSORT** (cont.)

\[ E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\( X_{i,j} = 1 \), i.e., comparison between \( x_i \) and \( x_j \) occurs only when

1. \( x_i, x_j \) are in the same sublist \( L \);
2. either is chosen to be the pivot;

\[ P(X_{i,j} = 1) = 2 \frac{1}{|L|} \], where \(|L|\) is the size of the sublist. **why?**

but we do not know the size of the sublist \( L \)!
Chapter 7. Quicksort

Analysis of Randomized-QuickSort (cont.)

\[ E(X) = E\left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\(X_{i,j} = 1\), i.e., comparison between \(x_i\) and \(x_j\) occurs only when

1. \(x_i, x_j\) are in the same sublist \(L\);
2. either is chosen to be the pivot;

\(P(X_{i,j} = 1) = 2 \frac{1}{|L|}\), where \(|L|\) is the size of the sublist. why?

but we do not know the size of the sublist \(L\)!

however, if \(x_i, x_j\) are so indexed in the final sorted list,
Chapter 7. Quicksort

Analysis of **RANDOMIZED-QUICKSORT** (cont.)

\[ E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\(X_{i,j} = 1\), i.e., comparison between \(x_i\) and \(x_j\) occurs only when

1. \(x_i, x_j\) are in the same sublist \(L\);
2. either is chosen to be the pivot;

\[ P(X_{i,j} = 1) = 2 \frac{1}{|L|}, \text{ where } |L| \text{ is the size of the sublist. why?} \]

but we do not know the size of the sublist \(L\)!

however, if \(x_i, x_j\) are so indexed in the final sorted list, then
Chapter 7. Quicksort

Analysis of Randomized-Quicksort (cont.)

\[ E(X) = E \left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\( X_{i,j} = 1 \), i.e., comparison between \( x_i \) and \( x_j \) occurs only when

1. \( x_i, x_j \) are in the same sublist \( L \);
2. either is chosen to be the pivot;

\[ P(X_{i,j} = 1) = 2 \frac{1}{|L|}, \text{ where } |L| \text{ is the size of the sublist. why?} \]

but we do not know the size of the sublist \( L \! \)!

however, if \( x_i, x_j \) are so indexed in the final sorted list, then

size of the sublist (which \( x_i, x_j \) belongs to)
\[ |L| \geq (j - i + 1) \]
Analysis of **RANDOMIZED-QUICKSORT** (cont.)

\[
E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1)
\]

\(X_{i,j} = 1\), i.e., comparison between \(x_i\) and \(x_j\) occurs only when

1. \(x_i, x_j\) are in the same sublist \(L\);
2. either is chosen to be the pivot;

\[P(X_{i,j} = 1) = 2 \frac{1}{|L|},\] where \(|L|\) is the size of the sublist. **why?**

but we do not know the size of the sublist \(L\)!

however, if \(x_i, x_j\) are so indexed in the final sorted list, then

1. size of the sublist \((\text{which } x_i, x_j \text{ belongs to})\)
   \[|L| \geq (j - i + 1)\]

2. \(P(X_{i,j} = 1) \leq 2 \frac{1}{|L|} \leq 2 \frac{1}{j - i + 1}\)
Chapter 7. Quicksort

original unsorted list

5 23 10

sublist L containing elements 5 and 10
10 is a pivot

5 10 ...

|L|

L has to contain elements between 5 and 10
i.e., L has to contain elements 6, 7, 8, 9
|L| ≥ j − i + 1 = 10 − 5 + 1 = 6

final sorted list

1 2 3 4 5 6 7 8 9 10

x_5  x_{10}
Analysis of \textsc{Randomized-QuickSort} (cont.)

\[ E(X) = E\left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} \leq \sum_{i<j} \frac{1}{n} \sum_{k=1}^{n} \frac{1}{k} + 1 \leq cn \log_2 n \]

for some constant \( c > 0 \). So \( E(X) = O(n \log_2 n) \).
Analysis of \texttt{RANDOMIZED-QUICKSORT} (cont.)

\[ E(X) = E\left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \leq n \sum_{i=1}^{n-1} \sum_{j=2}^{n} \frac{1}{j - i + 1} \leq c n \log_2 n \]

for some constant \( c > 0 \).

So \( E(X) = O(n \log_2 n) \).
Analysis of \textsc{Randomized-QuickSort} (cont.)

\[ E(X) = E\left( \sum_{i<j} X_{i,j} \right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \]

\[ \leq \sum_{i=1}^{n-1} 2 \sum_{k=1}^{n-1} \frac{1}{k + 1} \leq \]
Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT (cont.)

\[ E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \]

\[ \leq \sum_{i=1}^{n-1} 2 \sum_{k=1}^{n-1} \frac{1}{k + 1} \leq \sum_{i=1}^{n-1} c \log_2 n \]
Chapter 7. Quicksort

Analysis of \textsc{Randomized-QuickSort} (cont.)

\[ E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \]

\[ \leq \sum_{i=1}^{n-1} 2 \sum_{k=1}^{n-1} \frac{1}{k + 1} \leq \sum_{i=1}^{n-1} c \log_2 n \leq cn \log_2 n \]
Analysis of \texttt{RANDOMIZED-QUICKSORT} (cont.)

\[ E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \]

\[ \leq \sum_{i=1}^{n-1} 2 \sum_{k=1}^{n-1} \frac{1}{k + 1} \leq \sum_{i=1}^{n-1} c \log_2 n \leq cn \log_2 n \]

for some constant \( c > 0 \).
Analysis of Randomized-QuickSort (cont.)

\[ E(X) = E\left(\sum_{i<j} X_{i,j}\right) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} \text{Prob}(X_{i,j} = 1) \]

\[ \leq \sum_{i<j} 2 \frac{1}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2 \frac{1}{j - i + 1} \]

\[ \leq \sum_{i=1}^{n-1} 2 \sum_{k=1}^{n-1} \frac{1}{k + 1} \leq \sum_{i=1}^{n-1} c \log_2 n \leq cn \log_2 n \]

for some constant \( c > 0 \).

So \( E(X) = O(n \log_2 n) \).
Chapter 7. Quicksort
Chapter 8. Lower Bounds and Sorting in Linear Time

We have used $\mathcal{O}$ for upper bounds. We need another notation for lower bounds. Define $\Omega(g(n))$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$

• e.g., $\Omega(n \log n)$ includes the following functions: $14n \log n, 100n \log n, n^2, n^3 \log n, 37002n, n!, ...$

Proof techniques for $\Omega$ are similar to those for $\mathcal{O}$. 
Chapter 8. Lower Bounds and Sorting in Linear Time

- We have used Big-$O$ for upper bounds.
Chapter 8. Lower Bounds and Sorting in Linear Time

- We have used Big-$O$ for upper bounds.
- We need another notation for lower bounds.

$$\Omega(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k \}$$

- e.g., $\Omega(n \log n)$ includes the following functions:
  - $14n \log n$,
  - $\frac{1}{100}n \log n$,
  - $n^2$,
  - $n^3 \log n$,
  - $37002n$,
  - $n!$, ...

Proof techniques for Big-$\Omega$ are similar to those for Big-$O$. 
• We have used Big-$O$ for upper bounds.

• We need another notation for lower bounds.

Define $\Omega(g(n))$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$. 

• e.g., $\Omega(n \log n)$ includes the following functions: 

  \begin{itemize}
  \item $14n \log n$
  \item $\frac{1}{100}n \log n$
  \item $n^2$
  \item $n^3 \log n$
  \item $3^{7002}n$
  \item $n!$
  \end{itemize}
Chapter 8. Lower Bounds and Sorting in Linear Time

- We have used Big-$O$ for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n))$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k \}$$

- e.g., $\Omega(n \log n)$ includes the following functions: $14n \log n, 100n \log n, n^2, n^3 \log n, 7002n, n!$, ...
Chapter 8. Lower Bounds and Sorting in Linear Time

• We have used Big-$O$ for upper bounds.
• We need another notation for lower bounds.

Define $\Omega(g(n))$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k \}$$

• e.g., $\Omega(n \log n)$ includes the following functions:

$$14n \log n, \frac{1}{100} n \log n, n^2, n^3 \log n, \frac{3}{700} 2^n, n!, \ldots$$
We have used Big-$O$ for upper bounds.

We need another notation for lower bounds.

Define $\Omega(g(n))$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k \}$$

- e.g., $\Omega(n \log n)$ includes the following functions:
  $$14n \log n, \frac{1}{100}n \log n, n^2, n^3 \log n, \frac{3}{700}2^n, n!, \ldots$$

- Proof techniques for Big-$\Omega$ are similar to those for Big-$O$. 


Upper bound of an algorithm
A time sufficient (i.e., enough) for the algorithm to solve all instances.
We make sure an upper bound should covers all instances; e.g.,
MergeSort has upper bound $O(n \log n)$.
Is it correct to say MergeSort has upper bound $O(n^2)$?
It is correct for two reasons:
(1) since $cn \log n$ is sufficient, so is $cn^2$.
(2) $O(n^2)$ contains all functions that $O(n \log n)$ contains.
Is it correct to say MergeSort has upper bound $O(n)$?
Upper bound of an algorithm
Upper bound of an algorithm

A time sufficient (i.e., enough) for the algorithm to solve all instances.
Upper bound of **an algorithm**

A time **sufficient** (i.e., **enough**) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances;
Upper bound of an algorithm

A time sufficient (i.e., enough) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances;

e.g., MERGESORT has upper bound $O(n \log n)$.
Chapter 8. Lower Bounds and Sorting in Linear Time

Upper bound of an algorithm

A time sufficient (i.e., enough) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances;

e.g., MergeSort has upper bound $O(n \log n)$.

Is it correct to say MergeSort has upper bound $O(n^2)$?
Upper bound of an algorithm
A time sufficient (i.e., enough) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances;

e.g., MergeSort has upper bound $O(n \log n)$.

Is it correct to say MergeSort has upper bound $O(n^2)$?

It is correct for two reasons:
Upper bound of **an algorithm**

A time **sufficient** (i.e., **enough**) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances; e.g., MERGESORT has upper bound $O(n \log n)$.

Is it correct to say MERGESORT has upper bound $O(n^2)$?

It is correct for two reasons:

(1) since $cn \log n$ is sufficient, so is $cn^2$. 

Chapter 8. Lower Bounds and Sorting in Linear Time

Upper bound of an algorithm

A time sufficient (i.e., enough) for the algorithm to solve all instances. We make sure an upper bound should covers all instances; e.g., MergeSort has upper bound $O(n \log n)$.

Is it correct to say MergeSort has upper bound $O(n^2)$?

It is correct for two reasons:

1. since $cn \log n$ is sufficient, so is $cn^2$.
2. $O(n^2)$ contains all functions that $O(n \log n)$ contains.
Upper bound of **an algorithm**

A time **sufficient** (i.e., **enough**) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances;

e.g., **MergeSort** has upper bound $O(n \log n)$.

**Is it correct to say** **MergeSort** **has upper bound** $O(n^2)$?

It is correct for two reasons:

(1) since $cn \log n$ is sufficient, so is $cn^2$.

(2) $O(n^2)$ contains all functions that $O(n \log n)$ contains.
Upper bound of **an algorithm**

A time **sufficient** (i.e., **enough**) for the algorithm to solve all instances.

We make sure an upper bound should covers all instances; e.g., **MergeSort** has upper bound $O(n \log n)$.

**Is it correct to say MergeSort has upper bound $O(n^2)$?**

It is correct for two reasons:

(1) since $cn \log n$ is sufficient, so is $cn^2$.

(2) $O(n^2)$ contains all functions that $O(n \log n)$ contains.

**Is it correct to say MergeSort has upper bound $O(n)$?**
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances. $l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

Example: MergeSort has lower bound $\Omega(n \log n)$.

Is it correct to say MergeSort has lower bound $\Omega(n)$?

It is correct for two reasons:

1. Since $cn \log n$ is necessary, so is $cn$.

2. $\Omega(n)$ contains all functions that $\Omega(n \log n)$ contains.

Is it correct to say MergeSort has lower bound $\Omega(n^2)$?
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of an algorithm

It is correct to say MergeSort has lower bound $\Omega(n \log n)$:

(1) since $cn \log n$ is necessary, so is $cn$.

(2) $\Omega(n)$ contains all functions that $\Omega(n \log n)$ contains.

Is it correct to say MergeSort has lower bound $\Omega(n^2)$?
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of **an algorithm**

A time **necessary** (i.e., **needed**) for the algorithm to solve all instances.
Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

$l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.
Lower bound of **an algorithm**

A time **necessary** (i.e., **needed**) for the algorithm to solve all instances.

$l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

- e.g., *MergeSort* has lower bound $\Omega(n \log n)$
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

\( l(n) \) is a lower bound – if some (generic) instance requires time \( l(n) \) or more to be solved by the algorithm.

e.g., MergeSort has lower bound \( \Omega(n \log n) \)

Is it correct to say MergeSort has lower bound \( \Omega(n) \)?
Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

$l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

e.g., MergeSort has lower bound $\Omega(n \log n)$

Is it correct to say MergeSort has lower bound $\Omega(n)$?

It is correct for two reasons:
Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

$l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

e.g., \texttt{MergeSort} has lower bound $\Omega(n \log n)$

Is it correct to say \texttt{MergeSort} has lower bound $\Omega(n)$?

It is correct for two reasons:

(1) since $cn \log n$ is necessary, so is $cn$. 
Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

$l(n)$ is a lower bound – if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

e.g., MergeSort has lower bound $\Omega(n \log n)$

Is it correct to say MergeSort has lower bound $\Omega(n)$?

It is correct for two reasons:

1. since $cn \log n$ is necessary, so is $cn$.
2. $\Omega(n)$ contains all functions that $\Omega(n \log n)$ contains.
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances.

\( l(n) \) is a lower bound – if some (generic) instance requires time \( l(n) \) or more to be solved by the algorithm.

e.g., MergeSort has lower bound \( \Omega(n \log n) \)

Is it correct to say MergeSort has lower bound \( \Omega(n) \)?

It is correct for two reasons:

1. since \( cn \log n \) is necessary, so is \( cn \).
2. \( \Omega(n) \) contains all functions that \( \Omega(n \log n) \) contains.
Lower bound of an algorithm

A time necessary (i.e., needed) for the algorithm to solve all instances. $l(n)$ is a lower bound — if some (generic) instance requires time $l(n)$ or more to be solved by the algorithm.

e.g., MergeSort has lower bound $\Omega(n \log n)$

Is it correct to say MergeSort has lower bound $\Omega(n)$?

It is correct for two reasons:

(1) since $cn \log n$ is necessary, so is $cn$.

(2) $\Omega(n)$ contains all functions that $\Omega(n \log n)$ contains.

Is it correct to say MergeSort has lower bound $\Omega(n^2)$?
Chapter 8. Lower Bounds and Sorting in Linear Time

• The best known upper bound for MergeSort is $O(n \log n)$, coinciding with the best known lower bound $\Omega(n \log n)$.

• Both bounds are tight (i.e., optimal). Thus complexity is denoted with $\theta(n \log n)$, meaning both $O(n \log n)$ and $\Omega(n \log n)$.

• We may not be so lucky for some other algorithms.
Chapter 8. Lower Bounds and Sorting in Linear Time

- The best known upper bound for **MERGE SORT** is $O(n \log n)$,
• The best known upper bound for **MergeSort** is $O(n \log n)$, coinciding with the best known lower bound $\Omega(n \log n)$;
• The best known upper bound for MergeSort is $O(n \log n)$, coinciding with the best known lower bound $\Omega(n \log n)$;

• Both bounds are tight (i.e., optimal). Thus complexity is denoted with $\theta(n \log n)$, meaning both $O(n \log n)$ and $\Omega(n \log n)$. 


• The best known upper bound for MERGESORT is $O(n \log n)$, coinciding with the best known lower bound $\Omega(n \log n)$;

• Both bounds are tight (i.e., optimal). Thus complexity is denoted with $\theta(n \log n)$, meaning both $O(n \log n)$ and $\Omega(n \log n)$.

• We may not be so lucky for some other algorithms.
Rec-Fibonacci($n$)

if $n = 1$ or $n = 2$, return (1);
else
  $T_1 = \text{Rec-Fibonacci}(n - 1)$;
  $T_2 = \text{Rec-Fibonacci}(n - 2)$;
return ($T_1 + T_2$);
Chapter 8. Lower Bounds and Sorting in Linear Time

\begin{verbatim}
Rec-Fibonacci(n)
    if \( n = 1 \) or \( n = 2 \), return (1);
else
    \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
    \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
    return \( (T_1 + T_2) \);

Derive an upper bound:
\end{verbatim}
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci** \( (n) \)

if \( n = 1 \) or \( n = 2 \), return \((1)\);
else
    \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
    \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
    return \((T_1 + T_2)\);

Derive an upper bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

\textbf{Rec-Fibonacci}(n)

\textbf{if} \ n = 1 \ or \ n = 2, \ \textbf{return} \ (1);

\textbf{else}

\quad T_1 = \textbf{Rec-Fibonacci}(n - 1);

\quad T_2 = \textbf{Rec-Fibonacci}(n - 2);

\textbf{return} \ (T_1 + T_2);

Derive an upper bound:

\[ T(n) = c + T(n - 1) + T(n - 2), \] with \[ T(1) = T(2) = c \]

\[ \leq c + 2T(n - 1) \]
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci\((n)\)

\[
\begin{align*}
\text{if } n = 1 \text{ or } n = 2, & \text{ return } (1); \\
\text{else} & \\
& T_1 = \text{Rec-Fibonacci}(n - 1); \\
& T_2 = \text{Rec-Fibonacci}(n - 2); \\
& \text{return } (T_1 + T_2);
\end{align*}
\]

Derive an upper bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]
\[
\leq c + 2T(n - 1)
\]
\[
\leq c + 2c + 2^2 T(n - 2)
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else
    \( T_1 = \text{Rec-Fibonacci}(n - 1); \)
    \( T_2 = \text{Rec-Fibonacci}(n - 2); \)
    return \((T_1 + T_2)\);

Derive an upper bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \\
\leq c + 2T(n - 1) \\
\leq c + 2c + 2^2T(n - 2) \\
\ldots
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**

\[
\text{if } n = 1 \text{ or } n = 2, \text{ return } (1); \\
\text{else} \\
T_1 = \text{Rec-Fibonacci}(n - 1); \\
T_2 = \text{Rec-Fibonacci}(n - 2); \\
\text{return } (T_1 + T_2);
\]

Derive an upper bound:

\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \\
\leq c + 2T(n - 1) \\
\leq c + 2c + 2^2T(n - 2) \\
\ldots \\
\leq c + 2c + \ldots 2^{n-3}c + 2^{n-2}T(2)
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else

\( T_1 = \text{Rec-Fibonacci}(n - 1); \)
\( T_2 = \text{Rec-Fibonacci}(n - 2); \)

return \((T_1 + T_2); \)

Derive an upper bound:
\( T(n) = c + T(n - 1) + T(n - 2) \), with \( T(1) = T(2) = c \)
\( \leq c + 2T(n - 1) \)
\( \leq c + 2c + 2^2T(n - 2) \)
\( \ldots \)
\( \leq c + 2c + \ldots 2^{n-3}c + 2^{n-2}T(2) \)
\( = \frac{2^{n-2} - 1}{2-1}c + 2^{n-2}c \)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci** \((n)\)

\[
\text{if } n = 1 \text{ or } n = 2, \text{ return } (1);
\]

\[
\text{else}
\]

\[
T_1 = \text{Rec-Fibonacci}(n - 1);
\]

\[
T_2 = \text{Rec-Fibonacci}(n - 2);
\]

\[
\text{return } (T_1 + T_2);
\]

Derive an upper bound:

\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]

\[
\leq c + 2T(n - 1)
\]

\[
\leq c + 2c + 2^2T(n - 2)
\]

\[
\ldots
\]

\[
\leq c + 2c + \ldots 2^{n-3}c + 2^{n-2}T(2)
\]

\[
= \frac{2^{n-2}-1}{2-1}c + 2^{n-2}c
\]

\[
= (2^{n-2} - 1)c + 2^{n-2}c
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**

if \( n = 1 \) or \( n = 2 \), return \((1)\);
else

\[ T_1 = \text{Rec-Fibonacci}(n - 1); \]
\[ T_2 = \text{Rec-Fibonacci}(n - 2); \]

return \((T_1 + T_2)\);

Derive an upper bound:
\( T(n) = c + T(n - 1) + T(n - 2) \), with \( T(1) = T(2) = c \)

\[ \leq c + 2T(n - 1) \]
\[ \leq c + 2c + 2^2T(n - 2) \]

\[ \ldots \]
\[ \leq c + 2c + \ldots 2^{n-3}c + 2^{n-2}T(2) \]
\[ = \frac{2^{n-2}-1}{2-1}c + 2^{n-2}c \]
\[ = (2^{n-2} - 1)c + 2^{n-2}c \]
\[ = (2^{n-1} - 1)c \]

\[ = O(2^n) \]
Rec-Fibonacci($n$)

if $n = 1$ or $n = 2$, return (1);
else
    $T_1 = \text{Rec-Fibonacci}(n - 1);
    T_2 = \text{Rec-Fibonacci}(n - 2);
    \text{return } (T_1 + T_2);

Derive an upper bound:
$T(n) = c + T(n - 1) + T(n - 2)$, with $T(1) = T(2) = c$

$\leq c + 2T(n - 1)$
$\leq c + 2c + 2^2 T(n - 2)$

$\leq c + 2c + \ldots 2^{n-3}c + 2^{n-2}T(2)$
$= \frac{2^{n-2} - 1}{2-1}c + 2^{n-2}c$
$= (2^{n-2} - 1)c + 2^{n-2}c$
$= (2^{n-1} - 1)c$
$= O(2^n).$
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**

if $n = 1$ or $n = 2$, return (1);
else
    $T_1 = \text{Rec-Fibonacci}(n - 1)$;
    $T_2 = \text{Rec-Fibonacci}(n - 2)$;
return ($T_1 + T_2$);

Derive a lower bound:

$$T(n) = c + T(n - 1) + T(n - 2),$$
with $T(1) = T(2) = c \geq 2T(n - 2) \geq 2^2 T(n - 4) \cdots \geq 2^{n-2} T(2) = 2^{n-2} 2^2 c = 2^{n-2} 2^2 c = c^2 (212) \leq c^2 2^{n-2} \geq c^2 1.41 n.$$

while the derived upper bound is $O(2^n)$, not tight!
Chapter 8. Lower Bounds and Sorting in Linear Time

\textbf{Rec-Fibonacci}(n)

\begin{verbatim}
if \( n = 1 \) or \( n = 2 \), return (1);
else
  \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
  \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
return \( (T_1 + T_2) \);
\end{verbatim}

Derive a lower bound:

\[ T(n) = c + T(n - 1) + T(n - 2), \quad \text{with} \quad T(1) = T(2) = c \geq 2T(n - 2) \geq 2^2T(n - 4) \cdots \geq 2^nT(2) = 2^n2c = 2^n2c = c2^{1.41n} \geq c2^{1.41n}. \]

while the derived upper bound is \( O(2^n) \), not tight!
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci($n$)

    if $n = 1$ or $n = 2$, return (1);
else
t_1 = Rec-Fibonacci($n - 1$);
t_2 = Rec-Fibonacci($n - 2$);
return ($t_1 + t_2$);

Derive a lower bound:
$T(n) = c + T(n - 1) + T(n - 2)$, with $T(1) = T(2) = c$
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**

if \( n = 1 \) or \( n = 2 \), return (1);
else

\[ T_1 = \text{Rec-Fibonacci}(n - 1); \]
\[ T_2 = \text{Rec-Fibonacci}(n - 2); \]

return \((T_1 + T_2)\);

Derive a lower bound:

\[ T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \]
\[ \geq 2T(n - 2) \]

...
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci($n$)

if $n = 1$ or $n = 2$, return (1);
else

$T_1 = \text{Rec-Fibonacci}(n - 1)$;
$T_2 = \text{Rec-Fibonacci}(n - 2)$;

return ($T_1 + T_2$);

Derive a lower bound:
$T(n) = c + T(n - 1) + T(n - 2)$, with $T(1) = T(2) = c$

$\geq 2T(n - 2)$

$\geq 2^2T(n - 4)$
**Chapter 8. Lower Bounds and Sorting in Linear Time**

**Rec-Fibonacci (n)**

```cpp
if n = 1 or n = 2, return (1);
else
    T_1 = Rec-Fibonacci(n - 1);
    T_2 = Rec-Fibonacci(n - 2);
    return (T_1 + T_2);
```

Derive a lower bound:

\[ T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \]

\[ \geq 2T(n - 2) \]

\[ \geq 2^2T(n - 4) \]

\[ \ldots \]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**\( (n) \)

\[
\begin{align*}
\text{if } n = 1 \text{ or } n = 2, & \quad \text{return } (1); \\
\text{else} & \\
T_1 = \text{Rec-Fibonacci}(n - 1); \\
T_2 = \text{Rec-Fibonacci}(n - 2); \\
\text{return } (T_1 + T_2);
\end{align*}
\]

Derive a lower bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \\
\geq 2T(n - 2) \\
\geq 2^2 T(n - 4) \\
\ldots \\
\geq 2^{\frac{n-2}{2}} T(2)
\]

while the derived upper bound is \( O(2^n) \), not tight!
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else
\[ T_1 = \text{Rec-Fibonacci}(n - 1); \]
\[ T_2 = \text{Rec-Fibonacci}(n - 2); \]
return \((T_1 + T_2)\);

Derive a lower bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \quad \text{with} \quad T(1) = T(2) = c
\]
\[
\geq 2T(n - 2)
\]
\[
\geq 2^2T(n - 4)
\]
\[
\ldots
\]
\[
\geq 2^{\frac{n-2}{2}}T(2)
\]
\[
= 2^{\frac{n-2}{2}}c
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else
  \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
  \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
  return \( T_1 + T_2 \);

Derive a lower bound:
\[ T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \]
\[ \geq 2T(n - 2) \]
\[ \geq 2^2 T(n - 4) \]
\[ \ldots \]
\[ \geq 2 \frac{n-2}{2} T(2) \]
= \( 2 \frac{n-2}{2} c \)
= \( 2 \frac{n}{2} 2^{-2} c \)
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else
    \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
    \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
return \( T_1 + T_2 \);

Derive a lower bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]
\[
\geq 2T(n - 2)
\]
\[
\geq 2^2T(n - 4)
\]
\[
\cdots
\]
\[
\geq 2^{\frac{n-2}{2}}T(2)
\]
\[
= 2^{\frac{n-2}{2}} c
\]
\[
= 2^{\frac{n}{2}} 2^{\frac{2}{2}} c
\]
\[
= c \left(2^{\frac{1}{2}}\right)^n
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

Rec-Fibonacci\((n)\)

\[
\begin{align*}
\text{if } n = 1 \text{ or } n = 2, & \quad \text{return } (1); \\
\text{else} & \\
& \quad T_1 = \text{Rec-Fibonacci}(n - 1); \\
& \quad T_2 = \text{Rec-Fibonacci}(n - 2); \\
& \quad \text{return } (T_1 + T_2);
\end{align*}
\]

Derive a lower bound:

\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]

\[
\geq 2T(n - 2)
\]

\[
\geq 2^2T(n - 4)
\]

\[
\ldots
\]

\[
\geq 2\frac{n-2}{2} T(2)
\]

\[
= 2\frac{n-2}{2} c
\]

\[
= 2\frac{n}{2} 2\frac{-2}{2} c
\]

\[
= \frac{c}{2} (2^{\frac{1}{2}})^n
\]

\[
= \frac{c}{2} \sqrt{2^n}
\]

while the derived upper bound is $O(2^n)$, not tight!
Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else
    \( T_1 = \text{Rec-Fibonacci}(n - 1) \);
    \( T_2 = \text{Rec-Fibonacci}(n - 2) \);
    return \( (T_1 + T_2) \);

Derive a lower bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]
\[
\geq 2T(n - 2)
\]
\[
\geq 2^2 T(n - 4)
\]
\[
\ldots
\]
\[
\geq 2^{\frac{n-2}{2}} T(2)
\]
\[
= 2^{\frac{n-2}{2}} c
\]
\[
= 2^{\frac{n}{2}} 2^{-\frac{2}{2}} c
\]
\[
= \frac{c}{2} (2^{\frac{1}{2}})^n
\]
\[
= \frac{c}{2} \sqrt{2}^n
\]
\[
\geq \frac{c}{2} 1.41^n
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci**

\[
\text{if } n = 1 \text{ or } n = 2, \text{ return } (1); \\
\text{else} \\
T_1 = \text{Rec-Fibonacci}(n - 1); \\
T_2 = \text{Rec-Fibonacci}(n - 2); \\
\text{return } (T_1 + T_2);
\]

Derive a lower bound:
\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c \\
\geq 2T(n - 2) \\
\geq 2^2 T(n - 4) \\
\ldots \\
\geq 2^{\frac{n-2}{2}} T(2) \\
= 2^{\frac{n-2}{2}} c \\
= 2^{\frac{n}{2}} 2^{\frac{n-2}{2}} c \\
= \frac{c}{2} (2^{\frac{1}{2}})^n \\
= \frac{c}{2} \sqrt{2}^n \\
\geq \frac{c}{2} 1.41^n \\
= \Omega(1.41^n).
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Rec-Fibonacci** \(n\)

\[
\text{if } n = 1 \text{ or } n = 2, \text{ return } (1);
\]

\[
\text{else}
\]

\[
T_1 = \text{Rec-Fibonacci}(n - 1);
\]

\[
T_2 = \text{Rec-Fibonacci}(n - 2);
\]

\[
\text{return } (T_1 + T_2);
\]

Derive a lower bound:

\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]

\[
\geq 2T(n - 2)
\]

\[
\geq 2^2 T(n - 4)
\]

\[
\ldots
\]

\[
\geq 2 \frac{n-2}{2} T(2)
\]

\[
= 2 \frac{n-2}{2} c
\]

\[
= 2 \frac{n}{2} 2^{-\frac{n}{2}} c
\]

\[
= \frac{c}{2} (2^{\frac{1}{2}})^n
\]

\[
= \frac{c}{2} \sqrt{2}^n
\]

\[
\geq \frac{c}{2} 1.41^n
\]

\[
= \Omega(1.41^n). \text{ while the derived upper bound is } O(2^n),
\]
Rec-Fibonacci(n)

if \( n = 1 \) or \( n = 2 \), return (1);
else

\[
T_1 = \text{Rec-Fibonacci}(n - 1);
T_2 = \text{Rec-Fibonacci}(n - 2);
\]

return \((T_1 + T_2)\);

Derive a lower bound:

\[
T(n) = c + T(n - 1) + T(n - 2), \text{ with } T(1) = T(2) = c
\]

\[
\geq 2T(n - 2)
\]

\[
\geq 2^2T(n - 4)
\]

\[
\ldots
\]

\[
\geq 2^{\frac{n-2}{2}}T(2)
\]

\[
= 2^{\frac{n-2}{2}}c
\]

\[
= 2^{\frac{n}{2}}2^{-\frac{2}{2}}c
\]

\[
= \frac{c}{2}(2^{\frac{1}{2}})^n
\]

\[
= \frac{c}{2}\sqrt{2}^n
\]

\[
\geq \frac{c}{2}1.41^n
\]

\[
= \Omega(1.41^n). \text{ while the derived upper bound is } O(2^n), \text{ not tight!}
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

Upper bound of a problem: A time sufficient (i.e., enough) to solve all instances of the problem. To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem. e.g., $O(n^2)$ is an upper bound for Sorting (why?) $O(n \log n)$ is also an upper bound for Sorting (why?)

One important task in algorithm research: to design algorithms achieving better upper bounds (smaller time complexity)
Upper bound of a problem
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem;
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem.
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem.

e.g., $O(n^2)$ is an upper bound for **Sorting**
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem.

e.g., $O(n^2)$ is an upper bound for Sorting (why?)
Upper bound of a problem

A time sufficient (i.e., enough) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem.

e.g., $O(n^2)$ is an upper bound for Sorting (why?)

$O(n \log n)$ is also an upper bound for Sorting (why?)
Upper bound of a **problem**

A time **sufficient** (i.e., **enough**) to solve all instances of the problem.

To derive an upper bound, we can resort to algorithms solving the problem; an upper bound is of such an algorithm is also an upper bound for the problem.

- e.g., \(O(n^2)\) is an upper bound for **Sorting** (why?)
  - \(O(n \log n)\) is also an upper bound for **Sorting** (why?)

One important task in algorithm research: to design algorithms achieving better upper bounds (smaller time complexity)
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem: A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound? e.g., consider the Sorting problem. Insertion Sort has lower bound $\Omega(n^2)$ (why?). Can we say the Sorting problem has lower bound $\Omega(n^2)$? No! because MergeSort has upper bound $O(n \log n)$.

Likewise, we cannot say the Sorting has lower bound $\Omega(n \log n)$.

Statement "problem Sorting has lower bound $\Omega(n \log n)$" ⇐⇒ statement "there is no algorithm running faster than time $cn \log n$".
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem
A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?
e.g., consider Sorting problem,

Insertion Sort has lower bound $\Omega(n^2)$ (why?),
Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

\textbf{Insertion Sort} has lower bound $\Omega(n^2)$ (why?),
Can we say Sorting problem has lower bound $\Omega(n^2)$?
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a **problem**
A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider **Sorting** problem,

**Insertion** **Sort** has lower bound \( \Omega(n^2) \) (**why?**),
Can we say **Sorting** problem has lower bound \( \Omega(n^2) \)?
No!
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

**Insertion Sort** has lower bound $\Omega(n^2)$ (why?),
Can we say Sorting problem has lower bound $\Omega(n^2)$?
No! because ...
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem
A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

\textbf{Insertion Sort} has lower bound $\Omega(n^2)$ (why?),
Can we say \textbf{Sorting} problem has lower bound $\Omega(n^2)$?
\textbf{No!} because . . .
Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

**Insertion Sort** has lower bound $\Omega(n^2)$ (why?),
Can we say Sorting problem has lower bound $\Omega(n^2)$?
No! because ...

**MergeSort** has ...
Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider **Sorting** problem,

**Insertion Sort** has lower bound \( \Omega(n^2) \) (why?),

Can we say **Sorting** problem has lower bound \( \Omega(n^2) \)?

No! because . . .

**MergeSort** has . . . upper bound \( O(n \log n) \).
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

\textbf{Insertion Sort} has lower bound $\Omega(n^2)$ (why?),

Can we say Sorting problem has lower bound $\Omega(n^2)$?

No! because ...

\textbf{MergeSort} has ... upper bound $O(n \log n)$.

Likewise, we cannot say Sorting has lower bound $\Omega(n \log n)$. 
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

**Insertion Sort** has lower bound \( \Omega(n^2) \) (why?),
Can we say Sorting problem has lower bound \( \Omega(n^2) \)?
No! because ...

**MergeSort** has ... upper bound \( O(n \log n) \).
Likewise, we cannot say Sorting has lower bound \( \Omega(n \log n) \).
Statement “problem Sorting has lower bound \( \Omega(n \log n) \)”
Chapter 8. Lower Bounds and Sorting in Linear Time

Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

**Insertion Sort** has lower bound $\Omega(n^2)$ (why?),
Can we say **Sorting** problem has lower bound $\Omega(n^2)$?
No! because ...

**MergeSort** has ... upper bound $O(n \log n)$.

Likewise, we cannot say **Sorting** has lower bound $\Omega(n \log n)$.

Statement “problem **Sorting** has lower bound $\Omega(n \log n)$”
$\iff$
Lower bound of a problem

A time necessary (i.e., needed) for all instances in the problem to be solved.

Can we use an algorithm lower bound for the problem lower bound?

e.g., consider Sorting problem,

Insertion Sort has lower bound $\Omega(n^2)$ (why?),
Can we say Sorting problem has lower bound $\Omega(n^2)$?
No! because ...

MergeSort has ... upper bound $O(n \log n)$.
Likewise, we cannot say Sorting has lower bound $\Omega(n \log n)$.

Statement “problem Sorting has lower bound $\Omega(n \log n)$”
$\iff$
statement “there is no algorithm running faster than time $cn \log n$”.
Statement “problem Sorting has lower bound $\Omega(n \log n)$”
$\iff$
statement “there is no algorithm running faster than time $cn \log n$”. 
Statement “problem \textbf{Sorting} has lower bound \( \Omega(n \log n) \)”
\[ \iff \]
statement “there is no algorithm running faster than time \( cn \log n \)”.

• To derive a lower bound for a \textbf{problem}, we \textbf{cannot} examine an infinite number of algorithms!
Chapter 8. Lower Bounds and Sorting in Linear Time

Statement “problem **Sorting** has lower bound $\Omega(n \log n)$”

$\iff$

statement “there is no algorithm running faster than time $cn \log n$”.

- To derive a lower bound for a **problem**, we **cannot** examine an infinite number of algorithms!

- Lower bounds can only be derived mathematically, but not from existing algorithms.
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

Claim 1: Total number of leaves is $\geq n!$.

Claim 2: The height of the tree at least $\geq \log n!$.

(The minimum of heights of all such trees!)
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting with decision tree as algorithm/computation model

- each internal node denotes \( x_i \leq x_j \), with two outcomes

**Claim 1**: Total number of leaves is \( \geq n! \).

**Claim 2**: The height of the tree at least \( \geq \log n! \).
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes \((x_i \leq x_j)\), with two outcomes
- each path corresponds to one possible outcome of the algorithm

Claim 1: Total number of leaves is \(\geq n!\).

Claim 2: The height of the tree at least \(\geq \log n!\).

(The minimum of heights of all such trees!)
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting with decision tree as algorithm/computation model

- each internal node denotes \((x_i \leq x_j)\), with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list \((1, 2, \ldots, n)\)

Claim 1: Total number of leaves is \(\geq n!\).

Claim 2: The height of the tree at least \(\geq \log n!\).

(The minimum of heights of all such trees!)
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting with decision tree as algorithm/computation model

- each internal node denotes \((x_i \leq x_j)\), with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list \((1, 2, \ldots, n)\)

Claim 1: Total number of leaves is \(\geq n!\).

Claim 2: The height of the tree at least \(\geq \log n!\).

(The minimum of heights of all such trees!)

![Decision tree diagram](image.png)
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting
with decision tree as algorithm/computation model

- each internal node denotes \((x_i \leq x_j)\), with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list \((1, 2, \ldots, n)\)

Claim 1: Total number of leaves is \(\geq n!\).
Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting with decision tree as algorithm/computation model

- each internal node denotes \( x_i \leq x_j \), with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list \((1, 2, \ldots, n)\)

Claim 1: Total number of leaves is \( \geq n! \).

Claim 2: The height of the tree at least \( \geq \log n! \).
   (The minimum of heights of all such trees!)
Chapter 8. Lower Bounds and Sorting in Linear Time

\textit{Claim 1:} Total number of leaves is $\geq n!$. 

Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.
Chapter 8. Lower Bounds and Sorting in Linear Time

Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.
Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.

Claim 2: The height of every decision tree for Sorting is at least $\log_2 n!$.
Chapter 8. Lower Bounds and Sorting in Linear Time

Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.

Claim 2: The height of every decision tree for Sorting is at least $\log_2 n!$.

Proof: [proof by contradiction]
Chapter 8. Lower Bounds and Sorting in Linear Time

Claim 1: Total number of leaves is \( \geq n! \).

Proof: Because any decision tree for Sorting problem has to take care of all possible \( n! \) outcomes.

Claim 2: The height of every decision tree for Sorting is at least \( \log_2 n! \).

Proof: [proof by contradiction]

Assume some decision tree \( T \) for Sorting has height \( h < \log_2 n! \).
Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.

Claim 2: The height of every decision tree for Sorting is at least $\log_2 n!$.

Proof: [proof by contradiction]

Assume some decision tree $T$ for Sorting has height $h < \log_2 n!$.

This means every branch in $T$ has length $< \log_2 n!$. 

Claim 1: Total number of leaves is $\geq n!$.

Proof: Because any decision tree for Sorting problem has to take care of all possible $n!$ outcomes.

Claim 2: The height of every decision tree for Sorting is at least $\log_2 n!$.

Proof: [proof by contradiction]

Assume some decision tree $T$ for Sorting has height $h < \log_2 n!$.

This means every branch in $T$ has length $< \log_2 n!$.

The number of leaves is at most $< 2^h - 1 < 2^h = n!$, contradicts Claim 1.
**Theorem**: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.
Chapter 8. Lower Bounds and Sorting in Linear Time

**Theorem:** Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

**Prove.**
The longest path from the root to a leaf is $\Omega(\log n!)$. I.e., the number of comparisons needed in the worst case is $\Omega(\log n!)$. 

$n! = n(n-1)(n-2)\cdots(n-n+2)(n-n+1)\cdot\cdot\cdot2\times1 \geq (n/2)^n$ 

or by Stirling's formula:

$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n))$

$\Omega(\log(n!)) = \Omega(n \log n)$
Chapter 8. Lower Bounds and Sorting in Linear Time

**Theorem:** Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

**Prove.**
The longest path from the root to a leave is $\Omega(\log n!)$. I.e., the number of comparisons needed in the worst case is $\Omega(\log n!)$.  

\[
n! = n(n - 1)(n - 2) \cdots (n - \frac{n}{2})(n - \frac{n}{2} - 1) \cdots 2 \times 1 \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \times 2^{\frac{n}{2} - 1} \geq \frac{1}{2} n \frac{n}{2}
\]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Theorem**: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

**Prove.**
The longest path from the root to a leaf is $\Omega(\log n!)$. I.e., the number of comparisons needed in the worst case is $\Omega(\log n!)$.

$$n! = n(n-1)(n-2) \cdots (n - \frac{n}{2})(n - \frac{n}{2} - 1) \cdots 2 \times 1$$

$$\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \times 2^{\frac{n}{2} - 1} \geq \frac{1}{2} n^\frac{n}{2}$$

or by Stirling’s formula:

$$n! = \sqrt{2\pi n} (n/e)^n (1 + O(1/n))$$
Theorem: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

Prove.
The longest path from the root to a leaf is $\Omega(\log n!)$. I.e., the number of comparisons needed in the worst case is $\Omega(\log n!)$. 

$$n! = n(n-1)(n-2) \cdots (n-\frac{n}{2})(n-\frac{n}{2}-1) \cdots 2 \times 1$$

$$\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \times 2^{\frac{n}{2} - 1} \geq \frac{1}{2} n^{\frac{n}{2}}$$

or by Stirling’s formula:

$$n! = \sqrt{2\pi n} (n/e)^n (1 + O(1/n))$$

$$\Omega(\log(n!)) = \Omega(n \log n)$$
Sorting algorithms with worst case linear time
(To be covered after the next chapter)
Sorting algorithms with worst case linear time
(To be covered after the next chapter)

• count sort
• radix sort
• bucket sort
Count sort

Algorithm Counting-Sort \((A, B, k)\) {
1. \(A\) contains \(n\) integers;
2. \(k\) is the max;
3. for \(i = 0\) to \(k\)
4. \(C[i] = 0\)
5. for \(j = 1\) to length \([A]\)
6. \(C[A[j]] = C[A[j]] + 1\)
7. for \(i = 1\) to \(k\)
8. \(C[i] = C[i] + C[i - 1]\)
9. \(C[i]\) contains the number of elements whose values \(\leq i\);
10. for \(j = \text{length}[A]\) downto \(1\)
11. \(B[C[A[j]]] = A[j]\)
12. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2 5 3 0 2 3 0 3, k = 5, C: 2 0 2 3 0 1\)

analysis: \(T(n) = O(k + n)\)
**Count sort**

Algorithm **COUNTING-SORT** \((A, B, k)\)  
{\(A\) contains \(n\) integers; \(k\) is the max}
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT \((A, B, k)\) \{\(A\) contains \(n\) integers; \(k\) is the max\}
1. for \(i = 0\) to \(k\)
Count sort

Algorithm COUNTING-SORT \((A, B, k)\) \(\{A\) contains \(n\) integers; \(k\) is the max\}
1. \textbf{for} \(i = 0\) \textbf{to} \(k\)
2. \(C[i] = 0\)
### Count sort

Algorithm **COUNTING-SORT** \((A, B, k)\) \(\{A \text{ contains } n \text{ integers; } k \text{ is the max}\}\)

1. \textbf{for} \(i = 0\) \textbf{to} \(k\)
2. \(C[i] = 0\)
3. \textbf{for} \(j = 1\) \textbf{to} \(\text{length}[A]\)
Count sort

Algorithm COUNTING-SORT ($A$, $B$, $k$) \{ $A$ contains $n$ integers; $k$ is the max \}
1. for $i = 0$ to $k$
2. \hspace{1em} $C[i] = 0$
3. for $j = 1$ to length[$A$]
4. \hspace{1em} $C[A[j]] = C[A[j]] + 1$
Count sort

Algorithm Counting-Sort \((A, B, k)\) \(\{A\) contains \(n\) integers; \(k\) is the max\}

1. \(\textbf{for} \ i = 0 \ \textbf{to} \ k\)
2. \(C[i] = 0\)
3. \(\textbf{for} \ j = 1 \ \textbf{to} \ \text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values } = i\}\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm **COUNTING-SORT** \((A, B, k)\) \{\(A\) contains \(n\) integers; \(k\) is the max\}

1. for \(i = 0\) to \(k\)
2. \(C[i] = 0\)
3. for \(j = 1\) to \(\text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i]\} \text{ contains the number of elements whose values } = i\}\)
6. for \(i = 1\) to \(k\)
7. \(C[i] = C[i] + C[i-1]\)
8. \(\{C[i]\} \text{ contains the number of elements whose values } \leq i\}\)
9. for \(j = \text{length}[A]\) down to \(1\)
10. \(B[C[A[j]]] = A[j]\)
11. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3\), \(k = 5\), \(C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)

Analysis: \(T(n) = O(k + n)\)
Count sort

Algorithm Counting-Sort \((A, B, k)\) \{\(A\) contains \(n\) integers; \(k\) is the max\}

1. \(\text{for} \ i = 0 \ \text{to} \ k\)
2. \(C[i] = 0\)
3. \(\text{for} \ j = 1 \ \text{to} \ \text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values} = i\}\)
6. \(\text{for} \ i = 1 \ \text{to} \ k\)
7. \(C[i] = C[i] + C[i - 1]\)

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3\), \(k = 5\), \(C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)

Analysis: \(T(n) = O(k + n)\)
Count sort

Algorithm COUNTING-SORT ($A, B, k$)  

1. \textbf{for} $i = 0$ \textbf{to} $k$
2. \hfill $C[i] = 0$
3. \textbf{for} $j = 1$ \textbf{to} length[$A$]
4. \hfill $C[A[j]] = C[A[j]] + 1$
5. \hfill \{$C[i]$ contains the number of elements whose values $= i$\}
6. \textbf{for} $i = 1$ \textbf{to} $k$
7. \hfill $C[i] = C[i] + C[i - 1]$
8. \hfill \{$C[i]$ contains the number of elements whose values $\leq i$\}
Count sort

Algorithm Counting-Sort \((A, B, k)\) \{\(A\) contains \(n\) integers; \(k\) is the max\}
1. \textbf{for} \(i = 0\ \textbf{to} \ k\)
2. \hspace{1em} \(C[i] = 0\)
3. \textbf{for} \(j = 1\ \textbf{to} \ \text{length}[A]\)
4. \hspace{1em} \(C[A[j]] = C[A[j]] + 1\)
5. \{\(C[i]\) contains the number of elements whose values = \(i\)\}
6. \textbf{for} \(i = 1\ \textbf{to} \ k\)
7. \hspace{1em} \(C[i] = C[i] + C[i - 1]\)
8. \{\(C[i]\) contains the number of elements whose values \(\leq i\)\}
9. \textbf{for} \(j = \text{length}[A]\ \textbf{down to} \ 1\)
Count sort

Algorithm Counting-Sort $(A, B, k)$ \{ $A$ contains $n$ integers; $k$ is the max\}

1. for $i = 0$ to $k$
2. \hspace{1em} $C[i] = 0$
3. for $j = 1$ to length[$A$]
4. \hspace{1em} $C[A[j]] = C[A[j]] + 1$
5. \hspace{1em} \{ $C[i]$ contains the number of elements whose values $= i$ \}
6. for $i = 1$ to $k$
7. \hspace{1em} $C[i] = C[i] + C[i - 1]$
8. \hspace{1em} \{ $C[i]$ contains the number of elements whose values $\leq i$ \}
9. for $j =$ length[$A$] downto 1
10. \hspace{1em} $B[C[A[j]]] = A[j]$
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT \((A, B, k)\) \{A contains \(n\) integers; \(k\) is the max\}

1. \(\text{for } i = 0 \text{ to } k\)
2. \(C[i] = 0\)
3. \(\text{for } j = 1 \text{ to } \text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values } = i\}\)
6. \(\text{for } i = 1 \text{ to } k\)
7. \(C[i] = C[i] + C[i - 1]\)
8. \(\{C[i] \text{ contains the number of elements whose values } \leq i\}\)
9. \(\text{for } j = \text{length}[A] \text{ downto } 1\)
10. \(B[C[A[j]]] = A[j]\)
11. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3, \ k = 5, \ C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT \((A, B, k)\) \(\{A \text{ contains } n \text{ integers; } k \text{ is the max}\}\)

1. \(\text{for } i = 0 \text{ to } k\)
2. \(C[i] = 0\)
3. \(\text{for } j = 1 \text{ to } \text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values } = i\}\)
6. \(\text{for } i = 1 \text{ to } k\)
7. \(C[i] = C[i] + C[i - 1]\)
8. \(\{C[i] \text{ contains the number of elements whose values } \leq i\}\)
9. \(\text{for } j = \text{length}[A] \text{ down to } 1\)
10. \(B[C[A[j]]] = A[j]\)
11. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3, \ k = 5,\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT \((A, B, k)\) \(\{A\) contains \(n\) integers; \(k\) is the max\}

1. \textbf{for} \(i = 0\) \textbf{to} \(k\)
2. \hspace{.5cm} \(C[i] = 0\)
3. \textbf{for} \(j = 1\) \textbf{to} \(\text{length}[A]\)
4. \hspace{.5cm} \(C[A[j]] = C[A[j]] + 1\)
5. \hspace{.5cm} \{\(C[i]\) contains the number of elements whose values \(= i\)\}
6. \textbf{for} \(i = 1\) \textbf{to} \(k\)
7. \hspace{.5cm} \(C[i] = C[i] + C[i - 1]\)
8. \hspace{.5cm} \{\(C[i]\) contains the number of elements whose values \(\leq i\)\}
9. \textbf{for} \(j = \text{length}[A]\) \textbf{downto} 1
10. \hspace{.5cm} \(B[C[A[j]]] = A[j]\)
11. \hspace{.5cm} \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2\ 5\ 3\ 0\ 2\ 3\ 0\ 3,\ k = 5,\ C: 2\ 0\ 2\ 3\ 0\ 1\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm Counting-Sort \((A, B, k)\) \{ \(A\) contains \(n\) integers; \(k\) is the max\}

1. for \(i = 0\) to \(k\)
2. \(C[i] = 0\)
3. for \(j = 1\) to length\([A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values } = i\}\)
6. for \(i = 1\) to \(k\)
7. \(C[i] = C[i] + C[i - 1]\)
8. \(\{C[i] \text{ contains the number of elements whose values } \leq i\}\)
9. for \(j = \text{length}[A]\) \textbf{downto} 1
10. \(B[C[A[j]]] = A[j]\)
11. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2\ 5\ 3\ 0\ 2\ 3\ 0\ 3\), \(k = 5\), \(C: 2\ 0\ 2\ 3\ 0\ 1\)

analysis:
Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT \((A, B, k)\)

\{\(A\) contains \(n\) integers; \(k\) is the max\}

1. \(\text{for } i = 0 \text{ to } k\)
2. \(C[i] = 0\)
3. \(\text{for } j = 1 \text{ to } \text{length}[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i] \text{ contains the number of elements whose values } = i\}\)
6. \(\text{for } i = 1 \text{ to } k\)
7. \(C[i] = C[i] + C[i - 1]\)
8. \(\{C[i] \text{ contains the number of elements whose values } \leq i\}\)
9. \(\text{for } j = \text{length}[A] \text{ downto } 1\)
10. \(B[C[A[j]]] = A[j]\)
11. \(C[A[j]] = C[A[j]] - 1\)

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3\), \(k = 5\), \(C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)

analysis: \(T(n) = O(k + n)\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

Algorithm Radix-Sort \((A, d)\)

1. for \(i = 1\) to \(d\)
2. sort \(A\) on the \(i\)th digit

Lemma. Given \(n\) \(b\)-bit binary numbers and any positive \(r \leq b\). Radix-Sort uses \(\Theta(\lceil b/r \rceil (n + 2r))\) time.
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>720</td>
<td>720</td>
<td>329</td>
</tr>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
<td>839</td>
</tr>
</tbody>
</table>
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329  720  720  329
457  355  329  355
657  436  436  436
839  457  839  457
436  657  355  657
720  329  457  720
355  839  657  839

Algorithm \textsc{Radix-Sort}(A, d)
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329  720  720  329
457  355  329  355
657  436  436  436
839  457  839  457
436  657  355  657
720  329  457  720
355  839  657  839

Algorithm Radix-Sort\((A, d)\)

1. \textbf{for} \(i = 1\) \textbf{to} \(d\)
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329  720  720  329
457  355  329  355
657  436  436  436
839  457  839  457
436  657  355  657
720  329  457  720
355  839  657  839

Algorithm \textsc{Radix-Sort}(A, d)
1. \textbf{for} \( i = 1 \) \textbf{to} \( d \)
2. \textbf{sort} \( A \) on the \( i \)th digit
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329  720  720  329  
457  355  329  355  
657  436  436  436  
839  457  839  457  
436  657  355  657  
720  329  457  720  
355  839  657  839  

Algorithm \texttt{Radix-Sort}(A, d)

1. \textbf{for} $i = 1$ \textbf{to} $d$
2. \textbf{sort} $A$ on the $i$th digit

\textbf{Lemma}. Given $n$ $b$-bit binary numbers and any positive $r \leq b$.
\texttt{Radix-Sort} uses $\Theta([b/r](n + 2^r))$ time.
Lemma. Given $n$ $b$-bit binary numbers and any positive $r \leq b$. \textsc{Radix-Sort} uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.
Lemma. Given \( n \) \( b \)-bit binary numbers and any positive \( r \leq b \). Radix-Sort uses \( \Theta(\lceil b/r \rceil (n + 2^r)) \) time.

Proof. Each \( b \)-digit binary number can be regarded as \( \lceil b/r \rceil \) \( r \)-digit binary numbers. These \( r \)-digit binary numbers are of integer values in the range of \( \{0, 1, \ldots, 2^r - 1\} \).
Chapter 8. Lower Bounds and Sorting in Linear Time

**Lemma.** Given $n$ $b$-bit binary numbers and any positive $r \leq b$. Radix-Sort uses $\Theta(\lceil b/r \rceil (n + 2r))$ time.

**Proof.** Each $b$-digit binary number can be regarded as $\lceil b/r \rceil$ $r$-digit binary numbers. These $r$-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run Radix-Sort on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.
Lemma. Given $n$ $b$-bit binary numbers and any positive $r \leq b$. Radix-Sort uses $\Theta([b/r](n + 2^r))$ time.

Proof. Each $b$-digit binary number can be regarded as $[b/r]$ $r$-digit binary numbers. These $r$-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run Radix-Sort on the original binary numbers assumed to be $[b/r]$ columns.

For every column, sorting by Counting-Sort with $2^r - 1$ being the maximum.
Lemma. Given $n$ $b$-bit binary numbers and any positive $r \leq b$. \textsc{Radix-Sort} uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each $b$-digit binary number can be regarded as $\lceil b/r \rceil$ $r$-digit binary numbers. These $r$-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run \textsc{Radix-Sort} on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by \textsc{Counting-Sort} with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for \textsc{Counting-Sort}.
Lemma. Given $n$ $b$-bit binary numbers and any positive $r \leq b$. Radix-Sort uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each $b$-digit binary number can be regarded as $\lceil b/r \rceil$ $r$-digit binary numbers. These $r$-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run Radix-Sort on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by Counting-Sort with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for Counting-Sort.

Since all steps in the two algorithms are mandatory, the total time is also $\Omega(\lceil b/r \rceil (n + 2^r))$, thus $\Theta(\lceil b/r \rceil (n + 2^r))$. 
Lemma. Given $n$ $b$-bit binary numbers and any positive $r \leq b$. Radix-Sort uses $\Theta\left(\lceil b/r \rceil (n + 2^r)\right)$ time.

Proof. Each $b$-digit binary number can be regarded as $\lceil b/r \rceil$ $r$-digit binary numbers. These $r$-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run Radix-Sort on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by Counting-Sort with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for Counting-Sort.

Since all steps in the two algorithms are mandatory, the total time is also $\Omega(\lceil b/r \rceil (n + 2^r))$, thus $\Theta(\lceil b/r \rceil (n + 2^r))$.

Once $b$ and $n$ are given, we can choose $r$ to minimize the quantity $\lceil b/r \rceil (n + 2^r)$. 
Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)
Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm $\text{BUCKET-SORT}(A)$
1. $n = \text{length}[A]$
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**\((A)\)

1. \(n = \text{length}[A]\)
2. for \(i = 1\) to \(n\)
3. insert \(A[i]\) into list \(B[\lfloor nA[i] \rfloor]\)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**$(A)$
1. $n = length[A]$
2. for $i = 1$ to $n$
3. insert $A[i]$ into list $B[[nA[i]]]$  
4. for $i = 0$ to $n - 1$
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm `Bucket-Sort(A)`
1. \( n = length[A] \)
2. for \( i = 1 \) to \( n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. for \( i = 0 \) to \( n - 1 \)
5. sort list \( B[i] \) with **Insertion Sort**
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort$(A)$
1. $n = length[A]$
2. for $i = 1$ to $n$
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. for $i = 0$ to $n - 1$
5. sort list $B[i]$ with Insertion Sort
6. concatenate the lists $B[0], B[1], \ldots, B[n - 1]$
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**\( (A) \)

1. \( n = length[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
3. \( \text{insert } A[i] \text{ into list } B[[nA[i]]] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. \( \text{sort list } B[i] \text{ with Insertion Sort} \)
6. \( \text{concatenate the lists } B[0], B[1], ..., B[n - 1] \)

A: \( .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68 \)
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \text{for } i = 1 \\text{ to } n \)
3. \( \text{insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. \( \text{sort list } B[i] \text{ with Insertion Sort} \)
6. \( \text{concatenate the lists } B[0], B[1], ..., B[n - 1] \)

A: \(.78 .17 .39 .26 .72 .94 .21 .12 .23 .68\)

B: 0 /
\(1 \rightarrow .12 \rightarrow .17\)
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \textbf{for} \; i = 1 \; \textbf{to} \; n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. \( \textbf{for} \; i = 0 \; \textbf{to} \; n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: \( .78 \; .17 \; .39 \; .26 \; .72 \; .94 \; .21 \; .12 \; .23 \; .68 \)

B: \( 0 / \)
\( 1 \to .12 \to .17 \)
\( 2 \to .21 \to .23 \to .26 \)
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. for \( i = 1 \) to \( n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. for \( i = 0 \) to \( n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: \( .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68 \)

B: \( 0 / \)
   1 → \( .12 \) → \( .17 \)
   2 → \( .21 \) → \( .23 \) → \( .26 \)
   3 → \( .39 \)
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 \( \rightarrow \) .12 \( \rightarrow \) .17
2 \( \rightarrow \) .21 \( \rightarrow \) .23 \( \rightarrow \) .26
3 \( \rightarrow \) .39
4 /

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \]
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**$(A)$

1. $n = length[A]$
2. for $i = 1$ to $n$
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. for $i = 0$ to $n - 1$
5. sort list $B[i]$ with **Insertion Sort**
6. concatenate the lists $B[0], B[1], \ldots, B[n - 1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
   1 → .12 → .17
   2 → .21 → .23 → .26
   3 → .39
   4 /
   5 /
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort\( (A) \)
1. \( n = \text{length}[A] \)
2. \( \textbf{for}\ i = 1 \ \textbf{to} \ n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. \( \textbf{for}\ i = 0 \ \textbf{to} \ n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: 0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68

B: 0 /
1 \( \rightarrow \) 0.12 \( \rightarrow \) 0.17
2 \( \rightarrow \) 0.21 \( \rightarrow \) 0.23 \( \rightarrow \) 0.26
3 \( \rightarrow \) 0.39
4 /
5 /
6 \( \rightarrow \) 0.68
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = \text{length}[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
3. insert \( A[i] \) into list \( B[\lfloor nA[i] \rfloor] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. sort list \( B[i] \) with Insertion Sort
6. concatenate the lists \( B[0], B[1], ..., B[n - 1] \)

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
  1 → .12 → .17
  2 → .21 → .23 → .26
  3 → .39
  4 /
  5 /
  6 → .68
  7 → .72 → .78
Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \[ n = \text{length}[A] \]
2. \[ \textbf{for } i = 1 \textbf{ to } n \]
3. \[ \text{insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor] \]
4. \[ \textbf{for } i = 0 \textbf{ to } n - 1 \]
5. \[ \text{sort list } B[i] \text{ with Insertion Sort} \]
6. \[ \text{concatenate the lists } B[0], B[1], ..., B[n - 1] \]

A: \[ .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68 \]

B: \[ 0 / \]
   \[ 1 \rightarrow .12 \rightarrow .17 \]
   \[ 2 \rightarrow .21 \rightarrow .23 \rightarrow .26 \]
   \[ 3 \rightarrow .39 \]
   \[ 4 / \]
   \[ 5 / \]
   \[ 6 \rightarrow .68 \]
   \[ 7 \rightarrow .72 \rightarrow .78 \]
   \[ 8 / \]
Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm Bucket-Sort(A)
1. \( n = length[A] \)
2. \( \text{for } i = 1 \text{ to } n \)
3. \( \text{insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. \( \text{sort list } B[i] \text{ with Insertion Sort} \)
6. \( \text{concatenate the lists } B[0], B[1], \ldots, B[n − 1] \)

A: \( .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68 \)

B: \( 0 / \)
\( 1 \rightarrow .12 \rightarrow .17 \)
\( 2 \rightarrow .21 \rightarrow .23 \rightarrow .26 \)
\( 3 \rightarrow .39 \)
\( 4 / \)
\( 5 / \)
\( 6 \rightarrow .68 \)
\( 7 \rightarrow .72 \rightarrow .78 \)
\( 8 / \)
\( 9 \rightarrow .94 \)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**($A$)

1. $n = \text{length}[A]$
2. for $i = 1$ to $n$
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor ]$
4. for $i = 0$ to $n - 1$
5. sort list $B[i]$ with **Insertion Sort**
6. concatenate the lists $B[0], B[1], ..., B[n - 1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
    1 → .12 → .17
    2 → .21 → .23 → .26
    3 → .39
    4 /
    5 /
    6 → .68
    7 → .72 → .78
    8 /
    9 → .94
Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics
Chapter 9. Medians and Order Statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element)? The problem has upper bound $O(n \log_2 n)$. Why?
- Can we do better?
Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

• find the maximum: linear time
• find the minimum: linear time
• find the median (i.e., the $\frac{n}{2}$th smallest element)?
Chapter 9. Medians and Order Statistics

**Chapter 9. Medians and order statistics**

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element)?
  
  the problem has upper bound $O(n \log_2 n)$. 
Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element)?
  
The problem has upper bound $O(n \log_2 n)$. why?
Chapter 9. Medians and order statistics

• find the maximum: linear time
• find the minimum: linear time
• find the median (i.e., the \( \frac{n}{2} \)th smallest element) ?
  the problem has upper bound \( O(n \log_2 n) \). why?
  Can we do better?
Chapter 9. Medians and Order Statistics

Selection problem

Input: a list \( A \) of elements, an integer \( i \);

Output: the \( i \)th smallest element in \( A \);

There are algorithms solving it in linear time.

Two types of algorithms:

• Selection in worst case linear time
• Selection in expected linear time (but worst case \( \Theta(n^2) \))
Selection problem
Selection problem

**Input:** a list $A$ of elements, an integer $i$;
Selection problem

**Input:** a list $A$ of elements, an integer $i$;
**Output:** the $i$th smallest element in $A$;
Chapter 9. Medians and Order Statistics

Selection problem

**INPUT:** a list $A$ of elements, an integer $i$;

**OUTPUT:** the $i$th smallest element in $A$;

There are algorithms solving it in linear time.
Selection problem

**Input:** a list $A$ of elements, an integer $i$;  
**Output:** the $i$th smallest element in $A$;

There are algorithms solving it in linear time.

Two types of algorithms:
Selection problem

**Input:** a list $A$ of elements, an integer $i$;

**Output:** the $i$th smallest element in $A$;

There are algorithms solving it in linear time.

Two types of algorithms:

- Selection in worst case linear time
Selection problem

**Input:** a list $A$ of elements, an integer $i$;

**Output:** the $i$th smallest element in $A$;

There are algorithms solving it in linear time.

Two types of algorithms:

- Selection in worst case linear time
- Selection in *expected* linear time (but worst case $\Theta(n^2)$)
Chapter 9. Medians and Order Statistics

Selection in worst case linear time

Input: set $S$ of $n$ elements and $i$;

Output: the $i$th smallest element in $S$;

Main idea:
• find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$, such that $\forall y \in S_1 y < x$ and $\forall z \in S_2 z > x$;
• both $S_1$ and $S_2$ are guaranteed only a fraction of $S$;
• the $i$th smallest element is either $x$, or in $S_1$ or in $S_2$ (but not both);
• in either of the latter two cases, the algorithm is applied recursively.
Chapter 9. Medians and Order Statistics

Selection in worst case linear time
Chapter 9. Medians and Order Statistics

Selection in worst case linear time

**INPUT:** set $S$ of $n$ elements and $i$;

**OUTPUT:** the $i$th smallest element in $S$;
Selection in worst case linear time

**INPUT:** set $S$ of $n$ elements and $i$;

**OUTPUT:** the $i$th smallest element in $S$;

Main idea:
Selection in worst case linear time

**INPUT:** set $S$ of $n$ elements and $i$;

**OUTPUT:** the $i$th smallest element in $S$;

Main idea:

- find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$,
Selection in worst case linear time

**Input:** set $S$ of $n$ elements and $i$;

**Output:** the $i$th smallest element in $S$;

**Main idea:**

- Find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
Selection in worst case linear time

**Input:** set $S$ of $n$ elements and $i$;

**Output:** the $i$th smallest element in $S$;

Main idea:

- find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
- both $S_1$ and $S_2$ are guaranteed only a fraction of $S$;
Selection in worst case linear time

**Input:** set $S$ of $n$ elements and $i$;
**Output:** the $i$th smallest element in $S$;

Main idea:

- find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
- both $S_1$ and $S_2$ are guaranteed only a fraction of $S$;
- the $i$th smallest element is either $x$, or in $S_1$ or in $S_2$ (but not both);
Chapter 9. Medians and Order Statistics

Selection in worst case linear time

**Input:** set $S$ of $n$ elements and $i$;

**Output:** the $i$th smallest element in $S$;

Main idea:

- find a pivot $x$ to partition the list $S$ into two sublists $S_1$ and $S_2$, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
- both $S_1$ and $S_2$ are guaranteed only a fraction of $S$;
- the $i$th smallest element is either $x$, or in $S_1$ or in $S_2$ (but not both);
- in either of the latter two cases, the algorithm is applied recursively.
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$ if time for finding pivot: $cn$ and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$.

Then $T(n) \leq T(\beta n) + cn \leq cn + c\beta n + T(\beta^2 n) + \cdots + c\beta^{m-1} n + T(\beta^m n)$ (where $\beta^m n = 1$).

$\leq cn (1 - \beta^m) + c' \leq c\frac{1}{1 - \beta n} + c' = O(n)$.
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\[ \text{if time for finding pivot: } cn \]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

if time for finding pivot: $cn$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\textbf{if} time for finding pivot: $cn$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$
Assume total time complexity $T(n)$

- if time for finding pivot: $cn$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn +$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

*if* time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$
Assume total time complexity $T(n)$

if time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

\[ T(n) \leq cn + c\beta n + T(\beta^2 n) \]

$T(n) \leq cn +$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

if time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$T(n) \leq cn + c\beta n +$
Assume total time complexity $T(n)$

if time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$T(n) \leq cn + c\beta n + c\beta^2 n +$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

**if** time for finding pivot: $cn$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$$T(n) \leq cn + c\beta n + c\beta^2 n + \cdots + c\beta^{m-1} n + T(\beta^m n)$$
Assume total time complexity $T(n)$

if time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$$T(n) \leq cn + c\beta n + c\beta^2 n + \cdots + c\beta^{m-1} n + T(\beta^m n) \quad \text{(where } \beta^m n = 1)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

- if time for finding pivot: $cn$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$$T(n) \leq cn + c\beta n + c\beta^2 n + \cdots + c\beta^{m-1} n + T(\beta^m n) \quad \text{(where } \beta^m n = 1)$$

$$\leq cn \left( \frac{1 - \beta^m}{1 - \beta} \right) + c'$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$ if time for finding pivot: $cn$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

\begin{align*}
T(n) & \leq cn + c\beta n + T(\beta^2 n) \\
T(n) & \leq cn + c\beta n + c\beta^2 n + \cdots + c\beta^{m-1} n + T(\beta^m n) \text{ (where } \beta^m n = 1) \\
& \leq cn\left(1 - \frac{\beta^m}{1 - \beta}\right) + c' \leq c \frac{1}{1 - \beta} n + c'
\end{align*}
Assume total time complexity $T(n)$

- if time for finding pivot: $cn$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

Then $T(n) \leq T(\beta n) + cn$

$$T(n) \leq cn + c\beta n + T(\beta^2 n)$$

$$T(n) \leq cn + c\beta n + c\beta^2 n + \cdots + c\beta^{m-1} n + T(\beta^m n) \quad \text{(where } \beta^m n = 1)$$

$$\leq cn\left(\frac{1 - \beta^m}{1 - \beta}\right) + c' \leq c \frac{1}{1 - \beta} n + c' = O(n)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$ if time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$ and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$ assume $\alpha + \beta < 1$, then $T(n) \leq cn + T(\alpha n) + T(\beta n)$.

With the recursive tree method (you draw a picture): $T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta^2 n)$.

\[= cn + c(\alpha + \beta) n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)\]

\[\leq cn + c(\alpha + \beta) n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^m n + c'\]

where $m = \max\{i,j\}$, for such $i,j$ that $\alpha^i n = 1$ and $\beta^j n = 1$.

And $c' = 2T(1)$, the base case.

Therefore, $T(n) \leq c(1 - (\alpha + \beta)) n \leq c(1 - (\alpha + \beta)) n = O(n)$. 

Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\[
\text{if time for finding pivot: } cn + T(\alpha n), \text{ for some } 0 < \alpha < 1
\]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

if time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$
assume $\alpha + \beta < 1$, 

where $m = \max\{i, j\}$, for such $i, j$ that $\alpha^i n = 1$ and $\beta^j n = 1$. 
and $c' = 2T(1)$, the base case.

Therefore, $T(n) \leq c_1 - (\alpha + \beta)^m n \leq c_1 - (\alpha + \beta) n = O(n)$. 
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

- **if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$
- **assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

**if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):
Chapter 9. Medians and Order Statistics

Assume total time complexity \( T(n) \)

\[
\text{if } \text{time for finding pivot: } cn + T(\alpha n), \text{ for some } 0 < \alpha < 1 \\
\text{and time for the recursive step: } T(\beta n), \text{ for some } 0 < \beta < 1
\]

\text{assume } \alpha + \beta < 1, \text{ then}

\[
T(n) \leq cn + T(\alpha n) + T(\beta n)
\]

With the recursive tree method (you draw a picture):

\[
T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n)
\]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

- **if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha\beta n) + c\beta n + T(\beta\alpha n) + T(\beta^2 n)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity \( T(n) \)

\[
\text{if time for finding pivot: } cn + T(\alpha n), \text{ for some } 0 < \alpha < 1
\]

and time for the recursive step: \( T(\beta n) \), for some \( 0 < \beta < 1 \)

\textbf{assume} \( \alpha + \beta < 1 \), then

\[
T(n) \leq cn + T(\alpha n) + T(\beta n)
\]

With the recursive tree method (you draw a picture):

\[
T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)
\]

\[
= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)
\]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

**if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)$$

$$= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)$$

$$\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

- **if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)$$

$$= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)$$

$$\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

**if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)$$

$$= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)$$

$$\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$

$$= cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\[
\text{if time for finding pivot: } cn + T(\alpha n), \text{ for some } 0 < \alpha < 1
\]
and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

\[
T(n) \leq cn + T(\alpha n) + T(\beta n)
\]

With the recursive tree method (you draw a picture):

\[
T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)
\]

\[
= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)
\]

\[
\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)
\]

\[
= cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)
\]

\[
\leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n +
\]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\[ T(n) \leq cn + T(\alpha n) + T(\beta n) \]

if time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

\[ \text{assume } \alpha + \beta < 1, \text{ then } \]

With the recursive tree method (you draw a picture):

\[ T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n) \]

\[ = cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n) \]

\[ \leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n) \]

\[ = cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n) \]

\[ \leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^{m-1} n + c' \]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

**if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)$$

$$= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)$$

$$\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$

$$= cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$

$$\leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^{m-1}n + c'$$

where $m = \max\{i, j\}$, for such $i, j$ that $\alpha^i n = 1$ and $\beta^j n = 1$.

and $c' = 2T(1)$, the base case.
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

- **if** time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$
- and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

**assume** $\alpha + \beta < 1$, then

$$T(n) \leq cn + T(\alpha n) + T(\beta n)$$

With the recursive tree method (you draw a picture):

$$T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)$$

$$= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)$$

$$\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$

$$= cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)$$

$$\leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^{m-1} n + c'$$

where $m = \max\{i, j\}$, for such $i, j$ that $\alpha^i n = 1$ and $\beta^j n = 1$.

and $c' = 2T(1)$, the base case.

Therefore,

$$T(n) \leq c \frac{1 - (\alpha + \beta)^m}{1 - (\alpha + \beta)} n$$
Chapter 9. Medians and Order Statistics

Assume total time complexity \( T(n) \)

\[ \text{if time for finding pivot: } cn + T(\alpha n), \text{ for some } 0 < \alpha < 1 \]

and time for the recursive step: \( T(\beta n) \), for some \( 0 < \beta < 1 \)

\text{assume } \alpha + \beta < 1, \text{ then}

\[ T(n) \leq cn + T(\alpha n) + T(\beta n) \]

With the recursive tree method (you draw a picture):

\[ T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha\beta n) + c\beta n + T(\beta\alpha n) + T(\beta^2 n) \]

\[ = cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha\beta n) + T(\beta^2 n) \]

\[ \leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha\beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n) \]

\[ = cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n) \]

\[ \leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^{m-1} n + c' \]

where \( m = \max\{i, j\} \), for such \( i, j \) that \( \alpha^i n = 1 \) and \( \beta^j n = 1 \).

and \( c' = 2T(1) \), the base case.

Therefore,

\[ T(n) \leq c \frac{1-(\alpha+\beta)^m}{1-(\alpha+\beta)} n \leq c \frac{1}{1-(\alpha+\beta)} n \]
Chapter 9. Medians and Order Statistics

Assume total time complexity $T(n)$

\[ T(n) \leq cn + T(\alpha n) + T(\beta n) \]

if time for finding pivot: $cn + T(\alpha n)$, for some $0 < \alpha < 1$

and time for the recursive step: $T(\beta n)$, for some $0 < \beta < 1$

assume $\alpha + \beta < 1$, then

With the recursive tree method (you draw a picture):

\[
T(n) \leq cn + c\alpha n + T(\alpha^2 n) + T(\alpha \beta n) + c\beta n + T(\beta \alpha n) + T(\beta^2 n)
\]

\[
= cn + c\alpha n + c\beta n + T(\alpha^2 n) + 2T(\alpha \beta n) + T(\beta^2 n)
\]

\[
\leq cn + c(\alpha + \beta)n + c\alpha^2 n + 2c\alpha \beta n + c\beta^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)
\]

\[
= cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + T(\alpha^3 n) + 3T(\alpha^2 \beta n) + 3T(\alpha \beta^2 n) + T(\beta^3 n)
\]

\[
\leq cn + c(\alpha + \beta)n + c(\alpha + \beta)^2 n + c(\alpha + \beta)^3 n + \cdots + c(\alpha + \beta)^{m-1} n + c'
\]

where $m = \max\{i, j\}$, for such $i, j$ that $\alpha^i n = 1$ and $\beta^j n = 1$.

and $c' = 2T(1)$, the base case.

Therefore, \[ T(n) \leq c \frac{1 - (\alpha + \beta)^m}{1 - (\alpha + \beta)} n \leq c \frac{1}{1 - (\alpha + \beta)} n = O(n). \]
Chapter 9. Medians and Order Statistics

How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction $\alpha n$ of the original list $S$, $|S| = n$
- the total time actually is $T(n) \leq T(\alpha n) + T(\beta n) + cn$ where $\alpha + \beta < 1$
How to find such a pivot?
Chapter 9. Medians and Order Statistics

How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction $\alpha n$
Chapter 9. Medians and Order Statistics

How to find such a pivot?

- **the very selection algorithm is recursively called for finding the pivot**
- the size of the sublist to find the pivot is also a fraction $\alpha n$
  of the original list $S$, $|S| = n$;
How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction $\alpha n$ of the original list $S$, $|S| = n$;
- the total time actually is

$$T(n) \leq T(\alpha n) + T(\beta n) + cn$$

where $\alpha + \beta < 1$
Algorithm \textsc{Select} ($S, i$); \{ where $S$ contains $n$ distinct elements\}
Chapter 9. Medians and Order Statistics

Algorithm \texttt{Select} \((S, i)\); \{ where \(S\) contains \(n\) distinct elements\}

(1) divide \(S\) into \([n/5]\) groups of 5 elements
Algorithm `SELECT (S, i); { where S contains n distinct elements}`

(1) divide S into \( \lceil n/5 \rceil \) groups of 5 elements
(2) sort each group (of 5) and find the median of each group;}
Algorithm `SELECT (S, i);` {where $S$ contains $n$ distinct elements}
(1) divide $S$ into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let $M$ contain all these medians; where $|M| = \lceil n/5 \rceil$
Algorithm $\text{Select} \ (S, i); \ \{ \text{where } S \text{ contains } n \text{ distinct elements} \}$

1. divide $S$ into $\lceil n/5 \rceil$ groups of 5 elements
2. sort each group (of 5) and find the median of each group;
   let $M$ contain all these medians; where $|M| = \lceil n/5 \rceil$
3. recursively call $\text{Select}(M, \lceil n/10 \rceil)$;
Chapter 9. Medians and Order Statistics

Algorithm \textsc{Select} \((S, i)\); \{ where \(S\) contains \(n\) distinct elements\}
(1) divide \(S\) into \(\lceil n/5 \rceil\) groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let \(M\) contain all these medians; where \(|M| = \lceil n/5 \rceil\)
(3) \textbf{recursively call} \textsc{Select}(\(M, \lceil n/10 \rceil\));
    let the result be \(x\) and let the rank of \(x\) be \(k\) in \(S\)
(4) if \(i = k\) return \((x)\)
(5) else use \(x\) as the pivot to partition \(S\) resulting in \(S_1\) and \(S_2\),
    such that \(\forall y \in S_1\) \(y < x\) and \(\forall z \in S_2\) \(z > x\)
Algorithm $\text{Select} \ (S, i)$; \{ where $S$ contains $n$ distinct elements \}

(1) divide $S$ into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let $M$ contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) recursively call $\text{Select}(M, \lceil n/10 \rceil)$;
    let the result be $x$ and let the rank of $x$ be $k$ in $S$
(4) if $i = k$ return ($x$)
Chapter 9. Medians and Order Statistics

Algorithm $\text{SELECT} (S, i); \{ \text{ where } S \text{ contains } n \text{ distinct elements} \}$

1. divide $S$ into $\lceil n/5 \rceil$ groups of 5 elements
2. sort each group (of 5) and find the median of each group;
   let $M$ contain all these medians; where $|M| = \lceil n/5 \rceil$
3. **recursively call** $\text{SELECT}(M, \lceil n/10 \rceil);$  
   let the result be $x$ and let the rank of $x$ be $k$ in $S$
4. if $i = k$ return $(x)$
5. else use $x$ as the pivot to partition $S$ resulting in $S_1$ and $S_2,$
Algorithm \texttt{Select} \((S, i)\); \{ where \(S\) contains \(n\) distinct elements\}

1. divide \(S\) into \(\lceil n/5 \rceil\) groups of 5 elements
2. sort each group (of 5) and find the median of each group;
   let \(M\) contain all these medians; where \(|M| = \lceil n/5 \rceil\)
3. recursively call \texttt{Select}(\(M, \lceil n/10 \rceil\));
   let the result be \(x\) and let the rank of \(x\) be \(k\) in \(S\)
4. if \(i = k\) return \((x)\)
5. else use \(x\) as the pivot to partition \(S\) resulting in \(S_1\) and \(S_2\),
   such that \(\forall y \in S_1 \ y < x\) and \(\forall z \in S_2 \ z > x\)
Algorithm `Select (S, i); { where S contains n distinct elements}`
(1) divide S into ⌈n/5⌉ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
   let M contain all these medians; where |M| = ⌈n/5⌉
(3) recursively call `Select(M, ⌈n/10⌉);`
   let the result be x and let the rank of x be k in S
(4) if \( i = k \) return \( (x) \)
(5) else use x as the pivot to partition S resulting in \( S_1 \) and \( S_2 \),
   such that \( \forall y \in S_1 \ y < x \) and \( \forall z \in S_2 \ z > x \)
(6) if \( i < k \) recursively call `Select(S_1, i)`
Algorithm \texttt{Select} \((S, i)\); \{ where \(S\) contains \(n\) distinct elements\}

1. divide \(S\) into \([n/5]\) groups of 5 elements
2. sort each group (of 5) and find the median of each group;
   let \(M\) contain all these medians; where \(|M| = [n/5]\)
3. recursively call \texttt{Select} \((M, [n/10])\);
   let the result be \(x\) and let the rank of \(x\) be \(k\) in \(S\)
4. if \(i = k\) return \((x)\)
5. else use \(x\) as the pivot to partition \(S\) resulting in \(S_1\) and \(S_2\),
   such that \(\forall y \in S_1 \ y < x\) and \(\forall z \in S_2 \ z > x\)
6. if \(i < k\) recursively call \texttt{Select} \((S_1, i)\)
   else recursively call \texttt{Select} \((S_2, i - k)\)
Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left\lceil \frac{n}{5} \right\rceil^2 \geq \frac{3n}{10} \Rightarrow |S_2| < n - 3\frac{n}{10} = 7\frac{n}{10}$$

Similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left(\left\lceil \frac{n}{5} \right\rceil^2 - 2\right) \geq \frac{3n}{10} - 6 \geq \frac{3n}{10} \Rightarrow |S_1| < n - 3\frac{n}{10} + 6 = 7\frac{n}{10} + 6$$

So a time upper bound for $\text{Select}$ is $T(n) \leq T_{\text{mom}} + T_{\text{sub}} + cn$ when $n \geq 140$ (why?)
Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left(\frac{n/5}{2}\right) \geq 3n/10$$
Note: the number of elements \( \leq x \) is at least:

\[
|S_1| \geq 3\left(\frac{n/5}{2}\right) \geq 3n/10 \quad \Rightarrow
\]
Chapter 9. Medians and Order Statistics

Note: the number of elements \( \leq x \) is at least:

\[
|S_1| \geq 3 \left( \left\lfloor \frac{n}{5} \right\rfloor \right) \geq 3n/10 \implies |S_2| < n - 3n/10 = 7n/10
\]
Chapter 9. Medians and Order Statistics

Note: the number of elements \( \leq x \) is at least:

\[ |S_1| \geq 3\left(\frac{\lceil n/5 \rceil}{2}\right) \geq 3n/10 \implies |S_2| < n - 3n/10 = 7n/10 \]

similarly, the number of elements \( \geq x \) is at least:

\[ |S_2| \geq 3\left(\frac{\lceil n/5 \rceil}{2} - 2\right) \geq 3n/10 - 6 \geq 3n/10 \]
Chapter 9. Medians and Order Statistics

Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left\lceil \frac{n}{5} \right\rceil \geq 3n/10 \implies |S_2| < n - 3n/10 = 7n/10$$

similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left\lceil \frac{n}{5} \right\rceil - 2 \geq 3n/10 - 6 \geq 3n/10 \implies$$
Chapter 9. Medians and Order Statistics

Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left(\frac{\lceil n/5 \rceil}{2}\right) \geq 3n/10 \implies |S_2| < n - 3n/10 = 7n/10$$

similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left(\frac{\lceil n/5 \rceil}{2} - 2\right) \geq 3n/10 - 6 \geq 3n/10 \implies |S_1| < n - 3n/10 + 6 = 7n/10 + 6$$
Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left(\frac{\left\lceil \frac{n}{5} \right\rceil}{2}\right) \geq 3n/10 \quad \implies \quad |S_2| < n - 3n/10 = 7n/10$$

Similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left(\frac{\left\lceil \frac{n}{5} \right\rceil}{2} - 2\right) \geq 3n/10 - 6 \geq 3n/10 \quad \implies \quad |S_1| < n - 3n/10 + 6 = 7n/10 + 6$$

So a time upper bound for $\text{SELECT}$ is
Chapter 9. Medians and Order Statistics

Note: the number of elements \( \leq x \) is at least:

\[
|S_1| \geq 3\left(\frac{\lceil n/5 \rceil}{2}\right) \geq \frac{3n}{10} \quad \Longrightarrow \quad |S_2| < n - \frac{3n}{10} = \frac{7n}{10}
\]

similarly, the number of elements \( \geq x \) is at least:

\[
|S_2| \geq 3\left(\frac{\lceil n/5 \rceil}{2} - 2\right) \geq \frac{3n}{10} - 6 \geq \frac{3n}{10} \quad \Longrightarrow \quad |S_1| < n - \frac{3n}{10} + 6 = \frac{7n}{10} + 6
\]

So a time upper bound for \( \text{SELECT} \) is \( T(n) \leq T_{\text{mom}} + T_{\text{sub}} + cn \)
Note: the number of elements $\leq x$ is at least:

$$|S_1| \geq 3\left(\left\lfloor \frac{n/5}{2} \right\rfloor \right) \geq 3n/10 \implies |S_2| < n - 3n/10 = 7n/10$$

similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left(\left\lfloor \frac{n/5}{2} \right\rfloor - 2 \right) \geq 3n/10 - 6 \geq 3n/10 \implies |S_1| < n - 3n/10 + 6 = 7n/10 + 6$$

So a time upper bound for $\text{SELECT}$ is $T(n) \leq T_{mom} + T_{sub} + cn$

$$T(n) \leq T(\left\lfloor n/5 \right\rfloor) + T(\left\lceil 7n/10 + 6 \right\rceil) + cn$$

when $n \geq 140$ (why?)
Chapter 9. Medians and Order Statistics

Selection in *expected* linear time
Selection in *expected* linear time

**Input:** a list $A$ of elements, an integer $i$;
Selection in expected linear time

**Input:** a list \(A\) of elements, an integer \(i\);
**Output:** the \(ith\) smallest element in \(A\);
Chapter 9. Medians and Order Statistics

Selection in \textit{expected} linear time

\textbf{Input:} a list $A$ of elements, an integer $i$;

\textbf{Output:} the $i$\textit{th} smallest element in $A$;

Idea of the algorithm:

- randomly identify a pivot $x$ and partition the list $A$ into two sublists $A_l$ and $A_u$; assume the \textbf{rank} of $x$ is $k$;
Selection in *expected* linear time

**Input:** a list $A$ of elements, an integer $i$;  
**Output:** the $i$th smallest element in $A$;

Idea of the algorithm:

- randomly identify a pivot $x$ and partition the list $A$ into two sublists $A_l$ and $A_u$; assume the **rank** of $x$ is $k$;
- if $i = k$, done, return $(x)$;
Selection in *expected* linear time

**Input:** a list $A$ of elements, an integer $i$;

**Output:** the $i$th smallest element in $A$;

Idea of the algorithm:

- randomly identify a pivot $x$ and partition the list $A$ into two sublists $A_l$ and $A_u$; assume the rank of $x$ is $k$;
- if $i = k$, done, return $(x)$;
- if $k > i$, recursively do for $A_l$ with $i$;
Selection in expected linear time

**Input:** a list $A$ of elements, an integer $i$;  
**Output:** the $i$th smallest element in $A$;

Idea of the algorithm:

- randomly identify a pivot $x$ and partition the list $A$ into two sublists $A_l$ and $A_u$; assume the rank of $x$ is $k$;
- if $i = k$, done, return $(x)$;
  - else if $k > i$, recursively do for $A_l$ with $i$;
    - else recursively do for $A_u$ with $i - k$;
Chapter 9. Medians and Order Statistics

Algorithm `RANDOMIZED-SELECT (A, p, r, i)`
Algorithm \textsc{Randomized-Select} \ ($A, p, r, i$)

1. \textbf{if} $p = r$

If pivots always partition lists into $n$ \textit{r} \ $r_n$, for some $r > 1$, time $T(n)$ would have the recurrence

$$T(n) \leq \max\{T(n/r), T((r-1)n/r)\} + cn$$

assuming $r \geq 2$, $T(n) \leq cn(r-1)^r + cn(r-1)^{r-1} + \ldots cn(r-1)^{m-1} = O(n)$ where $(r-1)^m = n$. 

Chapter 9. Medians and Order Statistics

Algorithm $\text{RANDOMIZED-SELECT} (A, p, r, i)$
1. if $p = r$
2. return $(A[p])$
Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT \((A, p, r, i)\)

1. if \(p = r\)
2. return \((A[p])\)
3. \(q = \text{RANDOMIZED PARTITION} (A, p, r)\)

If pivots always partition lists into \(n\) \(r\):

\(r - 1\) \(r\) \(n\), for some \(r > 1\),

\(T(n)\) would have the recurrence

\[ T(n) \leq \max\{T(n^r), T((r - 1) n^r)\} + cn \leq T((r - 1) n^r) + cn \] assuming \(r \geq 2\).

\(T(n) \leq cn((r - 1) r^m + cn((r - 1) r^m)^2 + \ldots cn((r - 1) r^m)^m = O(n)\)
Algorithm RANDOMIZED-SELECT \((A, p, r, i)\)

1. \(\textbf{if } p = r\)
2. \(\textbf{return } (A[p])\)
3. \(q = \textsc{Randomized Partition} \((A, p, r)\)\)
4. \(k = q - p + 1\)
Algorithm RANDOMIZED-SELECT \((A, p, r, i)\)

1. \textbf{if} \(p = r\)
2. \hspace{2em} \textbf{return} \((A[p])\)
3. \hspace{2em} \(q =\) RANDOMIZED PARTITION \((A, p, r)\)
4. \hspace{2em} \(k = q - p + 1\)
5. \hspace{2em} \textbf{if} \(i = k\)
Chapter 9. Medians and Order Statistics

Algorithm \textsc{Randomized-Select} \((A, p, r, i)\)
1. \textbf{if} \(p = r\)
2. \hspace{1em} \textbf{return} \((A[p])\)
3. \(q = \textsc{Randomized Partition} \((A, p, r)\)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \hspace{1em} \textbf{return} \((A[q])\)
Algorithm \textsc{Randomized-Select} \((A, p, r, i)\)
1. \textbf{if} \(p = r\)
2. \hspace{1em} \textbf{return} \((A[p])\)
3. \(q = \textsc{Randomized Partition} \((A, p, r)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \hspace{1em} \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)

If pivots always partition lists into \(n/r\), \(r - 1\) \(r n\), for some \(r > 1\), time \(T(n)\) would have the recurrence
\[
T(n) \leq \max\{T(n r), T((r - 1) n r)\} + cn \leq T((r - 1) n r) + cn (r - 1) r + cn (r - 1) r 2 + ... cn (r - 1) r m = O(n)
\]
Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT \((A, p, r, i)\)
1. if \(p = r\)
2. return \((A[p])\)
3. \(q = \text{RANDOMIZED PARTITION} (A, p, r)\)
4. \(k = q - p + 1\)
5. if \(i = k\)
6. return \((A[q])\)
7. else if \(i < k\)
8. return \(\text{RANDOMIZED-SELECT} (A, p, q - 1, i)\)
Chapter 9. Medians and Order Statistics

Algorithm \textsc{Randomized-Select} \((A, p, r, i)\)

1. \textbf{if} \(p = r\)
2. \hspace{1em} \textbf{return} \((A[p])\)
3. \(q = \textsc{Randomized Partition} \((A, p, r)\)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \hspace{1em} \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)
8. \hspace{1em} \textbf{return} \((\textsc{Randomized-Select} \((A, p, q - 1, i)\))\)
9. \textbf{else return} \((\textsc{Randomized-Select} \((A, q + 1, r, i - k)\))\)
Chapter 9. Medians and Order Statistics

Algorithm Randomized-Select \((A, p, r, i)\)
1.  \textbf{if} \(p = r\)
2.     \textbf{return} \((A[p])\)
3.  \(q = \text{Randomized Partition} \((A, p, r)\)\)
4.  \(k = q - p + 1\)
5.  \textbf{if} \(i = k\)
6.     \textbf{return} \((A[q])\)
7.  \textbf{else if} \(i < k\)
8.     \textbf{return} \((\text{Randomized-Select} \((A, p, q - 1, i)\))\)
9.  \textbf{else return} \((\text{Randomized-Select} \((A, q + 1, r, i - k)\))\)

If pivots always partition lists into \(\frac{n}{r} : \frac{r-1}{r} n\), for some \(r > 1\),
Chapter 9. Medians and Order Statistics

Algorithm \texttt{Randomized-Select} \((A, p, r, i)\)

1. \textbf{if} \(p = r\)
2. \hspace{1em} \textbf{return} \((A[p])\)
3. \(q = \texttt{Randomized Partition} (A, p, r)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \hspace{1em} \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)
8. \hspace{1em} \textbf{return} \((\texttt{Randomized-Select} (A, p, q - 1, i))\)
9. \textbf{else return} \((\texttt{Randomized-Select} (A, q + 1, r, i - k))\)

If pivots always partition lists into \(\frac{n}{r^m} : \frac{r-1}{r^n} n\), for some \(r > 1\),
time \(T(n)\) would have the recurrence

\[ T(n) \leq \max\{T\left(\frac{n}{r}\right), T\left(\frac{(r-1)n}{r}\right)\} + nc \]
Chapter 9. Medians and Order Statistics

Algorithm Randomized-Select \((A, p, r, i)\)
1. \textbf{if} \(p = r\)
2. \textbf{return} \((A[p])\)
3. \(q = \text{Randomized Partition} \((A, p, r)\)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)
8. \textbf{return} \((\text{Randomized-Select} \((A, p, q - 1, i)\))\)
9. \textbf{else return} \((\text{Randomized-Select} \((A, q + 1, r, i - k)\))\)

If pivots always partition lists into \(\frac{n}{r} : \frac{r-1}{r} n\), for some \(r > 1\),

time \(T(n)\) would have the recurrence

\[
T(n) \leq \max\{T\left(\frac{n}{r}\right), T\left(\frac{(r - 1)n}{r}\right)\} + nc \leq T\left(\frac{(r - 1)n}{r}\right) + cn
\]

assuming \(r \geq 2\),
Algorithm `RANDOMIZED-SELECT` \((A, p, r, i)\)

1. \textbf{if} \(p = r\)
2. \hspace{1em} \textbf{return} \((A[p])\)
3. \(q = \text{RANDOMIZED PARTITION} \((A, p, r)\)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \hspace{1em} \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)
8. \hspace{1em} \textbf{return} \((\text{RANDOMIZED-SELECT} \((A, p, q - 1, i)\))\)
9. \hspace{1em} \textbf{else} \textbf{return} \((\text{RANDOMIZED-SELECT} \((A, q + 1, r, i - k)\))\)

If pivots always partition lists into \(\frac{n}{r} : \frac{r-1}{r} n\), for some \(r > 1\),
time \(T(n)\) would have the recurrence

\[
T(n) \leq \max\{T\left(\frac{n}{r}\right), T\left(\frac{(r-1)n}{r}\right)\} + nc \leq T\left(\frac{(r-1)n}{r}\right) + cn
\]

assuming \(r \geq 2\),

\[
T(n) \leq cn\frac{(r-1)}{r} + cn\left(\frac{(r-1)}{r}\right)^2 + cn\left(\frac{(r-1)}{r}\right)^3 + \ldots cn\left(\frac{(r-1)}{r}\right)^m = O(n)
\]

where \(\left(\frac{(r-1)}{r}\right)^m n = 1\), \(m = \log_{\frac{r-1}{r}} n\)
Chapter 9. Medians and Order Statistics

Performance analysis
Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$. 
Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$
Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;
Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;
- the algorithm identifies a pivot and recursively computes on sublist $A[p..q]$ (or $A[q+1..r]$);
Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;
- the algorithm identifies a pivot and recursively computes on sublist $A[p..q]$ (or $A[q + 1..r]$);
- the pivot is chosen with probability $\frac{1}{n}$;
Average case: $E[T(n)]$ (cont’)

- so the expected time $E[T(n)]$ needs to include the average
time of recursion on the case when sublist $A[p..q]$ possibly has
lengths $k = 0, 1, 2, \ldots, n - 1$
Average case: $E[T(n)]$ (cont’)

- so the expected time $E[T(n)]$ needs to include the average
time of recursion on the case when sublist $A[p..q]$ possibly has
lengths $k = 0, 1, 2, \ldots, n - 1$

- thus the expected time $E[T(n)]$ is computed as

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k - 1, n - k\})] + an,$$
for some constant $a > 0$
Chapter 9. Medians and Order Statistics

Average case: $E[T(n)]$ (cont’)

- so the expected time $E[T(n)]$ needs to include the average time of recursion on the case when sublist $A[p..q]$ possibly has lengths $k = 0, 1, 2, \ldots, n − 1$

- thus the expected time $E[T(n)]$ is computed as

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k−1, n−k\})] + an, \text{ for some constant } a > 0$$

because $\max\{k−1, n−k\} = k − 1$ if $k > n/2$ and $\max\{k−1, n−k\} = n − k$ if $k \leq n/2$

$$E[T(n)] \leq \frac{2}{n} \sum_{k=n/2}^{n-1} E[T(k)] + an$$
We conclude that $E[T(n)] \leq \frac{2}{n} \sum_{k=1}^{n-1} k = \frac{n}{2}$.

**Theorem.** $E[T(n)] = O(n)$.

**Proof (by substitution method).**

We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = ?$, we will decide later;
- **Assumption:** for all $k \leq n-1$, $E[T(k)] \leq ck$;
- **Induction:**

$$E[T(n)] \leq \frac{2}{n} \sum_{k=1}^{n-1} k = \frac{n}{2} E[T(k)] + an \leq \frac{2}{n} \sum_{k=1}^{n-1} k = \frac{n}{2} \left( \frac{n}{2} - 1 \right) + \left( \frac{1}{2} - \frac{1}{2}\right)n + an = \cdots = 3cn/4 + c/2 + an \leq cn$$ when $(cn/4 - c/2 - an) \geq 0$.

- **Base case:** $T(n) \leq cn$, for $n \geq \frac{2c}{(c-4a)}$.
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$. 

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = ?$, we will decide later;
- **Assumption:** for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- **Induction:**

  $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} k = n/2 E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} k = n/2 \cdot \frac{n}{2} - \frac{n}{2} \sum_{k=1}^{k=n/2} k + an = \cdots = 3cn/4 + c/2 + an \leq cn$ when $(cn/4 - c/2 - an) \geq 0$.

That is when $n \geq 2c/(c-4a)$. How to prove?
Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method).
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$. 

We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = \ldots$
We conclude that $E[T(n)] \leq \frac{2}{n} \sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = \_\_\_\_$, we will decide later;
We conclude that \( E[T(n)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an \)

**Theorem.** \( E[T(n)] = O(n) \).

**Proof** (by substitution method). We will prove that \( E[T(n)] \leq cn \) for some \( c > 0 \).

- Base case: \( n = ? \), we will decide later;
- Assumption: for all \( k \leq n - 1 \), \( E[T(k)] \leq ck \);
Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = 2$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an$$
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = ?$, we will decide later;
- **Assumption:** for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- **Induction:**

  $$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1}ck + an$$
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = ?,$ we will decide later;
- **Assumption:** for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- **Induction:**

$$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an$$
Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

• Base case: $n = ?$, we will decide later;

• Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;

• Induction:

\[
E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an
\]

\[
= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n(n-1)}{2} - \frac{n(n/2-1)}{2} (n/2) \right] + an
\]
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \frac{n/2-1}{n/2} \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2-1}{2} (n/2) \right] + an$$

$$= \cdots$$
Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case**: $n = ?$, we will decide later;
- **Assumption**: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- **Induction**:

  $$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

  $$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2 - 1}{2} (n/2) \right] + an$$

  $$= \cdots = 3cn/4 + c/2 + an$$
Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

  
  $$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

  $$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2 - 1}{2} (n/2) \right] + an$$

  $$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- **Base case:** $n = ?$, we will decide later;
- **Assumption:** for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- **Induction:**

  $$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

  $$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2-1}{2} (n/2) \right] + an$$

  $$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

  when $(cn/4 - c/2 - an) \geq 0$. 
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = \?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2 - 1}{2} (n/2) \right] + an$$

$$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

when $(cn/4 - c/2 - an) \geq 0$. That is when $n \geq 2c/(c - 4a)$
We conclude that $E[T(n)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$.

**Proof** (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n \sum_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2-1}{2} (n/2) \right] + an$$

$$= \cdots = \frac{3cn}{4} + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

when $(cn/4 - c/2 - an) \geq 0$. That is when $n \geq 2c/(c - 4a)$

- Base case: $T(n) \leq cn$, for $n < 2c/(c - 4a)$,
We conclude that \( E[T(n)] \leq 2/n \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + an \)

**Theorem.** \( E[T(n)] = O(n) \).

**Proof** (by substitution method). We will prove that \( E[T(n)] \leq cn \) for some \( c > 0 \).

- Base case: \( n = ? \), we will decide later;
- Assumption: for all \( k \leq n - 1 \), \( E[T(k)] \leq ck \);
- Induction:

\[
E[T(k)] \leq 2/n \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + an \leq 2/n \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an
\]

\[
= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} (n) - \frac{n/2-1}{2} (n/2) \right] + an
\]

\[
= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn
\]

when \( (cn/4 - c/2 - an) \geq 0 \). That is when \( n \geq 2c/(c - 4a) \)

- Base case: \( T(n) \leq cn \), for \( n < 2c/(c - 4a) \), How to prove?
Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

• analyzing time $T(n)$ of the algorithm
• obtain an expression $T(n) = \ldots$
• guess an upper (or lower) bound (e.g., $T(n) = O(\ldots)$)
• prove the correctness of the bound.

For example, given Insertion Sort:

• we first analyzed the algorithm and obtained
  $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 n \sum_{j=2}^{n} t_j + c_6 n \sum_{j=2}^{n} (t_j - 1) + c_7 n \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$

• we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
• and finally proved that it was indeed the case.
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

• analyzing time $T(n)$ of the algorithm
• obtain an expression $T(n) = \ldots$
• guess an upper (or lower) bound (e.g., $T(n) = O( )$)
• prove the correctness of the bound.

For example, given Insertion Sort:

• we first analyzed the algorithm and obtained $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 n \sum_{j=2} t_j + c_6 n \sum_{j=2} (t_j - 1) + c_7 n \sum_{j=2} (t_j - 1) + c_8 (n - 1)$
• we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
• and finally proved that it was indeed the case.
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
Chapter 9. Medians and Order Statistics

**Summary of Algorithm Analysis Scenarios**

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ldots)$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm and obtained $T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n} (t_j - 1) + c_7\sum_{j=2}^{n} (t_j - 1) + c_8(n - 1)$
- we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
- and finally proved that it was indeed the case.
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm and obtained $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 n \sum_{j=2}^{n} t_j + c_6 n \sum_{j=2}^{n} (t_j - 1) + c_7 n \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$
- we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
- and finally proved that it was indeed the case.
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

• analyzing time $T(n)$ of the algorithm
• obtain an expression $T(n) = \ldots$
• guess an upper (or lower) bound (e.g., $T(n) = O(\cdot)$)
• prove the correctness of the bound.
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given **Insertion Sort**: 

$$T(n) = c_1 n + c_2 (n - 1) + c_3 n \sum_{j=2}^{n} t_j + c_4 (n - 1)$$

we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$; and finally proved that it was indeed the case.
Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\cdot)$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm and obtained

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$
Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm and obtained

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$

- we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

• analyzing time $T(n)$ of the algorithm
• obtain an expression $T(n) = \ldots$
• guess an upper (or lower) bound (e.g., $T(n) = O(\) )
• prove the correctness of the bound.

For example, given **Insertion Sort**:

• we first analyzed the algorithm and obtained

$$T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$

• we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
• and finally proved that it was indeed the case.
Summary of Algorithm Analysis Scenarios

For recursive algorithms
For example, given the Binary Search algorithm,
• we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$T(n) \leq T(\lfloor n/2 \rfloor) + c$
• we guess upper bound $T(n) = O(\log_2 n)$, i.e.,
$T(n) \leq c \log_2 n$;
• we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function;
or
(2) we can use the substitution method by the principle of induction.

But we need the recurrence to apply induction.
using the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + c$ to prove $T(n) \leq c \log_2 n$.
see previous lecture notes.
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given **Binary Search** algorithm,

- we first analyze the time $T(n)$ of the algorithm
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

• we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

\[ T(n) \leq T(\lfloor n/2 \rfloor) + c \]

• we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

\[
T(n) \leq T(\lfloor n/2 \rfloor) + c
\]

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function;
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given \textsc{Binary Search} algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by \textit{unfolding} the time function; or
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

- we first analyze the time \( T(n) \) of the algorithm and obtained a recurrence for \( T(n) \)

\[
T(n) \leq T(\lfloor n/2 \rfloor) + c
\]

- we guess upper bound \( T(n) = O(\log_2 n) \), i.e., \( T(n) \leq c \log_2 n \);
- we prove the guessed bound.

  (1) we can use the recursive tree method by unfolding the time function; or
  (2) we can use the substitution method by the principle of induction.

see previous lecture notes
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given \textsc{Binary Search} algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

\[
T(n) \leq T(\lfloor n/2 \rfloor) + c
\]

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;

- we prove the guessed bound.

  (1) we can use the recursive tree method by unfolding the time function; or

  (2) we can use the substitution method by the principle of induction.

  \textbf{But we need the recurrence to apply induction.}
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

• we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

\[ T(n) \leq T(\lfloor n/2 \rfloor) + c \]

• we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
• we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or
(2) we can use the substitution method by the principle of induction.

But we need the recurrence to apply induction.

using the recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$

to prove $T(n) \leq c \log_2 n$. 
Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given Binary Search algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

\[ T(n) \leq T(\lfloor n/2 \rfloor) + c \]

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

1. we can use the recursive tree method by unfolding the time function; or
2. we can use the substitution method by the principle of induction.

But we need the recurrence to apply induction.

using the recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$

to prove $T(n) \leq c \log_2 n$. see previous lecture notes