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

- Chapter 6. Heapsort, the use of priority queue
- 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

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

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

- $\text{key(parent)} \geq \text{key(leftChild)}$, $\text{key(rightChild)}$;
- 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:
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**($A, i$)
The heap sort algorithm consists of subroutines:

- \textbf{Build-Max-Heap}(A)
- \textbf{Max-Heapify}(A, i)
- \textbf{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
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**\(^1\)(\(A\))
- **Max-Heapify**\(^1\)(\(A, i\))
- **HeapSort**\(^1\)(\(A\))

heaps as priority queues

- **Heap-Maximum**\(^1\)(\(A\))
- **Heap-Extract-Max**\(^1\)(\(A\))
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:

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

Heaps as priority queues

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

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. for i = length[A] downto 2
4. heapsize[A] = heapsize[A] − 1
5. Max-Heapify(A, 1)

T_{HS}(n) = T_{BMH}(n) + (n−1)T_{MH}(n, 1)

Subroutine Build-Max-Heap(A)

1. heapsize[A] = length[A]
2. for i = ⌊ length[A] / 2 ⌋ downto 1
3. Max-Heapify(A, i)
Chapter 6. Heapsort

Algorithm HeapSort(A)

1. BUILD-MAX-HEAP(A)
Chapter 6. Heapsort

Algorithm HeapSort$(A)$

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

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. for i = length[A] downto 2
Chapter 6. Heapsort

Algorithm \textsc{HeapSort}(A)

1. \texttt{Build-Max-Heap}(A)
2. \texttt{for } i = \texttt{length}[A] \texttt{ downto } 2
3. \hspace{1em} exchange \texttt{A}[1] \leftrightarrow \texttt{A}[i]
4. \hspace{1em} heapsize[A] = heapsize[A] \, - \, 1
Chapter 6. Heapsort

Algorithm HEAPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for $i = \text{length}[A]$ downto 2
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. MAX-HEAPIFY(A, 1)
Chapter 6. Heapsort

Algorithm \textsc{HeapSort}(A)
\begin{enumerate}
\item \textbf{Build-Max-Heap}(A)
\item \textbf{for} \(i = \text{length}[A]\) \textbf{downto} 2
\item exchange \(A[1] \rightleftharpoons A[i]\)
\item \(\text{heapsize}[A] = \text{heapsize}[A] - 1\)
\item \textbf{Max-Heapify}(A, 1)
\end{enumerate}

\(T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1)\)
Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. for $i = \text{length}[A]$ downto 2
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. Max-Heapify(A, 1)

$T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1)$

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

Algorithm HeapSort($A$)

1. Build-Max-Heap($A$)
2. for $i = \text{length}[A]$ downto 2
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. Max-Heapify($A, 1$)

$T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1)$

Subroutine Build-Max-Heap($A$)

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

Algorithm \textsc{HeapSort}(A)

1. \textbf{Build-Max-Heap}(A)
2. for $i = \text{length}[A]$ \textbf{downto} 2
4. $\text{heapsize}[A] = \text{heapsize}[A] - 1$
5. \textbf{Max-Heapify}(A, 1)

$T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1)$

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

1. $\text{heapsize}[A] = \text{length}[A]$
2. for $i = \lfloor \frac{1}{2} \text{length}[A] \rfloor$ \textbf{downto} 1
Chapter 6. Heapsort

Algorithm \textsc{HeapSort}(A)

1. \textbf{Build-Max-Heap}(A)
2. \textbf{for} \( i = \text{length}[A] \) \textbf{downto} 2
4. \( \text{heapsize}[A] = \text{heapsize}[A] - 1 \)
5. \textbf{Max-Heapify}(A, 1)

\[ T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1) \]

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

1. \( \text{heapsize}[A] = \text{length}[A] \)
2. \textbf{for} \( i = \lfloor \frac{1}{2}\text{length}[A] \rfloor \) \textbf{downto} 1
3. \textbf{Max-Heapify}(A, i)
Chapter 6. Heapsort

Algorithm HeapSort(A)
1. Build-Max-Heap(A)
2. for i = length[A] downto 2
4. heapsize[A] = heapsize[A] − 1
5. Max-Heapify(A, 1)

\[ T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1) \]

Subroutine Build-Max-Heap(A)
1. heapsize[A] = length[A]
2. for i = ⌊\frac{1}{2}length[A]⌋ downto 1
3. Max-Heapify(A, i)

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

Subroutine $\text{MAX-HEAPIFY}(A, i)$
Subroutine MAX-HEAPIFY($A, i$)

1. \( l = \text{LEFT}[i] \)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. \( l = \text{LEFT}[i] \)
2. \( r = \text{RIGHT}[i] \)
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

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

Subroutine \texttt{MAX-HEAPIFY}(A, i)

1. \quad l = \texttt{LEFT}[i]
2. \quad r = \texttt{RIGHT}[i]
3. \quad \textbf{if} (l \leq \texttt{heapsize}[A]) \textbf{and} (A[l] > A[i])
4. \quad \textbf{then} largest = l
Chapter 6. Heapsort

Subroutine Max-Heapify\( (A, i) \)

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

Subroutine \texttt{MAX-HEAPIFY}(A, i)

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

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

1. $l = \text{LEFT}[i]$
2. $r = \text{RIGHT}[i]$
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 $\text{largest} = r$

$T_{\text{MH}}(n,i) = c + T_{\text{MH}}(n,2i)$

$T_{\text{MH}}(n,i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$T_{\text{BMH}}(n) = \lfloor \frac{n}{2} \rfloor \sum_{i=1}^{n} T_{\text{MH}}(n,i)$

$T_{\text{BS}}(n) = T_{\text{BMH}}(n) + (n-1)T_{\text{MH}}(n,1)$

$T_{\text{BS}}(n) \leq c n^2 \log_2 n \leq O(n \log n)$
Chapter 6. Heapsort

Subroutine $\text{MAX-HEAPIFY}(A, i)$

1. $l = \text{LEFT}[i]$
2. $r = \text{RIGHT}[i]$
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 $\neq i$
Chapter 6. Heapsort

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

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

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

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

1. \[l = \text{LEFT}[i]\]
2. \[r = \text{RIGHT}[i]\]
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. \[\text{Max-Heapify}(A, \text{largest})\]

\[T_{MH}(n, i) = c + T_{MH}(n, 2i)\]
Chapter 6. Heapsort

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

1. \hspace{1em} l = \textsc{LEFT}[i]
2. \hspace{1em} r = \textsc{RIGHT}[i]
3. \hspace{1em} \textbf{if} (l \leq \text{heapsize}[A]) \textbf{and} (A[l] > A[i])
4. \hspace{1em} \hspace{1em} \textbf{then} largest = l
5. \hspace{1em} \hspace{1em} \textbf{else} largest = i
6. \hspace{1em} \textbf{if} (r \leq \text{heapsize}[A]) \textbf{and} (A[r] > A[\text{largest}])
7. \hspace{1em} \hspace{1em} \textbf{then} largest = r
8. \hspace{1em} \hspace{1em} \textbf{if} largest \neq i
9. \hspace{1em} \hspace{1em} \hspace{1em} \textbf{then} exchange A[i] \longleftrightarrow A[\text{largest}]
10. \hspace{1em} \hspace{1em} \textsc{Max-Heapify}(A, \text{largest})

\[ T_{\text{MH}}(n, i) = c + T_{\text{MH}}(n, 2i) \]

\[ T_{\text{MH}}(n, i) \leq c \log_2 n, \text{ for all } i = 1, 2, \ldots, n. \]
Chapter 6. Heapsort

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

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

\( T_{MH}(n,i) = c + T_{MH}(n,2i) \)

\( T_{MH}(n,i) \leq c \log_2 n, \text{ for all } \text{i} = 1, 2, \ldots, n. \)

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

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

1. \hspace{0.5em} l = LEFT[i]
2. \hspace{0.5em} r = RIGHT[i]
3. \textbf{if} \hspace{0.5em} (l \leq \text{heapsize}[A]) \hspace{0.5em} \textbf{and} \hspace{0.5em} (A[l] > A[i])
4. \hspace{0.5em} \textbf{then} \hspace{0.5em} \text{largest} = l
5. \hspace{0.5em} \textbf{else} \hspace{0.5em} \text{largest} = i
6. \textbf{if} \hspace{0.5em} (r \leq \text{heapsize}[A]) \hspace{0.5em} \textbf{and} \hspace{0.5em} (A[r] > A[\text{largest}])
7. \hspace{0.5em} \textbf{then} \hspace{0.5em} \text{largest} = r
8. \hspace{0.5em} \textbf{if} \hspace{0.5em} \text{largest} \neq i
9. \hspace{0.5em} \textbf{then} \hspace{0.5em} \text{exchange} \hspace{0.5em} A[i] \leftrightarrow A[\text{largest}]
10. \hspace{0.5em} \textsc{Max-Heapify}(A, \text{largest})

\[ T_{MH}(n, i) = c + T_{MH}(n, 2i) \]

\[ T_{MH}(n, i) \leq c \log_2 n, \text{ for all } i = 1, 2, \ldots, n. \]

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

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

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

\[
T_{MH}(n, i) = c + T_{MH}(n, 2i)
\]

\[
T_{MH}(n, i) \leq c \log_2 n, \text{ for all } i = 1, 2, \ldots, n.
\]

\[
T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c \frac{n}{2} \log_2 n
\]

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

Subroutine $\text{MAX-HEAPIFY}(A, i)$

1. $l = \text{LEFT}[i]$
2. $r = \text{RIGHT}[i]$
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 $\text{largest} = r$
8. if $\text{largest} \neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. $\text{MAX-HEAPIFY}(A, \text{largest})$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c \frac{n}{2} \log_2 n$

$T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1) \leq c \frac{n}{2} \log_2 n + (n - 1)c \log_2 n$
Chapter 6. Heapsort

Subroutine MAX-HEAPIFY($A, i$)

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
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 $A[i] \leftarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$T_{BMH}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} T_{MH}(n, i) \leq c \frac{n}{2} \log_2 n$

$T_{HS}(n) = T_{BMH}(n) + (n - 1)T_{MH}(n, 1) \leq c \frac{n}{2} \log_2 n + (n - 1)c \log_2 n \leq O(n \log n)$
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")
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) replace a key with a larger value
1. if key < A[i] then return ("new key is smaller than current key")
3. A[i] = key
4. while i > 1 and A[PARENT[i]] < A[i]
6. 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 \textsc{Heap-Maximum}(A) obtain the maximum
1. \texttt{return} \((A[1])\)
Chapter 6. Heapsort

Operations on heaps:

Function \textsc{Heap-Maximum}(A)
1. \textbf{return} \((A[1])\)

Function \textsc{Heap-Extract-Max}(A)
1. \textbf{if} \text{heapsize}[A] < 1
2. \textbf{then return} ("heap underflow")
3. \textit{max} = A[1]
5. \text{heapsize}[A] = \text{heapsize}[A] − 1
6. \text{Max-Heapify}(A, 1)
7. \textbf{return} (\textit{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[PARENT[*i*]] < A[*i*]`
5. `exchange A[*i*] ←→ A[PARENT[*i*]]`
6. `i = PARENT[*i*]`
Chapter 6. Heapsort

Operations on heaps:

Function \textsc{Heap-Maximum}(A)
1. \textbf{return} \(A[1]\)

Function \textsc{Heap-Extract-Max}(A)
1. \textbf{if} \ heapsize[A] < 1
2. \textbf{then return} ("heap underflow")
3. \(max = A[1]\)
5. \(\text{heapsize}[A] = \text{heapsize}[A] - 1\)
6. \textsc{Max-Heapify}(A, 1)
7. \textbf{return} \(max\)

Function \textsc{Heap-Increase-Key}(A, i, key)
1. \textbf{if} \ key < A[i]
2. \textbf{then return} ("new key is smaller than current key")
3. \(A[i] = key\)
4. \textbf{while} \(i > 1\) \textbf{and} \(A[\text{Parent}[i]] < A[i]\)
5. \textbf{exchange} \(A[i] \leftrightarrow A[\text{Parent}[i]]\)
6. \(i = \text{Parent}[i]\)

Function \textsc{Max-Heap-Insert}(A, key)
1. \(\text{heapsize}[A] = \text{heapsize}[A] + 1\)
2. \(A[\text{heapsize}[A]] = -\infty\)
3. \textsc{Heap-Increase-Key}(A, \text{heapsize}[A], key)
Chapter 7. Quicksort

Chapter 7. Quicksort and randomized algorithms
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:

(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
  
  \( A[i] \leq A[q] \) for all \( i = p, \cdots, q - 1 \)
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

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

1. if \(p < r\) then
2. \(q = \text{Partition}(A, p, r)\)
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

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + cn = O(n \log_2 n)
\]

- Assume \(A[q]\) partitions the list 20% vs 80%, then

\[
T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + cn = O(n \log_2 n)
\]

- Assume \(A[q]\) partitions the list 1% vs 99%, then

\[
T(n) \leq T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right) + cn = O(n \log_2 n)
\]

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

Chapter 7. Quicksort and Randomized algorithms

Algorithm \textsc{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 and Randomized algorithms

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

How can we identify such a pivot?
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. \textbf{QUICKSORT} \( (A, p, q - 1) \)
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.

- 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
  \[
  T(n) \leq T(5n/6) + T(4n/6) + cn = O(n \log_2 n)
  \]

- Assume \(A[q]\) partitions the list 1\% vs 99\%, then
  \[
  T(n) \leq T(100n/101) + T(99n/100) + cn = O(n \log_2 n)
  \]
Chapter 7. Quicksort and Randomized algorithms

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

1. \textbf{if} \(p < r\)
2. \textbf{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.
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. \textbf{QUICKSORT} \((A, p, q - 1)\)
4. \textbf{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 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 
  \[ T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + cn = O(n \log_2 n) \]
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
  \[T(n) \leq T(n/5) + T(4n/5) + cn = O(n \log_2 n)\]

- Assume \(A[q]\) partitions the list 1% vs 99%, then
  \[T(n) \leq T(n/100) + T(99n/100) + cn = O(n \log_2 n)\]
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
  \[T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + cn = O(n \log_2 n)\]

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

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

```
2 8 7 1 3 5 6
```

```
2 8 7 1 3 5 6
```

```
2 8 7 1 3 5 6
```

```
2 1 7 8 3 5 6
```

```
2 1 7 8 3 5 6
```

```
2 1 3 8 7 5 6
```

```
2 1 3 8 7 5 6
```

```
2 1 3 8 7 5 6
```

```
2 1 3 4 7 5 6 8
```

quicksort(A,p,q-1)  quicksort(A,q+1,r)
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 \(i + 1\)
Chapter 7. Quicksort

\texttt{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.
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, 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, we have

\[ l_0: \quad cn \quad cn \]
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, we have

\[ \begin{align*}
  l_0: & \quad & cn \\
  l_1: & \quad & cn/10 \quad 9cn/10
\end{align*} \]
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, we have

\[
\begin{align*}
  l_0: & & cn & & cn \\
  l_1: & & cn/10 & & 9cn/10 \\
  l_2: & & cn/10^2 & & 9cn/10^2 & & 9cn/10^2 & & 9^2cn/10^2 & & \cdots
\end{align*}
\]
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, we have

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

\[ c' n \log_2 n = O(n \log n) \]

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, we have

- \( l_0: \) \( cn \)
- \( l_1: \) \( cn/10 \) \( 9cn/10 \)
- \( l_2: \) \( cn/10^2 \) \( 9cn/10^2 \) \( 9^2cn/10^2 \)
- \( l_h: \) \( cn/10^h \) \( \cdots \) \( c9^h n/10^h \) \( cn \)

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, we have

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

\[ c/\log_{10} 9 \]
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, we have

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

where \( \left( \frac{1}{10} \right)^h n = 1 \), 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, we have

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

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, we have

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

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

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

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

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, we have

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

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

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

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

\[ T(n) \leq cn \log_{\frac{9}{10}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} \]
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, we have

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

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., \( k = \log_{\frac{9}{10}} n \)

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

\[ T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c' n \log_2 n \]
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, we have

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

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

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

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

\[ T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c'n \log_2 n = O(n \log_2 n) \]
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, we have

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

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., \( k = \log_{\frac{9}{10}} n \)

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

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

where \( c' = c / \log_2 \frac{10}{9} \)
Instead of analyzing \texttt{QuickSort} (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.
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 $\left(\text{PARTITION}(A, p, r)\right)$

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

Algorithm Randomized QuickSort \((A, p, r)\)
1. if \(p < r\)
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} \((A, p, r)\)

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

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

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

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

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

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

Analysis of RANDOMIZED-QUICKSORT
Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between $x_i$ and $x_j$;
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 $X_{i,j}$ indicating if a comparison between $x_i$ and $x_j$ 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 \(X_{i,j}\) indicating if a comparison between \(x_i\) and \(x_j\) occurs.

• \(X_{i,j} \in \{0,1\}\)
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 \(X_{i,j}\) indicating
  if a comparison between \(x_i\) and \(x_j\) occurs.

\(X_{i,j} \in \{0, 1\}\)

• let \(X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}\), written as \(X = \sum_{i<j} X_{i,j}\)
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 $X_{i,j}$ indicating if a comparison between $x_i$ and $x_j$ occurs.

- $X_{i,j} \in \{0, 1\}$

- let $X = \sum_{i=1}^{n} \sum_{j=1,i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$

- the expected number of comparisons is

$$E(X) = E\left(\sum_{i<j} X_{i,j}\right)$$
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 $X_{i,j}$ indicating
  if a comparison between $x_i$ and $x_j$ occurs.

• $X_{i,j} \in \{0, 1\}$

• let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$

• 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 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 $X_{i,j}$ indicating if a comparison between $x_i$ and $x_j$ occurs.

- $X_{i,j} \in \{0, 1\}$

- let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$

- the expected number of comparisons is

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} P(X_{i,j} = 1)$$
Chapter 7. Quicksort

Analysis of \texttt{RANDOMIZED-QUICKSORT} (cont.)

\[
E(X) = E(\sum_{i<j} X_{i,j})
\]
Analysis of **RANDOMIZED-QUICKSORT** (cont.)

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

\[ \sum_{i<j} \text{Prob}(X_{i,j} = 1)X_{i,j} = 1, \text{ i.e., comparison between } x_i \text{ and } x_j \text{ occurs only when } \]

(1) \( x_i, x_j \) are in the same sublist \( L \);

(2) either is chosen to be the pivot;

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

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

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

So

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

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

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

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

\(X_{i,j} = 1\), i.e., comparison between \(x_i\) and \(x_j\) occurs only when
Analysis of \textsc{Randomized-Quicksort} (cont.)

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

\( X_{i,j} = 1, \) 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 \textsc{Randomized-QuickSort} (cont.)

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

\(X_{i,j} = 1\), 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;

...
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;

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

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

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

\( X_{i,j} = 1 \), 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 \( L \)!
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;

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

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,
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, then
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;

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

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)

\[ |L| \geq (j - i + 1) \]

So \[ P(X_{i,j} = 1) \leq 2 \frac{1}{|L|} \leq 2 \frac{1}{j-i+1} \]
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

x₅ x₁₀
Analysis of \texttt{RANDOMIZED-QUICKSORT} (cont.)

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

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

\[
\leq \sum_{i<j} c \log_2 n \leq cn \log_2 n
\]

for some constant \(c > 0\).

So \(E(X) = O(n \log_2 n)\).
Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT (cont.)

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

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

Analysis of RANDOMIZED-QUICKSORT (cont.)

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

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

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

for some constant \( c > 0 \).

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

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

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

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

Analysis of **Randomized-QuickSort** (cont.)

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

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

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

Analysis of `RANDOMIZED-QUICKSORT` (cont.)

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

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

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

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

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

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

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

for some constant \( c > 0 \).

So \( E(X) = O(n \log_2 n) \).
Chapter 7. Quicksort

O(n log n) Sorting Algorithms

---

Diagram showing the comparison of Heap, Merge, and Quick sorting algorithms with respect to time in seconds for different input sizes (n).
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 $c > 0$.

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

In other words, $\lim_{n \to \infty} f(n)/g(n) = \text{constant} > 0$ or $\infty$. For example, we have shown $T(n) = \Omega(n^2)$ for Insertion Sort.

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

Chapter 8. Lower bounds and sorting in linear time
Chapter 8. Lower Bounds and Sorting in Linear Time

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.

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

In other words, \(\lim_{n \to \infty} f(n) / g(n) = \text{constant} > 0\) or \(\infty\).

For example, we have shown \(T(n) = \Omega(n^2)\) for Insertion Sort.

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


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 $c > 0$. In other words, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant} > 0$ or $\infty$ for example, we have shown $T(n) = \Omega(n^2)$ for Insertion Sort.

Proof techniques for Big-$\Omega$ are similar to those for Big-$O$. 
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 $c > 0$.

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

In other word, $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$
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 $c > 0$.

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

In other word, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant} > 0$ or
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 $c > 0$.

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

In other word, $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant} > 0 \text{ or } \infty$$
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 $c > 0$.

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

In other word, $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant } > 0 \text{ or } \infty$$

For example, we have shown $T(n) = \Omega(n^2)$ for Insertion Sort.
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 \( c > 0 \).

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

In other word, \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant} > 0 \text{ or } \infty \)

For example, we have shown \( T(n) = \Omega(n^2) \) for Insertion Sort.

Proof techniques for Big-\(\Omega\) are similar to those for Big-\(O\).
Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound
## Chapter 8. Lower Bounds and Sorting in Linear Time

### Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log_2 n)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
</tbody>
</table>

So we say

- **Insertion Sort** runs in time $\Theta(n^2)$,
- **MergeSort** runs in time $\Theta(n \log_2 n)$,
- the sorting problem has $\Theta(n \log_2 n)$ time complexity.
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td>INSERTION SORT $O(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td>INSERTION SORT $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

So we say
- Insertion Sort runs in time $\Theta(n^2)$,
- MergeSort runs in time $\Theta(n \log_2 n)$,
- the sorting problem has $\Theta(n \log_2 n)$ time complexity.
Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td>INSERTION SORT $O(n^2)$  $\rightarrow$  $O(n^2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MERGE SORT $O(n \log_2 n)$</td>
<td></td>
</tr>
</tbody>
</table>

So we say
- Insertion Sort runs in time $\Theta(n^2)$,
- MergeSort runs in time $\Theta(n \log_2 n)$,
- the sorting problem has $\Theta(n \log_2 n)$ time complexity.
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>INSERTION SORT</strong> $O(n^2)$  $\rightarrow$  $O(n^2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>MERGE SORT</strong> $O(n \log_2 n)$  $\rightarrow$  $O(n \log_2 n)$</td>
<td></td>
</tr>
</tbody>
</table>
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>Insertion Sort</strong> $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>Insertion Sort</strong> $\Omega(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>
### Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td>Insertion Sort $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>Merge Sort $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td>Insertion Sort $\Omega(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
</tbody>
</table>
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>INSERTION SORT</strong> $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td><strong>MERGE SORT</strong> $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>INSERTION SORT</strong> $\Omega(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td><strong>MERGE SORT</strong> $\Omega(n \log_2 n)$</td>
<td></td>
</tr>
</tbody>
</table>
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>Insertion Sort</strong> $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>Insertion Sort</strong> $\Omega(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $\Omega(n \log_2 n)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
</tbody>
</table>
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>Insertion Sort</strong> $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>Insertion Sort</strong> $\Omega(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $\Omega(n \log_2 n)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
</tbody>
</table>

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$. 
Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td>Insertion Sort $O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>Merge Sort $O(n \log_2 n)$</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td>Insertion Sort $\Omega(n^2)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td>Merge Sort $\Omega(n \log_2 n)$</td>
<td>$\Omega(n \log_2 n)$</td>
</tr>
</tbody>
</table>

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

So we say

- **Insertion Sort** runs in time $\Theta(n^2)$,
## Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>Insertion Sort</strong> (O(n^2)) (\rightarrow) (O(n^2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> (O(n \log_2 n)) (\rightarrow) (O(n \log_2 n))</td>
<td></td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>Insertion Sort</strong> (\Omega(n^2)) (\leftarrow) (\Omega(n \log_2 n))</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> (\Omega(n \log_2 n)) (\leftarrow) (\Omega(n \log_2 n))</td>
<td></td>
</tr>
</tbody>
</table>

\(f(n) = O(g(n))\) and \(f(n) = \Omega(g(n))\) if and only if \(f(n) = \Theta(g(n))\).

So we say
- **Insertion Sort** runs in time \(\Theta(n^2)\),
- **MergeSort** runs in time \(\Theta(n \log_2 n)\),
Important notes on lower bound and upper bound

<table>
<thead>
<tr>
<th>Bounds</th>
<th>Algorithms</th>
<th>Sorting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper bound</td>
<td><strong>Insertion Sort</strong> $O(n^2)$ $→$ $O(n^2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $O(n \log_2 n)$ $→$ $O(n \log_2 n)$</td>
<td></td>
</tr>
<tr>
<td>lower bound</td>
<td><strong>Insertion Sort</strong> $\Omega(n^2)$ $←$ $\Omega(n \log_2 n)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Merge Sort</strong> $\Omega(n \log_2 n)$ $←$ $\Omega(n \log_2 n)$</td>
<td></td>
</tr>
</tbody>
</table>

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

So we say

- **Insertion Sort** runs in time $\Theta(n^2)$,
- **MergeSort** runs in time $\Theta(n \log_2 n)$,
- the sorting problem has $\Theta(n \log_2 n)$ time complexity.
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!)
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 \(\geq n!\).
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!)
**Theorem:** Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.
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 \( \Omega(\log n!) \).
**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 $\Omega(\log n!)$. 

\[
n! = n(n-1)(n-2) \cdots \left(n - \frac{n}{2}\right) \left(n - \frac{n}{2} - 1\right) \cdots 2 \times 1 \\
\geq \left(\frac{n}{2}\right)^{n/2} \times 2^{n/2 - 1} \geq \frac{1}{2} n^{n/2}
\]

or by Stirling’s formula:

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

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

Sorting algorithms with worst case linear time
Chapter 8. Lower Bounds and Sorting in Linear Time

Sorting algorithms with worst case linear time

- count sort
- radix sort
- bucket sort
Count sort

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]
5. {C[i] 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] contains the number of elements whose values ≤ i}
9. for j = length[A] downto 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

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. \textbf{for} \(i = 0\) \textbf{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. \(C[i] = 0\)
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 \(\text{length}[A]\)
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} \(i = 0\ \text{to}\ k\)
2. \hspace{1em} \(C[i] = 0\)
3. \textbf{for} \(j = 1\ \text{to}\ \text{length}[A]\)
4. \hspace{1em} \(C[A[j]] = C[A[j]] + 1\)
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. \textbf{for} \(i = 0\) \textbf{to} \(k\)
2. \hspace{1em} \(C[i] = 0\)
3. \textbf{for} \(j = 1\) \textbf{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 = \(i\}\)
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. \(C[i] = 0\)
3. \textbf{for} \(j = 1\ \textbf{to} \ length[A]\)
4. \(C[A[j]] = C[A[j]] + 1\)
5. \(\{C[i]\ \text{contains the number of elements whose values} = i\}\)
6. \textbf{for} \(i = 1\ \textbf{to} \ k\)
Count sort

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

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

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

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

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

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

Example: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3\), \(k = 5\), \(C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)
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{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. \hspace{1em} \{\text{\(C[i]\) 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. \hspace{1em} \{\text{\(C[i]\) contains the number of elements whose values } \leq i\}\)
9. \textbf{for } \(j = \text{length}[A] \ \textbf{downto } 1\)
10. \hspace{1em} \(B[C[A[j]]] = A[j]\)
11. \hspace{1em} \(C[A[j]] = C[A[j]] - 1\)

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. \(\{C[i]\) 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. \(\{C[i]\) 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: \(A: 2 \ 5 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3, \quad k = 5, \quad C: 2 \ 0 \ 2 \ 3 \ 0 \ 1\)

analysis: \(T(n) = O(k + n)\)
Radix Sort:

Algorithm Radix-Sort \( \langle A, d \rangle \)

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 \( \Theta\left(\lceil \frac{b}{r} \rceil (n + 2r)\right) \) time.
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
### Radix Sort:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>720</td>
<td>720</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
<td></td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
<td></td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
<td></td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
<td></td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
<td>839</td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm** `Radix-Sort(A, d)`
Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

\[
\begin{array}{cccc}
329 & 720 & 720 & 329 \\
457 & 355 & 329 & 355 \\
657 & 436 & 436 & 436 \\
839 & 457 & 839 & 457 \\
436 & 657 & 355 & 657 \\
720 & 329 & 457 & 720 \\
355 & 839 & 657 & 839 \\
\end{array}
\]

Algorithm **Radix-Sort** \((A, d)\)

1. for \(i = 1\) to \(d\)
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>

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 Radix-Sort\((A, d)\)

1.  \textbf{for} \(i = 1\) \textbf{to} \(d\)
2.  \textbf{sort} \(A\) on the \(i\)th digit

Lemma. Given \(n\) \(b\)-bit binary numbers and any positive \(r \leq b\). 
\textbf{Radix-Sort} uses \(\Theta([b/r](n + 2^r))\) 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.
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 $\{0, 1, \ldots, 2^r - 1\}$. 
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.
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.
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.
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 \( \Theta([b/r](n + 2^r)) \) 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.

The total time is \( O([b/r](n + 2^r)) \), where \((n + 2^r)\) is time for Counting-Sort.

Since all steps in the two algorithms are mandatory, the total time is also \( \Omega([b/r](n + 2^r)) \), thus \( \Theta([b/r](n + 2^r)) \).

Once \( b \) and \( n \) are given, we can choose \( r \) to minimize the quantity \([b/r](n + 2^r)\).
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 **BUCKET-SORT**(A)
1. \( n = \text{length}[A] \)
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. \( \text{for } i = 1 \text{ to } n \)
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. \( \text{insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor] \)
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. \( \text{for } i = 0 \text{ to } n - 1 \)
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. \( \text{insert } A[i] \text{ into list } B[[nA[i]]] \)
4. \( \text{for } i = 0 \text{ to } n - 1 \)
5. \( \text{sort list } B[i] \text{ with Insertion Sort} \)
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 \( B[0], B[1], \ldots, B[n - 1] \)
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. \( \text{insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor] \)
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
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. \( \textbf{for } i = 1 \textbf{ to } n \)
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: \( 0 / \)
\( 1 \rightarrow .12 \rightarrow .17 \)
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm **Bucket-Sort**($A$)
1. $n = |A|$
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 $\rightarrow$.12 $\rightarrow$.17
2 $\rightarrow$.21 $\rightarrow$.23 $\rightarrow$.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
Chapter 8. Lower Bounds and Sorting in Linear Time

**Bucket Sort** (assuming uniform distribution of inputs)

Algorithm \textsc{Bucket-Sort}(A)

1. \hspace{1em} \textbf{n} = \text{length}[A]
2. \hspace{1em} \textbf{for} \hspace{0.5em} i = 1 \ \textbf{to} \ n
3. \hspace{1em} \quad \text{insert} \ A[i] \ \text{into list} \ B[\lfloor nA[i] \rfloor]
4. \hspace{1em} \textbf{for} \hspace{0.5em} i = 0 \ \textbf{to} \ n - 1
5. \hspace{1em} \quad \text{sort list} \ B[i] \ \text{with} \ \textsc{Insertion Sort}
6. \hspace{1em} \text{concatenate the lists} \ B[0], B[1], ..., B[n - 1]

\textbf{A}: \ .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68

\textbf{B}: \ 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
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. \( \text{for } i = 1 \text{ to } n \)
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: \(0 /
1 \rightarrow .12 \rightarrow .17
2 \rightarrow .21 \rightarrow .23 \rightarrow .26
3 \rightarrow .39
4 /
5 /\)
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 \( B[0], B[1], ..., B[n - 1] \)

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
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 \( B[0], B[1], ..., B[n - 1] \)

A: \( .78 \ .17 \ .39 \ .26 \ .72 \ .94 \ .21 \ .12 \ .23 \ .68 \)

B: \( 0 / \)
\( 1 \rightarrow .12 \rightarrow .17 \)
\( 2 \rightarrow .21 \rightarrow .23 \rightarrow .26 \)
\( 3 \rightarrow .39 \)
\( 4 / \)
\( 5 / \)
\( 6 \rightarrow .68 \)
\( 7 \rightarrow .72 \rightarrow .78 \)
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** $n - 1$
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 → .12 → .17
   2 → .21 → .23 → .26
   3 → .39
   4 /
   5 /
   6 → .68
   7 → .72 → .78
   8 /
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
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
• 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
- find the median (i.e., the $n/2$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

- 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 expected linear time (but worst case $\Theta(n^2)$)
• Selection in worst case linear time
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 $A$;
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 *expected* linear time (but worst case $\Theta(n^2)$)
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 *expected* linear time (but worst case $\Theta(n^2)$)
- Selection in worst case linear time
Selection in *expected* linear time
Selection in *expected* linear time

**Input:** a list $A$ of elements, an integer $i$;
Chapter 9. Medians and Order Statistics

Selection in expected linear time

**Input:** a list $A$ of elements, an integer $i$;

**Output:** the $i$th smallest element in $A$;
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 $k$;
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 $(x)$;
  - else 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 ($A, p, r, i$)
Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)
1. if p = r
2. return A[p]
3. q = 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 Randomized-Select (A, p, q − 1, i)
9. else return Randomized-Select (A, q + 1, r, i − k)

If pivots always partition lists into $n^r/r$, for some $r > 1$, time $T(n)$ would have the recurrence

$T(n) \leq \max\{T(n^r/r), T((r−1)n^r/r)\} + cn$

assuming $r \geq 2$, $T(n) \leq cn^r/r + cn((r−1)n^r/r)^2 + \ldots cn((r−1)n^r/r)^m = O(n)$

where $(r−1)n^r/r = 1$, $m = \log r/r − 1$. 
Chapter 9. Medians and Order Statistics

Algorithm 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. \(q = \text{Randomized Partition} (A, p, r)\)
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. $k = q - p + 1$
Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT \((A, p, r, i)\)

1. \(\textbf{if} \ p = r\)
2. \(\quad \textbf{return} \ (A[p])\)
3. \(q = \text{RANDOMIZED PARTITION} \ (A, p, r)\)
4. \(k = q - p + 1\)
5. \(\textbf{if} \ i = k\)
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} \((A[q])\)
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} \(i < k\)
Chapter 9. Medians and Order Statistics

Algorithm Randomized-Select \((A, p, r, i)\)
1. \(\textbf{if} \ p = r\)
2. \(\quad \textbf{return} \ (A[p])\)
3. \(q = \text{Randomized Partition} \ (A, p, r)\)
4. \(k = q - p + 1\)
5. \(\textbf{if} \ i = k\)
6. \(\quad \textbf{return} \ (A[q])\)
7. \(\quad \textbf{else if} \ i < k\)
8. \(\quad \quad \textbf{return} \ (\text{Randomized-Select} \ (A, p, q - 1, i))\)
Algorithm \textsc{Randomized-Select}\,(A,\,p,\,r,\,i)

1. \textbf{if} \ p = r
2. \quad \textbf{return} \ (A[p])
3. \quad q = \textsc{Randomized\,Partition}\,(A,\,p,\,r)
4. \quad k = q - p + 1
5. \quad \textbf{if} \ i = k
6. \quad \quad \textbf{return} \ (A[q])
7. \quad \textbf{else if} \ i < k
8. \quad \quad \textbf{return} \ (\textsc{Randomized-Select}\,(A,\,p,\,q - 1,\,i))
9. \quad \textbf{else} \ \textbf{return} \ (\textsc{Randomized-Select}\,(A,\,q + 1,\,r,\,i - k))
Chapter 9. Medians and Order Statistics

Algorithm \textsc{Randomized-Select} \((A, p, r, i)\)

1. \textbf{if} \(p = r\)
2. \quad \textbf{return} \((A[p])\)
3. \(q = \textsc{Randomized Partition} \((A, p, r)\)\)
4. \(k = q - p + 1\)
5. \textbf{if} \(i = k\)
6. \quad \textbf{return} \((A[q])\)
7. \textbf{else if} \(i < k\)
8. \quad \textbf{return} \((\textsc{Randomized-Select} \((A, p, q - 1, i))\))
9. \textbf{else return} \((\textsc{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 \textsc{Randomized-Select} \((A, p, r, i)\)
1. \textbf{if} \(p = r\
2. \quad \textbf{return} (A[p])
3. \quad q = \textsc{Randomized Partition} \((A, p, r)\)
4. \quad k = q - p + 1
5. \textbf{if} \(i = k\
6. \quad \textbf{return} (A[q])
7. \textbf{else if} \(i < k\
8. \quad \textbf{return} (\textsc{Randomized-Select} \((A, p, q - 1, i)\))
9. \textbf{else return} (\textsc{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

\[
T(n) \leq \max\{T\left(\frac{n}{r}\right), T\left(\frac{(r-1)n}{r}\right)\} + nc
\]
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 \(r \geq 2\),

\[
T(n) \leq cn\frac{(r-1)}{r} + cn\left(\frac{(r-1)}{r}\right)^2 + cn\left(\frac{(r-1)}{r}\right)^4 + \ldots cn\left(\frac{(r-1)}{r}\right)^m = O(n)
\]

where \(\left(\frac{(r-1)}{r}\right)^m n = 1\), \(m = \log \frac{r}{r-1} n\)
Performance analysis
Chapter 9. Medians and Order Statistics

Performance analysis

   The worst case: running time $\Theta(n^2)$. 
Chapter 9. Medians and Order Statistics

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 $n = r - p + 1$;
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 $n = r - p + 1$;

• the algorithm identifies a pivot and recursively computes on sublist $A[p..q]$ (or $A[q + 1..r]$);
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 $n = r - p + 1$;
- 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

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k-1, n-k\})] + an, \text{ for some constant } a > 0$$
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

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k-1, n-k\})] + an,$$

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 $k \leq n/2$

$$E[T(n)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an$$
Chapter 9. Medians and Order Statistics

We conclude that \( E[T(n)] \leq \frac{2}{n} \sum_{k=1}^{n-1} k \leq \frac{n}{2} E[T(k)] + an \)

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: \( E[T(k)] \leq \frac{2}{n} \sum_{k=1}^{n-1} k = \frac{n}{2} E[T(k)] + an \leq 2c n \left( \frac{n-1}{2} \right) + an = \cdots = 3cn/4 + c/2 + an \leq cn \) when \( (cn/4 - c/2 - an) \geq 0 \).

- Base case: assume \( T(n) \leq cn \), for \( n < 2c/(c-4a) \)
Chapter 9. Medians and Order Statistics

We conclude that \( E[T(n)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an \)
We conclude that $E[T(n)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an$

**Theorem.** $E[T(n)] = O(n)$. 
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: $n =$
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;
Chapter 9. Medians and Order Statistics

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;
- **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: $n = ?$, 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:

$$E[T(k)] \leq 2/n\sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\sum_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an$$
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:**

$E[T(k)] \leq 2/n \sum_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n \sum_{k=n/2}^{n-1} ck + an$

\[
= \frac{2c}{n} \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = \frac{2c}{n} \left[ \frac{n-1}{2} \right] (n) - \frac{n/2-1}{2} \left( n/2 \right) + an
\]

\[
= \cdots =
\]
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: $n = \;\text{?}$, 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 $n \geq 2c/(c - 4a)$
We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

**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:

  $$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

  $$= 2c \left[ \sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right] + an = 2c \left[ \frac{n-1}{2} (n) - \frac{n/2 - 1}{2} (n/2) \right] + an$$

  $$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

  when $(cn/4 - c/2 - an) \geq 0$. That is $n \geq 2c/(c - 4a)$

- **Base case:** assume $T(n) \leq cn$, for $n < 2c/(c - 4a)$
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.

$T(n) \leq T(\beta n) + cn$ where $0 < \beta < 1$.
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:

• 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.

$$T(n) \leq T(\beta n) + cn$$ where $0 < \beta < 1$
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.
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.
- $T(n) \leq T(\beta n) + cn$ where $0 < \beta < 1$
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 $S$;
• the total time actually is $T(n) \leq T(\alpha n) + T(\beta n) + cn$ where $\alpha + \beta < 1$.
Chapter 9. Medians and Order Statistics

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
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 $\alpha n$
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 $\text{SELECT} (S, i); \{ \text{where } S \text{ contains } n \text{ distinct elements} \}$
Algorithm $\text{SELECT} (S, i)$; \{ where $S$ contains $n$ distinct elements\}

(1) divide $S$ into $\lceil n/5 \rceil$ groups of 5 elements

Chapter 9. Medians and Order Statistics
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
(2) sort each group (of 5) and find the median of each group;
Chapter 9. Medians and Order Statistics

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 \( |M| = \lceil n/5 \rceil \)
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⌉);
(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 ∀y ∈ S_1 y < x and ∀z ∈ S_2 z > x
(6) if i < k recursively call `Select(S_1, i)
else recursively call `Select(S_2, i−k)"
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 \(|M| = \lceil n/5 \rceil\)
3. recursively call Select\((M, \lceil n/10 \rceil)\);
   - let the result be \(x\) and let the rank of \(x\) be \(k\) in \(S\)

\(\text{Algorithm Select } (S, i); \{ \text{where } S \text{ contains } n \text{ distinct elements} \}\)

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 Select\((M, \lceil n/10 \rceil)\);
   - let the result be \(x\) and let the rank of \(x\) be \(k\) in \(S\)
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 \textsc{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) \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) \textbf{if} \(i = k\) \textbf{return} \((x)\)
(5) \textbf{else} use \(x\) as the pivot to partition \(S\) resulting in \(S_1\) and \(S_2\),
Chapter 9. Medians and Order Statistics

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) \textbf{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) \textbf{if} \(i = k\) \textbf{return} \((x)\)
(5) \textbf{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 $\lceil n/5 \rceil$ 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** `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 $\forall z \in S_2$ $z > x$
6. if $i < k$ **recursively call** `Select(S_1, i)`
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\((S_1, i)\)
   else recursively call Select\((S_2, i - k)\)
Chapter 9. Medians and Order Statistics

Note: the number of elements $\leq x$ is at least:

$$|S_1| = 3\left(\frac{\lceil n/5 \rceil}{2}\right) \geq 3n/10$$

Similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3\left(\frac{\lceil n/5 \rceil}{2} - 2\right) \geq 3n/10 - 6$$

So a time upper bound for $\text{SELECT}$ is

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 6 \rceil) + O(n)$$

when $n \geq 140$
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:
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

For example, given Insertion Sort:

- we first analyzed the algorithm and obtained $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 n \sum_{j=2}^{t_j} + c_6 n \sum_{j=2}^{t_j} (t_j - 1) + c_7 n \sum_{j=2}^{t_j} (t_j - 1) + c_8 (n - 1)$

- 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 $T(n) = \ldots$
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., $T(n) = O(\ldots)$)

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.
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.
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:
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.

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

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$
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

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1) \]

- 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., $T(n) = O(\cdot)$)
- 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., $T(n) \leq cn^2$;
- 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
Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given **Binary Search** 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
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 $T(n)$

\[
T(n) \leq T(\lfloor n/2 \rfloor) + c
\]
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 \texttt{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;
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
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.
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.
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 $T(n) \leq c \log_2 n$. 

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 $T(n) \leq c \log_2 n$. see previous lecture notes