CSCI 4470/6470 Algorithms, Fall 2019

Liming Cai
Department of Computer Science, UGA
Syllabus: http://cobweb.cs.uga.edu/~cai/courses/algo/2019Fall/

August 22, 2019
An Introduction to the Introduction

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare (1980 Turing Award recipient)

Which way do we take in algorithm design?

Both correctness and efficiency are desired.
An Introduction to the Introduction

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
An Introduction to the Introduction

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare (1980 Turing Award recipient)
An Introduction to the Introduction

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare (1980 Turing Award recipient)

Which way do we take in algorithm design?
There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare (1980 Turing Award recipient)

Which way do we take in algorithm design?

Both correctness and efficiency are desired.
An Introduction to the Introduction

Example 1

Sequence Homology Reveals Functions

- Homology reveals evolution of structure/function
  - FOS_RAT: MMPSGFNADYEASRSSCSSASSPAGDSLSYYHRSPADSFSSMGSPVNTQDFCADLSVSSANF 60
  - FOS_MOUSE: MMPSGFNADYEASRSSCSSASSPAGDSLSYYHRSPADSFSSMGSPVNTQDFCADLSVSSANF 60
  - FOS_CHICK: MMYQGFAGEYEAPSSRSSCSSASSPAGDSLTYYPSADSFSSMGSPVNSQDFCTDLAVSSANF 60
  - FOSB_MOUSE: -MFQAPFGDYDS-GRSCS-SPSAESQ--YLSSVDSFGSPPTAAASOE-CAGLGEQMPGSF 54
  - FOSB_HUMAN: -MFQAPFGDYDS-GRSCS-SPSAESQ--YLSSVDSFGPPTAAASOE-CAGLGEQMPGSF 54

- Homology reveals regulatory structure (E. Coli promoters)
  - tyr RNA: TCTCAACGTAACACTTTTGACTACGCGGCG + CGTATCTTGTATATACGCCCTTCCTTCCGATAAGGS
  - rm D1: GATCAAAAAATATCTTGCGAAAATA + TTGGGATCTTCTATATAGCGCTCGCTTGGAGACACCAAG
  - rm X1: ATGCTTTCTCCTTGTCTTCCTGA + GCGCGCTTCTATATAGCGCTCGCTTGGAGACACCAAG
  - rm D2: CCTGAATTCACGCTTTTGTACCTTGAAA + GAGGAAGACGCTATAACGCGACCTCGGAACGTGAC
  - rm E1: TCCAATTTTTATCTTTGATATAGCGCTCGCTTGGAGACACCAAG
  - rm A1: TTTATATCTTTCTTTGATATAGCGCTCGCTTGGAGACACCAAG
  - rm A2: GCAAAATTTTTATCTTTGATATAGCGCTCGCTTGGAGACACCAAG
  - θI: TARACTGCTGCTTGTACTTTTA + CCTTTGGCTTGTATATAGCGCTCGCTTGGAGACACCAAG
  - ΔF1: TATCTGTGCCTGTGCTTGTATATAGCGCTCGCTTGGAGACACCAAG
  - T7 A3: GTCGTCACGAAACACAGGTCAACAGCGACTAGT + AGTAAAAGACGCTAGTATCTAAGAGCTAGGAT
  - T7 A1: TATCAAAGAAGATACTTTTGATATAGCGCTCGCTTGGAGACACCAAG
  - T7 A2: AGCAGAAACACAGGTCAACAGCGACTAGT + AGTAAAAGACGCTAGTATCTAAGAGCTAGGAT
  - Fd VIII: GATACGAAACACAGGTCAACAGCGACTAGT + AGTAAAAGACGCTAGTATCTAAGAGCTAGGAT
An Introduction to the Introduction

Example 1

called **Multiple Sequence Alignment.**
Example 1

Problem MULTIPLE SEQUENCE ALIGNMENT:
An Introduction to the Introduction

Example 1

Problem **MULTIPLE SEQUENCE ALIGNMENT:**

Input: \( k \) sequences, each of length \( \approx n \);
Example 1

Problem **Multiple Sequence Alignment**:

Input: $k$ sequences, each of length $\approx n$;
Output: a biologically most “plausible” alignment
An Introduction to the Introduction

Example 1

Problem **Multiple Sequence Alignment:**

Input: \( k \) sequences, each of length \( \approx n \);
Output: a biologically most “plausible” alignment

- e.g., for \( k = 2 \)

```
FOS_CHICK    MMYQGFAGEYEAPSSRCASAPAGDSLTYYPSPADSFSSMGSPVNSQDFCTDLAVSSANF 60
FOSB_MOUSE   -RFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAAASQE-CAGLGEMPGSF 54
```
An Introduction to the Introduction

Example 1

Problem Multiple Sequence Alignment:

Input: $k$ sequences, each of length $\approx n$;
Output: a biologically most “plausible” alignment

• e.g., for $k = 2$

```
FOS_CHICK  MMYQGFAGEYEAPSSRCSSASPAGDSLTYYPSPADSFSMGSVPVNSQDFCTDLAVSSANF 60
FOSB_MOUSE  MFQAFPGDYDS-GSRCSS-SPSAEQ--YLSSVDSFGSPPTAAASQ--CAGLEMPUGSF 54
```

editing through substitutions, insertions, deletions
Example 1

Problem MULTIPLE SEQUENCE ALIGNMENT:

Input: $k$ sequences, each of length $\approx n$;
Output: a biologically most “plausible” alignment

- e.g., for $k = 2$

```
<table>
<thead>
<tr>
<th>FOS_CHICK</th>
<th>MMYQGFAGEYEAPSSRCSSASPAGDSTTYYPSPADSFSSMSGPVNSQDFCTDIAVSSANF 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOSB_MOUSE</td>
<td>MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSPSPPTAASQEE-CAGLEMPFGSF 54</td>
</tr>
</tbody>
</table>
```

editing through substitutions, insertions, deletions

possible alignments $\geq 2^n$, naive methods may not be efficient
Example 1

Problem **Multiple Sequence Alignment:**

Input: $k$ sequences, each of length $\approx n$;
Output: a biologically most “plausible” alignment

- e.g., for $k = 2$

```
FOS_CHICK  MMYGFAEYAEAGSSPAGSLTYYPSPADSFSMSPVNSQDFCLDLAVSSANF 60
FOSB_MOUSE  -MFQAFPGDYDS-GSRCS-SPSAEQ--YLSSVDSFGSPPTAASQ-E-CGLGEMPGSF 54
```

editing through substitutions, insertions, deletions

possible alignments $\geq 2^n$, naive methods may not be efficient

Algorithm design goal 1:
An Introduction to the Introduction

Example 1

Problem Multiple Sequence Alignment:

Input: $k$ sequences, each of length $\approx n$;
Output: a biologically most “plausible” alignment

• e.g., for $k = 2$

| FOS_CHICK | MMYQGFAGEYEAPSSRCSSASPAGDSLTYYPSPADSFSSMGSPVNSQDFCDTDLSVSSANF 60 |
| FOSB_MOUSE | -MFQAFPGDYDS-GSRCSS-SPSAESEQ--YLSSVDSFGSPPTAASQ-E-CAGLGMPSF 54 |

editing through substitutions, insertions, deletions

possible alignments $\geq 2^n$, naive methods may not be efficient

Algorithm design goal 1:

To craft algorithms as efficient as possible
An Introduction to the Introduction

Example 1

Problem **Multiple Sequence Alignment:**

Input: \( k \) sequences, each of length \( \approx n \);
Output: a biologically most “plausible” alignment

• e.g., for \( k = 2 \)

\[
\begin{array}{c}
	\text{FOS_CHICK: MMYQGFA}G\text{EYPSSRCSSASPAGDSLTYYPSPADSFSSMGSPVNSQDFCTDLAVSSANF 60} \\
	\text{FOSB_MOUSE: -MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAASQE-CAGLGMEMPGSF 54}
\end{array}
\]

editing through substitutions, insertions, deletions

possible alignments \( \geq 2^n \), naive methods may not be efficient

Algorithm design goal 1:

To craft algorithms as efficient as possible
(to develop skills more than naive ideas!)
An Introduction to the Introduction

Example 2
An Introduction to the Introduction

Example 2
An Introduction to the Introduction
An Introduction to the Introduction

- 128 players in total, single elimination
- How many matches are played to determine a champion?
- Is it possible to just play fewer matches?
An Introduction to the Introduction

- 128 players in total, single elimination
• 128 players in total, single elimination
• How many matches are played to determine a champion?
128 players in total, single elimination

How many matches are played to determine a champion? 127
• 128 players in total, single elimination
• How many matches are played to determine a champion? 127
• Is it possible to just play fewer matches?
An Introduction to the Introduction

- 7 matches is sufficient to determine the champion.
- What does it mean?
  - The specific arrangement of 7 matches finds the champion.
- Are 7 matches necessary to determine the champion?
  - ≤ 6 matches, regardless arrangement, cannot find a champion.

```
First Round    Semi-finals    Final    Winner

1  Abbott
2  Bolton
3  Cross
4  Dent
5  Elphick
6  Flower
7  Gough
8  Handy
```

Winner: Abbott
7 matches is sufficient to determine the champion.
• 7 matches is **sufficient** to determine the champion.

What does it mean?
An Introduction to the Introduction

- 7 matches is **sufficient** to determine the champion.

**What does it mean?**

The specific arrangement of 7 matches finds the champion.
An Introduction to the Introduction

- 7 matches is **sufficient** to determine the champion.

**What does it mean?**

The specific arrangement of 7 matches finds the champion.

- Are 7 matches **necessary** to determine the champion?
An Introduction to the Introduction

- 7 matches is **sufficient** to determine the champion.

**What does it mean?**

The specific arrangement of 7 matches finds the champion.

- Are 7 matches **necessary** to determine the champion?

**What does that mean?**
An Introduction to the Introduction

• 7 matches is sufficient to determine the champion.

What does it mean?
The specific arrangement of 7 matches finds the champion.

• Are 7 matches necessary to determine the champion?

What does that mean?
≤ 6 matches, regardless arrangement, cannot find a champion.
An Introduction to the Introduction

Problem **Find Max:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ (1 $\leq i \leq n$), such that for $j = 1, 2, \ldots, n$, $x_i \geq x_j$.

The problem can be solved with $n - 1$ comparisons.

There is a way to find the maximum using $n - 1$ comparisons.

Regardless how to compare these numbers.
Problem **Find Max:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Problem **Find Max:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$
Problem **FIND MAX**: 

Input: a set of $n$ numbers $x_1, \ldots, x_n$;  
Output: $x_i$, for some $i$ (1 ≤ $i$ ≤ $n$), such that for $j = 1, 2, \ldots, n$,  

$$x_i \geq x_j$$

- The problem can be solved with $n - 1$ comparisons.
An Introduction to the Introduction

Problem **Find Max:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$

- The problem can be solved with $n - 1$ comparisons.
  There is a way to find the maximum using $n - 1$ comparisons.
An Introduction to the Introduction

Problem FIND MAX:

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

\[ x_i \geq x_j \]

- The problem can be solved with $n - 1$ comparisons.
  There is a way to find the maximum using $n - 1$ comparisons.
- The problem requires at least $n - 1$ comparisons.
Problem **Find Max:**

- **Input:** a set of \( n \) numbers \( x_1, \ldots, x_n \);
- **Output:** \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),

\[
x_i \geq x_j
\]

- The problem can be solved with \( n - 1 \) comparisons.
  There is a way to find the maximum using \( n - 1 \) comparisons.

- The problem requires at least \( n - 1 \) comparisons.
  Regardless how to compare these numbers.
An Introduction to the Introduction

Problem **Find Max**:

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$
An Introduction to the Introduction

Problem **FIND MAX:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$

- $n - 1$ comparisons are **sufficient** to find the maximum called an upper bound;
An Introduction to the Introduction

Problem **Find Max**:

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),

\[
    x_i \geq x_j
\]

• \( n - 1 \) comparisons are **sufficient** to find the maximum called an upper bound;

You only need to show an **algorithm** to show an upper bound;
Problem **Find Max**: 

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);

Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),

\[
x_i \geq x_j
\]

- \( n - 1 \) comparisons are **sufficient** to find the maximum called an **upper bound**;

You only need to show an **algorithm** to show an upper bound;

- \( n - 1 \) comparisons are **necessary** to find the maximum called a **lower bound**;
Problem **FIND MAX**:

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),

\[
x_i \geq x_j
\]

- \( n - 1 \) comparisons are **sufficient** to find the maximum called an upper bound;

You only need to show an **algorithm** to show an upper bound;

- \( n - 1 \) comparisons are **necessary** to find the maximum called a lower bound;

you need a (mathematical) **proof** to show a lower bound.
An Introduction to the Introduction

Problem **Find Max:**

- Input: a set of $n$ numbers $x_1, \ldots, x_n$;
- Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$, $x_i \geq x_j$
An Introduction to the Introduction

Problem \texttt{Find Max}:

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$

- Is $n + 3$ an upper bound for problem \texttt{Find Max}?
Problem \textsc{Find Max}:

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),
\[ x_i \geq x_j \]

\begin{itemize}
  \item Is \( n + 3 \) an upper bound for problem \textsc{Find Max}?
  \item Is \( n + 3 \) a lower bound for the problem?
\end{itemize}
Problem \textbf{Find Max}:

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),
\[ x_i \geq x_j \]

- Is \( n + 3 \) an upper bound for problem \textbf{Find Max}? 
- Is \( n + 3 \) a lower bound for the problem?
- Is \( n - 3 \) an upper bound? How about a lower bound?
An Introduction to the Introduction

Problem FIND MAX:

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \);
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \),

\[ x_i \geq x_j \]

• Is \( n + 3 \) an upper bound for problem FIND MAX?
• Is \( n + 3 \) a lower bound for the problem?
• Is \( n - 3 \) an upper bound? How about a lower bound?

Algorithm design goal 2:
An Introduction to the Introduction

Problem **Find Max**: 

Input: a set of \( n \) numbers \( x_1, \ldots, x_n \); 
Output: \( x_i \), for some \( i \) (\( 1 \leq i \leq n \)), such that for \( j = 1, 2, \ldots, n \), 
\[ x_i \geq x_j \]

- Is \( n + 3 \) an upper bound for problem **Find Max**?
- Is \( n + 3 \) a lower bound for the problem?
- Is \( n - 3 \) an upper bound? How about a lower bound?

Algorithm design goal 2:

To understand challenges posed by a problem to be solved
An Introduction to the Introduction

Problem **Find Max:**

Input: a set of $n$ numbers $x_1, \ldots, x_n$;
Output: $x_i$, for some $i$ ($1 \leq i \leq n$), such that for $j = 1, 2, \ldots, n$,

$$x_i \geq x_j$$

- Is $n + 3$ an upper bound for problem **Find Max**?
- Is $n + 3$ a lower bound for the problem?
- Is $n - 3$ an upper bound? How about a lower bound?

**Algorithm design goal 2:**

To understand challenges posed by a problem to be solved (need to develop capability to observe and analyze!)
Example 3 Toss coin over the phone
An Introduction to the Introduction

Example 3 Toss coin over the phone
• Two distant cities toss coin to decide who to a football match;
An Introduction to the Introduction

Example 3 Toss coin over the phone

- Two distant cities toss coin to decide who to a football match;
- Lack of visual communication tools, coin toss is done via phone calls;
An Introduction to the Introduction

Example 3 Toss coin over the phone

- Two distant cities toss coin to decide who to a football match;
- Lack of visual communication tools, coin toss is done via phone calls;
Example 3 Toss coin over the phone

- Two distant cities toss coin to decide who to a football match;
- Lack of visual communication tools, coin toss is done via phone calls;

- How would it work?
An Introduction to the Introduction

They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with a binary string $Y$, the output produced from some binary string input $X$ (but the input string $X$ is not given to B);
- Then B guesses if the number of 0's in $X$ is odd or even;
They decided the following protocol:

- A told B a huge Boolean circuit through the phone;
An Introduction to the Introduction

They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with a
They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with a binary string $Y$, the output produced from some binary string input $X$ (but the input string $X$ is not given to $B$);
They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with a binary string $Y$, the output produced from some binary string input $X$ (but the input string $X$ is not given to $B$);
- Then B guesses if the number of 0’s in $X$ is odd or even;
From $Y$, it would be very difficult for $B$ to figure out which $X$ is used to produce $Y$ (one-way function, an intractable problem); so the best $B$ can do is to randomly guess (odd/even), which has the same effect as tossing a coin.

Algorithm design goal 3: To understand inherent difficulties of some prominent problems (need to study math underlying the intractability)
An Introduction to the Introduction

- From $Y$, it would be very difficult for $B$ to figure out which $X$ is $X$ used to produce $Y$ (one-way function, an intractable problem);
• From $Y$, it would be very difficult for $B$ to figure out which $X$ is $X$ used to produce $Y$ (one-way function, an intractable problem);

• So the best $B$ can do is to randomly guess (odd/even), which has the same effect as tossing a coin.
• From \( Y \), it would be very difficult for \( B \) to figure out which \( X \) is \( X \) used to produce \( Y \) (one-way function, an intractable problem);

• So the best \( B \) can do is to randomly guess (odd/even), which has the same effect as tossing a coin.

**Algorithm design goal 3:**
An Introduction to the Introduction

- From $Y$, it would be very difficult for $B$ to figure out which $X$ is $X$ used to produce $Y$ (one-way function, an intractable problem);
- So the best $B$ can do is to randomly guess (odd/even), which has the same effect as tossing a coin.

**Algorithm design goal 3:**

To understand inherent difficulties of some prominent problems
• From \( Y \), it would be very difficult for \( B \) to figure out which \( X \) is used to produce \( Y \) (one-way function, an \textit{intractable problem});

• So the best \( B \) can do is to randomly guess (odd/even), which has the same effect as tossing a coin.

\underline{Algorithm design goal 3:}

To understand inherent difficulties of some prominent problems (need to study math underlying the intractability)
An Introduction to the Introduction

To design good algorithms for computational problems,
An Introduction to the Introduction

To design good algorithms for computational problems,
• need to be familiar with techniques for complexity analysis;
An Introduction to the Introduction

To design good algorithms for computational problems,

- need to be familiar with techniques for complexity analysis;
- need to master skills for efficient algorithm design;
An Introduction to the Introduction

To design good algorithms for computational problems,

• need to be familiar with techniques for complexity analysis;
• need to master skills for efficient algorithm design;
• need to be able to know if an algorithm is optimal (how?).
The Introduction

What is this course about (and why is it needed)?
• About basic yet indispensable skills for problem solving
• Algorithm design leads to writing code, i.e., creative thinking without programming languages

How different is this course from other algorithm courses?
• Design technique-oriented, not application-oriented
• Emphasis on guaranteed performance (typically in efficiency)

Goals to achieve
• To learn to measure performance of algorithms
• To master some fundamental algorithmic techniques
• To study advanced algorithmic skills
• To understand computational intractability
The Introduction

▷ What is this course about (and why is it needed)?
The Introduction

► What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
The Introduction

- What is this course about (and why is it needed)?
  - about basic yet indispensable skills for problem solving
  - algorithm design leads to writing code,
The Introduction

▶ What is this course about (and why is it needed)?

- about basic yet indispensable skills for problem solving
- algorithm design leads to writing code,
  i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?

- design technique-oriented, not application-oriented
- emphasis on guaranteed performance (typically in efficiency)

▶ Goals to achieve

- to learn to measure performance of algorithms
- to master some fundamental algorithmic techniques
- to study advanced algorithmic skills
- to understand computational intractability
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
  • emphasis on guaranteed performance (typically in efficiency)
The Introduction

- **What is this course about (and why is it needed)?**
  - about basic yet indispensable skills for problem solving
  - algorithm design leads to writing code,
    i.e., creative thinking without programming languages

- **How different is this course from other algorithm courses?**
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)

- **Goals to achieve**
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
  • emphasis on guaranteed performance (typically in efficiency)

▶ Goals to achieve
  • to learn to measure performance of algorithms
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
  • emphasis on guaranteed performance (typically in efficiency)

▶ Goals to achieve
  • to learn to measure performance of algorithms
  • to master some fundamental algorithmic techniques
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
  • emphasis on guaranteed performance (typically in efficiency)

▶ Goals to achieve
  • to learn to measure performance of algorithms
  • to master some fundamental algorithmic techniques
  • to study advanced algorithmic skills
The Introduction

▶ What is this course about (and why is it needed)?
  • about basic yet indispensable skills for problem solving
  • algorithm design leads to writing code,
    i.e., creative thinking without programming languages

▶ How different is this course from other algorithm courses?
  • design technique-oriented, not application-oriented
  • emphasis on guaranteed performance (typically in efficiency)

▶ Goals to achieve
  • to learn to measure performance of algorithms
  • to master some fundamental algorithmic techniques
  • to study advanced algorithmic skills
  • to understand computational intractability
Part I. Foundations
Part I. Foundations

- Chapter 1. The role of algorithms in computing
- Chapter 2. Getting started
- Chapter 3. Growth of functions
- Chapter 4. Solving recurrences
- Chapter 5. Probabilistic analysis and randomized algorithms
Part I. Foundations

The theme of the course
Part I. Foundations

The theme of the course

• Goal: learning techniques to design efficient algorithms
Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms
- mean: through developing skills to analyze algorithms
Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms
- mean: through developing skills to analyze algorithms

Design and analysis of algorithms are closely related.
Part I. Foundations

Example: the Fibonacci sequence.

\[ f(n) = \begin{cases} 
  f(n - 1) + f(n - 2) & \text{if } n \geq 3 \\
  1, & \text{otherwise}
\end{cases} \]
Example: the Fibonacci sequence.

\[ f(n) = \begin{cases} 
  f(n - 1) + f(n - 2) & \text{if } n \geq 3 \\
  1, & \text{otherwise}
\end{cases} \]

That is:

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>...</td>
</tr>
</tbody>
</table>
Part I. Foundations

Problem 1: Computing the $n$th Fibonacci number:
Problem 1: Computing the $n$th Fibonacci number:

**Input:** $n \geq 1$;
**Output:** the $n$th number in the Fibonacci sequence.
Problem 1: Computing the $n$th Fibonacci number:

**Input:** $n \geq 1$;

**Output:** the $n$th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*
Part I. Foundations

Problem 1: Computing the $n$th Fibonacci number:

**Input:** $n \geq 1$;
**Output:** the $n$th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*

- *recursive*: task decomposition, top-down, recursive calls;
Problem 1: Computing the $n$th Fibonacci number:

**Input:** $n \geq 1$;

**Output:** the $n$th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*

- *recursive*: task decomposition, top-down, recursive calls;
- *iterative*: more tightly coupled tasks, bottom-up approaches;
Part I. Foundations

Rec-Fibonacci(n)

But how efficient is it? Or how slow is it? Its execution is via a run-time stack:

- suitable for execution of subroutines
- but oblivious, cannot remember any completed subroutine.
Part I. Foundations

Rec-Fibonacci($n$)

if $n = 1$ or $n = 2$,

But how efficient is it? Or how slow is it? Its execution is via a run-time stack

• suitable for execution of subroutines
• but oblivious, cannot remember any completed subroutine.
Rec-Fibonacci \( (n) \)

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

if \( n = 1 \) or \( n = 2 \), return (1);
else
    \( T_1 = \) Rec-Fibonacci(n - 1);
Rec-Fibonacci($n$)

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

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

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

return ($T_1 + T_2$);
Rec-Fibonacci\( (n) \)

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

\text{else}

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

\text{return } (T_1 + T_2);

But how efficient is it? Or how slow is it?
Rec-Fibonacci(n)

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

But how efficient is it? Or how slow is it?
its execution is via a *run-time stack*
Part I. Foundations

\[ \text{Rec-Fibonacci}(n) \]

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

But how efficient is it? Or how slow is it?

its execution is via a \textit{run-time stack}

- suitable for execution of subroutines
Part I. Foundations

Rec-Fibonacci\((n)\)

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

But how efficient is it? Or how slow is it?

its execution is via a *run-time stack*

- suitable for execution of subroutines
- but oblivious, cannot remember any completed subroutine.
Part I. Foundations

**Rec-Fibonacci**

```plaintext
if n = 1 or n = 2, return 1;
else
T1 = Rec-Fibonacci(n - 1);
T2 = Rec-Fibonacci(n - 2);
return (T1 + T2);
```

**Rec-Fibonacci**

```plaintext
if n = 1 or n = 2, return 1;
else
T1 = Rec-Fibonacci(n - 1);
T2 = Rec-Fibonacci(n - 2);
return (T1 + T2);
```
Part I. Foundations

Repeated computations everywhere!
The size of tree is the number of recursive calls;
Part I. Foundations

The size of tree is the number of recursive calls; How big is it?
Part I. Foundations

\[
\begin{align*}
\text{small triangle} & \leq \text{size of tree} & \leq \text{large triangle} \\
\text{roughly: } & 2^n & \leq \text{size of tree} & \leq 2^{n+1}
\end{align*}
\]
Part I. Foundations

small triangle $\leq$ size of tree $\leq$ large triangle
Part I. Foundations

small triangle $\leq$ size of tree $\leq$ large triangle

roughly: $2^{\frac{n}{2}} \leq$ size of tree $\leq 2^n$
Iterative-Fibonacci($n$)
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1);
Part I. Foundations

Iterative-Fibonacci\((n)\)

if \(n = 1\) or \(n = 2\) return \((1)\);
else
  \(M[1] = 1, M[2] = 1\)
Part I. Foundations

**Iterative-Fibonacci** \((n)\)

```
if \(n = 1\) or \(n = 2\) return \((1)\);
else
  \(M[1] = 1, M[2] = 1\)
  for \(i = 3\) to \(n\) do
    \(M[i] = M[i - 1] + M[i - 2]\)
```

How fast is it?

\(T_{\text{total}} = \max\{T_{\text{if}}, T_{\text{else}}\}\)

where
\(T_{\text{if}} = c_1\), \(T_{\text{else}} = c_2 + T_{\text{for}} = c_2 + d \times (n - 2)\)

\(T_{\text{total}} \leq c_1 + c_2 + d \times (n - 2)\), a linear function in \(n\)
Part I. Foundations

Iterative-Fibonacci\((n)\)

\[
\text{if } n = 1 \text{ or } n = 2 \text{ return } (1); \\
\text{else} \\
\quad M[1] = 1, \ M[2] = 1 \\
\quad \text{for } i = 3 \text{ to } n \text{ do} \\
\quad \quad M[i] = M[i - 1] + M[i - 2] \\
\text{return } (M[n])
\]
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1);
else
    $M[1] = 1, M[2] = 1$
    for $i = 3$ to $n$ do
        $M[i] = M[i - 1] + M[i - 2]$
    return ($M[n]$)

How fast is it?
Part I. Foundations

**Iterative-Fibonacci**\((n)\)

\[
\begin{align*}
\text{if } n &= 1 \text{ or } n = 2 \text{ return } (1); \\
\text{else} \\
&\text{for } i = 3 \text{ to } n \text{ do} \\
&M[i] = M[i - 1] + M[i - 2] \\
&\text{return } (M[n])
\end{align*}
\]

How fast is it?

\[
T_{total} = \max\{T_{if}, T_{else}\}
\]
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1); 
else 
    $M[1] = 1, M[2] = 1$
    for $i = 3$ to $n$ do 
        $M[i] = M[i - 1] + M[i - 2]$
    return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, 

Iterative-Fibonacci(n)

if $n = 1$ or $n = 2$ return (1);
else
    $M[1] = 1, M[2] = 1$
    for $i = 3$ to $n$ do
        $M[i] = M[i - 1] + M[i - 2]$
    return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for}$
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1);
else
  $M[1] = 1, M[2] = 1$
  for $i = 3$ to $n$ do
    $M[i] = M[i - 1] + M[i - 2]$
  return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1, \ T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1);
else
  $M[1] = 1, M[2] = 1$
  for $i = 3$ to $n$ do
    $M[i] = M[i - 1] + M[i - 2]$
  return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$$T_{total} \leq c_1$$
Iterative-Fibonacci($n$)

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

$M[1] = 1$, $M[2] = 1$

for $i = 3$ to $n$ do

$M[i] = M[i - 1] + M[i - 2]$

return $(M[n])$

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$$T_{total} \leq c_1 + c_2 + d(n - 2),$$
Iterative-Fibonacci\((n)\)

\[
\text{if } n = 1 \text{ or } n = 2 \text{ return } (1); \\
\text{else} \\
M[1] = 1, \ M[2] = 1 \\
\text{for } i = 3 \text{ to } n \text{ do} \\
M[i] = M[i - 1] + M[i - 2] \\
\text{return } (M[n])
\]

How fast is it?

\[
T_{\text{total}} = \max\{T_{\text{if}}, T_{\text{else}}\}
\]

where \(T_{\text{if}} = c_1, \ T_{\text{else}} = c_2 + T_{\text{for}} = c_2 + d \times (n - 2)\)

\[
T_{\text{total}} \leq c_1 + c_2 + d(n - 2), \text{ a linear function in } n
\]
Iterative-Fibonacci(n)

if $n = 1$ or $n = 2$ return (1);
else
  $M[1] = 1$, $M[2] = 1$
  for $i = 3$ to $n$ do
    $M[i] = M[i - 1] + M[i - 2]$
  return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$$T_{total} \leq c_1 + c_2 + d(n - 2), \text{ a linear function in } n$$

Iterative-Fibonacci(n) is a simple dynamic programming algorithm.
Actually, all the algorithm \textsc{Iterative-Fibonacci}(n) does is:
Actually, all the algorithm $\text{ITERATIVE-FIBONACCI}(n)$ does is:

To fill out a table of size $n$, with
Part I. Foundations

Actually, all the algorithm \textsc{Iterative-Fibonacci}(n) does is:

To fill out a table of size $n$, with

- each entry being filled out exactly once, and
Part I. Foundations

Actually, all the algorithm \textsc{Iterative-Fibonacci}(n) does is:

To fill out a table of size \( n \), with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say \( c \) steps.
Part I. Foundations

Actually, all the algorithm \textsc{Iterative-Fibonacci}(n) does is:

To fill out a table of size $n$, with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say $c$ steps.

So the total time \textsc{Iterative-Fibonacci}(n) uses is
Actually, all the algorithm \texttt{ITERATIVE-FIBONACCI}(n) does is:

To fill out a table of size \( n \), with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say \( c \) steps.

So the total time \texttt{ITERATIVE-FIBONACCI}(n) uses is

\[
T(n) = c \times n
\]
Chapter 1. The role of algorithms in computing
Chapter 1. The role of algorithms in computing

What is an Algorithm: a well-defined, finite procedure that takes an input and produces an output.
Chapter 1. The role of algorithms in computing

What is an Algorithm: a well-defined, finite procedure that takes an input and produces an output.

Example 2: An algorithm skeleton;

Algorithm Maximum;

Input: list $X = \{a_1, \cdots, a_n\}$;

Body that is a series of instructions;

Output: $y$, the maximum of $a_1, \cdots, a_n$. 

Chapter 1. The Role of Algorithms in Computing

Alternatively, an algorithm specifies a finite process to compute a function or a relation.
Alternatively, an algorithm specifies a finite process to compute a function or a relation.

e.g., algorithm Maximum computes the following function:

\[ f_{\text{max}}(X) = y, \text{ where } \forall a \in X, y \geq a, \]
Chapter 1. The Role of Algorithms in Computing

Alternatively, an algorithm specifies a finite process to compute a function or a relation.

e.g., algorithm $\text{MAXIMUM}$ computes the following function:

$$f_{\text{max}}(X) = y, \text{ where } \forall a \in X, y \geq a,$$

For some problems, the functions computed are predicates, i.e., output $y \in \{\text{TRUE, FALSE}\}$.
Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues
Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.
Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:
Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:

- very large input data for “easy” problems;
Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:

• very large input data for “easy” problems;
• moderately large input data for “hard” problems.
The Sorting Problem

Input: \(n\) numbers \(\langle a_1, \cdots, a_n \rangle\);

Output: a reordering \(\langle a'_1, \cdots, a'_n \rangle\) of the input such that \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\).

**Insertion Sort**

Idea: an iterative process to produce a new list such that at each iteration, the new list consists of two sublists,

- a sorted sublist followed by an unsorted sublist,
- the leftmost number of the unsorted is being inserted into the sorted.

As the process goes, the sorted sublist gets longer, the unsorted sublist gets shorter, until the unsorted becomes empty.
Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: $n$ numbers $\langle a_1, \cdots, a_n \rangle$;
Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

**Input:** $n$ numbers $\langle a_1, \cdots, a_n \rangle$;

**Output:** a reordering $\langle a'_1, \cdots, a'_n \rangle$ of the input such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$. 

**Insertion Sort**

Idea: an iterative process to produce a new list such that at each iteration, the new list consists of two sublists,

- a sorted sublist followed by an unsorted sublist,
- the leftmost number of the unsorted is being inserted into the sorted.

As the process goes, the sorted sublist gets longer, the unsorted sublist gets shorter, until the unsorted becomes empty.
Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

**Input:** \( n \) numbers \( \langle a_1, \cdots, a_n \rangle \);

**Output:** a reordering \( \langle a'_1, \cdots, a'_n \rangle \) of the input such that
\[
a'_1 \leq a'_2 \leq \cdots \leq a'_n.
\]

Insertion Sort

**idea:** an iterative process to produce a new list such that
Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

**Input:** \( n \) numbers \( \langle a_1, \cdots, a_n \rangle \);

**Output:** a reordering \( \langle a'_1, \cdots, a'_n \rangle \) of the input such that
\[
a'_1 \leq a'_2 \leq \cdots \leq a'_n.
\]

Insertion Sort

**Idea:** an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,
Chapter 2. Getting started

The Sorting Problem

**INPUT:** $n$ numbers $\langle a_1, \cdots, a_n \rangle$;

**OUTPUT:** a reordering $\langle a'_1, \cdots, a'_n \rangle$ of the input such that

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n.$$

Insertion Sort

**idea:** an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,

- a **sorted sublist** followed by an **unsorted sublist**, and
Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: $n$ numbers $\langle a_1, \ldots, a_n \rangle$;
OUTPUT: a reordering $\langle a'_1, \ldots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \cdots \leq a'_n.$$ 

Insertion Sort

idea: an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,

- a sorted sublist followed by an unsorted sublist, and
- the leftmost number of the unsorted is being inserted into the sorted.
Chapter 2. Getting started

The Sorting Problem

**Input:** \( n \) numbers \( \langle a_1, \cdots, a_n \rangle \);

**Output:** a reordering \( \langle a'_1, \cdots, a'_n \rangle \) of the input such that

\[ a'_1 \leq a'_2 \leq \cdots \leq a'_n. \]

**Insertion Sort**

**Idea:** an iterative process to produce a new list such that at each iteration, the new list consists of two sublists,

- a **sorted sublist** followed by an **unsorted sublist**, and
- the leftmost number of the **unsorted** is being inserted into the **sorted**.

As the process goes, the **sorted sublist** gets longer, the **unsorted sublist** gets shorter, until the **unsorted** becomes empty.
Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

for j = 2 to length[A]

key = A[j]

{Insert A[j] into sorted A[1..j-1]}

i = j - 1

while i > 0 and A[i] > key

A[i+1] = A[i]

i = i - 1

A[i+1] = key

Analysis of the algorithm:
• (correctness proof): to show that the algorithm is as desired;
• (efficiency proof): to show a guaranteed efficiency of the algorithm.
Algorithm INSERTION-SORT($A$)

1. for $j = 2$ to length[$A$] do
Chapter 2. Getting Started

Algorithm Insertion-Sort($A$)

1. for $j = 2$ to $\text{length}[A]$ do
2.     $key = A[j]$
Chapter 2. Getting Started

Algorithm Insertion-Sort($A$)

1. for $j = 2$ to $\text{length}[A]$ do
2.     $key = A[j]$
Algorithm **INSERTION-SORT**(A)

1. for $j = 2$ to $\text{length}[A]$ do
2.     $key = A[j]$
4.     $i = j - 1$
Algorithm Insertion-Sort($A$)

1. for $j = 2$ to length[$A$] do
2.   key = $A[j]$
4.   $i = j - 1$
5. while $i > 0$ and $A[i] > key$
Algorithm \textsc{Insertion-Sort}(A)

1. \textbf{for} \( j = 2 \) \textbf{to} \( \text{length}[A] \) \textbf{do}
2. \hspace{1em} \textit{key} = A[j]
3. \hspace{1em} \{ \text{Insert } A[j] \text{ into sorted } A[1..j-1] \}
4. \hspace{1em} \textit{i} = j - 1
5. \hspace{1em} \textbf{while} \( i > 0 \) \textbf{and} \( A[i] > \textit{key} \)
6. \hspace{1.5em} \textbf{do} \( A[i+1] = A[i] \)
Algorithm Insertion-Sort($A$)

1. for $j = 2$ to length[$A$] do
2. \hspace{1em} key = $A[j]$
3. \hspace{1em} \{Insert $A[j]$ into sorted $A[1..j - 1]$\}
4. \hspace{1em} $i = j - 1$
5. while $i > 0$ and $A[i] > key$
6. \hspace{1em} do $A[i + 1] = A[i]$
7. \hspace{1em} \hspace{1em} \hspace{1em} $i = i - 1$

Analysis of the algorithm:
• (correctness proof): to show that the algorithm is as desired;
• (efficiency proof): to show a guaranteed efficiency of the algorithm.
Algorithm **INSERTION-SORT**(\(A\))

1. \textbf{for} \(j = 2\) to \(\text{length}[A]\) \textbf{do}
2. \hskip 0.5cm \textit{key} = \(A[j]\)
3. \{Insert \(A[j]\) into sorted \(A[1..j - 1]\)\}
4. \(i = j - 1\)
5. \textbf{while} \(i > 0\) \textbf{and} \(A[i] > \textit{key}\) \textbf{do}
6. \hskip 0.5cm \(A[i + 1] = A[i]\)
7. \hskip 0.5cm \(i = i - 1\)
8. \(A[i + 1] = \textit{key}\)
Algorithm Insertion-Sort($A$)

1. for $j = 2$ to $\text{length}[A]$ do
2.   $key = A[j]$
4.   $i = j - 1$
5. while $i > 0$ and $A[i] > key$ do
7.   $i = i - 1$
8. $A[i + 1] = key$

Analysis of the algorithm:
Chapter 2. Getting Started

Algorithm **INSERTION-SORT**(*A*)

1. **for** *j* = 2 to *length*[*A*] **do**
2. \hspace{1em} **key** = *A*[*j*]
3. \hspace{1em} \{Insert *A*[*j*] into sorted *A*[1..*j* − 1]\}
4. \hspace{1em} *i* = *j* − 1
5. **while** *i* > 0 and *A*[*i*] > **key**
6. \hspace{1em} **do** *A*[*i* + 1] = *A*[*i*]
7. \hspace{1em} \hspace{1em} *i* = *i* − 1
8. \hspace{1em} *A*[*i* + 1] = **key**

Analysis of the algorithm:

- (correctness proof): to show that the algorithm is as desired;
Algorithm Insertion-Sort($A$)

1. for $j = 2$ to $\text{length}[A]$ do
2. \hspace{1em} $key = A[j]$ \hspace{1em} \{Insert $A[j]$ into sorted $A[1..j-1]$\}
3. \hspace{1em} $i = j - 1$
4. \hspace{1em} while $i > 0$ and $A[i] > key$
5. \hspace{2em} do $A[i+1] = A[i]$
6. \hspace{2em} $i = i - 1$
7. \hspace{1em} $A[i+1] = key$

Analysis of the algorithm:

- (correctness proof): to show that the algorithm is as desired;
- (efficiency proof): to show a guaranteed efficiency of the algorithm
Chapter 2. Getting Started

Correctness proof: this is to prove
Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)
Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by the algorithm to
Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by the algorithm to

the post-condition (condition for the output)
Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by the algorithm to

the post-condition (condition for the output)

If the algorithm consists of sequential blocks of instructions, the task is to prove the correct transformation by each block.
Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by the algorithm to

the post-condition (condition for the output)

If the algorithm consists of sequential blocks of instructions,
the task is to prove the correct transformation by each block.

This means we need to prove that every sequential statement
in the algorithm transforms the given pre-condition to
the given post-condition.
Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.
The most difficult task is to do this for a loop statement.
Finding loop invariant becomes necessary and sufficient.
The most difficult task is to do this for a loop statement. Finding loop invariant becomes necessary and sufficient. In Insertion-Sort, the loop invariant is
The most difficult task is to do this for a loop statement. Finding loop invariant becomes necessary and sufficient.

In Insertion-Sort, the loop invariant is

at each iteration, the sublist $A[1..j - 1]$ consists of the elements originally in the positions $[1..j-1]$ but in sorted order.
The most difficult task is to do this for a loop statement. Finding loop invariant becomes necessary and sufficient.

In Insertion-Sort, the loop invariant is

at each iteration, the sublist $A[1..j-1]$ consists of the elements originally in the positions $[1..j-1]$ but in sorted order.

However, finding loop invariants is difficult!
Chapter 2. Getting Started

Efficiency analysis: This is to show that
Chapter 2. Getting Started

Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.
Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.
- resources can be CPU time and memory space used in the computation.
Chapter 2. Getting Started

Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.
- Resources can be CPU time and memory space used in the computation.
- However, the unit measured is not real time or memory unit.
Chapter 2. Getting Started

Time of an algorithm $A(x)$

Input Instances $x$
Chapter 2. Getting Started

Time of an algorithm $A(x)$

Input Instances  worst case

$x$
Chapter 2. Getting Started

![Graph with y-axis labeled "Time of an algorithm A(x)" and x-axis labeled "Input Instances" and "worst case". There is a blue dashed line marked as "Upper bound for algorithm A, bounding all cases of instances".](image)
Chapter 2. Getting Started

Resource measurement based on

• random-access machine (RAM)
• counting primitive operations: addition, subtraction, floor, ceiling, multiplication, jump, memory movement, these operations differ in time by a constant multiplicative factor.
• speed between different machines: a constant multiplicative factor.
Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)

- counting primitive operations: addition, substraction, floor, ceiling, multiplication, jump, memory movement,
Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)

- counting primitive operations: addition, subtraction, floor, ceiling, multiplication, jump, memory movement,

these operations differs in time by a constant multiplicative factor.
Resource measurement based on

- random-access machine (RAM)

- counting primitive operations: addition, substraction, floor, ceiling, multiplication, jump, memory movement,

  these operations differs in time by a constant multiplicative factor.

- speed between different machines: a constant multiplicative factor.
Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)
Analysis of

Algorithm **INSERTION-SORT**(A)

1. **for** \( j = 2 \) **to** \( \text{length}[A] \) **do**
2. \( \text{key} = A[j] \)
3. \{**Insert** \( A[j] \) **into** sorted \( A[1..j-1] \}\}
4. \( i = j - 1 \)
5. **while** \( i > 0 \) **and** \( A[i] > \text{key} \)
7. \( i = i - 1 \)
8. \( A[i + 1] = \text{key} \)
Chapter 2. Getting Started

Analysis of

Algorithm Insertion-Sort($A$)

1. for $j = 2$ to length[$A$] do
2.     key = $A[j]$
4.     $i = j - 1$
5. while $i > 0$ and $A[i] > key$
7.     $i = i - 1$
8. $A[i + 1] = key$

Assume $t_j$ to be the number of times while is executed for every $j$.

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

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

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

\[ T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \]

for some constants \( a, b, c, \)
Chapter 2. Getting Started

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

\[ T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \]

for some constants \( a, b, c \), for example, \( a \geq c_5 + c_6 + c_7 \), \( b \geq c_1 + c_2 + c_4 + c_8 \).
Chapter 2. Getting Started

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

\[ T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \]

for some constants \( a, b, c \), for example, \( a \geq c_5 + c_6 + c_7 \), \( b \geq c_1 + c_2 + c_4 + c_8 \).

Because \( t_j = j \) in the worst case (e.g., list is reversely sorted).
Chapter 2. Getting Started

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

\[ T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \]

for some constants \(a, b, c\), for example, \(a \geq c_5 + c_6 + c_7\), \(b \geq c_1 + c_2 + c_4 + c_8\).

Because \(t_j = j\) in the worst case (e.g., list is reversely sorted).

\[ T(n) \leq a \frac{n}{2} (n + 1) + bn + c - a \]
Chapter 2. Getting Started

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

\[ T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \]

for some constants \( a, b, c \), for example, \( a \geq c_5 + c_6 + c_7 \), \( b \geq c_1 + c_2 + c_4 + c_8 \).

Because \( t_j = j \) in the worst case (e.g., list is reversely sorted).

\[ T(n) \leq a \frac{n}{2} (n + 1) + bn + c - a \leq xn^2 + yn + z \]

for some constants \( x, y, z \).
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq x n^2 + y n + z \]

for some constants \( x, y, z \) \( \leq \) means in all cases for which \( \leq \) holds; \( x n^2 + y n + z \) is a complexity upper bound for \( T(n) \)
So we have proved:

\[ T(n) \leq xn^2 + yn + z \]

for some constants \( x, y, z \)

\( \leq \) means in all cases for which \( \leq \) holds;
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \]

for some constants \( x, y, z \)

\( \leq \) means in all cases for which \( \leq \) holds;

\( xn^2 + yn + z \) is a complexity upper bound for \( T(n) \)
Chapter 2. Getting Started

Important complexity issues:

1. \( n \): the number of bits encoding input \( x \), i.e., \( n = |x| \).

   It is inaccurate for \( n \) to represent the number of items in the input.

   Consider to sort 4 items \( \langle x_1, x_2, x_3, x_4 \rangle \) of values in the scale of \( 2^N \), for some very large \( N \).

   \( n \) is the number of items, \( n = 4 \), then any sorting algorithm would run in constant time.

   However, since \( x_1, x_2, x_3, x_4 \) are of very large values, a single comparison \( x_1 \leq x_2 ? \) would need a time proportional to \( N \).

   Hence, if \( n = |\langle x_1, x_2, x_3, x_4 \rangle| \), then \( n \approx N \).

   To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in \( N \) (i.e., total time is linear in \( n \)).
Important complexity issues:

1. size of input $n$: the number of bits encoding input $x$, i.e., $n = |x|$. It is inaccurate for $n$ to represent the number of items in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of $2^N$, for some very large $N$.

- If $n$ is the number of items, $n = 4$, then any sorting algorithm would run in constant time.

- However, since $x_1, x_2, x_3, x_4$ are of very large values, a single comparison $x_1 \leq x_2$ would need a time proportional to $N$.

Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$.

To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in $N$ (i.e., total time is linear in $n$).
Chapter 2. Getting Started

Important complexity issues:

1. **size of input** $n$: the number of bits encoding input $x$, i.e., $n = |x|$. 

   If $n$ is the number of items, $n = 4$, then any sorting algorithm would run in constant time. However, since $x_1, x_2, x_3, x_4$ are of very large values, a single comparison $x_1 \leq x_2$? would need a time proportional to $N$. Hence, if $n = |(x_1, x_2, x_3, x_4)|$, then $n \approx N$.

   To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in $N$ (i.e., total time is linear in $n$).
Important complexity issues:

1. **size of input** $n$: the number of bits encoding input $x$, i.e., $n = |x|$. 
   
   *It is inaccurate* for $n$ to represent the number of items in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of $2^N$, for some very large $N$.

- If $n$ is the number of items, $n = 4$, then any sorting algorithm would run in constant time.
- However, since $x_1, x_2, x_3, x_4$ are of very large values, a single comparison $x_1 \leq x_2$ would need a time proportional to $N$.

Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$.

To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in $N$ (i.e., total time is linear in $n$).
Important complexity issues:

1. **size of input** \( n \): the number of bits encoding input \( x \), i.e., \( n = |x| \).

   It is inaccurate for \( n \) to represent the number of items in the input.

Consider to sort 4 items \( \langle x_1, x_2, x_3, x_4 \rangle \) of values in the scale of \( 2^N \), for some very large \( N \).
Important complexity issues:

1. **size of input** \( n \): the number of bits encoding input \( x \), i.e., \( n = |x| \).

   It is inaccurate for \( n \) to represent the number of items in the input.

Consider to sort 4 items \( \langle x_1, x_2, x_3, x_4 \rangle \) of values in the scale of \( 2^N \), for some very large \( N \).

- If \( n \) is the number of items, \( n = 4 \), then any sorting algorithm would run in **constant time**.
Important complexity issues:

1. **size of input** \( n \): the number of bits encoding input \( x \), i.e., \( n = |x| \).

   It is inaccurate for \( n \) to represent the number of items in the input.

Consider to sort 4 items \( \langle x_1, x_2, x_3, x_4 \rangle \) of values in the scale of \( 2^N \), for some very large \( N \).

- If \( n \) is the number of items, \( n = 4 \), then any sorting algorithm would run in **constant time**.

- However, since \( x_1, x_2, x_3, x_4 \) are of very large values, a single comparison \( x_1 \leq x_2 \) would need a **time proportional to** \( N \).
Important complexity issues:

1. **size of input** $n$: the number of bits encoding input $x$, i.e., $n = |x|$.

   *It is inaccurate for $n$ to represent the number of items in the input.*

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of $2^N$, for some very large $N$.

- If $n$ is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.

- However, since $x_1, x_2, x_3, x_4$ are of very large values, a single comparison $x_1 \leq x_2$? would need a time proportional to $N$.

Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$. 
Important complexity issues:

1. **size of input** $n$: the number of bits encoding input $x$, i.e., $n = |x|$.  

   **It is inaccurate** for $n$ to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of $2^N$, for some very large $N$.

- If $n$ is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.

- However, since $x_1, x_2, x_3, x_4$ are of very large values, a single comparison $x_1 \leq x_2$? would need a **time proportional to** $N$.

  Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$.

To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in $N$ (i.e., **total time is linear in** $n$).
Chapter 2. Getting Started

Running time

$T(n)$: the number of primitive operations executed,

worst-case running time: the running time upper bound for all inputs.

order of growth:

$T(n) = an^2 + bn + c$ grows the same rate as $an^2$ (if $a > 0$).
Chapter 2. Getting Started

2. Running time $T(n)$: the number of primitive operations executed, a function in $n$
2. Running time $T(n)$: the number of primitive operations executed, a function in $n$

worst-case running time: the running time upper bound for all inputs.
2. Running time $T(n)$: the number of primitive operations executed, a function in $n$

**worst-case running time**: the running time upper bound for all inputs.

**order of growth**: $T(n) = an^2 + bn + c$ grows the same rate as $an^2$ (if $a > 0$).
Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$. So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as $T(n) = O(n^2)$.

In general, $O(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k \}$.
Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as $T(n) = O(n^2)$.

In general, $O(g(n)) = \{f(n): \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k\}$.
Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$. 
Chapter 3. Growth of Functions

**Big-O:** set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$
Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$

In general,

$$O(g(n)) = \{ f(n) : \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k \}$$
Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$

In general,

$$O(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k\}$$
Chapter 3. Growth of Functions

For example: the following functions are all of the order of \( O(n^2) \):

\[
(1) \quad 3n^2 \\
(2) \quad 5n^2 + 6n \log_2 n + 90 \\
(3) \quad 0.001n^2 - 20n - 5000 \\
(4) \quad 3n \log^2 n + 6n \\
(5) \quad \sqrt{n} - 20 \log_2 n \\
(6) \quad \log_2 n + 56 \\
(7) \quad 345
\]

But the following are not:

\[
(8) \quad 3n^2 \log_2 n - 400n \\
(9) \quad n^2 .001
\]

Let us do proofs for some of these examples.
Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:
For example: the following functions are all of the order of $O(n^2)$:

1. $3n^2$
2. $5.3n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log_2 n + 6n$
5. $\sqrt{n} - 20 \log_2 n$
6. $\log_2 n + 56$
7. $345$
Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:

1. $3n^2$
2. $5.3n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log_2 n + 6n$
5. $\sqrt{n} - 20 \log_2 n$
6. $\log_2 n + 56$
7. $345$

But the following are not:
For example: the following functions are all of the order of $O(n^2)$:

1. $3n^2$
2. $5.3n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log_2 n + 6n$
5. $\sqrt{n} - 20 \log_2 n$
6. $\log_2 n + 56$
7. $345$

But the following are not:

8. $3n^2 \log_2 n - 400n$
9. $n^{2.001}$
For example: the following functions are all of the order of $O(n^2)$:

1. $3n^2$
2. $5.3n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log^n_2 + 6n$
5. $\sqrt{n} - 20 \log_2 n$
6. $\log_2 n + 56$
7. $345$

But the following are not:

8. $3n^2 \log_2 n - 400n$
9. $n^{2.001}$

Let us do proofs for some of these examples.
Chapter 4. Solving Recurrences

Example: binary search algorithm

Algorithm Binary Search \((A, p, r, \text{key})\)

1. if \(p > r\) return (NULL)
2. else
3. \(q = \lfloor \frac{p + r}{2} \rfloor\)
4. if \(A[q] = \text{key}\) return \((k)\)
5. else
6. if \(A[q] > \text{key}\) Binary Search \((A, p, q - 1, \text{key})\)
7. else Binary Search \((A, q + 1, r, \text{key})\)

Let \(T(n)\) be the worst case time complexity for Binary Search, where parameter \(n = r - p + 1\).

The time \(T(n)\) can be upper-bounded with the recurrence,

\[
T(n) \leq T\left(\frac{n}{2}\right) + c
\]

and

\(T(0) \leq c\)

where \(c > 0\) is a constant.

Note: we can also set base case \(T(1) \leq c\).
Chapter 4. Solving Recurrences

Example: binary search algorithm

Algorithm \textsc{Binary Search}(A, p, r, key)

1. \textbf{if} \( p > r \) \textbf{return} (NULL)
2. \textbf{else}
3. \hspace{1em} \( q = \left\lfloor \frac{p + r}{2} \right\rfloor \)
4. \hspace{1em} \textbf{if} \( A[q] = key \) \textbf{return} (k)
5. \hspace{1em} \textbf{else}
6. \hspace{2em} \textbf{if} \( A[q] > key \) \textsc{Binary Search} (A, p, q - 1, key)
7. \hspace{2em} \textbf{else}
8. \hspace{3em} \textsc{Binary Search} (A, q + 1, r, key)

Let \( T(n) \) be the worst case time complexity for \textsc{Binary Search}, where parameter \( n = r - p + 1 \).

The time \( T(n) \) can be upper-bounded with the recurrence,

\[ T(n) \leq T\left( \frac{n}{2} \right) + c \]

and \( T(0) \leq c \) where \( c > 0 \) is a constant.

Note: we can also set base case \( T(1) \leq c \).
Chapter 4. Solving Recurrences

Example: binary search algorithm

Algorithm Binary Search\((A, p, r, key)\)

1. \(\text{if } p > r \text{ return } (\text{NULL})\)
2. \(\text{else}\)
3. \(q = \lfloor \frac{p+r}{2} \rfloor\)
4. \(\text{if } A[q] = key \text{ return } (k)\)
5. \(\text{else}\)
6. \(\text{if } A[q] > key \text{ Binary Search } (A, p, q-1, key)\)
7. \(\text{else}\)
8. \(\text{Binary Search } (A, q + 1, r, key)\)

Let \(T(n)\) be the worst case time complexity for Binary Search, where parameter \(n = r - p + 1\).
Example: binary search algorithm

Algorithm Binary Search \((A, p, r, key)\)

1. if \(p > r\) return (NULL)
2. else
3. \(q = \left\lfloor \frac{p+r}{2} \right\rfloor\)
4. if \(A[q] = key\) return \((k)\)
5. else
6. if \(A[q] > key\) Binary Search \((A, p, q - 1, key)\)
7. else
8. Binary Search \((A, q + 1, r, key)\)

Let \(T(n)\) be the worst case time complexity for Binary Search, where parameter \(n = r - p + 1\).

The time \(T(n)\) can be upper-bounded with the recurrence,

\[
T(n) \leq T\left(\frac{n}{2}\right) + c \quad \text{and} \quad T(0) \leq c
\]

where \(c > 0\) is a constant.
Example: binary search algorithm

Algorithm **Binary Search** \((A, p, r, \text{key})\)

1. **if** \(p > r\) **return** (NULL)
2. **else**
3. \(q = \lfloor \frac{p+r}{2} \rfloor\)
4. **if** \(A[q] = \text{key}\) **return** (k)
5. **else**
6. **if** \(A[q] > \text{key}\) **Binary Search** \((A, p, q - 1, \text{key})\)
7. **else**
8. **Binary Search** \((A, q + 1, r, \text{key})\)

Let \(T(n)\) be the worst case time complexity for **Binary Search**, where parameter \(n = r - p + 1\).

The time \(T(n)\) can be upper-bounded with the recurrence,

\[
T(n) \leq T\left(\frac{n}{2}\right) + c \quad \text{and} \quad T(0) \leq c
\]

where \(c > 0\) is a constant.

**Note:** we can also set base case \(T(1) \leq c\).
Chapter 4. Solving Recurrences

We have recurrence

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

and

\[ T(1) \leq c \]

for the Binary Search algorithm.

The recurrence can be resolved by unfolding the recursive terms. The process to unfold the recurrence \( T(n) \) can be:

1. straightforward unfolding (for simpler recurrences)
2. graph based (called recursive tree method)
3. guess and induction based (called substitution method)
Chapter 4. Solving Recurrences

We have recurrence

\[ T(n) \leq T\left(\frac{n}{2}\right) + c \] and \( T(1) \leq c \)

for the Binary Search algorithm.
Chapter 4. Solving Recurrences

We have recurrence

\[ T(n) \leq T\left(\frac{n}{2}\right) + c \text{ and } T(1) \leq c \]

for the Binary Search algorithm.

The recurrence can be resolved by unfolding the recursive terms. The process to unfold the recurrence \( T(n) \) can be:
We have recurrence
\[ T(n) \leq T\left(\frac{n}{2}\right) + c \text{ and } T(1) \leq c \]
for the Binary Search algorithm.

The recurrence can be resolved by unfolding the recursive terms. The process to unfold the recurrence \( T(n) \) can be:

1. straightforward unfolding (for simpler recurrences)
We have recurrence

\[ T(n) \leq T\left(\frac{n}{2}\right) + c \quad \text{and} \quad T(1) \leq c \]

for the Binary Search algorithm.

The recurrence can be resolved by unfolding the recursive terms. The process to unfold the recurrence \( T(n) \) can be:

1. straightforward unfolding (for simpler recurrences)
2. graph based (called recursive tree method).
3. guess and induction based (called substitution method)
We have recurrence

\[ T(n) \leq T\left(\frac{n}{2}\right) + c \quad \text{and} \quad T(1) \leq c \]

for the Binary Search algorithm.

The recurrence can be resolved by unfolding the recursive terms. The process to unfold the recurrence \( T(n) \) can be:

1. straightforward unfolding (for simpler recurrences)
2. graph based (called recursive tree method).
3. guess and induction based (called substitution method)
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

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

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

\[ \vdots \]

\[ T(n^{2^h}) \leq T(n^{2^{h+1}}) + c = T(n^{2^{h+1}} + 1) + c \]

where \( n^{2^{h+1}} + 1 = 1 \), \( 2^{h+1} = n \), \( h + 1 = \log_2 n \), etc.

There are \( h + 1 \) inequalities.

\[ T(n) \leq T(n^{2^{h+1}} + 1) + c(h + 1) \]

\[ = c + c \log_2 n = O(\log_2 n) \]

\[ \Rightarrow \] proved this equation using definition of big-O.
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

   use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;
1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

$T(n) \leq T\left(\frac{n}{2}\right) + c$
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use \( T(n) \leq T\left(\frac{n}{2}\right) + c \) as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c
\end{align*}
\]
1. Simple, straightforward unfolding:

use \( T(n) \leq T\left(\frac{n}{2}\right) + c \) as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) & \leq T\left(\frac{n}{2^3}\right) + c
\end{align*}
\]
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) & \leq T\left(\frac{n}{2^3}\right) + c \\
\ldots & \ldots
\end{align*}
\]

\[T\left(\frac{n}{2^h}\right) \leq T\left(\frac{n}{2^{h+1}}\right) + c\]

where $n_{2^h+1} = 1$,

\[2^{h+1} = n, h + 1 = \log_2 n, T(1) + c(h+1) = c + c\log_2 n = O(\log_2 n)\]

\[\Rightarrow \text{proved this equation using definition of big-O}\]
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use \( T(n) \leq T\left(\frac{n}{2}\right) + c \) as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) & \leq T\left(\frac{n}{2^3}\right) + c \\
\ldots & \\
T\left(\frac{n}{2^h}\right) & \leq T\left(\frac{n}{2^{h+1}}\right) + c \\
\end{align*}
\]

where \( \frac{n}{2^{h+1}} = 1, 2^{h+1} = n, h + 1 = \log_2 n \)
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

   use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

   
   $T(n) \leq T\left(\frac{n}{2}\right) + c$

   $T\left(\frac{n}{2}\right) \leq T\left(\frac{n}{2^2}\right) + c$

   $T\left(\frac{n}{2^2}\right) \leq T\left(\frac{n}{2^3}\right) + c$

   
   

   $T\left(\frac{n}{2^h}\right) \leq T\left(\frac{n}{2^{h+1}}\right) + c$ where $\frac{n}{2^{h+1}} = 1$, $2^{h+1} = n$, $h + 1 = \log_2 n$
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

$$
T(n) \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) \leq T\left(\frac{n}{2^3}\right) + c \\
\cdots \\
T\left(\frac{n}{2^h}\right) \leq T\left(\frac{n}{2^{h+1}}\right) + c
$$

where $\frac{n}{2^{h+1}} = 1$, $2^{h+1} = n$, $h + 1 = \log_2 n$

$$
T(n) \leq T\left(\frac{n}{2^{h+1}}\right) + c(h + 1)
$$

there are $h + 1$ inequalities
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) & \leq T\left(\frac{n}{2^3}\right) + c \\
\ldots & \\
T\left(\frac{n}{2^h}\right) & \leq T\left(\frac{n}{2^{h+1}}\right) + c
\end{align*}
\]

where $\frac{n}{2^{h+1}} = 1$, $2^{h+1} = n$, $h + 1 = \log_2 n$

\[
T(n) \leq T\left(\frac{n}{2^{h+1}}\right) + c(h + 1) = T(1) + c(h + 1)
\]

there are $h + 1$ inequalities
1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

\[
T(n) \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) \leq T\left(\frac{n}{2^3}\right) + c \\
\ldots \\
T\left(\frac{n}{2^h}\right) \leq T\left(\frac{n}{2^{h+1}}\right) + c
\]

where $\frac{n}{2^{h+1}} = 1$, $2^{h+1} = n$, $h + 1 = \log_2 n$

\[
T(n) \leq T\left(\frac{n}{2^{h+1}}\right) + c(h + 1)
= T(1) + c(h + 1) \\
= c + c \log_2 n
\]
1. Simple, straightforward unfolding:

use $T(n) \leq T(\frac{n}{2}) + c$ as a template;

\[
\begin{align*}
T(n) &\leq T(\frac{n}{2}) + c \\
T(\frac{n}{2}) &\leq T(\frac{n}{2^2}) + c \\
T(\frac{n}{2^2}) &\leq T(\frac{n}{2^3}) + c \\
\cdots &
\end{align*}
\]

\[
T(\frac{n}{2^h}) \leq T(\frac{n}{2^{h+1}}) + c \quad \text{where} \quad \frac{n}{2^{h+1}} = 1, \ 2^{h+1} = n, \ h + 1 = \log_2 n
\]

\[
\begin{align*}
T(n) &\leq T(\frac{n}{2^{h+1}}) + c(h + 1) \\
&= T(1) + c(h + 1) \\
&= c + c \log_2 n \\
&= O(\log_2 n)
\end{align*}
\]
Chapter 4. Solving Recurrences

1. Simple, straightforward unfolding:

use $T(n) \leq T\left(\frac{n}{2}\right) + c$ as a template;

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2}\right) + c \\
T\left(\frac{n}{2}\right) & \leq T\left(\frac{n}{2^2}\right) + c \\
T\left(\frac{n}{2^2}\right) & \leq T\left(\frac{n}{2^3}\right) + c \\
\cdots & \\
T\left(\frac{n}{2^h}\right) & \leq T\left(\frac{n}{2^{h+1}}\right) + c \quad \text{where } \frac{n}{2^{h+1}} = 1, \ 2^{h+1} = n, \ h + 1 = \log_2 n
\end{align*}
\]

\[
\begin{align*}
T(n) & \leq T\left(\frac{n}{2^{h+1}}\right) + c(h + 1) \\
& = T(1) + c(h + 1) \\
& = c + c \log_2 n \\
& = O(\log_2 n) \quad \iff \text{proved this equation using definition of big-}O
\end{align*}
\]
Chapter 4. Solving Recurrences

Another example:

Algorithm Merge Sort \((A, p, r)\)

1. if \(p < r\)
2. then \(q = \lfloor \frac{p + r}{2} \rfloor\)
3. Merge Sort \((A, p, q)\)
4. Merge Sort \((A, q + 1, r)\)
5. Merging2Lists \((A, p, q, r)\)

Analysis of the algorithm.

- Assume \(n = r - p + 1\), a power of 2; also assume \(T(n)\) is time for Merge Sort \((A, p, r)\).
- \(t_1, t_2 = c \cdot t_3 = t_4 = T(n/2)\)
- \(t_5 \leq n\) (why?)

\[ T(n) = t_1, t_2 + t_3 + t_4 + t_5 \leq 2T(n/2) + n + c \]

base case: \(T(1) = d\), constant
Chapter 4. Solving Recurrences

Another example:

Algorithm Merge Sort \((A, p, r)\)

1. \(\text{if } p < r\)
2. \(q = \lfloor \frac{p + r}{2} \rfloor\)
3. Merge Sort \((A, p, q)\)
4. Merge Sort \((A, q + 1, r)\)
5. Merging2Lists \((A, p, q, r)\)

Analysis of the algorithm.

- Assume \(n = r - p + 1\), a power of 2; also assume \(T(n)\) is time for Merge Sort \((A, p, r)\).

- \(t_1, 2 = c\)
- \(t_3 = t_4 = T(n/2)\)
- \(t_5 \leq n\) (why?)

\[ T(n) = t_{1, 2} + t_3 + t_4 + t_5 \leq 2T(n/2) + n + c \]

Base case: \(T(1) = d\), constant
Another example:

Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \ p < r
2. \hspace{2em} \textbf{then} \ q = \left\lfloor \frac{p+r}{2} \right\rfloor
3. \hspace{2em} \textsc{Merge Sort}(A, p, q)
4. \hspace{2em} \textsc{Merge Sort}(A, q + 1, r)
5. \hspace{2em} \textsc{Merging2Lists}(A, p, q, r)
Another example:

Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \(p < r\)
2. \textbf{then} \(q = \left\lfloor \frac{p+r}{2} \right\rfloor\)
3. \textsc{Merge Sort}(A, p, q)
4. \textsc{Merge Sort}(A, q + 1, r)
5. \textsc{Merging2Lists}(A.p, q, r)

Analysis of the algorithm.

- Assume \(n = r - p + 1\), a power of 2;
  also assume \(T(n)\) is time for \textsc{Merge Sort}(A, p, r). Then
Another example:

Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \ p < r
2. \textbf{then} \ q = \left\lfloor \frac{p+r}{2} \right\rfloor
3. \textsc{Merge Sort}(A, p, q)
4. \textsc{Merge Sort}(A, q + 1, r)
5. \textsc{Merging2Lists}(A.p, q, r)

Analysis of the algorithm.

- Assume \( n = r - p + 1 \), a power of 2; also assume \( T(n) \) is time for \textsc{Merge Sort}(A, p, r). Then

- \( t_{1,2} = c \)
Chapter 4. Solving Recurrences

Another example:

Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \ p < r
2. \hspace{1em} \textbf{then} \ \ q = \lfloor \frac{p+r}{2} \rfloor
3. \hspace{1em} \textsc{Merge Sort}(A, p, q)
4. \hspace{1em} \textsc{Merge Sort}(A, q + 1, r)
5. \hspace{1em} \textsc{Merging2Lists}(A.p, q, r)

Analysis of the algorithm.

- Assume \( n = r - p + 1 \), a power of 2;
  also assume \( T(n) \) is time for \textsc{Merge Sort}(A, p, r). Then

- \( t_{1,2} = c \)
- \( t_3 = t_4 = T\left( \frac{n}{2} \right) \)
Chapter 4. Solving Recurrences

Another example:

Algorithm \( \text{Merge Sort}(A, p, r) \)

1. \( \textbf{if} \ p < r \)
2. \( \textbf{then} \ q = \lfloor \frac{p+r}{2} \rfloor \)
3. \( \text{Merge Sort}(A, p, q) \)
4. \( \text{Merge Sort}(A, q + 1, r) \)
5. \( \text{Merging2Lists}(A[p, q, r]) \)

Analysis of the algorithm.

- Assume \( n = r - p + 1 \), a power of 2;
  also assume \( T(n) \) is time for \( \text{Merge Sort}(A, p, r) \). Then

- \( t_{1,2} = c \)
- \( t_3 = t_4 = T\left(\frac{n}{2}\right) \)
- \( t_5 \leq n \) (why?)
Chapter 4. Solving Recurrences

Another example:

Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \ p < r
2. \hspace{1em} \textbf{then} \ q = \left\lfloor \frac{p+r}{2} \right\rfloor
3. \hspace{2em} \textsc{Merge Sort}(A, p, q)
4. \hspace{2em} \textsc{Merge Sort}(A, q + 1, r)
5. \hspace{2em} \textsc{Merging2Lists}(A.p, q, r)

Analysis of the algorithm.

- Assume \( n = r - p + 1 \), a power of 2;
  also assume \( T(n) \) is time for \textsc{Merge Sort}(A, p, r). Then

- \( t_{1,2} = c \)
- \( t_3 = t_4 = T(\frac{n}{2}) \)
- \( t_5 \leq n \) (why?)

\[
T(n) = t_{1,2} + t_3 + t_4 + t_5 \leq 2T(\frac{n}{2}) + n + c
\]
Chapter 4. Solving Recurrences

Another example:

Algorithm MERGE SORT($A, p, r$)

1. if $p < r$
2. then $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
3. MERGE SORT($A, p, q$)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS($A.p, q, r$)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
- also assume $T(n)$ is time for MERGE SORT($A, p, r$). Then

- $t_{1,2} = c$
- $t_3 = t_4 = T\left(\frac{n}{2}\right)$
- $t_5 \leq n$ (why?)

$$T(n) = t_{1,2} + t_3 + t_4 + t_5 \leq 2T\left(\frac{n}{2}\right) + n + c$$

base case: $T(1) = d$, constant
Chapter 4. Solving Recurrences

Solve recurrence \( T(n) \leq 2T(n^2) + n + c \) with base case \( T(1) = d \) with a simple method:

\[
T(n) \leq 2T(n^2) + n + c \\
\leq 2T(n^2) + n^2 + c \\
\leq 2T(n^2^2) + n^2 + c \\
\leq 2T(n^2^3) + n^2 + c \\
\vdots \\
\leq 2T(n^{2^{h}}) + n^{2^{h}} + c
\]

where \( n^{2^{h}+1} = 1 \), multiplying \( 2, 2^2, \ldots \) to the second, third, \ldots inequalities, respectively,

\[
T(n) \leq 2^{h}T(n^{2^{h}+1}) + \left( 2^{h} \times n^{2^{h+1}} + 2^{h}c \right)
\]

\[
T(n) \leq 2^{h+1}T(n^{2^{h}+1}) + \left( 2^{h+1} \times n^{2^{h+1}} + 2^{h+1}c \right)
\]

\[
T(n) \leq 2^{h+2}T(n^{2^{h+2}}) + \left( 2^{h+2} \times n^{2^{h+2}} + 2^{h+2}c \right)
\]

\[
T(n) \leq 2^{h+3}T(n^{2^{h+3}}) + \left( 2^{h+3} \times n^{2^{h+3}} + 2^{h+3}c \right)
\]

\[
T(n) \leq 2^{h+n}T(n^{2^{h+n}}) + \left( 2^{n} \times n^{2^{n}} + 2^{n}c \right)
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case} \quad T(1) = d \]

with a simple method:
Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) &\leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) &\leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]  with base case \( T(1) = d \)

with a simple method:

\[
\begin{align*}
T(n) &\leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) &\leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) &\leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]
\[ T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \]
\[ T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \]
\[ \ldots \]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case} \quad T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
& \ldots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \\
& \quad \text{where} \quad \frac{n}{2^{h+1}} = 1
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]  with base case \( T(1) = d \)

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
\cdots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying \( 2, 2^2, \ldots \) to the second, third, \ldots inequalities, respectively,
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \] with base case \( T(1) = d \)

with a simple method:

\[
\begin{align*}
T(n) &\leq 2T(\frac{n}{2}) + n + c \\
T(\frac{n}{2}) &\leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T(\frac{n}{2^2}) &\leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
\vdots \\
T(\frac{n}{2^h}) &\leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying \(2, 2^2, \ldots\) to the second, third, \ldots inequalities, respectively,

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) &\leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) &\leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) &\leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
&\quad \vdots \\
T\left(\frac{n}{2^h}\right) &\leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying 2, 2², \ldots to the second, third, \ldots inequalities, respectively,

\[
\begin{align*}
T(n) &\leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) &\leq 2^2 T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]

with base case \( T(1) = d \)

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
\ldots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \\
\text{where } \frac{n}{2^{h+1}} & = 1
\end{align*}
\]

multiplying \( 2, 2^2, \ldots \) to the second, third, \ldots inequalities, respectively,

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{2^2}\right) & \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
\ldots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying \(2, 2^2, \ldots\) to the second, third, \ldots inequalities, respectively,

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{2^2}\right) & \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \\
\ldots \\
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]

with base case \( T(1) = d \)

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
& \vdots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \\
\end{align*}
\]

where \( \frac{n}{2^{h+1}} = 1 \)

Multiplying 2, 2^2, \ldots to the second, third, \ldots inequalities, respectively,

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{2^2}\right) & \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \\
& \vdots \\
2^hT\left(\frac{n}{2^h}\right) & \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \\
\end{align*}
\]

where \( \frac{n}{2^{h+1}} = 1 \)
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
& \vdots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying \(2, 2^2, \ldots\) to the second, third, \ldots\ inequalities, respectively,

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{2^2}\right) & \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \\
& \vdots \\
2^hT\left(\frac{n}{2^h}\right) & \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = d \]

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{2^2}\right) & \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c \\
& \vdots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

multiplying 2, 2^2, \ldots to the second, third, \ldots inequalities, respectively,

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{2^2}\right) & \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \\
& \vdots \\
2^hT\left(\frac{n}{2^h}\right) & \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

\[
T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i
\]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T(n^{2h+1}) + (h+1)n + c h \sum_{i=0}^{2^h} i \]

With \( n^{2h+1} = 1 \), we have \( n = 2^{h+1} \) or \( h+1 = \log_2 n \).

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c (2^h - 1) \]

\[ = n \log_2 n + (d+c)n - c \]

\[ = O(n \log_2 n) \]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that

\[ n \log_2 n + (c+d)n - c \leq an \log_2 n \]

when \( n > k \).

Proof:

Choose \( a = 2 \). Then to make (1) holds, we need \( \log_2 n > c+d \). So \( k = 2c + d \) suffices.

That is, \( n \log_2 n + (c+d)n - c \leq 2n \log_2 n \)

when \( n > k = 2c + d \).

So \( T(n) = O(n \log_2 n) \).
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left( \frac{n}{2^{h+1}} \right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \).
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \]
\[ = nd + n \log_2 n + c(n - 1) \]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \]
\[ = nd + n \log_2 n + c(n - 1) \]
\[ = n \log_2 n + (d + c)n - c \]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \]
\[ = nd + n \log_2 n + c(n - 1) \]
\[ = n \log_2 n + (d + c)n - c \]
\[ = O(n \log_2 n) \]
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[
T(n) \leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1)
= nd + n \log_2 n + c(n - 1)
= n \log_2 n + (d + c)n - c
= O(n \log_2 n)
\]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that

\[
n \log_2 n + (c + d)n - c \leq an \log_2 n \quad \text{(1)}
\]

when \( n > k \).
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[
T(n) \leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\
= nd + n \log_2 n + c(n - 1) \\
= n \log_2 n + (d + c)n - c \\
= O(n \log_2 n)
\]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that

\[
n \log_2 n + (c + d)n - c \leq an \log_2 n \quad (1)
\]

when \( n > k \).

Proof:

Choose \( a = 2 \). Then to make (1) holds, we need \( \log_2 n > c + d \). So \( k = 2^{c+d} \) suffices.
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \]
\[ = nd + n \log_2 n + c(n - 1) \]
\[ = n \log_2 n + (d + c)n - c \]
\[ = O(n \log_2 n) \]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that

\[ n \log_2 n + (c + d)n - c \leq an \log_2 n \quad (1) \]

when \( n > k \).

Proof:

Choose \( a = 2 \). Then to make (1) holds, we need \( \log_2 n > c + d \). So \( k = 2^{c+d} \) suffices. That is, \( n \log_2 n + (c + d)n - c \leq 2n \log_2 n \) when \( n > k = 2^{c+d} \).
Chapter 4. Solving Recurrences

\[ T(n) \leq 2^{h+1} T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^{h} 2^{i} \]

With \( \frac{n}{2^{h+1}} = 1 \), we have \( n = 2^{h+1} \) or \( h + 1 = \log_2 n \)

\[ T(n) \leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \]
\[ = nd + n \log_2 n + c(n - 1) \]
\[ = n \log_2 n + (d + c)n - c \]
\[ = O(n \log_2 n) \]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that

\[ n \log_2 n + (c + d)n - c \leq an \log_2 n \quad \text{(1)} \]

when \( n > k \).

Proof:

Choose \( a = 2 \). Then to make (1) holds, we need \( \log_2 n > c + d \). So \( k = 2^{c+d} \) suffices.

That is, \( n \log_2 n + (c + d)n - c \leq 2n \log_2 n \) when \( n > k = 2^{c+d} \).

So

\[ T(n) = O(n \log_2 n) \]
Chapter 4. Solving Recurrences

When \( n \) is not a power of 2

• choose \( m \) \( n \) such that \( m \) \( n \) is a power of 2 and the smallest such that \( n \leq m \) \( n \);

• \( T(n) \leq T(m) \) \( n \), why?
  assume \( T \) to be monotonic;

• use the analysis we just did, \( T(m) = O(m \log_2 m) \);
  that is, \( \exists c, k, T(m) \leq cm \log_2 m \) when \( m \geq k \);

• but \( m < 2^n \), why?
  because \( m < 2^n \);

• So \( T(n) \leq T(m) \leq cm \log_2 m \leq 2cn \log_2 (2^n) \leq 2cn \log_2 n \leq c' n \log_2 n \), here \( c' = 4c \), when \( m \geq k \) (\( \geq 4 \)), i.e., when \( n \geq \lceil k/2 \rceil \) (\( \geq 2 \));

• therefore, \( T(n) = O(n \log_2 n) \).
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- Choose $m \cdot n$ such that $m \cdot n$ is a power of 2 and the smallest such that $n \leq m \cdot n$;
- $T(n) \leq T(m \cdot n)$, why? Assume $T$ to be monotonic;
- Use the analysis we just did, $T(m \cdot n) = O(m \cdot n \log_2 m \cdot n)$; that is, $\exists c, k, T(m \cdot n) \leq cm \cdot n \log_2 m \cdot n$ when $m \cdot n \geq k$;
- But $m \cdot n < 2^n$, why? Because $m \cdot n < 2^n$;
- So $T(n) \leq T(m \cdot n) \leq cm \cdot n \log_2 m \cdot n \leq 2^c n \log_2 (2^n) \leq 2^c n \log_2 n = c' n \log_2 n$, here $c' = 4c$, when $m \cdot n \geq k$ ($\geq 4$), i.e., when $n \geq \lceil k \cdot 2 \rceil$ ($\geq 2$); therefore, $T(n) = O(n \log_2 n)$. 
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why?
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
When \( n \) is not a power of 2

• choose \( m_n \) such that \( m_n \) is a power of 2 and the smallest such that \( n \leq m_n \);

• \( T(n) \leq T(m_n) \), why? assume \( T \) to be monotonic;

• use the analysis we just did, \( T(m_n) = O(m_n \log_2 m_n) \);
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
- but $m_n < 2n$, why?
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;

- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k$, $T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;

- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;
When \( n \) is not a power of 2

- choose \( m_n \) such that \( m_n \) is a power of 2 and the smallest such that \( n \leq m_n \);
- \( T(n) \leq T(m_n) \), why? assume \( T \) to be monotonic;
- use the analysis we just did, \( T(m_n) = O(m_n \log_2 m_n) \);
  that is, \( \exists c, k, T(m_n) \leq cm_n \log_2 m_n \) when \( m_n \geq k \);
- but \( m_n < 2n \), why? because \( \frac{m_n}{2} < n \);
- So \( T(n) \)
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;

- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k$, $T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;

- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;

- So $T(n) \leq T(m_n)$
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;
- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n$
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;

- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k$, $T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;

- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;

- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2(2n)$
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$;
  that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;
- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2$
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;

- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;

- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;

- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2 = 4cn \log_2 n =$
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;

- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;

- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;

- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;

- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2 = 4cn \log_2 n = c'n \log_2 n$, here $c' = 4c$,
When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$;
  that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;
- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2$
  $= 4cn \log_2 n = c'n \log_2 n$, here $c' = 4c$,
  when $m_n \geq k(\geq 4)$,
Chapter 4. Solving Recurrences

When $n$ is not a power of 2

- choose $m_n$ such that $m_n$ is a power of 2 and the smallest such that $n \leq m_n$;
- $T(n) \leq T(m_n)$, why? assume $T$ to be monotonic;
- use the analysis we just did, $T(m_n) = O(m_n \log_2 m_n)$; that is, $\exists c, k, T(m_n) \leq cm_n \log_2 m_n$ when $m_n \geq k$;
- but $m_n < 2n$, why? because $\frac{m_n}{2} < n$;
- So $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2$
  $= 4cn \log_2 n = c' n \log_2 n$, here $c' = 4c$, when $m_n \geq k(\geq 4)$, i.e., when $n \geq \left\lceil \frac{k}{2} \right\rceil (\geq 2)$;
Chapter 4. Solving Recurrences

When \( n \) is not a power of 2

- choose \( m_n \) such that \( m_n \) is a power of 2 and the smallest such that \( n \leq m_n \);

- \( T(n) \leq T(m_n) \), why? assume \( T \) to be monotonic;

- use the analysis we just did, \( T(m_n) = O(m_n \log_2 m_n) \);
  that is, \( \exists c, k, T(m_n) \leq cm_n \log_2 m_n \) when \( m_n \geq k \);

- but \( m_n < 2n \), why? because \( \frac{m_n}{2} < n \);

- So \( T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2 \)
  \[ = 4cn \log_2 n = c' n \log_2 n, \] here \( c' = 4c \),
  when \( m_n \geq k(\geq 4) \), i.e., when \( n \geq \lceil \frac{k}{2} \rceil(\geq 2) \);

- therefore, \( T(n) = O(n \log_2 n) \).
Chapter 4. Solving Recurrences

Methods for solving recurrences
Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)
Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $P(n)$ for every natural number $n \geq 1$, it suffices to prove

- $P(1)$ holds;
Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $P(n)$ for every natural number $n \geq 1$, it suffices to prove

- $P(1)$ holds;
- for every $k \geq 1$, if $P(k)$ holds, then $P(k + 1)$ holds.
Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $\mathcal{P}(n)$ for every natural number $n \geq 1$, it suffices to prove

- $\mathcal{P}(1)$ holds;
- for every $k \geq 1$, if $\mathcal{P}(k)$ holds, then $\mathcal{P}(k + 1)$ holds.

"The principle for dominos to fall".
Math induction comes with different forms or variants
Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $P(3)$ instead of $P(1)$;
Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $P(3)$ instead of $P(1)$;
- the statement to prove: $P(k) \rightarrow P(k + 1)$ has the variant:
Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \rightarrow \mathcal{P}(k + 1)$ has the variant:

$$\mathcal{P}(1) \land \mathcal{P}(2) \land \cdots \land \mathcal{P}(k) \rightarrow \mathcal{P}(k + 1)$$
Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $P(3)$ instead of $P(1)$;
- the statement to prove: $P(k) \rightarrow P(k + 1)$ has the variant:

$$P(1) \land P(2) \land \cdots \land P(k) \rightarrow P(k + 1)$$

- other variants: e.g., $P(k) \rightarrow P(2k)$
Math induction comes with different forms or variants

- the base case can be for any integer, e.g., \( P(3) \) instead of \( P(1) \);
- the statement to prove: \( P(k) \rightarrow P(k+1) \) has the variant:

\[
P(1) \land P(2) \land \cdots \land P(k) \rightarrow P(k+1)
\]

- other variants: e.g., \( P(k) \rightarrow P(2k) \)
  
but we need to make sure all \( n \)'s beyond the base cases are covered.
Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

• the base case can be for any integer, e.g., \( P(3) \) instead of \( P(1) \);

• the statement to prove: \( P(k) \rightarrow P(k + 1) \) has the variant:

\[
P(1) \land P(2) \land \cdots \land P(k) \rightarrow P(k + 1)
\]

• other variants: e.g., \( P(k) \rightarrow P(2k) \)
  but we need to make sure all \( n \)’s beyond the base cases are covered.

\[
P(k) \rightarrow P(2k) \land P(2k + 1)
\]
Chapter 4. Solving Recurrences

**Theorem:** Arithmetic sequence of first $n$ terms

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$
Chapter 4. Solving Recurrences

**Theorem**: Arithmetic sequence of first $n$ terms

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$

**Proof**: (use math induction)
Chapter 4. Solving Recurrences

**Theorem**: Arithmetic sequence of first $n$ terms

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$

**Proof**: (use math induction)

step 1: base case: $n = 1$, left = 1, right = $\frac{1}{2}(1 + 1) = 1$;
step 2: assumption:

$$1 + 2 + 3 + \cdots + n - 1 = \frac{n - 1}{2}(n - 1 + 1)$$
Chapter 4. Solving Recurrences

**Theorem**: Arithmetic sequence of first $n$ terms

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$

**Proof**: (use math induction)

step 1: base case: $n = 1$, left = 1, right = $\frac{1}{2}(1 + 1) = 1$;

step 2: assumption:

$$1 + 2 + 3 + \cdots + n - 1 = \frac{n - 1}{2}(n - 1 + 1) \text{ which is } = \frac{n - 1}{2}n$$

step 3: induction:

$$1 + 2 + 3 + \cdots + n$$
Chapter 4. Solving Recurrences

**Theorem:** Arithmetic sequence of first \( n \) terms

\[
1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)
\]

**Proof:** (use math induction)

step 1: base case: \( n = 1 \), left = 1, right = \( \frac{1}{2}(1 + 1) = 1 \);

step 2: assumption:

\[
1 + 2 + 3 + \cdots + n - 1 = \frac{n-1}{2}(n - 1 + 1) \text{ which is } = \frac{n-1}{2}n
\]

step 3: induction:

\[
1 + 2 + 3 + \cdots + n = (1 + 2 + 3 + \cdots + n - 1) + n
\]
Chapter 4. Solving Recurrences

**Theorem:** Arithmetic sequence of first \( n \) terms

\[
1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)
\]

**Proof:** (use math induction)

step 1: base case: \( n = 1 \), left = 1, right = \( \frac{1}{2}(1 + 1) = 1 \);

step 2: assumption:

\[
1 + 2 + 3 + \cdots + n - 1 = \frac{n - 1}{2} (n - 1 + 1) \text{ which is } = \frac{n - 1}{2} n
\]

step 3: induction:

\[
1 + 2 + 3 + \cdots + n = (1 + 2 + 3 + \cdots + n - 1) + n = \frac{n - 1}{2} n + n
\]
Chapter 4. Solving Recurrences

**Theorem:** Arithmetic sequence of first \( n \) terms

\[
1 + 2 + 3 + \cdots + n = \frac{n}{2} (n + 1)
\]

**Proof:** (use math induction)

1. **base case:** \( n = 1 \), left = 1, right = \( \frac{1}{2} (1 + 1) = 1 \);
2. **assumption:**
   
   \[
   1 + 2 + 3 + \cdots + n - 1 = \frac{n - 1}{2} (n - 1 + 1) \text{ which is } \frac{n - 1}{2} n
   \]
3. **induction:**
   
   \[
   1 + 2 + 3 + \cdots + n = (1 + 2 + 3 + \cdots + n - 1) + n = \frac{n - 1}{2} n + n = \frac{n}{2} (n + 1)
   \]
Chapter 4. Solving Recurrences

**Theorem:** Arithmetic sequence of first $n$ terms

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$

**Proof:** (use math induction)

step 1: base case: $n = 1$, left $= 1$, right $= \frac{1}{2}(1 + 1) = 1$;
step 2: assumption:

$$1 + 2 + 3 + \cdots + n - 1 = \frac{n - 1}{2}(n - 1 + 1) \text{ which is } = \frac{n - 1}{2}n$$

step 3: induction:

$$1 + 2 + 3 + \cdots + n = (1 + 2 + 3 + \cdots + n - 1) + n = \frac{n - 1}{2}n + n = \frac{n}{2}(n + 1)$$

So

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n + 1)$$
Algorithm MERGE SORT$(A, p, r)$

1. if $p < r$
2. then $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
3. MERGE SORT$(A, p, q)$
4. MERGE SORT$(A, q + 1, r)$
5. MERGING2LISTS$(A.p, q, r)$
Algorithm \textsc{Merge Sort}(A, p, r)

1. \textbf{if} \: p < r \\
2. \quad \textbf{then} \: q = \left\lfloor \frac{p+r}{2} \right\rfloor \\
3. \quad \textsc{Merge Sort}(A, p, q) \\
4. \quad \textsc{Merge Sort}(A, q + 1, r) \\
5. \quad \textsc{Merging2Lists}(A.p, q, r)
Chapter 4. Solving Recurrences

Algorithm Merge Sort \((A, p, r)\)

1. \textbf{if} \(p < r\)
2. \textbf{then} \(q = \lfloor \frac{p+r}{2} \rfloor\)
3. \text{Merge Sort} \((A, p, q)\)
4. \text{Merge Sort} \((A, q + 1, r)\)
5. \text{Merging2Lists} \((A, p, q, r)\)

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = c \]
**Chapter 4. Solving Recurrences**

**MergeSort** has the time complexity:

\[ T(n) \leq 2T\left( \frac{n}{2} \right) + n + c \quad \text{with base case} \quad T(1) = c \]

Use **substitution method** to prove that \( T(n) = O(n \log_2 n) \).
MergeSort has the time complexity:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case} \ T(1) = c \]

Use substitution method to prove that \( T(n) = O(n \log_2 n) \).

Proof. We claim that there are constants \( a, k \) such that

\[ T(n) \leq an \log_2 n \quad \text{when} \ n \geq k \quad (2) \]
Chapter 4. Solving Recurrences

**MergeSort** has the time complexity:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case} \quad T(1) = c \]

Use **substitution method** to prove that \( T(n) = O(n \log_2 n) \).

**Proof.** We claim that there are constants \( a, k \) such that

\[ T(n) \leq an \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]

step 1. base case: when \( n = 1, \ T(1) = c, \)
MergerSort has the time complexity:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = c \]

Use substitution method to prove that \( T(n) = O(n \log_2 n) \).

Proof. We claim that there are constants \( a, k \) such that

\[ T(n) \leq an \log_2 n \quad \text{when } n \geq k \quad (2) \]

step 1. base case: when \( n = 1 \), \( T(1) = c \), to make \( T(1) \leq an \log_2 n \),
Chapter 4. Solving Recurrences

MergeSort has the time complexity:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = c \]

Use substitution method to prove that \( T(n) = O(n \log_2 n) \).

**Proof.** We claim that there are constants \( a, k \) such that

\[ T(n) \leq an \log_2 n \quad \text{when } n \geq k \quad (2) \]

step 1. base case: when \( n = 1 \), \( T(1) = c \), to make \( T(1) \leq an \log_2 n \),
we need \( n \geq k = 2 \), **So we need to use \( T(2) \) as base case.**
Chapter 4. Solving Recurrences

**MergeSort** has the time complexity:

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) = c
\]

Use substitution method to prove that \( T(n) = O(n \log_2 n) \).

**Proof.** We claim that there are constants \( a, k \) such that

\[
T(n) \leq an \log_2 n \quad \text{when } n \geq k \quad (2)
\]

step 1. base case: when \( n = 1 \), \( T(1) = c \), to make \( T(1) \leq an \log_2 n \),
we need \( n \geq k = 2 \), **So we need to use** \( T(2) \) **as base case**.

from the recurrence, we known

\[
T(2) \leq 2T(1) + 2 + c = 3c + 2 \leq an \log_2 n
\]

when **we choose** \( a = 3c + 2 \);
Chapter 4. Solving Recurrences

MergeSort has the time complexity:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \] 

with base case \( T(1) = c \)

Use substitution method to prove that \( T(n) = O(n \log_2 n) \).

**Proof.** We claim that there are constants \( a, k \) such that

\[ T(n) \leq an \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]

*step 1. base case: when \( n = 1 \), \( T(1) = c \), to make \( T(1) \leq an \log_2 n \), we need \( n \geq k = 2 \), **So we need to use \( T(2) \) as base case.**

from the recurrence, we known

\[ T(2) \leq 2T(1) + 2 + c = 3c + 2 \leq an \log_2 n \]

when we choose \( a = 3c + 2; \)
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, 

\[ T\left(\frac{n}{2}\right) \leq a\left(\frac{n}{2}\right)^2 \log_2 \left(\frac{n}{2}\right), \]

when $n \geq 4$
step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$
step 2. assumption: for \( n / 2 \), the claim is true, i.e.,

\[
T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \quad \text{when } n \geq 4
\]

step 3. induction:

\[
T(n)
\]
step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a\frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c$$
step 2. assumption: for $n/2$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \quad \text{when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c$$
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c$$
Chapter 4. Solving Recurrences

step 2. assumption: for \( \frac{n}{2} \), the claim is true, i.e.,

\[
T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \quad \text{when } n \geq 4
\]

step 3. induction:

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

\[
= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c = an \log_2 n - (3c+2)n + n + c
\]
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c = an \log_2 n - (3c+2)n + n + c$$

$$= an \log_2 n - 3cn - 2n + n + c$$
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \quad \text{when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c = an \log_2 n - (3c+2)n + n + c$$

$$= an \log_2 n - c3n - 2n + n + c = an \log_2 n + c(1 - 3n) - n$$
step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \text{ when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c = an \log_2 n - (3c + 2)n + n + c$$

$$= an \log_2 n - c3n - 2n + n + c = an \log_2 n + c(1 - 3n) - n \leq an \log_2 n$$

because $c(1 - 3n) - n < 0$
Chapter 4. Solving Recurrences

step 2. assumption: for $\frac{n}{2}$, the claim is true, i.e.,

$$T\left(\frac{n}{2}\right) \leq a \frac{n}{2} \log_2 \frac{n}{2}, \quad \text{when } n \geq 4$$

step 3. induction:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c = 2T\left(\frac{n}{2}\right) + n + c \leq 2a \frac{n}{2} \log_2 \frac{n}{2} + n + c$$

$$= an(\log_2 n - \log_2 2) + n + c = an \log_2 n - an + n + c = an \log_2 n - (3c + 2)n + n + c$$

$$= an \log_2 n - c3n - 2n + n + c = an \log_2 n + c(1 - 3n) - n \leq an \log_2 n$$

because $c(1 - 3n) - n < 0$
Chapter 4. Solving Recurrences

A little review on logarithm functions:

- \( \log_a n + \log_a m = \log_a nm \);
- \( \log_a n^b = b \log_a n \), especially \( \log_a 1^n = -\log_a n \);
- \( a^{\log_a n} = n \);
- \( \log_m a^n = (\log_a n)^m \neq \log_a n^m \).
Chapter 4. Solving Recurrences

A little review on logarithm functions:

\[ \log_a n + \log_a m = \log_a nm; \]
Chapter 4. Solving Recurrences

A little review on logarithm functions:

- \( \log_a n + \log_a m = \log_a nm; \)
- \( \log_a n^b = b \log_a n, \) especially \( \log_a \frac{1}{n} = -\log_a n; \)
Chapter 4. Solving Recurrences

A little review on logarithm functions:

- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;
A little review on logarithm functions:

- \( \log_a n + \log_a m = \log_a nm; \)
- \( \log_a n^b = b \log_a n, \) especially \( \log_a \frac{1}{n} = -\log_a n; \)
- \( a^{\log_a n} = n; \)
- \( \log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n; \)
Chapter 4. Solving Recurrences

A little review on logarithm functions:

- \( \log_a n + \log_a m = \log_a nm \);
- \( \log_a n^b = b \log_a n \), especially \( \log_a \frac{1}{n} = -\log_a n \);
- \( a^{\log_a n} = n \);
- \( \log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n \);
- \( \log_a^m n = (\log_a n)^m \neq \log_a n^m \).
Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n, \quad \text{where } T(1) = 2 \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \quad \text{where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later. Verify with induction:
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n, \text{ where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.
Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n, \quad \text{where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.
Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[ T(2) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \quad \text{where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[ T(2) = \frac{3}{2} T(\lfloor \frac{2}{3} \rfloor) + n = \frac{3}{2} T(\lfloor \frac{4}{3} \rfloor) + 2 \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \quad \text{where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[ T(2) = \frac{3}{2} T(\lfloor \frac{2 \cdot 2}{3} \rfloor) + 2 = \frac{3}{2} T(\lfloor \frac{4}{3} \rfloor) + 2 = \frac{3}{2} T(1) + 2 = 5 \]

We make \( T(2) = 5 \leq cn \log_2 n \) holds for \( c = 3 \), and when \( n \geq k = 2 \),
Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[
T(2) = \frac{3}{2} T(\lfloor \frac{2 \cdot 2}{3} \rfloor) + 2 = \frac{3}{2} T(\lfloor \frac{4}{3} \rfloor) + 2 = \frac{3}{2} T(1) + 2 = 5
\]

We make \( T(2) = 5 \leq cn \log_2 n \) holds for \( c = 3 \), and when \( n \geq k = 2 \),
Chapter 4. Solving Recurrences

Solving recurrence with the **substitution method** (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2 \]

**Guess** \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

**Verify with induction:**

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[ T(2) = \frac{3}{2} T(\lfloor \frac{2}{3} \rfloor) + n = \frac{3}{2} T(\lfloor \frac{4}{3} \rfloor) + 2 = \frac{3}{2} T(1) + 2 = 5 \]

We make \( T(2) = 5 \leq cn \log_2 n \) holds for \( c = 3 \), and when \( n \geq k = 2 \),

Step 2, assumption: assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + n, \text{ where } T(1) = 2 \]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[ T(2) = \frac{3}{2} T\left(\left\lfloor \frac{4}{3} \right\rfloor\right) + 2 = \frac{3}{2} T(1) + 2 = 5 \]

We make \( T(2) = 5 \leq cn \log_2 n \) holds for \( c = 3 \), and when \( n \geq k = 2 \),

Step 2, assumption: assume the guessed upper bound holds for \( \left\lfloor \frac{2n}{3} \right\rfloor \), i.e.,

\[ T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \leq c\left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor \]

Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + n \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[
T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2
\]

Guess \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

Verify with induction:

Step 1, base case: \( T(1) = 2 \leq cn \log_2 n = 0 \) does not hold for the guessed inequality.

Instead, we choose \( T(2) \) to be the base case. From the recurrence, we have

\[
T(2) = \frac{3}{2} T(\lfloor \frac{4}{3} \rfloor) + 2 = \frac{3}{2} T(1) + 2 = 5
\]

We make \( T(2) = 5 \leq cn \log_2 n \) holds for \( c = 3 \), and when \( n \geq k = 2 \),

Step 2, assumption: assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[
T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor
\]

Step 3, substitute it in the recurrence, we get

\[
T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} \left( c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor \right) + n
\]
Chapter 4. Solving Recurrences

Step 3, substitute it in the recurrence, we get

\[
T(n) = \frac{3}{2}T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n
\]
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} \left( c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor \right) + n \]
Chapter 4. Solving Recurrences

Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} \left( c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor \right) + n \]

\[ \leq \frac{3}{2} \left( c \frac{2n}{3} \log_2 \frac{2n}{3} \right) + n \]
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lceil \frac{2n}{3} \rceil) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor) + n \]

\[ \leq \frac{3}{2} (c \frac{2n}{3} \log_2 \frac{2n}{3}) + n \leq cn(\log_2 n + \log_2 \frac{2}{3}) + n \]
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2} \left( c \left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor \right) + n \]

\[ \leq \frac{3}{2} \left( c \frac{2n}{3} \log_2 \frac{2n}{3} \right) + n \leq cn \left( \log_2 n + \log_2 \frac{2}{3} \right) + n \]

\[ \leq cn \left( \log_2 n - \log_2 \frac{3}{2} \right) + n \]
Chapter 4. Solving Recurrences

Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor) + n \]

\[ \leq \frac{3}{2} (c \frac{2n}{3} \log_2 \frac{2n}{3}) + n \leq cn(\log_2 n + \log_2 \frac{2}{3}) + n \]

\[ \leq cn(\log_2 n - \log_2 \frac{3}{2}) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]
Chapter 4. Solving Recurrences

Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor) + n \]

\[ \leq \frac{3}{2} (c \frac{2n}{3} \log_2 \frac{2n}{3}) + n \leq cn (\log_2 n + \log_2 \frac{2}{3}) + n \]

\[ \leq cn (\log_2 n - \log_2 \frac{3}{2}) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]

Since \( \log_2 \frac{3}{2} > 0.5 \),
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor) + n \]

\[ \leq \frac{3}{2} (c \frac{2n}{3} \log_2 \frac{2n}{3}) + n \leq cn(\log_2 n + \log_2 \frac{2}{3}) + n \]

\[ \leq cn(\log_2 n - \log_2 \frac{3}{2}) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]

Since \( \log_2 \frac{3}{2} > 0.5 \), \(-cn \log_2 \frac{3}{2} + n < 0 \) when \( c \geq 2 \).
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2} \left( c \left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor \right) + n \]

\[ \leq \frac{3}{2} \left( c \frac{2n}{3} \log_2 \frac{2n}{3} \right) + n \leq cn(\log_2 n + \log_2 \frac{2}{3}) + n \]

\[ \leq cn(\log_2 n - \log_2 \frac{3}{2}) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]

Since \( \log_2 \frac{3}{2} > 0.5 \), \(-cn \log_2 \frac{3}{2} + n < 0\) when \( c \geq 2 \).

But we already know \( c = 3 \) from the base case proof,
Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T\left(\lfloor \frac{2n}{3} \rfloor \right) + n \leq \frac{3}{2} \left( c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor \right) + n \]

\[ \leq \frac{3}{2} \left( c \frac{2n}{3} \log_2 \frac{2n}{3} \right) + n \leq cn \left( \log_2 n + \log_2 \frac{2}{3} \right) + n \]

\[ \leq cn \left( \log_2 n - \log_2 \frac{3}{2} \right) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]

Since \( \log_2 \frac{3}{2} > 0.5 \), \(-cn \log_2 \frac{3}{2} + n < 0\) when \( c \geq 2 \).

But we already know \( c = 3 \) from the base case proof, we have

\[ T(n) \leq cn \log_2 n - cn \log_2 \frac{3}{2} + n \]
Chapter 4. Solving Recurrences

Step 3, substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor) + n \]

\[ \leq \frac{3}{2} (c \frac{2n}{3} \log_2 \frac{2n}{3}) + n \leq cn(\log_2 n + \log_2 \frac{2}{3}) + n \]

\[ \leq cn(\log_2 n - \log_2 \frac{3}{2}) + n = cn \log_2 n - cn \log_2 \frac{3}{2} + n \]

Since \( \log_2 \frac{3}{2} > 0.5 \), \(-cn \log_2 \frac{3}{2} + n < 0 \) when \( c \geq 2 \).

But we already know \( c = 3 \) from the base case proof, we have

\[ T(n) \leq cn \log_2 n - cn \log_2 \frac{3}{2} + n \leq cn \log_2 n \]
Chapter 4. Solving Recurrences

2. Changing variables

Example:
\[ T(n) = 2T(\sqrt{n}) + \log_2 n \]

Define \( m = \log_2 n \), i.e., \( n = 2^m \).

\[ T(2^m) = 2T(2^{m/2}) + m \]

rename the function:
\[ S(m) = T(2^m) \]

\[ S(m) = 2S(m/2) + m \]

solve it, we have
\[ S(m) = O(m \log m) \]

so
\[ T(n) = T(2^m) = O(m \log m) = O(\log n \log \log n) \].
2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)
2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \),
Chapter 4. Solving Recurrences

2. Changing variables

Example: \[ T(n) = 2T(\sqrt{n}) + \log_2 n \]

Define \( m = \log_2 n \), i.e., \( n = 2^m \)
Chapter 4. Solving Recurrences

2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \), i.e., \( n = 2^m \)

\[ T(2^m) = 2T(2^{m/2}) + m \]
2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \), i.e., \( n = 2^m \)

\( T(2^m) = 2T(2^{m/2}) + m \)

rename the function: \( S(m) = T(2^m) \)
Chapter 4. Solving Recurrences

2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \), i.e., \( n = 2^m \)

\( T(2^m) = 2T(2^{m/2}) + m \)

rename the function: \( S(m) = T(2^m) \)

\( S(m) = 2S(m/2) + m \)
Chapter 4. Solving Recurrences

2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \), i.e., \( n = 2^m \)

\( T(2^m) = 2T(2^{m/2}) + m \)

rename the function: \( S(m) = T(2^m) \)

\( S(m) = 2S(m/2) + m \)

solve it, we have \( S(m) = O(m \log m) \)
Chapter 4. Solving Recurrences

2. Changing variables

Example: \( T(n) = 2T(\sqrt{n}) + \log_2 n \)

Define \( m = \log_2 n \), i.e., \( n = 2^m \)

\[ T(2^m) = 2T(2^{m/2}) + m \]

rename the function: \( S(m) = T(2^m) \)

\[ S(m) = 2S(m/2) + m \]

solve it, we have \( S(m) = O(m \log m) \)

so \( T(n) = T(2^m) = O(m \log m) = O(\log n \log \log n) \).
Chapter 4. Solving Recurrences

3. Recursive tree method
Chapter 4. Solving Recurrences

3. Recursive tree method

By unfolding the recurrence to make a recursive-tree.
3. Recursive tree method

By unfolding the recurrence to make a recursive-tree.

(1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.
3. Recursive tree method

By *unfolding* the recurrence to make a recursive-tree.

(1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.

(2) for each child, replace it with then non-recursive terms and produce children that are then recursive terms.
3. Recursive tree method

By unfolding the recurrence to make a recursive-tree.

(1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.

(2) for each child, replace it with then non-recursive terms and produce children that are then recursive terms

(3) repeat (2), expand the tree until all children are the base case.
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ l_0: \quad T(n) \]
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

| $l_0$ | $T(n)$ | $l_1$ | $T(n/4)$ | $T(n/4)$ | $T(n/4)$ |
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$:

$l_1$:

$T(n/4)$

$T(n/4)$

$T(n/4)$

$n^2$
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

<table>
<thead>
<tr>
<th>$l_0$:</th>
<th>$T(n)$</th>
<th>$T(n)$</th>
<th>$T(n)$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$: $T(n/4)$</td>
<td>$T(n/4)$</td>
<td>$T(n/4)$</td>
<td>$n^2$</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

<table>
<thead>
<tr>
<th>$l_0:$</th>
<th>$T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1:$</td>
<td>$T(n/4)$</td>
</tr>
<tr>
<td>$l_2:$</td>
<td>$T(n/4)$</td>
</tr>
</tbody>
</table>

$n^2$
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

<table>
<thead>
<tr>
<th>Level</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0$</td>
<td>$T(n)$</td>
</tr>
<tr>
<td>$l_1$</td>
<td>$T(n/4)$</td>
</tr>
<tr>
<td>$l_2$</td>
<td>$T(n/16)$</td>
</tr>
</tbody>
</table>

Then $T(n)$ is the sum $T(n) = n^2[1 + 3(1/4)^2 + 3(2/4)^2 + 3(3/4)^2 + \cdots + 3(m-1/4)^2] + 3mT(1) = n^2[1 + 3(1/4)^2 + 3(2/4)^2 + 3(3/4)^2 + \cdots + 3[(m-1)/4]^2] + 3mT(1)$ for all $n > 0$. 

$n^2 \leq 16 \frac{13}{16} n^2$ for all $n > 0$. 

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: 

\[
\begin{array}{cccc}
T(n) & & & \\
l_1: & T(n/4) & T(n/4) & T(n/4) \\
l_2: & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & 3(n/4^2)
\end{array}
\]

\[n^2\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$:

$l_1$:

$l_2$:

$l_3$:

\[
\begin{array}{cccccc}
T(n) & T(n) & T(n/4) & T(n/4) & T(n/4) & n^2 \\
T(n/4) & T(n/4) & T(n/4) & T(n/4) & T(n/4) & 3(n/4)^2 \\
T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & 3^2(n/4^2)^2 \\
\end{array}
\]
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$L_0$: $T(n)$
$L_1$: $T(n/4)$
$L_2$: $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$
$L_3$: ...
$L_4$: ...

\[ T(n) = \sum_{i=0}^{\log_4 n} 3^i \cdot \left( \frac{n}{4^i} \right)^2 \]

for all $n > 0$. 

\[ T(n) \leq n^2 \sum_{i=0}^{\log_4 n} \left( \frac{3}{4} \right)^i = n^2 \left( 1 - \left( \frac{3}{4} \right)^{\log_4 n} \right) \leq n^2 \cdot \frac{4}{1 - \frac{3}{4}} = 16n^2 \]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{array}{cccccccccc}
  l_0: & T(n) & & & & & & & & \\
  l_1: & T(n/4) & T(n/4) & & & & & & & \\
  l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & & & & & & \\
  l_3: & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
  l_4: & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]

$3(\frac{n}{4})^2$ $3^2(\frac{n}{4^2})^2$ $3^3(\frac{n}{4^3})^2$
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$:

$l_1$:

$l_2$:

$l_3$: ...

$l_4$: ...

$l_5$: ...
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\begin{align*}
l_0: & & T(n) \\
l_1: & & T(n/4) & & T(n/4) \\
l_2: & & T(n/4^2) & & T(n/4^2) & & T(n/4^2) & & T(n/4^2) \\
l_3: & & \ldots \\
l_4: & & \ldots \\
l_5: & & \ldots \\
l_{m-1}: & & \ldots \\
\end{align*}

where $n/4^m = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{n}{4} \right)^2 + 3^2 \left( \frac{n}{4^2} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 3^m T(1). \]

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{n}{4} \right)^2 + 3^2 \left( \frac{n}{4^2} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 3^m \times 1 \]

\[ = n^2 \left[ 1 + 3 \left( \frac{n}{4} \right)^2 + 3^2 \left( \frac{n}{4^2} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 3^m \times \frac{1}{1-\frac{3}{4}} \]

\[ = n^2 \left[ 1 + 3 \left( \frac{n}{4} \right)^2 + 3^2 \left( \frac{n}{4^2} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 16 \times \frac{1}{1-\frac{3}{4}} \]

\[ \leq n^2 \left[ 1 + 3 \left( \frac{n}{4} \right)^2 + 3^2 \left( \frac{n}{4^2} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 16 \times \frac{1}{1-\frac{3}{4}} \]

for all $n > 0$. 
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ l_0: \quad T(n/4) \quad T(n/4) \quad T(n) \]
\[ l_1: \quad T(n/4) \quad T(n/4) \quad T(n/4) \quad n^2 \]
\[ l_2: \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad 3(n/4)^2 \]
\[ l_3: \quad \ldots \]
\[ l_4: \quad \ldots \]
\[ l_5: \quad \ldots \]
\[ l_{m-1}: \quad \ldots \]
\[ l_m: \quad \ldots \quad 3^{m-2} \left( \frac{n}{4^{m-2}} \right)^2 \]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ l_0: \quad T(n/4) \quad T(n/4) \quad T(n/4) \quad n^2 \]
\[ l_1: \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad \frac{3(n/4)^2}{n^2} \]
\[ l_2: \quad 3^2 \left( \frac{n}{4^2} \right)^2 \quad 3^3 \left( \frac{n}{4^3} \right)^2 \]
\[ l_3: \quad \ldots \quad \ldots \quad \ldots \]
\[ lm-1: \quad \ldots \quad \ldots \quad \ldots \quad 3^{m-2} \left( \frac{n}{4^{m-2}} \right)^2 \]
\[ lm: \quad T(1), T(1), T(1), T(1), T(1), \ldots, T(1) \quad 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{align*}
  l_0: & & T(n) \\
  l_1: & & T(n/4) & & T(n/4) \\
  l_2: & & T(n/4^2) & & T(n/4^2) & & T(n/4^2) \\
  l_3: & & & & & & & \\
  l_4: & & & & & & & \\
  l_5: & & & & & & & \\
  l_{m-1}: & & & & & & & \\
  l_m: & & T(1), T(1), T(1), T(1), T(1), \ldots, T(1) & & & & & & & & & & & \\
\end{align*}
\]

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$. 

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$l_0: \quad T(n)$$
$$l_1: \quad T(n/4) \quad T(n/4) \quad T(n/4) \quad n^2$$
$$l_2: \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad T(\frac{n}{4^2}) \quad 3(\frac{n}{4})^2$$
$$l_3: \quad \ldots$$
$$l_4: \quad \ldots$$
$$l_5: \quad \ldots$$
$$l_{m-1}: \quad \ldots$$
$$l_m: \quad T(1), T(1), T(1), T(1), T(1), \ldots, T(1)$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1)$$
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: $T(n)$  

$l_1$: $T(n/4)$  

$l_2$: $T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2)$  

$l_3$: $\ldots$  

$l_4$: $\ldots$  

$l_5$: $\ldots$  

$l_{m-1}$: $\ldots$  

$l_m$: $T(1), T(1), T(1), T(1), T(1), \ldots, T(1)$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1)$

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1$
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ l_0: \]
\[ l_1: \]
\[ l_2: T(n/4) T(n/4) T(n/4) \]
\[ l_3: \cdots \]
\[ l_4: \cdots \]
\[ l_5: \cdots \]
\[ l_{m-1}: \cdots \]
\[ l_m: T(1), T(1), T(1), T(1), T(1), \ldots, T(1) \]

where \( \frac{n}{4^m} = 1 \), i.e., \( m = \log_4 n \).

Then $T(n)$ is the sum

\[
T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m T(1)
\]

\[
T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \times 1
\]

\[
= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \left(\frac{n}{4^m}\right)^2
\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: $T(n)$

$l_1$: $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

$l_2$: $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $n^2$

$l_3$: $3(n/4)^2$

$l_4$: $3^2(n/4)^2$

$l_5$: $3^3(n/4)^2$

$l_{m-1}$: $3^{m-2}(n/4m-2)^2$

$l_m$: $3^{m-1}(n/4m-1)^2$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4})^2 + 3^3(\frac{1}{4^2})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1)$

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1$

$= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m(\frac{n}{4^m})^2$

$= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \cdots + (\frac{3}{16})^m]$
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{align*}
  l_0: & & T(n) \\
  l_1: & & T(n/4) \quad T(n/4) \\
  l_2: & & T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \\
  l_3: & & \ldots \ldots \\
  l_4: & & \ldots \ldots \\
  l_5: & & \ldots \ldots \\
  l_{m-1}: & & \ldots \ldots \\
  l_m: & & T(1), T(1), T(1), T(1), T(1), \ldots, T(1)
\end{align*}
\]

where $n/4^m = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[
\begin{align*}
  T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
  T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
  &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m(\frac{n}{4^m})^2 \\
  &= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \cdots + (\frac{3}{16})^m] \\
  &= n^2(\frac{1-(\frac{3}{16})^{m+1}}{1-\frac{3}{16}})
\end{align*}
\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{align*}
  l_0: & \quad T(n) \\
  l_1: & \quad T(n/4) \\
  l_2: & \quad T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)T(n/4^2)3(n/4)^2 \\
  l_3: & \quad \ldots \ldots \\
  l_4: & \quad \ldots \ldots \\
  l_5: & \quad \ldots \ldots \\
  l_{m-1}: & \quad \ldots \ldots \\
  l_m: & \quad T(1), T(1), T(1), T(1), T(1), \ldots, T(1)
\end{align*}
\]

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[
T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m(\frac{n}{4^m})^2 \\
= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \cdots + (\frac{3}{16})^m] \\
= n^2(\frac{1-(\frac{3}{16})^{m+1}}{1-\frac{3}{16}}) \\
\leq n^2(\frac{1}{1-\frac{3}{16}})
\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: $T(n)$
$l_1$: $T(n/4)$ $T(n)$
$l_2$: $T(n/4^2)$ $T(n/4)$ $T(n)$
$l_3$: $T(n/4^4)$ $T(n/4^2)$ $T(n/4)$ $T(n)$
$l_4$: $T(n/4^8)$ $T(n/4^4)$ $T(n/4^2)$ $T(n/4)$ $T(n)$
$l_5$: $T(n/4^{16})$ $T(n/4^8)$ $T(n/4^4)$ $T(n/4^2)$ $T(n/4)$ $T(n)$
$l_{m-1}$: $T(n/4^{2^{m-1}})$ $T(n/4^{2^{m-2}})$ $T(n/4^{2^{m-3}})$ $\ldots$ $T(n/4)$ $T(n)$
$l_m$: $T(1), T(1), T(1), T(1), T(1), \ldots, T(1)$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m T(1)$

$T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \times 1$

$= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \left(\frac{n}{4^m}\right)^2$

$= n^2[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \cdots + \left(\frac{3}{16}\right)^m]$

$= n^2\left(\frac{1 - \left(\frac{3}{16}\right)^{m+1}}{1 - \frac{3}{16}}\right)$

$\leq n^2\left(\frac{1}{1 - \frac{3}{16}}\right)$

$= \frac{16}{13} n^2$

for all $n > 0$. 

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\[ l_0: \quad n^2 \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\begin{align*}
  l_0: & & n^2 \\
  l_1: & & (n/4)^2 & & (n/4)^2 & & (n/4)^2
\end{align*}
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\[ l_0: \quad n^2 \]
\[ l_1: \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \]
\[ l_2: \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\begin{align*}
  l_0: & \quad n^2 \\
  l_1: & \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \\
  l_2: & \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \\
  l_3: & \quad \ldots \ldots \\
  \end{align*}
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\[ l_0: \quad n^2 \]
\[ l_1: \quad (n/4)^2 (n/4)^2 (n/4)^2 \]
\[ l_2: \quad (n/4)^2 (n/4)^2 (n/4)^2 (n/4)^2 (n/4)^2 (n/4)^2 \]
\[ l_3: \quad \ldots \]
\[ l_4: \quad \ldots \]

\[ 3^3 \text{ nodes of } (n/4^3)^2 \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

- \( l_0: \) \( n^2 \)
- \( l_1: \) \( (n/4)^2 \) \( (n/4)^2 \) \( (n/4)^2 \)
- \( l_2: \) \( (n/4^2)^2 \) \( (n/4^2)^2 \) \( (n/4^2)^2 \) \( (n/4^2)^2 \) \( (n/4^2)^2 \)
- \( l_3: \ldots \)
- \( l_4: \ldots \)
- \( l_{m-1}: \ldots \)

\( 3^3 \) nodes of \( (n/4^3)^2 \)
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\[ l_0: \]
\[ l_1: \] \[ (n/4)^2 \]
\[ l_2: \] \[ (n/4)^2 \] \[ (n/4)^2 \] \[ n^2 \]
\[ l_3: \ldots \]
\[ l_4: \ldots \]
\[ l_{m-1}: \ldots \]

\[ 3^3 \text{ nodes of } (n/4^3)^2 \]
\[ 3^{m-1} \text{ of } (n/4^{m-1})^2 \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \] with base case \( T(1) = 1 \)

\( l_0: \) 
\( l_1: \) \( (n/4)^2 \) 
\( l_2: \) \( (n/4)^2 \) \( (n/4)^2 \) \( (n/4)^2 \) \( (n/4)^2 \)

\( l_3: \) \ldots

\( l_4: \) \ldots

\( l_{m-1}: \) \ldots

\( l_m: \) \( T(1), T(1), T(1), T(1), T(1), \ldots, \)

\( 3^3 \) nodes of \( (n/4^3)^2 \)

\( 3^{m-1} \) of \( (n/4^{m-1})^2 \)

\( 3^m \) nodes of \( T(1) \)
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

- \( l_0: \) \( n^2 \)
- \( l_1: \) \( (n/4)^2 \) \( (n/4)^2 \)
- \( l_2: \) \( (n/4)^2 \) \( (n/4)^2 \) \( (n/4)^2 \) \( (n/4)^2 \)
- \( l_3: \ldots \)
- \( l_4: \ldots \)
- \( l_{m-1}: \ldots \)
- \( l_m: \) \( T(1), T(1), T(1), T(1), T(1), \ldots \)

3\(^m\) nodes of \( (\frac{n}{4^m})^2 \)

3\(^m-1\) of \( (\frac{n}{4^{m-1}})^2 \)
Another example (page 91 textbook):

Assume time function $T(n)$ of some algorithm has the recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

with base case $T(1) = T(2) = T(3) = c > 0$, a constant. We assume $n$ is a power of 3.
Another example (page 91 textbook):

Assume time function $T(n)$ of some algorithm has the recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

with base case $T(1) = T(2) = T(3) = c > 0$, a constant. We assume $n$ is a power of 3.

(1) using recursive tree method to derive an upper bound
Another example (page 91 textbook):

Assume time function $T(n)$ of some algorithm has the recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

with base case $T(1) = T(2) = T(3) = c > 0$, a constant.

We assume $n$ is a power of 3.

(1) using recursive tree method to derive an upper bound
(2) using substitution method to verify the upper bound
Chapter 4. Solving Recurrences

(1) using recursive tree method to derive an upper bound
Chapter 4. Solving Recurrences

(1) using recursive tree method to derive an upper bound

\(l_0:\)
\[ T(n) \]

\(l_1:\)
\[ T\left(\frac{n}{3}\right) \quad T\left(\frac{2n}{3}\right) \]
Chapter 4. Solving Recurrences

(1) using recursive tree method to derive an upper bound

\[ T(n) \]

\[ T\left(\frac{n}{3}\right) \quad T\left(\frac{2n}{3}\right) \quad n \]

\[ T\left(\frac{n}{3^2}\right) \quad T\left(\frac{2n}{3^2}\right) \quad T\left(\frac{2^2n}{3^2}\right) \quad \frac{n}{3} + \frac{2n}{3} = n \]
(1) using recursive tree method to derive an upper bound

\[ l_0: \quad T(n) \]
\[ l_1: \quad T\left(\frac{n}{3}\right) \quad T\left(\frac{2n}{3}\right) \quad n \]
\[ l_2: \quad T\left(\frac{n}{3^2}\right) \quad T\left(\frac{2n}{3^2}\right) \quad T\left(\frac{2^2n}{3^2}\right) \quad \frac{n}{3} + \frac{2n}{3} = n \]
\[ \vdots \]
\[ l_{m-1}: \quad T\left(\frac{n}{3^{m-1}}\right) \quad \ldots \]
\[ l_m: \quad T\left(\frac{n}{3^m}\right) \quad T\left(\frac{2^{m-1}n}{3^m}\right) \quad n \]
(1) using recursive tree method to derive an upper bound

\[ l_0: \quad T(n) \]
\[ l_1: \quad T\left(\frac{n}{3}\right) \quad T\left(\frac{2n}{3}\right) \quad n \]
\[ l_2: \quad T\left(\frac{n}{3^2}\right) \quad T\left(\frac{2n}{3^2}\right) \quad T\left(\frac{2^2n}{3^2}\right) \quad \frac{n}{3} + \frac{2n}{3} = n \]
\[ \ldots \]
\[ l_{m-1}: \quad T\left(\frac{n}{3^{m-1}}\right) \quad \ldots \quad T\left(\frac{2^{m-1}n}{3^{m-1}}\right) \quad n \]
\[ l_m: \quad T\left(\frac{n}{3^m}\right) \quad T\left(\frac{2n}{3^m}\right) \quad \ldots \quad T\left(\frac{2^mn}{3^m}\right) \quad n \]
(1) using recursive tree method to derive an upper bound

\[ l_0: \quad T(n) \]

\[ l_1: \quad T\left(\frac{n}{3}\right), \quad T\left(\frac{2n}{3}\right) \quad n = \frac{n}{3} + \frac{2n}{3} \]

\[ l_2: \quad T\left(\frac{n}{3^2}\right), \quad T\left(\frac{2n}{3^2}\right), \quad T\left(\frac{2^2n}{3^2}\right) \quad n = \frac{n}{3^2} + \frac{2n}{3^2} + \frac{2^2n}{3^2} \]

\[ \ldots \]

\[ l_{m-1}: \quad T\left(\frac{n}{3^{m-1}}\right) \ldots \quad n = T\left(\frac{2^{m-1}n}{3^{m-1}}\right), \quad T\left(\frac{2^m n}{3^m}\right) \]

\[ l_m: \quad T\left(\frac{n}{3^m}\right), \quad T\left(\frac{2n}{3^m}\right) \ldots \quad n = \frac{n}{3^m} + \frac{2n}{3^m} \]

\[ l_{m+1}: \quad \ldots \quad n = T\left(\frac{2^{m+1}n}{3^{m+1}}\right) \]
Chapter 4. Solving Recurrences

(1) using recursive tree method to derive an upper bound

\[ l_0: \quad T(n) \]
\[ l_1: \quad T\left(\frac{n}{3}\right), \quad T\left(\frac{2n}{3}\right) \]
\[ l_2: \quad T\left(\frac{n}{3^2}\right), \quad T\left(\frac{2n}{3^2}\right), \quad T\left(\frac{2^2n}{3^2}\right) \]
\[ \vdots \]
\[ l_{m-1}: \quad T\left(\frac{n}{3^{m-1}}\right), \quad \ldots \]
\[ l_m: \quad T\left(\frac{n}{3^m}\right), \quad T\left(\frac{2n}{3^m}\right), \quad \ldots \]
\[ l_{m+1}: \quad \ldots \]
\[ \vdots \]
\[ l_r: \quad \ldots \]
(1) using recursive tree method to derive an upper bound

\[
\begin{align*}
    l_0: & \quad T(n) \\
    l_1: & \quad T(\frac{n}{3}) \quad T(\frac{2n}{3}) \\
    l_2: & \quad T(\frac{n}{3^2}) \quad T(\frac{2n}{3^2}) \quad T(\frac{2^2n}{3^2}) \\
    \vdots \\
    l_{m-1}: & \quad T(\frac{n}{3^{m-1}}) \quad \ldots \\
    l_m: & \quad T(\frac{n}{3^m}) \quad T(\frac{2n}{3^m}) \quad \ldots \\
    l_{m+1}: & \quad \ldots \\
    \vdots \\
    l_r: & \quad \ldots \\
\end{align*}
\]

\[
\begin{align*}
    n & = \frac{n}{3} + \frac{2n}{3} \\
    \frac{2^{m-1}n}{3^{m-1}} & < n \\
    \frac{2^r n}{3^r} & < n
\end{align*}
\]
Chapter 4. Solving Recurrences

(1) using recursive tree method to derive an upper bound
(1) using recursive tree method to derive an upper bound

• the leftmost branch is the shortest; the rightmost is the longest;

• \( \frac{n}{3m} = 1, \, m = \log_3 n; \, \frac{2^r n}{3^r} = 1, \, r = \log_3 \frac{2}{n}. \)

• beginning from level \( m + 1 \), some nodes will gradually disappear, so the number of leaves is NOT \( 2^{\log_\frac{3}{2} n} \neq O(n \log_2 n) \) (why?),

• we do not need to give an accurate account for the sum of all quantities in blue color and those at leaves, but only an estimated upper bound.

• we estimate an upper bound to be \( O(n \log_2 n). \)
Chapter 4. Solving Recurrences

(2) using the substitution method to verify the upper bound
Chapter 4. Solving Recurrences

(2) using the substitution method to verify the upper bound

We prove that $T(n) = O(n \log_2 n)$.

That is to prove: $\exists a, k > 0$ such that

$$T(n) \leq an \log_2 n \text{ when } n \geq k$$
Chapter 4. Solving Recurrences

(2) using the substitution method to verify the upper bound

We prove that \( T(n) = O(n \log_2 n) \).

That is to prove: \( \exists a, k > 0 \) such that

\[
T(n) \leq an \log_2 n \quad \text{when } n \geq k
\]

Base case: for \( n = 3 \), \( T(3) = c \leq a3 \log_2 3 \) will be true if we choose \( a \geq c \) because \( \log_2 3 > 1 \).
Chapter 4. Solving Recurrences

(2) using the substitution method to verify the upper bound

We prove that $T(n) = O(n \log_2 n)$.

That is to prove: $\exists a, k > 0$ such that

$$T(n) \leq an \log_2 n \text{ when } n \geq k$$

Base case: for $n = 3$, $T(3) = c \leq a3 \log_2 3$ will be true if we choose $a \geq c$ because $\log_2 3 > 1$.

Assume that $T\left(\frac{n}{3}\right) \leq a \frac{n}{3} \log_2 \frac{n}{3}$; and $T\left(\frac{2n}{3}\right) \leq a \frac{2n}{3} \log_2 \frac{2n}{3}$;
Chapter 4. Solving Recurrences

(2) using the substitution method to verify the upper bound
We prove that \( T(n) = O(n \log_2 n) \).

That is to prove: \( \exists a, k > 0 \) such that

\[
T(n) \leq an \log_2 n \quad \text{when } n \geq k
\]

Base case: for \( n = 3 \), \( T(3) = c \leq a3 \log_2 3 \) will be true if we choose \( a \geq c \) because \( \log_2 3 > 1 \).

Assume that \( T\left(\frac{n}{3}\right) \leq a \frac{n}{3} \log_2 \frac{n}{3} \); and \( T\left(\frac{2n}{3}\right) \leq a \frac{2n}{3} \log_2 \frac{2n}{3} \);

Then

\[
T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n
\]

\[
\leq a \frac{n}{3} \log_2 \frac{n}{3} + a \frac{2n}{3} \log_2 \frac{2n}{3} + n
\]

\[
= \frac{an}{3} \log_2 n + \frac{an}{3} \log_2 \frac{1}{3} + \frac{2an}{3} \log_2 n + \frac{2an}{3} \log_2 \frac{2}{3} + n
\]

\[
= an \log_2 n - \frac{an}{3} \left(\log_2 3 + 2 \log_2 \frac{3}{2}\right) + n
\]
Therefore,

\[ T(n) \leq an \log_2 n - \frac{an}{3} \left( \log_2 3 + \log_2 \frac{9}{4} \right) + n \]

\[ = an \log_2 n - \frac{an}{3} \log_2 \frac{27}{4} + n \]

\[ \leq an \log_2 n \]  \hspace{1cm} (2)

when \( a \) is chosen such that \( a \geq \max\{3, c\} \), since it makes

\[ -\frac{an}{3} \log_2 \frac{27}{4} + n \leq 0 \]
Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms
Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;
Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;
- With assumption that input data are in a probabilistic distribution
Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;
- With assumption that input data are in a probabilistic distribution
- Close relationship with randomized algorithms
Chapter 5. Probabilistic Analysis of Algorithms

A good example: **Quick Sort** algorithm [Hoare'1959]
A good example: **Quick Sort** algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$. 
A good example: **Quick Sort** algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$. 

Alternatively, we can enforce the desired distribution by using randomness (tossing coins) in the algorithms. That is, we use randomized algorithms.
Chapter 5. Probabilistic Analysis of Algorithms

A good example: Quick Sort algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn\log_2n$ for a constant $b > 0$. In other word, it is efficient in most cases;
A good example: **Quick Sort** algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

In other words, it is efficient in most cases;
assumption: the input data are of the uniform distribution.
A good example: Quick Sort algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

In other words, it is efficient in most cases;
assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.
A good example: Quick Sort algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

In other words, it is efficient in most cases; assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.

Alternatively, we can enforce the desired distribution by using randomness (tossing coins) in the algorithms.
A good example: **Quick Sort** algorithm [Hoare’1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

  In other word, it is efficient in most cases; assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.

  Alternatively, we can **enforce the desired distribution by using randomness (tossing coins)** in the algorithms.

  That is, we use **randomized algorithms**.
Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms:

- actions in the algorithm are considered random events.
In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms:

- actions in the algorithm are considered random events
- such random events are driven by random data
In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms:

- actions in the algorithm are considered random events
- such random events are driven by random data which are either input data or randomly tossed coins
Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms:

- actions in the algorithm are considered random events
- such random events are driven by random data which are either input data or randomly tossed coins
- So the running time come with a probability
In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms

1. actions in the algorithm are considered random events
2. such random events are driven by random data which are either input data or randomly tossed coins
3. So the running time comes with a probability you can compute the expected time (i.e., averaged time)
In both probabilistic analysis of deterministic algorithms and analysis of randomized algorithms

- actions in the algorithm are considered random events
- such random events are driven by random data which are either input data or randomly tossed coins
- So the running time come with a probability you can compute the expected time (i.e., averaged time)
There are two types of randomized algorithms:

- Las Vegas algorithms
- Monte Carlo algorithms
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;

- Monte Carlo algorithms
  - on 'NO' instances, 100% accuracy; \( \text{Prob}(\text{to answer 'NO' on 'NO' instance}) = 1 \)
  - on 'YES' instances, \( \geq 75\% \) accuracy; \( \text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75 \)

Accuracy 75% can be improved to 99.9% with multiple trials.

Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;
  - running time comes with a probability distribution

- Monte Carlo algorithms
  - on 'NO' instances, 100% accuracy;
  - probability of answering 'NO' on 'NO' instance = 1
  - on 'YES' instances, ≥ 75% accuracy;
  - probability of answering 'YES' on 'YES' instance ≥ 0.75

Accuracy 75% can be improved to 99.99% with multiple trials.

Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- **Las Vegas algorithms**
  - always gives answer correctly;
  - running time comes with a probability distribution
  e.g., for `QUICKSORT`, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$

- **Monte Carlo algorithms**
  - on 'NO' instances, 100\% accuracy;
  - on 'YES' instances, \geq 75\% accuracy;
    $\text{Prob}(\text{to answer } '\text{YES}' \text{ on } '\text{YES}' \text{ instance}) \geq 0.75$

Accuracy 75\% can be improved to 99.99\% with multiple trials.

• Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, \( \text{Prob}(T(n) \leq cn \log n) \geq 0.75 \)

- Monte Carlo algorithms

Accuracy 75% can be improved to 99.9% with multiple trials.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- **Las Vegas algorithms**
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, \( \text{Prob}(T(n) \leq cn \log n) \geq 0.75 \)

- **Monte Carlo algorithms**
  - on 'NO' instances, 100% accuracy;
  - on 'YES' instances, \( \geq 75\% \) accuracy;
    \( \text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75 \)

Accuracy 75\% can be improved to 99.9\% with multiple trials.

- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$

- Monte Carlo algorithms
  - on 'NO' instances, 100% accuracy;
    $\text{Prob( to answer 'NO' on 'NO' instance)} = 1$
  - on 'YES' instances, $\geq 75\%$ accuracy;
    $\text{Prob( to answer 'YES' on 'YES' instance)} \geq 0.75$

Accuracy 75\% can be improved to 99.99\% with multiple trials.

• Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- **Las Vegas algorithms**
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QuickSort, \( \text{Prob}(T(n) \leq cn \log n) \geq 0.75 \)

- **Monte Carlo algorithms**
  - on 'NO' instances, 100% accuracy;
    \( \text{Prob( to answer 'NO' on 'NO' instance) } = 1 \)
  - on 'YES' instances, \( \geq 75\% \) accuracy;
  - Accuracy 75% can be improved to 99.99% with multiple trials.

- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$

- Monte Carlo algorithms
  - on 'NO' instances, 100% accuracy;
    $\text{Prob}(\text{to answer 'NO' on 'NO' instance}) = 1$
  - on 'YES' instances, $\geq 75\%$ accuracy;
    $\text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$

Accuracy 75% can be improved to 99.99% with multiple trials.

- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- **Las Vegas algorithms**
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$

- **Monte Carlo algorithms**
  - on 'NO' instances, 100% accuracy;
    $\text{Prob}(\text{to answer 'NO' on 'NO' instance}) = 1$
  - on 'YES' instances, $\geq 75\%$ accuracy;
    $\text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$

Accuracy 75% can be improved to 99.99% with multiple trials.
There are two types of randomized algorithms:

- **Las Vegas algorithms**
  - always gives answer correctly;
  - running time comes with a probability distribution
    e.g., for QUICKSORT, \( \text{Prob}(T(n) \leq cn \log n) \geq 0.75 \)

- **Monte Carlo algorithms**
  - on 'NO' instances, 100% accuracy;
    \( \text{Prob}(\text{to answer 'NO' on 'NO' instance}) = 1 \)
  - on 'YES' instances, \( \geq 75\% \) accuracy;
    \( \text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75 \)

  Accuracy 75\% can be improved to 99.99\% with multiple trials.

- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.