CSCI 4470/6470 Algorithms, Spring 2018

Liming Cai
Department of Computer Science, UGA

Syllabus: http://cobweb.cs.uga.edu/~cai/courses/algo/2017Fall/

January 3, 2018
An Introduction to the Introduction
An Introduction to the Introduction

Sequence Homology Reveals Functions

- Homology reveals evolution of structure/function

<table>
<thead>
<tr>
<th></th>
<th>FOS_RAT</th>
<th>FOS_MOUSE</th>
<th>FOS_CHICK</th>
<th>FOSB_MOUSE</th>
<th>FOSB_HUMAN</th>
<th>Consensus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MMFSGFNADYEASSSRS</td>
<td>MMFSGFNADYEASSSRS</td>
<td>MMFSGFNADYEASSSRS</td>
<td>-MFQAPFGDYS-</td>
<td>-MFQAPFGDYS-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSASPAGDSL</td>
<td>SSASPAGDSL</td>
<td>SSASPAGDSL</td>
<td>GSRCS3-SPSAE</td>
<td>GSRCS3-SPSAE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSYYHS</td>
<td>LSYYHS</td>
<td>LSYYHS</td>
<td>ESQ--YLSSVD</td>
<td>ESQ--YLSSVD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PADFS</td>
<td>PADFS</td>
<td>PADFS</td>
<td>GFSPPTAA</td>
<td>GFSPPTAA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSMGSPVNTQDFCAD</td>
<td>SSMGSPVNTQDFCAD</td>
<td>SSMGSPVNTQDFCAD</td>
<td>D1.6VSSANF</td>
<td>D1.6VSSANF</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>54</td>
<td>54</td>
<td></td>
</tr>
</tbody>
</table>

- Homology reveals regulatory structure (E. Coli promoters)
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)

| FOS_CHICK | MMYQGFAGEYEAPSSRCSSASPDGSLLTYYPSPADSFSMSVPNSQDFCDLAVSSANF 60 |
| FOSB_MOUSE | MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAAASQE-CAGLGMPSGF 54 |
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)

- analogy in text searching/matching

```
FOS_CHICK: MMYQGFAGEYEAPSSRCSSASPAGDSLTYYPSPADSFSSMGSPVNSQDFCRTLAVSSANF 60
FOSB_MOUSE: MFQAFPDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAASQEQ-CAGLMEMPGSF 54
```

- requirement to design "smarter" algorithms that run much faster

- we will study various techniques for efficient algorithm design.
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)
  
  \[
  \text{FOS\_CHICK: } \text{MMYQGFAGEYEAPSSRCSSASPAGDLSLTYYPSPADSFSSMGSPVNSQDFCTDLAVSSANF 60}
  \]

  \[
  \text{FOSB\_MOUSE: } \text{-MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAASQE-CAGLGEMPGSF 54}
  \]

- analogy in text searching/matching
  but allowing substitutions, insertions, deletions
An Introduction to the Introduction

• main task: sequence comparison (consider just 2 sequences)

• analogy in text searching/matching
  but allowing substitutions, insertions, deletions

• total number of alignments $\geq 2^n$. 
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)

- analogy in text searching/matching but allowing substitutions, insertions, deletions

- total number of number of alignments $\geq 2^n$. To find the most plausible one, cannot afford to try them all
An Introduction to the Introduction

• main task: sequence comparison (consider just 2 sequences)

| FOS_CHICK | MMYQGFAGEYEAPSSRCSSASPAGDSLTYYPSPADSFSSMGSPVNSQDFCTDLAVSSANF 60 |
| FOSB_MOUSE | MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSVDSFGSPPTAAASQE-CAGLGEMPFSF 54 |

• analogy in text searching/matching
  but allowing substitutions, insertions, deletions

• total number of alignments $\geq 2^n$.
  to find the most plausible one, cannot afford to try them all

• require to design “smarter” algorithms that run much faster
An Introduction to the Introduction

- main task: sequence comparison (consider just 2 sequences)

- analogy in text searching/matching but allowing substitutions, insertions, deletions

- total number of number of alignments $\geq 2^n$. to find the most plausible one, cannot afford to try them all

- require to design “smarter” algorithms that run much faster

We will study various techniques for efficient algorithm design.
An Introduction to the Introduction
An Introduction to the Introduction
An Introduction to the Introduction
An Introduction to the Introduction

- 128 players in total;
- 127 matches were played to determine who was the champion.

Is it possible to just play fewer matches?
• 128 players in total;
• 128 players in total;
• 127 matches were played to determine who was the champion.
• 128 players in total;
• 127 matches were played to determine who was the champion.
• Is it possible to just play fewer matches?
An Introduction to the Introduction

Sounds crazy ... but if I don't do it, my job won't be secure...

See if you can reduce the number of matches!

Office of Director
USTA
You were called to answer this question;
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were necessary,
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were necessary, and finally came up with a mathematical proof for that.
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were necessary, and finally came up with a mathematical proof for that.

Sir, this won't happen. I have a mathematical proof that it needs 127 matches!

Hmmm, he is smart... I should give him more work to do.
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were necessary, and finally came up with a mathematical proof for that.
An Introduction to the Introduction

Same problems

Input:
Output:
Solution:
Time Efficiency:

TENNIS TOURNAMENT
128 players
Champion
a match scheme
number of matches

FINDING MAXIMUM
n numbers
the maximum number
an algorithm
number of comparisons
## An Introduction to the Introduction

<table>
<thead>
<tr>
<th>Same problems</th>
<th><strong>Tennis Tournament</strong></th>
<th><strong>Finding Maximum</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>128 players</td>
<td>$n$ numbers</td>
</tr>
<tr>
<td>Output:</td>
<td>Champion</td>
<td>the maximum number</td>
</tr>
<tr>
<td>Solution:</td>
<td>a match scheme</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Time Efficiency:</td>
<td>number of matches</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:
An Introduction to the Introduction

<table>
<thead>
<tr>
<th>Same problems</th>
<th><strong>TENNIS TOURNAMENT</strong></th>
<th><strong>Finding Maximum</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>128 players</td>
<td>$n$ numbers</td>
</tr>
<tr>
<td>Output:</td>
<td>Champion</td>
<td>the maximum number</td>
</tr>
<tr>
<td>Solution:</td>
<td>a match scheme</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Time Efficiency:</td>
<td>number of matches</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:

- **tried various algorithms** for **TENNIS TOURNAMENT** before giving up;
An Introduction to the Introduction

<table>
<thead>
<tr>
<th>Same problems</th>
<th>TENNIS TOURNAMENT</th>
<th>Finding Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>128 players</td>
<td>$n$ numbers</td>
</tr>
<tr>
<td>Output:</td>
<td>Champion</td>
<td>the maximum number</td>
</tr>
<tr>
<td>Solution:</td>
<td>a match scheme</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Time Efficiency:</td>
<td>number of matches</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:

- **tried various algorithms** for **TENNIS TOURNAMENT** before giving up;
- **proved that more efficient algorithms do not exist**;
An Introduction to the Introduction

<table>
<thead>
<tr>
<th>Same problems</th>
<th><strong>Tennis Tournament</strong></th>
<th><strong>Finding Maximum</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>128 players</td>
<td>$n$ numbers</td>
</tr>
<tr>
<td>Output:</td>
<td>Champion</td>
<td>the maximum number</td>
</tr>
<tr>
<td>Solution:</td>
<td>a match scheme</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Time Efficiency:</td>
<td>number of matches</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:

- **tried various algorithms** for **Tennis Tournament** before giving up;
- proved that more efficient algorithms do not exist;
- actually you proved the “USTA algorithm” was already the optimal.
An Introduction to the Introduction

Same problems  |  **Tennis Tournament**  |  **Finding Maximum**
---|---|---
Input:  |  128 players  |  $n$ numbers
Output:  |  Champion  |  the maximum number
Solution:  |  a match scheme  |  an algorithm
Time Efficiency:  |  number of matches  |  number of comparisons

You actually accomplished two tasks:

- **tried various algorithms** for **Tennis Tournament** before giving up;
- proved that more efficient algorithms do not exist;
- actually you proved the “USTA algorithm” was already the optimal.

We will learn how to prove that an algorithm is already the best.
An Introduction to the Introduction

Toss a coin over the phone
Toss a coin over the phone

- how does one person know the other is telling the truth?
An Introduction to the Introduction

Toss a coin over the phone

- how does one person know the other is telling the truth?
- even if the person is, would the first person trust him?
An Introduction to the Introduction

Toss a coin over the phone

- how does one person know the other is telling the truth?
- even if the person is, would the first person trust him?

How to accomplish this task?
An Introduction to the Introduction

They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with an output $Y$ of binary bits (but NOT the input $X$);
- B guessed if the number of 0's in $X$ was odd or even;
They decided the following protocol:

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

- A told B a huge Boolean circuit through the phone; along with an output $Y$ of binary bits (but NOT the input $X$);
They decided the following protocol:

- A told B a huge Boolean circuit through the phone; along with an output $Y$ of binary bits (but NOT the input $X$);
- B guessed if the number of 0's in $X$ was odd or even;
An Introduction to the Introduction

from \( Y \), it would be very difficult to figure out which \( X \) had been used to generate \( Y \) (an intractable problem);

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

We will investigate some intractable problems and the theory behind it.
• from $Y$, it would be very difficult to figure out which $X$ had been used to generate $Y$ (an intractable problem);
from $Y$, it would be very difficult to figure out which $X$ had been used to generate $Y$ (an intractable problem);

So the best B can do is to randomly guess (odd/even), which has the same effect as guessing a coin toss.
An Introduction to the Introduction

- from $Y$, it would be very difficult to figure out which $X$ had been used to generate $Y$ (an intractable problem);
- So the best $B$ can do is to randomly guess (odd/even), which has the same effect as guessing a coin toss.

We will investigate some intractable problems and the theory behind it.
An Introduction to the Introduction

Now we are back to the tennis tournament problem.
An Introduction to the Introduction

Now we are back to the tennis tournament problem.

Same problems

Input: 128 players
Output: Champion
Solution: a match scheme
Time Efficiency: number of matches

**Tennis Tournament**

Finding Maximum

Input: \( n \) numbers
Output: the maximum number
Solution: an algorithm
Time Efficiency: number of comparisons
Now we are back to the tennis tournament problem.

Same problems

\begin{align*}
\text{Input:} & \quad 128 \text{ players} \quad & \text{Output:} & \quad \text{Champion} \\
\text{Solution:} & \quad \text{a match scheme} \quad & \text{Solution:} & \quad \text{an algorithm} \\
\text{Time Efficiency:} & \quad \text{number of matches} \quad & \text{Time Efficiency:} & \quad \text{number of comparisons}
\end{align*}

You actually accomplished two tasks:
Now we are back to the tennis tournament problem.

Same problems | **TENNIS TOURNAMENT** | **FINDING MAXIMUM**
---|---|---
Input: | 128 players | \( n \) numbers
Output: | Champion | the maximum number
Solution: | a match scheme | an algorithm
Time Efficiency: | number of matches | number of comparisons

You actually accomplished two tasks:

- tried various algorithms for **TENNIS TOURNAMENT** before giving up;
An Introduction to the Introduction

Now we are back to the tennis tournament problem.

Same problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input</th>
<th>Output</th>
<th>Solution</th>
<th>Time Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>TENNIS TOURNAMENT</td>
<td>128 players</td>
<td>Champion</td>
<td>a match scheme</td>
<td>number of matches</td>
</tr>
<tr>
<td>FINDING MAXIMUM</td>
<td>$n$ numbers</td>
<td>the maximum number</td>
<td>an algorithm</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:

- tried various algorithms for TENNIS TOURNAMENT before giving up;
- proved that more efficient algorithms do not exist;
Now we are back to the tennis tournament problem.

<table>
<thead>
<tr>
<th>Same problems</th>
<th>TENNIS TOURNAMENT</th>
<th>FINDING MAXIMUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>128 players</td>
<td>( n ) numbers</td>
</tr>
<tr>
<td>Output:</td>
<td>Champion</td>
<td>the maximum number</td>
</tr>
<tr>
<td>Solution:</td>
<td>a match scheme</td>
<td>an algorithm</td>
</tr>
<tr>
<td>Time Efficiency:</td>
<td>number of matches</td>
<td>number of comparisons</td>
</tr>
</tbody>
</table>

You actually accomplished two tasks:

- **tried various algorithms** for **TENNIS TOURNAMENT** before giving up;
- **proved that more efficient algorithms do not exist**;
- actually you proved the “USTA algorithm” was already the optimal.
An Introduction to the Introduction

- Algorithm A
- Algorithm Z
- USTA algorithm

**Number of matches**

- USTA algorithm gives an upper bound ($\leq 127$)
- No algorithm uses fewer than 127 matches
- Your proof gives a lower bound ($\geq 127$)

The Tennis Tournament (128) Problem
An Introduction to the Introduction

Definitions of complexity upper and lower bounds;
An Introduction to the Introduction

Definitions of complexity upper and lower bounds;

- **Time complexity upper bound** is a time threshold _in_ which all instances of a problem _can_ be solved.
Definitions of complexity upper and lower bounds;

- **Time complexity upper bound** is a time threshold *in* which all instances of a problem *can* be solved.

- **Time complexity lower bound** is a time threshold *below* which some instances of the problem *cannot* be solved.
An Introduction to the Introduction

Definitions of complexity upper and lower bounds;

- **Time complexity upper bound** is a time threshold *in* which all instances of a problem *can* be solved.

- **Time complexity lower bound** is a time threshold *below* which some instances of the problem *cannot* be solved.

- These notions apply to both problems and algorithms.
In general, for a given problem $\Pi$,
An Introduction to the Introduction

In general, for a given problem $\Pi$,

- an algorithm $A$ for $\Pi$ also gives a time upper bound $T_A$ for $\Pi$, where $T_A$ is the time complexity of algorithm $A$;
In general, for a given problem $\Pi$,

- an algorithm $A$ for $\Pi$ also gives a time upper bound $T_A$ for $\Pi$, where $T_A$ is the time complexity of algorithm $A$;

- a faster algorithm $B$ for $\Pi$, with $T_B < T_A$, gives a tighter upper bound $T_B$ for $\Pi$;
An Introduction to the Introduction

In general, for a given problem \( \Pi \),

- an algorithm \( A \) for \( \Pi \) also gives a time upper bound \( T_A \) for \( \Pi \), where \( T_A \) is the time complexity of algorithm \( A \);

  a faster algorithm \( B \) for \( \Pi \), with \( T_B < T_A \), gives a tighter upper bound \( T_B \) for \( \Pi \);

- a proof that \( \Pi \) cannot have algorithms faster than time \( T \),
An Introduction to the Introduction

In general, for a given problem $\Pi$,

- an algorithm $A$ for $\Pi$ also gives a time \textit{upper bound} $T_A$ for $\Pi$, where $T_A$ is the time complexity of algorithm $A$;

  a faster algorithm $B$ for $\Pi$, with $T_B < T_A$, gives a \textit{tighter upper bound} $T_B$ for $\Pi$;

- a proof that $\Pi$ cannot have algorithms faster than time $T$, gives a \textit{lower bound} $T$ for $\Pi$;
An Introduction to the Introduction

In general, for a given problem \( \Pi \),

- an algorithm \( A \) for \( \Pi \) also gives a time upper bound \( T_A \) for \( \Pi \), where \( T_A \) is the time complexity of algorithm \( A \);

  a faster algorithm \( B \) for \( \Pi \), with \( T_B < T_A \), gives a
tighter upper bound \( T_B \) for \( \Pi \);

- a proof that \( \Pi \) cannot have algorithms faster than time \( T \),
gives a lower bound \( T \) for \( \Pi \);

  a proof that \( \Pi \) cannot have algorithms faster than time \( S \), with \( S > T \),
In general, for a given problem $\Pi$,

- an algorithm $A$ for $\Pi$ also gives a time upper bound $T_A$ for $\Pi$, where $T_A$ is the time complexity of algorithm $A$;

  a faster algorithm $B$ for $\Pi$, with $T_B < T_A$, gives a tighter upper bound $T_B$ for $\Pi$;

- a proof that $\Pi$ cannot have algorithms faster than time $T$, gives a lower bound $T$ for $\Pi$;

  a proof that $\Pi$ cannot have algorithms faster than time $S$, with $S > T$, gives a tighter lower bound $S$ for $\Pi$. 


In general, for a given problem \( \Pi \),

- an algorithm \( A \) for \( \Pi \) also gives a time upper bound \( T_A \) for \( \Pi \), where \( T_A \) is the time complexity of algorithm \( A \);
  
a faster algorithm \( B \) for \( \Pi \), with \( T_B < T_A \), gives a tighter upper bound \( T_B \) for \( \Pi \);

- a proof that \( \Pi \) cannot have algorithms faster than time \( T \),
  gives a lower bound \( T \) for \( \Pi \);
  
a proof that \( \Pi \) cannot have algorithms faster than time \( S \), with \( S > T \),
  gives a tighter lower bound \( S \) for \( \Pi \).

- for problem \( \Pi \), time lower bounds \( \leq \) time upper bounds,
In general, for a given problem $\Pi$,

- an algorithm $A$ for $\Pi$ also gives a time upper bound $T_A$ for $\Pi$, where $T_A$ is the time complexity of algorithm $A$;

  a faster algorithm $B$ for $\Pi$, with $T_B < T_A$, gives a tighter upper bound $T_B$ for $\Pi$;

- a proof that $\Pi$ cannot have algorithms faster than time $T$, gives a lower bound $T$ for $\Pi$;

  a proof that $\Pi$ cannot have algorithms faster than time $S$, with $S > T$, gives a tighter lower bound $S$ for $\Pi$.

- for problem $\Pi$, time lower bounds $\leq$ time upper bounds;

  when a lower bound is the same as an upper bound, both the lower and upper bounds are called optimal.
In general, for a given problem \( \Pi \),

- an algorithm \( A \) for \( \Pi \) also gives a time upper bound \( T_A \) for \( \Pi \), where \( T_A \) is the time complexity of algorithm \( A \);
  
a faster algorithm \( B \) for \( \Pi \), with \( T_B < T_A \), gives a tighter upper bound \( T_B \) for \( \Pi \);

- a proof that \( \Pi \) cannot have algorithms faster than time \( T \), gives a lower bound \( T \) for \( \Pi \);
  
a proof that \( \Pi \) cannot have algorithms faster than time \( S \), with \( S > T \), gives a tighter lower bound \( S \) for \( \Pi \).

- for problem \( \Pi \), time lower bounds \( \leq \) time upper bounds;
  
  when a lower bound is the same as an upper bound, both the lower and upper bounds are called  \textbf{optimal}.
  Also the corresponding algorithms (that offer the upper bound) are called \textbf{optimal} for \( \Pi \).
An Introduction to the Introduction

Time complexity situation for problem $\Pi$

- Algorithm A
- Algorithm B
- a proved lower bound
- a proved lower bound

$T_A$
$T_B$
$S$
$T$

gap, the smaller the better

0
1
2
An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are
So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds
An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds
   - we need be familiar with techniques for algorithm complexity analysis;
So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds
   - we need to be familiar with techniques for algorithm complexity analysis;
   - we need to master efficient algorithm design skills;
An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds
   - we need be familiar with techniques for algorithm complexity analysis;
   - we need to master efficient algorithm design skills;
2. and to achieve tighter lower bounds,
An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds
   - we need be familiar with techniques for algorithm complexity analysis;
   - we need to master efficient algorithm design skills;

2. and to achieve tighter lower bounds,
   - we need to know the methods for lower bound proofs.
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?

• 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
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,

- 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?
  - 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.
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:

\[
\begin{array}{ccccccccccc}
  n & : & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \ldots \\
  f(n) & : & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & \ldots \\
\end{array}
\]
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*
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;
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:

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

if $n = 1$ or $n = 2$,
Rec-Fibonacci\( (n) \)

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

Rec-Fibonacci \((n)\)

\[
\text{if } n = 1 \text{ or } n = 2, \text{ return } (1); \\
\text{else} \\
T_1 = \text{Rec-Fibonacci}(n - 1);
\]
Part I. Foundations

Rec-Fibonacci($n$)

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

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)

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 ($T_1 + T_2$);

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

• suitable for execution of subroutines
Part I. Foundations

\texttt{Rec-Fibonacci}(n)

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

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

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

\begin{itemize}
    \item suitable for execution of subroutines
    \item but oblivious, cannot remember any completed subroutine.
\end{itemize}
Part I. Foundations

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

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

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

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

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

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

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

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

```plaintext
Rec-Fibonacci(n)
if n = 1 or n = 2, return (1);
else
    T_1 = Rec-Fibonacci(n - 1);
    T_2 = Rec-Fibonacci(n - 2);
    return (T_1 + T_2);
```
Part I. Foundations

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

\[ \text{small triangle} \leq \text{size of tree} \leq \text{large triangle} \]
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($n$) is a simple dynamic programming algorithm.
Iterative-Fibonacci($n$)

if $n = 1$ or $n = 2$ return (1);
Iterative-Fibonacci\( (n) \)

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

\[
\text{else}
\]

\[
M[1] = 1, \ M[2] = 1
\]
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

\[ M[i] = M[i - 1] + M[i - 2] \]
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 ($M[n]$)
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?
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?

\[
T_{total} = \max\{T_{if}, T_{else}\}
\]
ITERATIVE-FIBONACCI\((n)\)

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

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\((n)\)

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

How fast is it?

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

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

Iterative-Fibonacci($n$)

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

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\( (n) \)

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

else

\[
M[1] = 1, \quad M[2] = 1
\]

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($n$)

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($n$)

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.
Part I. Foundations

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

Actually, all the algorithm \texttt{ITERATIVE-FIBONACCI}(n) does is:

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

Actually, all the algorithm $\text{ITERATIVE-FIBONACCI}(n)$ 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.
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 \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

\[ T(n) = c \times n \]
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;

\begin{itemize}
  \item \textbf{INPUT:} list \( X = \{a_1, \cdots, a_n\} \);
  \item \textbf{Body} that is a series of instructions;
  \item \textbf{OUTPUT:} \( y \), the maximum of \( a_1, \cdots, a_n \).
\end{itemize}
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:

\[ f_{\text{max}}(X) = y, \quad \text{where } \forall a \in X, y \geq a, \]
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.
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,
• 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

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

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

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**.
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, 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*)
Chapter 2. Getting Started

Algorithm \texttt{INSERTION-SORT}(A)

1. \textbf{for} \( j = 2 \) \textbf{to} \texttt{length}[A] \textbf{do}

   \begin{itemize}
   \item \texttt{key} = A[j]
   \item \{ \textbf{Insert} A[j] into sorted A[1..j−1] \}
   \item \texttt{i} = j−1
   \item \textbf{while} \( i > 0 \) \textbf{and} A[i] > key \textbf{do}
   \item \qquad A[i+1] = A[i]
   \item \qquad \texttt{i} = i−1
   \item \qquad A[i+1] = key
   \end{itemize}
Chapter 2. Getting Started

Algorithm \textsc{Insertion-Sort}(A)

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

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

1. \textbf{for} $j = 2$ \textbf{to} $\text{length}[A]$ \textbf{do}
2. \hspace{1em} $key = A[j]$
3. \hspace{1em} \{Insert $A[j]$ into sorted $A[1..j - 1]$\}
4. \hspace{1em} $i = j - 1$
Algorithm $\text{INSERTION-SORT}(A)$

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

1. \textbf{for} $j = 2$ to $\text{length}[A]$ \textbf{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. \hspace{1em} \textbf{while} $i > 0$ and $A[i] > key$
6. \hspace{1em} \hspace{1em} \textbf{do} $A[i+1] = A[i]$

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

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.   $i = i - 1$
Chapter 2. Getting Started

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 $A[i] > key$
7.    $i = i - 1$
8. $A[i+1] = key$

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 \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] \}\n4. \hspace{1em} i = j - 1
5. \hspace{1em} \textbf{while} \hspace{1em} i > 0 \text{ and } A[i] > \textit{key}
6. \hspace{2em} \textbf{do} \hspace{1em} A[i + 1] = A[i]
7. \hspace{2em} \hspace{1em} i = i - 1
8. \hspace{1em} A[i + 1] = \textit{key}

Analysis of the algorithm:
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:

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

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.   $A[i + 1] = key$

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
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
Chapter 2. Getting Started

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.
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.
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 $A(x)$ vs Input Instances $x$
Chapter 2. Getting Started

Time of an algorithm $A(x)$

---

Input Instances  worst case

$x$
Chapter 2. Getting Started

**Upper bound** for algorithm A, bounding all cases of instances

- Time of an algorithm $A(x)$
- Input Instances
- worst case
Chapter 2. Getting Started

Time of an algorithm $A(x)$

- **Upper bound** for algorithm $A$, bounding all cases of instances

- **All these are lower bounds** for algorithm $A$
Chapter 2. Getting Started

Time of an algorithm $A(x)$

- **Upper bound** for algorithm $A$, bounding all cases of instances
- **tightest lower bound**

All these are lower bounds for algorithm $A$

Input Instances  worst case
Chapter 2. Getting Started

**Upper bound** for algorithm A, bounding all cases of instances

*All these are lower bounds* for algorithm A

\[ \text{lower bounds} \leq \text{worst case time} \leq \text{upper bounds} \]
Resource measurement based on
Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
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.
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.

- speed between different machines: a constant multiplicative factor.
Analysis of
Algorithm $\text{INSERTION-SORT}(A)$
Chapter 2. Getting Started

Analysis of

Algorithm \textsc{Insertion-Sort}(A)

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

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

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

Analysis of

Algorithm Insertion-Sort($A$)

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

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

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

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

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

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

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

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 \( x, y, z \).

We can also prove that (can you?)

\[ T(n) \geq u_n^2 + v_n + w \]

for some constants \( u, v, w \).

I.e.,

\[ u_n^2 + v_n + w \leq T(n) \leq x_n^2 + y_n + z \]

means at least one case for which \( \leq \) holds;

\[ u_n^2 + v_n + w \leq T(n) \leq x_n^2 + y_n + z \]

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

\[ x_n^2 + y_n + z \] is a complexity upper bound for \( T(n) \).
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \text{ for some constants } x, y, z \]
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \] for some constants \( x, y, z \)

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \] for some constants \( u, v, w \)
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z \]

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w \]

I.e.,

\[ un^2 + vn + w \leq T(n) \leq xn^2 + yn + z \]
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z \]

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w \]

I.e.,

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

\[ \leq \text{means at least one case for which } \leq \text{ holds;} \]
Chapter 2. Getting Started

So we have proved:

\[ T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z \]

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w \]

I.e.,

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

\leq \text{ means at least one case for which } \leq \text{ holds;}

\[ un^2 + vn + w \text{ is a complexity lower bound for } T(n) \]
So we have proved:

\[ T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z \]

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w \]

I.e.,

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

\( \leq \) means at least one case for which \( \leq \) holds;

\( un^2 + vn + w \) is a complexity lower bound for \( T(n) \)

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

So we have proved:

\[ T(n) \leq xn^2 + yn + z \text{ for some constants } x, y, z \]

We can also prove that (can you?)

\[ T(n) \geq un^2 + vn + w \text{ for some constants } u, v, w \]

I.e.,

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

\leq \text{ means at least one case for which } \leq \text{ holds;}

\[ un^2 + vn + w \text{ is a complexity lower bound for } T(n) \]

\leq \text{ means in all cases for which } \leq \text{ holds;}

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

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. The 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 sorting 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of very large 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., $n = |x|$. There are inaccuracies when $n$ represents the number of items in the input. For instance, to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of very large values on a scale of $2^N$, even though $n$ is the number of items and $n = 4$, a single comparison $x_1 \leq x_2$ would need 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.
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 $⟨x_1, x_2, x_3, x_4⟩$ 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** $N$. 
Chapter 2. Getting Started

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 $n \approx N$. 
Chapter 2. Getting Started

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

2. 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 $n$.

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

Big-O: $O(n^2)$ 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,

$$O(g(n)) = \{ f(n) : \exists c > 0, \ k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k \}$$
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. $3n^2$
2. $5 \cdot 3n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log_2 n^2 + 6n$
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
For example: the following functions are all of the order of $O(n^2)$:

1. $3n^2$
2. $5n^2 + 6n \log_2 n + 90$
3. $0.001n^2 - 200n - 5000$
4. $3n \log_2 n^2 + 6n$
5. $\sqrt{n} - 20 \log_2 n$
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. $345$
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) $n^{2.001}$
Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.

- Algorithm Insertion Sort has time complexity $O(n^2)$.
- Algorithm Iterative Fibonacci has time complexity $O(n)$.
- Algorithm Rec-Fibonacci has time complexity $O(2^n)$.

But these are not entirely correct!

Recall that $n$ should be the input size, not number of items, or value.

Assume for Insertion Sort, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size $n = |\langle x_1, x_2, \ldots, x_m \rangle|$. Insertion Sort has time complexity $O(m^2)$.

- if every $x_i$ constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.
- if $B$ is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2/(\log n)^2)$. 
Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.
Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.

- Algorithm Insertion Sort has time complexity $O(n^2)$.

- if every $x_i$ constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.

- if $B$ is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2 / (\log n)^2)$.
Chapter 3. Growth of Functions

The **Big-O** notation is used to denote upper bound running time complexity.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$.

But these are not entirely correct!

Recall that $n$ should be the input size, not number of items, or value. 

Assume for **Insertion Sort**, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size $n = |\langle x_1, x_2, \ldots, x_m \rangle|$.

- **Insertion Sort** has time complexity $O(m^2)$.
- If every $x_i$ is constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.
- If $B$ is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2 / (\log n)^2)$. 

Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.

- Algorithm Insertion Sort has time complexity $O(n^2)$.
- Algorithm Iterative Fibonacci has time complexity $O(n)$.
  
  But these are not entirely correct! Recall that $n$ should be the input size, not number of items, or value.
  
  Assume for Insertion Sort, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size $n = |\langle x_1, x_2, \ldots, x_m \rangle|$.
  
  Insertion Sort has time complexity $O(m^2)$.
  
  • If every $x_i$ constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.
  
  • If $B$ is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2 / (\log n)^2)$. 
Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.

• Algorithm Insertion Sort has time complexity $O(n^2)$.
• Algorithm Iterative Fibonacci has time complexity $O(n)$.
• Algorithm Rec-Fibonacci has time complexity
Chapter 3. Growth of Functions

The **Big-O** notation is used to denote upper bound running time complexity.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$
Chapter 3. Growth of Functions

The **Big-O** notation is used to denote *upper bound running time complexity*.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$

But these are not entirely correct!
The **Big-O** notation is used to denote *upper bound running time complexity*.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$

But these are not entirely correct!

Recall that $n$ should be the **input size**, not number of items, or value
Chapter 3. Growth of Functions

The Big-O notation is used to denote upper bound running time complexity.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$

But these are not entirely correct!

Recall that $n$ should be the **input size**, not number of items, or value

Assume for **Insertion Sort**, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \ldots, x_m \rangle|$$

**Insertion Sort** has time complexity $O(m^2)$
Chapter 3. Growth of Functions

The **Big-O** notation is used to denote *upper bound running time complexity*.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$

But these are not entirely correct!

Recall that $n$ should be the **input size**, not number of items, or value.

Assume for **Insertion Sort**, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \ldots, x_m \rangle|$$

**Insertion Sort** has time complexity $O(m^2)$

- if every $x_i$ constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.
The Big-O notation is used to denote upper bound running time complexity.

- Algorithm **Insertion Sort** has time complexity $O(n^2)$.
- Algorithm **Iterative Fibonacci** has time complexity $O(n)$.
- Algorithm **Rec-Fibonacci** has time complexity $O(2^n)$

But these are not entirely correct!

Recall that $n$ should be the **input size**, not number of items, or value

Assume for **Insertion Sort**, the input $\langle x_1, x_2, \ldots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \ldots, x_m \rangle|$$

**Insertion Sort** has time complexity $O(m^2)$

- if every $x_i$ constant $B$ bits, then $m \leq n/B$, the time is $O(n^2)$.
- if $B$ is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2/(\log_2 n)^2)$. 
Chapter 3. Growth of Functions

Could you also correct the time complexities for Iterative Fibonacci and Rec-Fibonacci measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value; let $n_b = \text{number of bits to represent } n$, then $n_b = \log_2 |n| = \log_2 n$; so $n = 2^{n_b}$. So you can say either (1) Iterative Fibonacci runs in time $O(n)$ on value $n$, or (2) it runs in time $O(2^{n_b})$ for on value encoded with $n_b$ bits.

- **Rec-Fibonacci**: the same discussion applies. (1) Recursive Fibonacci runs in time $O(2^n)$ on value $n$, or (2) it runs in time $O(2^{2n_b})$ for on value encoded with $n_b$ bits.
Could you also correct the time complexities for **ITERATIVE FIBONACCI** and **REC-FIBONACCI** measured by the input size?
Could you also correct the time complexities for Iterative Fibonacci and Rec-Fibonacci measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
- **Rec-Fibonacci**: the same discussion applies.
Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b = \text{the number of bits to represent } n$, then

- **Rec-Fibonacci**: the same discussion applies.
Could you also correct the time complexities for Iterative Fibonacci and Rec-Fibonacci measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b = \text{the number of bits to represent } n$, then $n_b = |n| = \log_2 n$;

- **Rec-Fibonacci**: the same discussion applies.

(1) Recursive Fibonacci runs in time $O(2^n)$ on value $n$, or
(2) it runs in time $O(2^{2n_b})$ for on value encoded with $n_b$ bits.
Could you also correct the time complexities for Iterative Fibonacci and Rec-Fibonacci measured by the input size?

- Iterative Fibonacci: $O(n)$ where $n$ is the number value;
  
  let $n_b$ = the number of bits to represent $n$, then $n_b = |n| = \log_2 n$; so $n = 2^{n_b}$.

- Rec-Fibonacci: the same discussion applies.
Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b =$ the number of bits to represent $n$, then $n_b = |n| = \log_2 n$; so $n = 2^{n_b}$. So you can say either
Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b$ = the number of bits to represent $n$, then $n_b = |n| = \log_2 n$;
  so $n = 2^{n_b}$. So you can say either

  (1) **Iterative Fibonacci** runs in time $O(n)$ on value $n$, or
Chapter 3. Growth of Functions

Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b = \text{the number of bits to represent } n$, then $n_b = \lvert n \rvert = \log_2 n$;
  
  so $n = 2^{n_b}$. So you can say either

  (1) **Iterative Fibonacci** runs in time $O(n)$ on value $n$, or
  
  (2) it runs in time $O(2^{n_b})$ for on value encoded with $n_b$ bits.
Could you also correct the time complexities for \textsc{Iterative Fibonacci} and \textsc{Rec-Fibonacci} measured by the input size?

- \textbf{Iterative Fibonacci}: $O(n)$ where $n$ is the number value;
  
  let $n_b = \text{the number of bits to represent } n$, then $n_b = |n| = \log_2 n$;
  
  so $n = 2^{n_b}$. So you can say either

  1. \textsc{Iterative Fibonacci} runs in time $O(n)$ on value $n$, or
  2. it runs in time $O(2^{n_b})$ for on value encoded with $n_b$ bits.

- \textbf{Rec-Fibonacci}: the same discussion applies.
Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b =$ the number of bits to represent $n$, then $n_b = |n| = \log_2 n$; so $n = 2^{n_b}$. So you can say either

  (1) **Iterative Fibonacci** runs in time $O(n)$ on value $n$, or
  (2) it runs in time $O(2^{n_b})$ for on value encoded with $n_b$ bits.

- **Rec-Fibonacci**: the same discussion applies.

  (1) **Recursive Fibonacci** runs in time $O(2^n)$ on value $n$, or
Chapter 3. Growth of Functions

Could you also correct the time complexities for **Iterative Fibonacci** and **Rec-Fibonacci** measured by the input size?

- **Iterative Fibonacci**: $O(n)$ where $n$ is the number value;
  
  let $n_b = \text{the number of bits to represent } n$, then $n_b = |n| = \log_2 n$;
  
  so $n = 2^{n_b}$. So you can say either

  (1) **Iterative Fibonacci** runs in time $O(n)$ on value $n$, or
  (2) it runs in time $O(2^{n_b})$ for on value encoded with $n_b$ bits.

- **Rec-Fibonacci**: the same discussion applies.

  (1) **Recursive Fibonacci** runs in time $O(2^n)$ on value $n$, or
  (2) it runs in time $O(2^{2^{n_b}})$ for on value encoded with $n_b$ bits.
Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has inevitably resulted in recurrences.

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

1. if \( p < r \) 
2. then 
3. \( q = \left\lfloor \frac{p + r}{2} \right\rfloor \) 
4. Merge Sort \((A, p, q)\) 
5. Merge Sort \((A, q + 1, r)\) 
6. 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: \( T(1) \leq c \).
Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm

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

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

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

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

- base case: \( T(1) \leq c \).
Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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

1. if \(p < r\)
2. \(\textbf{then } q = \lfloor \frac{p + r}{2} \rfloor\)
3. Merge Sort\((A, p, q)\)
4. Merge Sort\((A, q + 1, r)\)
5. Merging2Lists\((A.p, q, r)\)
A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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)`**. Then
Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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($A, p, r$). Then

• $t_{1,2} = c$
Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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

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

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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

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

- \( t_{1,2} = c \)
- \( t_{3} = t_{4} = T(\frac{n}{2}) \)
- \( t_{5} \leq n \) \textbf{(why?)}
A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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)`. 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

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

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($A, p, r$). Then

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

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

base case: $T(1) \leq c$. 


Chapter 4. Solving Recurrences

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

\[
T(n) \leq 2T(n^2) + n + c \\
T(n^2) \leq 2T(n^4) + n^2 + c \\
T(n^4) \leq 2T(n^8) + n^4 + c \\
\vdots \\
T(n^{2^h}) \leq 2T(n^{2^{h+1}}) + n^{2^h} + c
\]

where $n^{2^h+1} = 1$

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

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

where $n^{2^h+1} = 1$.
Chapter 4. Solving Recurrences

Solve recurrence

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

with base case \( T(1) \leq c \)

with a simple method:
Chapter 4. Solving Recurrences

Solve recurrence

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

with a simple method:

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

Solve recurrence

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

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

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

with a simple method:

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

Solve recurrence

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

with a simple method:

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

Solve recurrence

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

with a simple method:

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

Solve recurrence

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

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,
Solve recurrence

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

with a simple method:

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

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

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

with a simple method:

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

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

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

Solve recurrence

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

with base case \( T(1) \leq c \)

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{4}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{4}\right) & \leq 2T\left(\frac{n}{8}\right) + \frac{n}{4} + c \\
& \ldots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where} \quad \frac{n}{2^{h+1}} = 1
\end{align*}
\]

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

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
2T\left(\frac{n}{2}\right) & \leq 2^2T\left(\frac{n}{4}\right) + 2 \times \frac{n}{2} + 2c \\
2^2T\left(\frac{n}{4}\right) & \leq 2^3T\left(\frac{n}{8}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \\
& \ldots
\end{align*}
\]
Chapter 4. Solving Recurrences

Solve recurrence

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

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,

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

Solve recurrence

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

with a simple method:

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

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

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

Solve recurrence

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

with a simple method:

\[
\begin{align*}
T(n) & \leq 2T\left(\frac{n}{2}\right) + n + c \\
T\left(\frac{n}{2}\right) & \leq 2T\left(\frac{n}{4}\right) + \frac{n}{2} + c \\
T\left(\frac{n}{4}\right) & \leq 2T\left(\frac{n}{8}\right) + \frac{n}{4} + c \\
\vdots \\
T\left(\frac{n}{2^h}\right) & \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1
\end{align*}
\]

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

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

Solve recurrence

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

with a simple method:

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

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

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \]
\[ 2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c \]
\[ 2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c \]
\[ \ldots \]
\[ 2^hT\left(\frac{n}{2^h}\right) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1 \]
\[ + \]
\[ T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h + 1)n + c \sum_{i=0}^{h} 2^i \]
Chapter 4. Solving Recurrences

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

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

\[ T(n) \leq 2^h + 1 \cdot n + \log_2 n + c(2^h + 1) \leq cn + n \log_2 n + c(2^h - 1) \]
\[ = n \log_2 n + 2cn - c \]
\[ = O(n \log_2 n) \]

We need to prove the last equality, i.e., find constants \( a \) and \( k \) such that
\[ n \log_2 n + 2cn - c \leq an \log_2 n \]
when \( n > k \).

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

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

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 \( h + 1 = \log_2 n \)

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

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

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

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

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

\[ n \log_2 n + 2cn - c \leq an \log_2 n \quad (1) \]

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 \( h + 1 = \log_2 n \)

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

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

\[ n \log_2 n + 2cn - c \leq an \log_2 n \quad (1) \]

when \( n > k \).

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

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

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

\[ n \log_2 n + 2cn - c \leq an \log_2 n \quad (1) \]

when \( n > k \).

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

That is, \( n \log_2 n + 2cn - c \leq 2n \log_2 n \) when \( n > k = 2^{2c} \).
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 \( h + 1 = \log_2 n \)

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

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

\[ n \log_2 n + 2cn - c \leq an \log_2 n \quad (1) \]

when \( n > k \).

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

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

So

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

When $n$ is not a power of 2
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?
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 $n \leq m_n$;

- $T(n) \leq T(m_n), \text{ why?}$ assume $T$ to be monotonic;
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, $T(m_n) = O(m_n \log_2 m_n)$;
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 \( m_n \geq k \);
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?
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 \( \frac{m_n}{2} < n \);
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 $T(n)$
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 $T(n) \leq T(m_n)$
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 \( T(n) \leq T(m_n) \leq cm_n \log_2 m_n \)
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 $T(n) \leq T(m_n) \leq cm_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2 = 4cn \log_2 n = \ldots$
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, $\exists c, k, T(m_n) \leq c m_n \log_2 m_n$ when $m_n \geq k$;

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

- So $T(n) \leq T(m_n) \leq c m_n \log_2 m_n \leq 2cn \log_2 (2n) \leq 2cn \log_2 n^2$
  $= 4cn \log_2 n = c'n \log_2 n$, here $c' = 4c$,
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 $m_n \geq k (\geq 4)$. 

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 \( n \geq \lceil \frac{k}{2} \rceil(\geq 2) \);
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

The principle for dominos to fall.
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 $\mathcal{P}(n)$ for every natural number $n \geq 1$, it suffices to prove

- $\mathcal{P}(1)$ 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.
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.

"The principle for dominos to fall".
Chapter 4. Solving Recurrences

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., $P(3)$ instead of $P(1)$;
- the statement to prove: $P(k) \rightarrow P(k + 1)$ has the variant:

$$P(1) \land P(2) \land \cdots \land P(k) \rightarrow P(k + 1)$$
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., \( \mathcal{P}(3) \) instead of \( \mathcal{P}(1) \);
- the statement to prove: \( \mathcal{P}(k) \rightarrow \mathcal{P}(k + 1) \) has the variant:

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

- other variants: e.g., \( \mathcal{P}(k) \rightarrow \mathcal{P}(2k) \)

  but we need to make sure all \( n \)'s beyond the base cases are covered.
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.** $n - 1$ comparisons are needed to find the maximum of $n$ numbers.
Chapter 4. Solving Recurrences

**Theorem.** \( n - 1 \) comparisons are needed to find the maximum of \( n \) numbers.

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

**Theorem.** \( n - 1 \) comparisons are needed to find the maximum of \( n \) numbers.

**Proof:** (use math induction)

- Observation: every number should be compared *directly or indirectly* with the maximum.
Chapter 4. Solving Recurrences

**Theorem.** \(n - 1\) comparisons are needed to find the maximum of \(n\) numbers.

**Proof:** (use math induction)

- Observation: every number should be compared *directly or indirectly* with the maximum.

- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.
Chapter 4. Solving Recurrences

**Theorem.** $n - 1$ comparisons are needed to find the maximum of $n$ numbers.

**Proof:** (use math induction)

- Observation: every number should be compared *directly or indirectly* with the maximum.

- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.
Chapter 4. Solving Recurrences

**Theorem.** \( n - 1 \) comparisons are needed to find the maximum of \( n \) numbers.

**Proof:** (use math induction)

- Observation: every number should be compared *directly or indirectly* with the maximum.
- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.

- Lemma: every *connected graph* of \( n \) vertices has \( \geq n - 1 \) edges.
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

**Proof:**

Base case: $n = 1$, a connected graph of single vertex contains zero edges.
Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i, i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.

Induction: Let $G$ be any connected graph $G$ of $k + 1$ vertices and $v$ be an arbitrary vertex in it. Assume $v$ shares edges with $r$ other vertices, $r \geq 1$ (why?).

Let $G'$ be the result of removing $v$ from $G$. 
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.

Induction: Let $G$ be any connected graph $G$ of $k + 1$ vertices and $v$ be an arbitrary vertex in it. Assume $v$ shares edges with $r$ other vertices, $r \geq 1$ (why?).

Let $G'$ be the result of removing $v$ from $G$.

Assume that $G'$ consists of $t$ connected components $G_1, \ldots, G_t$. $1 \leq t \leq r$ (why?)
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.

Induction: Let $G$ be any connected graph $G$ of $k + 1$ vertices and $v$ be an arbitrary vertex in it. Assume $v$ shares edges with $r$ other vertices, $r \geq 1$ (why?).

Let $G'$ be the result of removing $v$ from $G$.

Assume that $G'$ consists of $t$ connected components $G_1, \ldots, G_t$. $1 \leq t \leq r$ (why?)

Let vertex counts of $G_1, \ldots, G_t$ be $m_1, \ldots, m_t$, respectively.
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i \), \( i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.
Note that \( \sum_{i=1}^{t} m_i = k \) (why?).
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

**Proof:**

**Base case:** \( n = 1 \), a connected graph of single vertex contains zero edges.

**Assumption:** Assume the lemma holds for every \( i, i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

**Induction:** Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.

Note that \( \sum_{i=1}^{t} m_i = k \) (why?).

By the assumption, these components contain at least \( m_1 - 1, \ldots m_t - 1 \) edges, respectively.
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.

Induction: Let $G$ be any connected graph $G$ of $k + 1$ vertices and $v$ be an arbitrary vertex in it. Assume $v$ shares edges with $r$ other vertices, $r \geq 1$ (why?).

Let $G'$ be the result of removing $v$ from $G$.

Assume that $G'$ consists of $t$ connected components $G_1, \ldots, G_t$. $1 \leq t \leq r$ (why?)

Let vertex counts of $G_1, \ldots, G_t$ be $m_1, \ldots, m_t$, respectively.

Note that $\sum_{i=1}^{t} m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \ldots m_t - 1$ edges, respectively.

The total count of edges in $G$ is

$$\geq \sum_{i=1}^{t} (m_i - 1)$$
Chapter 4. Solving Recurrences

Lemma: every connected graph of $n$ vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every $i$, $i \leq k$, where $k \geq 1$. That is, a connected graph of $i$ vertices contains at least $i - 1$ edges.

Induction: Let $G$ be any connected graph $G$ of $k + 1$ vertices and $v$ be an arbitrary vertex in it. Assume $v$ shares edges with $r$ other vertices, $r \geq 1$ (why?).

Let $G'$ be the result of removing $v$ from $G$.

Assume that $G'$ consists of $t$ connected components $G_1, \ldots, G_t$. $1 \leq t \leq r$ (why?)

Let vertex counts of $G_1, \ldots, G_t$ be $m_1, \ldots, m_t$, respectively.

Note that $\sum_{i=1}^{t} m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \ldots m_t - 1$ edges, respectively.

The total count of edges in $G$ is

$$\geq \sum_{i=1}^{t} (m_i - 1) + r$$
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i, i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.
Note that \( \sum_{i=1}^{t} m_i = k \) (why?).

By the assumption, these components contain at least \( m_1 - 1, \ldots m_t - 1 \) edges, respectively.

The total count of edges in \( G \) is

\[
\geq \sum_{i=1}^{t} (m_i - 1) + r = \sum_{i=1}^{t} m_i - \sum_{i=1}^{t} 1 + r
\]
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i, i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.

Note that \( \sum_{i=1}^{t} m_i = k \) (why?).

By the assumption, these components contain at least \( m_1 - 1, \ldots, m_t - 1 \) edges, respectively.

The total count of edges in \( G \) is

\[
\geq \sum_{i=1}^{t} (m_i - 1) + r = \sum_{i=1}^{t} m_i - \sum_{i=1}^{t} 1 + r = \sum_{i=1}^{t} m_i + r - t
\]
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i, i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.

Note that \( \sum_{i=1}^{t} m_i = k \) (why?).

By the assumption, these components contain at least \( m_1 - 1, \ldots m_t - 1 \) edges, respectively.

The total count of edges in \( G \) is

\[
\geq \sum_{i=1}^{t} (m_i - 1) + r = \sum_{i=1}^{t} m_i - \sum_{i=1}^{t} 1 + r = \sum_{i=1}^{t} m_i + r - t \geq \sum_{i=1}^{t} m_i =
\]
Chapter 4. Solving Recurrences

Lemma: every connected graph of \( n \) vertices has \( \geq n - 1 \) edges.

Proof:

Base case: \( n = 1 \), a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every \( i, \ i \leq k \), where \( k \geq 1 \). That is, a connected graph of \( i \) vertices contains at least \( i - 1 \) edges.

Induction: Let \( G \) be any connected graph \( G \) of \( k + 1 \) vertices and \( v \) be an arbitrary vertex in it. Assume \( v \) shares edges with \( r \) other vertices, \( r \geq 1 \) (why?).

Let \( G' \) be the result of removing \( v \) from \( G \).

Assume that \( G' \) consists of \( t \) connected components \( G_1, \ldots, G_t \). \( 1 \leq t \leq r \) (why?)

Let vertex counts of \( G_1, \ldots, G_t \) be \( m_1, \ldots, m_t \), respectively.

Note that \( \sum_{i=1}^{t} m_i = k \) (why?).

By the assumption, these components contain at least \( m_1 - 1, \ldots m_t - 1 \) edges, respectively.

The total count of edges in \( G \) is

\[
\geq \sum_{i=1}^{t} (m_i - 1) + r = \sum_{i=1}^{t} m_i - \sum_{i=1}^{t} 1 + r = \sum_{i=1}^{t} m_i + r - t \geq \sum_{i=1}^{t} m_i = k
\]
Example: binary search algorithm

Algorithm \textsc{Binary Search}(A, key, i, j)

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

Let $n = j - i + 1$. It has the time with recurrence,

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

and $T(1) = d$ where $c > 0$, $d > 0$ are constants.
Example: binary search algorithm

Algorithm **Binary Search**(*A, key, i, j*)

1. **if** *j < i* return (NULL)
2. **else**
3. \[ k = \lfloor \frac{i+j}{2} \rfloor \]
4. **if** *A[k] = key* return (*k*)
5. **else**
6. **if** *A[k] > key* *Binary Search* (*A, key, i, k − 1*)
7. **else**
8. *Binary Search* (*A, key, k + 1, j*)

Let \( n = j - i + 1 \). It has the time with recurrence,

\[
T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \quad \text{and} \quad T(1) = d
\]

where \( c > 0, d > 0 \) are constants.
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\left\lfloor \frac{n}{2} \right\rfloor) + c \text{ and } T(1) = d \]
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \quad \text{and} \quad T(1) = d \]

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

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \quad \text{and} \quad T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

Proof. We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lceil \frac{n}{2} \rceil) + c \text{ and } T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \text{ when } n \geq k \tag{2} \]

- First we show that if (2) holds for \( \lceil \frac{n}{2} \rceil \), i.e., \( T(\lceil \frac{n}{2} \rceil) \leq a \log_2 \frac{n}{2} \)
Chapter 4. Solving Recurrences

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

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]

- First we show that if (2) holds for \( \left\lfloor \frac{n}{2} \right\rfloor \), i.e., \( T(\left\lfloor \frac{n}{2} \right\rfloor) \leq a \log_2 \frac{n}{2} \)
- then \( T(n) \leq T(\left\lfloor \frac{n}{2} \right\rfloor) + c \)
Prove that $T(n) = O(\log_2 n)$.

**Proof.** We show that there are constants $a, k$ such that

$$T(n) \leq a \log_2 n \quad \text{when} \quad n \geq k \quad (2)$$

- First we show that if $(2)$ holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

then $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c$
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2) \]

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \)
  
  then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \quad \text{and} \quad T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \), then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)

Choosing \( a \geq c \) allows (2) holds for \( n \) (under the assumption (2) holds for \( \lfloor \frac{n}{2} \rfloor \))
Chapter 4. Solving Recurrences

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

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2) \]

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \)

then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)

Choosing \( a \geq c \) allows (2) holds for \( n \) (under the assumption (2) holds for \( \lfloor \frac{n}{2} \rfloor \))

- Then we need to find the \( k \) such that (2) holds for all \( n \geq k \).
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2) \]

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \)

  then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)

  Choosing \( a \geq c \) allows (2) holds for \( n \) (under the assumption (2) holds for \( \lfloor \frac{n}{2} \rfloor \))

- Then we need to find the \( k \) such that (2) holds for all \( n \geq k \).

  \( k = 1 \) would not work because base case \( T(1) = d > 0 \) does not guarantee for (2) to hold, because (2) requires \( T(1) \leq a \log_2 1 = 0 \).
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \text{ when } n \geq k \] (2)

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \)
  then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)
  Choosing \( a \geq c \) allows (2) holds for \( n \) (under the assumption (2) holds for \( \lfloor \frac{n}{2} \rfloor \))

- Then we need to find the \( k \) such that (2) holds for all \( n \geq k \).
  \( k = 1 \) would not work because base case \( T(1) = d > 0 \) does not guarantee for (2) to hold, because (2) requires \( T(1) \leq a \log_2 1 = 0 \).

  \( k = 2 \) will work because \( T(2) \leq T(1) + c = d + c \) guarantee for (2) to hold as long as \( d + c \leq a \), as (2) requires \( T(2) \leq a \log_2 2 \).
Chapter 4. Solving Recurrences

\[ T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \quad \text{and} \quad T(1) = d \]

Prove that \( T(n) = O(\log_2 n) \).

**Proof.** We show that there are constants \( a, k \) such that

\[ T(n) \leq a \log_2 n \quad \text{when} \quad n \geq k \quad (2) \]

- First we show that if (2) holds for \( \lfloor \frac{n}{2} \rfloor \), i.e., \( T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2} \)

  then \( T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a \)

  Choosing \( a \geq c \) allows (2) holds for \( n \) (under the assumption (2) holds for \( \lfloor \frac{n}{2} \rfloor \))

- Then we need to find the \( k \) such that (2) holds for all \( n \geq k \).

  \( k = 1 \) would not work because base case \( T(1) = d > 0 \) does not guarantee for (2) to hold, because (2) requires \( T(1) \leq a \log_2 1 = 0 \).

  \( k = 2 \) will work because \( T(2) \leq T(1) + c = d + c \) guarantee for (2) to hold as long as \( d + c \leq a \), as (2) requires \( T(2) \leq a \log_2 2 \).
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 \( \log_a 1^n = -\log_a n \);
- \( a^{\log_a n} = n \);
- \( \log_m a^n = \left( \log_a n \right)^m \neq \log_a n^m \).
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 \( \log_a \frac{1}{n} = -\log_a n; \)
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 \( \log_a \frac{1}{n} = -\log_a n \);
- \( a^{\log_a n} = n \);
- \( \log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n \);
- \( \log_a^m n = (\log_a n)^m \neq \log_a n^m \).
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n, \text{ where } T(1) = 2 \]
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lceil \frac{2n}{3} \rceil) + n, \text{ where } T(1) = 2 \]

**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)

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n, \quad \text{where } T(1) = 2 \]

**Guess** \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

**Verify:**
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2 \]

**Guess** \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

**Verify:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

(1) choose a different base case,
Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

\[ T(n) = \frac{3}{2} T(\lceil \frac{2n}{3} \rceil) + n, \quad \text{where } T(1) = 2 \]

**Guess** \( T(n) \leq cn \log_2 n \), for some constant \( c \) to be determined later.

**Verify:**

- **base case:** \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, 2. add a constant to the guessed upper bound
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, (2) add a constant to the guessed upper bound

We instead guess \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).
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**:  
- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now **assume the guessed upper bound holds for** \( \lfloor \frac{2n}{3} \rfloor \), i.e.,  
\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]

substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \]
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case,
2. add a constant to the guessed upper bound

We instead *guess* \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]

substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \]
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, (2) add a constant to the guessed upper bound

We instead guess \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]

substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, 2. add a constant to the guessed upper bound

We instead **guess** \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]

substitute it in the recurrence, we get

\[
T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \\
\leq \frac{3}{2} \left( c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \right) + n \\
\leq cn \log_2 \frac{2n}{3} + 3 + n \\
= cn (\log_2 n + \log_2 \frac{2}{3}) + 3 + n
\]
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:**

- base case: \( T(1) = 2 \leq 0 \) does not hold for the guessed inequality.

There are two remedies:

1. choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** \( T(n) \leq cn \log_2 n + 2 \), so it holds for \( n = 1 \).

Now assume the guessed upper bound holds for \( \lfloor \frac{2n}{3} \rfloor \), i.e.,

\[ T(\lfloor \frac{2n}{3} \rfloor) \leq c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \]

substitute it in the recurrence, we get

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} \left( c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2 \right) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2} \left( c\left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor + 2 \right) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn \left( \log_2 n + \log_2 \frac{2}{3} \right) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2} \left( c \left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor + 2 \right) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn \left( \log_2 n + \log_2 \frac{2}{3} \right) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]

To allow that the last term \( \leq cn \log_2 n + 2 \), the assumed upper bound,
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T(\left\lfloor \frac{2n}{3} \right\rfloor) + n \leq \frac{3}{2} \left( c \left\lfloor \frac{2n}{3} \right\rfloor \log_{2} \left\lfloor \frac{2n}{3} \right\rfloor + 2 \right) + n \leq cn \log_{2} \frac{2n}{3} + 3 + n \]

\[ = cn(\log_{2} n + \log_{2} \frac{2}{3}) + 3 + n = cn \log_{2} n + 3 + n - cn \log_{2} \frac{3}{2} \]

\[ \leq cn \log_{2} n + 2 + 1 + n - cn \log_{2} \frac{3}{2} \]

To allow that the last term \( \leq cn \log_{2} n + 2 \), the assumed upper bound, we have choose \( c \) such that \( 1 + n - cn \log_{2} \frac{3}{2} \leq 0 \).
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2} \left( c \left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left( \frac{2n}{3} \right) + 2 \right) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ = \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]

To allow that the last term \( \leq cn \log_2 n + 2 \), the assumed upper bound, we have choose \( c \) such that \( 1 + n - cn \log_2 \frac{3}{2} \leq 0 \).

for example, we can choose \( c = 2/\log_2 \frac{3}{2} \), such that

\[ 1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0 \]

for all \( n \geq 1 \).
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ = \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]

To allow that the last term \( \leq cn \log_2 n + 2 \), the assumed upper bound, we have choose \( c \) such that \( 1 + n - cn \log_2 \frac{3}{2} \leq 0 \).

For example, we can choose \( c = 2 / \log_2 \frac{3}{2} \), such that

\[ 1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0 \]

For all \( n \geq 1 \).

So we have shown \( T(n) \leq cn \log_2 n + 2 \), for all \( n \geq 1 \).
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2}T\left(\left\lfloor \frac{2n}{3} \right\rfloor \right) + n \leq \frac{3}{2}\left(c\left\lfloor \frac{2n}{3} \right\rfloor \log_2 \left\lfloor \frac{2n}{3} \right\rfloor + 2\right) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]

To allow that the last term \( \leq cn \log_2 n + 2 \), the assumed upper bound, we have choose \( c \) such that \( 1 + n - cn \log_2 \frac{3}{2} \leq 0 \).

for example, we can choose \( c = 2/\log_2 \frac{3}{2} \), such that

\[ 1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0 \]

for all \( n \geq 1 \).

So we have shown \( T(n) \leq cn \log_2 n + 2 \), for all \( n \geq 1 \). Are you sure?
Chapter 4. Solving Recurrences

\[ T(n) = \frac{3}{2} T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2} (c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \]

\[ = cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \]

\[ \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2} \]

To allow that the last term \( \leq cn \log_2 n + 2 \), the assumed upper bound, we have choose \( c \) such that \( 1 + n - cn \log_2 \frac{3}{2} \leq 0 \).

For example, we can choose \( c = \frac{2}{\log_2 \frac{3}{2}} \), such that

\[ 1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0 \]

for all \( n \geq 1 \).

So we have shown \( T(n) \leq cn \log_2 n + 2 \), for all \( n \geq 1 \). Are you sure?

Does this also imply

\[ T(n) = O(n \log_2 n) \text{ or } T(n) = O(n \log_2 n + 2) \]
$T(n) \leq cn \log_2 n + 2$
Chapter 4. Solving Recurrences

\[ T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n \ (\text{when } n \geq 2) \]
Chapter 4. Solving Recurrences

\[ T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n \quad \text{(when } n \geq 2) \]
\[ \leq cn \log_2 n + n \log_2 n \]
Chapter 4. Solving Recurrences

\[ T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n \quad (\text{when } n \geq 2) \]
\[ \leq cn \log_2 n + n \log_2 n \]
\[ = (c + 1)n \log_2 n \]
Chapter 4. Solving Recurrences

\[ T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n \quad (\text{when } n \geq 2) \]
\[ \leq cn \log_2 n + n \log_2 n \]
\[ = (c + 1)n \log_2 n \]
\[ = O(n \log_2 n), \]
Chapter 4. Solving Recurrences

\[ T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n \quad \text{(when } n \geq 2) \]
\[ \leq cn \log_2 n + n \log_2 n \]
\[ = (c + 1)n \log_2 n \]
\[ = O(n \log_2 n), \]
$T(n) \leq cn \log_2 n + 2 \leq cn \log_2 n + n$ (when $n \geq 2$)
\leq cn \log_2 n + n \log_2 n \\
= (c + 1)n \log_2 n \\
= O(n \log_2 n),$

So we choose $k = 2.$
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:

\[ S(m) = T(2^m) \]

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 $m = \log_2 n$, 
2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$
2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$T(2^m) = 2T(2^{m/2}) + m$
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: $S(m) = T(2^m)$
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: \( S(m) = T(2^m) \)

\( S(m) = 2S(m/2) + m \)
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 \( S(m) = O(m \log m) \)
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
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.
Chapter 4. Solving Recurrences

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.
Chapter 4. Solving Recurrences

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 $T(1) = 1$
Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\begin{align*}
  l_0: & & T(n) & & T(n) & & T(n) \\
  l_1: & & T(n/4) & & T(n/4) & & T(n/4) \\
  & & & & & n^2
\end{align*}
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) & 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) & 3(n/4)^2
\end{align*}
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[\begin{array}{cccccc}
  l_0: & & & & & T(n) \\
  l_1: & T(n/4) & & & T(n/4) & n^2 \\
  l_2: & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & 3\left(\frac{n}{4}\right)^2 \\
  l_3: & \ldots & & & & \end{array}\]
## 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>( )</th>
<th>( l_1 ):</th>
<th>( T(n/4) )</th>
<th>( T(n/4) )</th>
<th>( T(n) )</th>
<th>( n^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_2 ):</td>
<td>( T(n/4^2) )</td>
<td>( T(n/4^2) )</td>
<td>( T(n/4^2) )</td>
<td>( T(n/4^2) )</td>
<td>( T(n/4^2) )</td>
<td>( 3(n/4)^2 )</td>
</tr>
<tr>
<td>( l_3 ):</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( l_4 ):</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{array}{cccccc}
& & & T(n) & & \\
\ell_0: & & & T(n) & & \\
\ell_1: & T(n/4) & & T(n/4) & & \\
\ell_2: & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) & T(n/4^2) \\
\ell_3: & \ldots & \ldots & \ldots & \ldots & \ldots \\
\ell_4: & \ldots & \ldots & \ldots & \ldots & \ldots \\
\ell_{m-1}: & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccc}
& & & n^2 & & \\
& & & 3(n/4)^2 & & \\
& & & 3^2(n/4^2)^2 & & \\
& & & 3^3(n/4^3)^2 & & \\
\end{array}
\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ l_0: \quad T(n) = \frac{n^2}{1} \]
\[ l_1: \quad T(n/4) = \frac{n}{4^2} \]
\[ l_2: \quad T(n/4^2) = \frac{n}{4^4} \]
\[ l_3: \quad T(n/4^3) = \frac{n}{4^8} \]
\[ l_4: \quad T(n/4^4) = \frac{n}{4^{16}} \]
\[ l_{m-1}: \quad T(n/4^{m-1}) = \frac{n}{4^{2^m-1}} \]

Then $T(n)$ is the sum

\[ T(n) = \frac{n^2}{1} + 3\left(\frac{n}{4}\right)^2 + 3^2\left(\frac{n}{4^2}\right)^2 + 3^3\left(\frac{n}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \]

\[ = n^2 \left[ 1 + 3 \left(\frac{1}{4}\right)^2 + 3^2 \left(\frac{1}{4^2}\right)^2 + 3^3 \left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1} \left(\frac{1}{4^{m-1}}\right)^2 \right] \]

\[ \leq n^2 \left[ \frac{1}{1 - \frac{3}{4}} \right] \]

\[ = 16n^2 \]

for all $n > 0$.\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: $T(n)$
$l_1$: $T(n/4)$ $T(n/4)$ $T(n/4)$
$l_2$: $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$ $T(n/4^2)$
$l_3$: $\ldots$
$l_4$: $\ldots$
$l_{m-1}$: $\ldots$
$l_m$: $T(1), T(1), T(1), T(1), T(1), \ldots$, $3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2$
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) & & T(n/4) & & 3(n/4)^2 \\
  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) & & 3^2(n/4^2)^2 \\
  l_3: & & \ldots \ldots \\
  l_4: & & \ldots \ldots \\
  l_{m-1}: & & \ldots \ldots \\
  l_m: & & T(1), T(1), T(1), T(1), T(1), \ldots \\

\text{where } \frac{n}{4^m} = 1, \text{ i.e., } m = \log_4 n.
\end{align*}
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\begin{align*}
l_0: & & T(n) & & n^2 \\
l_1: & & T(n/4) & & T(n/4) & & T(n/4) \quad 3(n/4)^2 \\
l_2: & & T(n/4^2) & & T(n/4^2) & & T(n/4^2) & & T(n/4^2) \quad 3^2(n/4^2)^2 \\
l_3: \; & & \vdots \; & & \vdots \; & & \vdots \; \quad 3^{m-1}(n/4^m)^2 \\
l_4: \; & & \vdots \; & & \vdots \; & & \vdots \; & & \vdots \\
l_{m-1}: \; & & \vdots \; & & \vdots \; & & \vdots \; & & \vdots \\
l_m: \; & & T(1), T(1), T(1), T(1), T(1), \ldots , & & \quad T(1)
\end{align*}

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[ T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^mT(1) \]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[ \begin{align*}
  l_0: & \quad T(n) \\
  l_1: & \quad T(n/4) \quad T(n/4) \quad T(n/4) \\
  l_2: & \quad T(n/4^2) \quad T(n/4^2) \quad T(n/4^2) \\
  l_3: & \quad \ldots \\
  l_4: & \quad \ldots \\
  l_{m-1}: & \quad \ldots \\
  l_m: & \quad T(1), T(1), T(1), T(1), T(1), \ldots,
\end{align*} \]

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m T(1) \]

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m \times 1 \]
Chapter 4. Solving Recurrences

Example \( T(n) = 3T(n/4) + n^2 \), with base case \( T(1) = 1 \)

\[
\begin{align*}
l_0: & \quad T(n) \\
l_1: & \quad T(n/4) & \quad T(n/4) & \quad T(n/4) & \quad n^2 \\
l_2: & \quad T(n/4^2) & \quad T(n/4^2) & \quad T(n/4^2) & \quad T(n/4^2) & \quad 3\left(\frac{n}{4}\right)^2 \\
l_3: & \quad \ldots \\
l_4: & \quad \ldots \\
l_{m-1}: & \quad \ldots \\
l_m: & \quad T(1), T(1), T(1), T(1), T(1), \ldots , \\
\end{align*}
\]

where \( \frac{n}{4^m} = 1 \), i.e., \( m = \log_4 n \).

Then \( T(n) \) is the sum

\[
\begin{align*}
T(n) &= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m T(1) \\
T(n) &= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \times 1 \\
&= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2] + 3^m \left(\frac{n}{4^m}\right)^2 \\
\end{align*}
\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: $T(n)$

$l_1$: $T(n/4)$

$l_2$: $T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2) T(n/4^2)$

$l_3$: $T(n/4^3)$

$l_4$: $T(n/4^4)$

$l_{m-1}$: $T(n/4^m)$

$l_m$: $T(1)$, $T(1)$, $T(1)$, $T(1)$, $T(1)$, $T(1)$, $T(1)$, $T(1)$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1)$

$T(n) = n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1$

$= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \cdots + 3^{m-1}(\frac{n}{4^{m-1}})^2] + 3^m(\frac{n}{4^m})^2$

$= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \cdots + (\frac{3}{16})^m]$
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[\begin{align*}
l_0: & \quad T(n) \\
l_1: & \quad T(n/4) \\
l_2: & \quad T(n/4^2) T(n/4) \\
l_3: & \quad \ldots \\
l_m-1: & \quad \ldots \\
l_m: & \quad T(1), T(1), T(1), T(1), T(1), \ldots,
\end{align*}\]

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[\begin{align*}
T(n) &= n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m T(1) \\
T(n) &= n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m \times 1 \\
&= n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 3^m \left( \frac{n}{4^m} \right)^2 \\
&= n^2 \left[ 1 + \frac{3}{16} + \left( \frac{3}{16} \right)^2 + \left( \frac{3}{16} \right)^3 + \cdots + \left( \frac{3}{16} \right)^m \right] \\
&= n^2 \left( \frac{1-\left( \frac{3}{16} \right)^{m+1}}{1-\frac{3}{16}} \right)
\end{align*}\]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0$: \hspace{1cm} $T(n)$  \\
$l_1$: \hspace{1cm} $T(n/4)$  \\
$l_2$: \hspace{1cm} $T(n/4^2)$, $T(n/4^2)$, $T(n/4^2)$  \\
$l_3$: \hspace{1cm} $T(n/4^3)$, $T(n/4^3)$, $T(n/4^3)$  \\
$l_4$: \hspace{1cm} $T(n/4^4)$, $T(n/4^4)$, $T(n/4^4)$, $T(n/4^4)$  \\
$l_m$: $T(1), T(1), T(1), T(1), T(1), \ldots,$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m T(1) \]

\[ T(n) = n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{1}{4^{m-1}} \right)^2 \right] + 3^m \times 1 \]

\[ = n^2 \left[ 1 + 3 \left( \frac{1}{4} \right)^2 + 3^2 \left( \frac{1}{4^2} \right)^2 + 3^3 \left( \frac{1}{4^3} \right)^2 + \cdots + 3^{m-1} \left( \frac{n}{4^{m-1}} \right)^2 \right] + 3^m \left( \frac{n}{4^m} \right)^2 \]

\[ = n^2 \left[ 1 + \frac{3}{16} + \left( \frac{3}{16} \right)^2 + \left( \frac{3}{16} \right)^3 + \cdots + \left( \frac{3}{16} \right)^m \right] \]

\[ = n^2 \left( \frac{1 - \left( \frac{3}{16} \right)^{m+1}}{1 - \frac{3}{16}} \right) \]

\[ \leq n^2 \left( \frac{1}{1 - \frac{3}{16}} \right) \]
Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

\[
\begin{align*}
l_0: & \quad T(n) \\
l_1: & \quad T(n/4) \\
l_2: & \quad T(n/4^2) T(n/4^2) T(n/4^2) \\
l_3: & \quad \ldots \ldots \\
l_{m-1}: & \quad \ldots \ldots \\
l_m: & \quad T(1), T(1), T(1), T(1), T(1), \ldots ,
\end{align*}
\]

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

\[
T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m T(1)
\]

\[
T(n) = n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2] + 3^m \times 1
\]

\[
= n^2[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \cdots + 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2] + 3^m \left(\frac{n}{4^m}\right)^2
\]

\[
= n^2\left[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \cdots + \left(\frac{3}{16}\right)^m\right]
\]

\[
= n^2\left(\frac{1-\left(\frac{3}{16}\right)^{m+1}}{1-\frac{3}{16}}\right)
\]

\[
\leq n^2\left(\frac{1}{1-\frac{3}{16}}\right)
\]

\[
= \frac{16}{13}n^2
\]

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:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

- \( l_0 \):
  - \( n^2 \)

- \( l_1 \):
  - \( (n/4)^2 \)
  - \( (n/4)^2 \)
  - \( (n/4)^2 \)
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\( l_0: \quad n^2 \)

\( l_1: \quad \left(\frac{n}{4}\right)^2 \left(\frac{n}{4}\right)^2 \left(\frac{n}{4}\right)^2 \)

\( l_2: \quad \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \)
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \] with base case \( T(1) = 1 \)

\( l_0: \) 
\[ n^2 \]

\( l_1: \) 
\[ \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \]

\( l_2: \) 
\[ \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \left( \frac{n}{4^2} \right)^2 \]

\( l_3: \) 
\[ . . . . . . \]

\( l_m: \) 
\[ . . . . . . \]

\( l_{m-1}: \) 
\[ . . . . . . \]

\( l_m: \) 
\[ n^2 \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\[ l_0: \quad n^2 \]
\[ l_1: \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \]
\[ l_2: \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \]
\[ l_3: \quad \ldots \ldots \quad 3^3 \text{ nodes of } (n/4^3)^2 \]
\[ l_4: \quad \ldots \ldots \]
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

- \( l_0: \) 
  \[ n^2 \]

- \( l_1: \) 
  \[ (n/4)^2 \]

- \( l_2: \) 
  \[ \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \]

- \( l_3: \) 
  \[ \ldots \]

- \( l_4: \) 
  \[ \ldots \]

- \( l_{m-1}: \) 
  \[ \ldots \]

Description: The tree is shown with layers labeled from 0 to \( m-1 \), where each level \( l_i \) contains \( n^2 \) nodes, and the tree structure is expanded recursively down to the base case \( T(1) = 1 \). The layers are filled with \( (n/4)^2 \) for each level, and \( (n/4^2)^2 \) for the second level, continuing recursively until the base case is reached.
Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

- \( l_0: \)  
  \[ n^2 \]

- \( l_1: \)  
  \[ (n/4)^2 \]

- \( l_2: \)  
  \[ (n/4)^2 \]

- \( l_3: \)  
  \[ (n/4)^2 \]

- \( l_4: \)  
  \[ (n/4)^2 \]

- \( l_{m-1}: \)  
  \[ \text{3}^{m-1} \text{ of } (\frac{n}{4^{m-1}})^2 \]

- \( l_m: \)  
  \[ \text{3}^2 \text{ nodes of } (\frac{n}{4^2})^2 \]
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: & & n^2 \\
  l_1: & & (n/4)^2 \quad (n/4)^2 \\
  l_2: & & (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \\
  l_3: & \vdots \\
  l_4: & \vdots \\
  l_{m-1}: & \vdots \\
  l_m: & T(1), T(1), T(1), T(1), T(1), \ldots, \\
\end{align*}

3^3 \text{ nodes of } (\frac{n}{4^3})^2

3^{m-1} \text{ of } (\frac{n}{4^{m-1}})^2

3^m \text{ nodes of } T(1)
Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

\[ T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1 \]

\( l_0: \) 
\[ \begin{array}{c}
  & n^2 \\
 l_1: & (n/4)^2 \quad (n/4)^2 \\
 l_2: & (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \quad (n/4^2)^2 \\
 l_3: & \cdots \\
 l_4: & \cdots \\
 l_{m-1}: & \cdots \\
 l_m: & T(1), T(1), T(1), T(1), T(1), \ldots, \\
\end{array} \]

3\(^3\) nodes of \((n/4^3)^2\)

3\(^{m-1}\) of \((n/4^{m-1})^2\)

3\(^m\) nodes of \(T(1)\)
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 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]
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 $a > 0$. 
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 $b > 0$. 
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;
A good example: **Quick Sort** algorithm [Hoare’1959]

- It has the worst case time $T_{\text{wc}}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{\text{ac}}(n) \leq bn \log_2 n$ for a constant $b > 0$.

  in other word, it is efficient in most cases;
  assumption: the input data are of the uniform distribution.
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.
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 $T_{ac}(n) \leq bn \log_2 n$ 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.
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 $T_{ac}(n) \leq bn \log_2 n$ 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**.
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 \textit{random events}
- such random events are driven by random data
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
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

- 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)
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 you can compute the expected time (i.e., averaged time)
There are two types of randomized algorithms:

- Las Vegas algorithms
- 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.
There are two types of randomized algorithms:

- Las Vegas algorithms
  - always gives answer correctly;
- Monte Carlo algorithms
  - on 'NO' instances, 100% accuracy;
  - on 'YES' instances, ≥ 75% accuracy;

Accuracy 75% can be improved to 99.9% with multiple trials.

Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.
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;
  - $\text{Prob}(\text{to answer } \text{'NO'} \text{ on } \text{'NO'} \text{ instance}) = 1$
  - 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**
  - 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;
    
  \[ \text{Prob\ (to\ answer\ ‘NO’\ on\ ‘NO’\ instance)} = 1 \]
  - on ‘YES’ instances, ≥ 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.
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;
  - 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, $Prob(T(n) \leq cn \log n) \geq 0.75$

• Monte Carlo algorithms
  - on ’NO’ instances, 100% accuracy;
    $Prob(\text{to answer ’NO’ on ’NO’ instance}) = 1$
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;
    - 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;
    $\text{Prob}(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$
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.
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.