CSCI 4470/6470 Algorithms, Spring 2017

Liming Cai

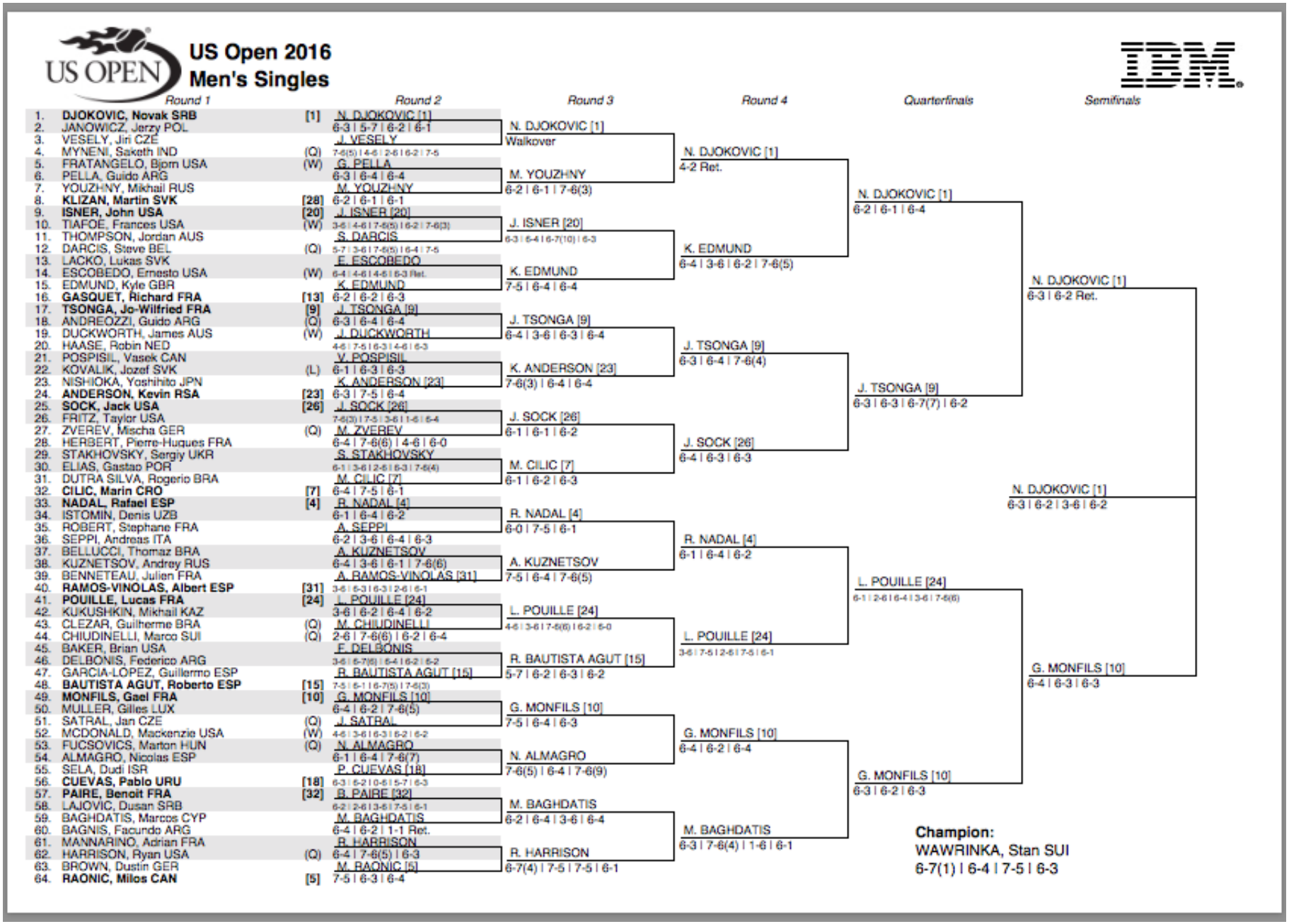
Department of Computer Science, UGA

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

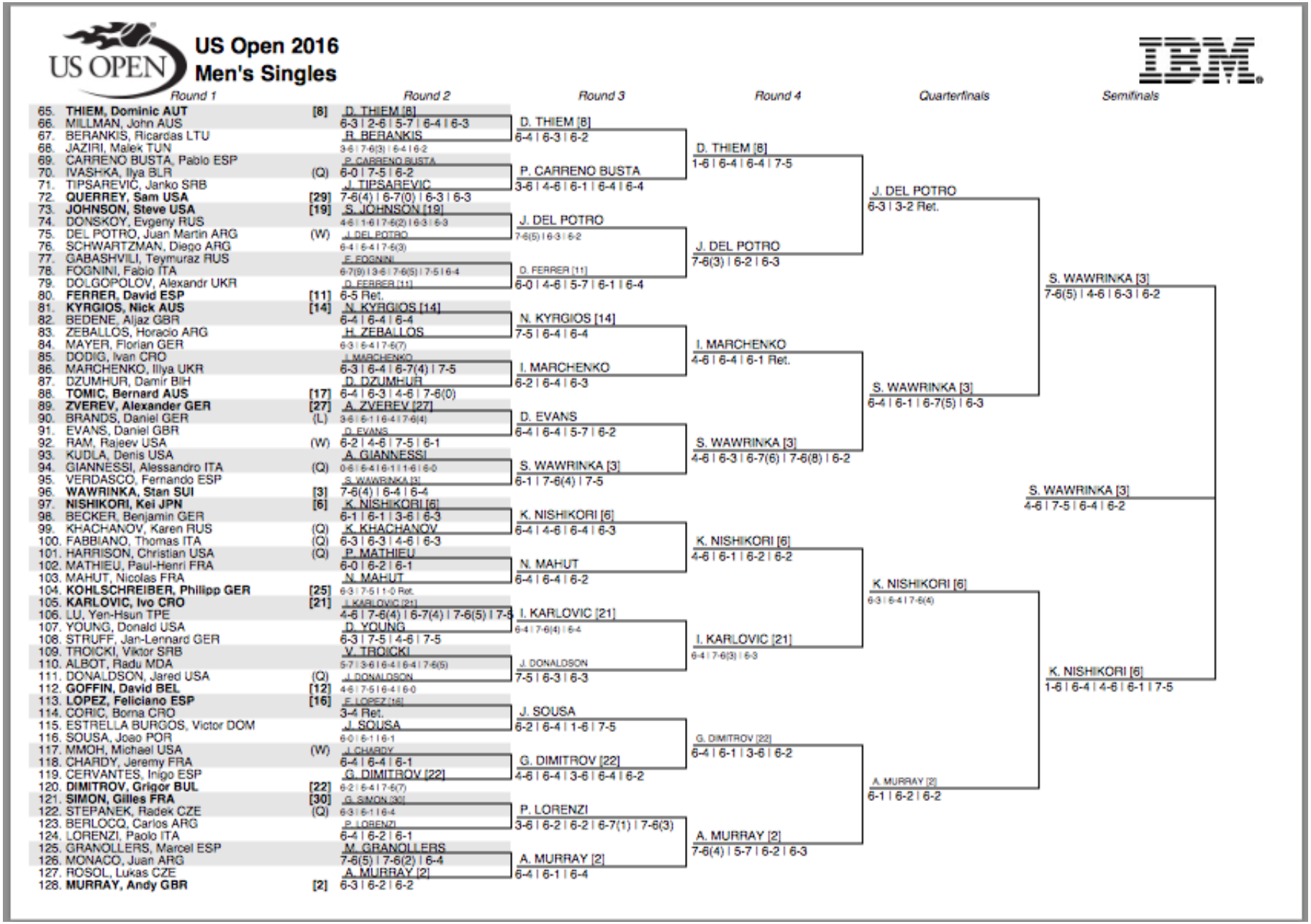
January 17, 2017

An Introduction to the Introduction

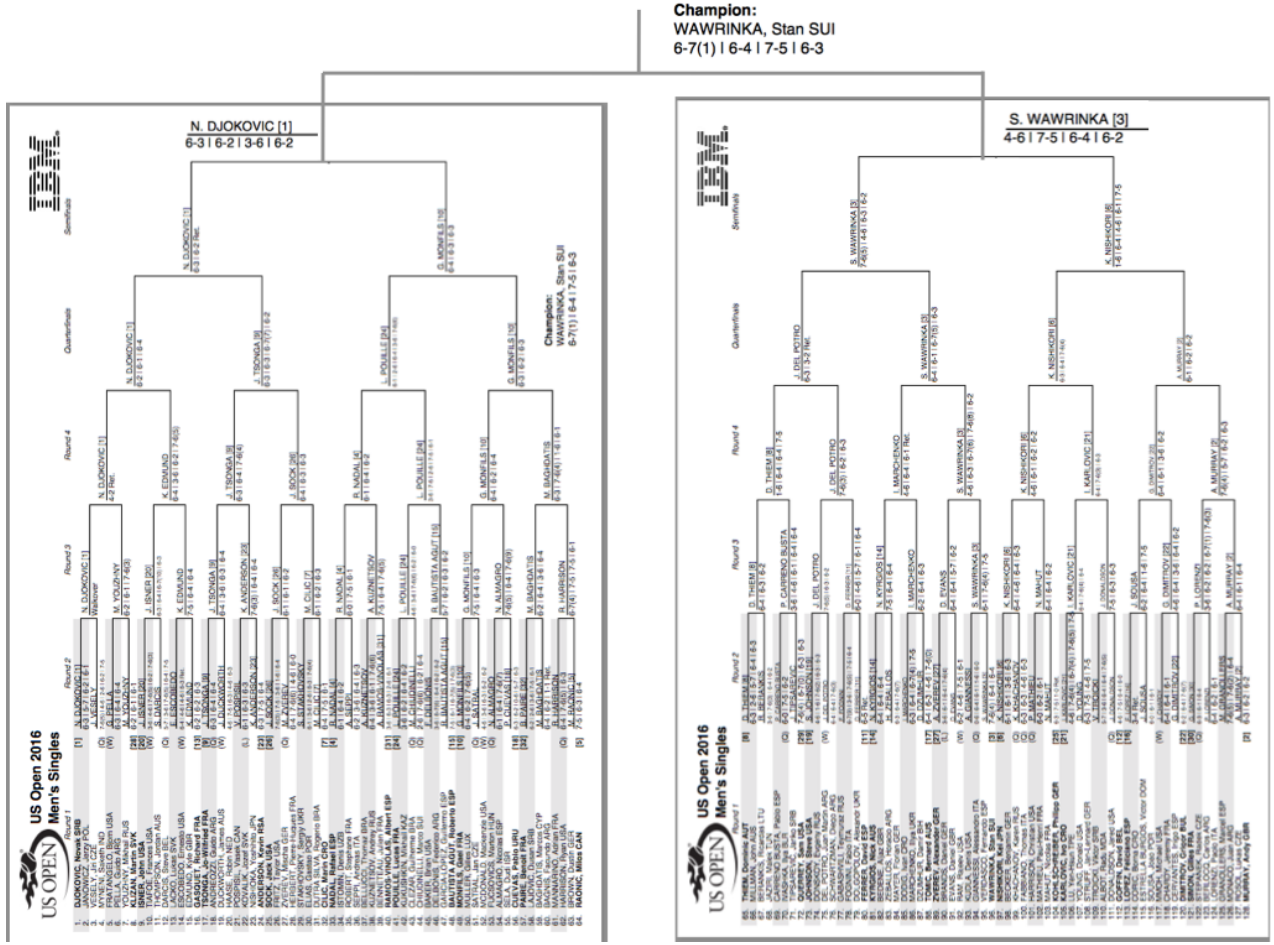
An Introduction to the Introduction



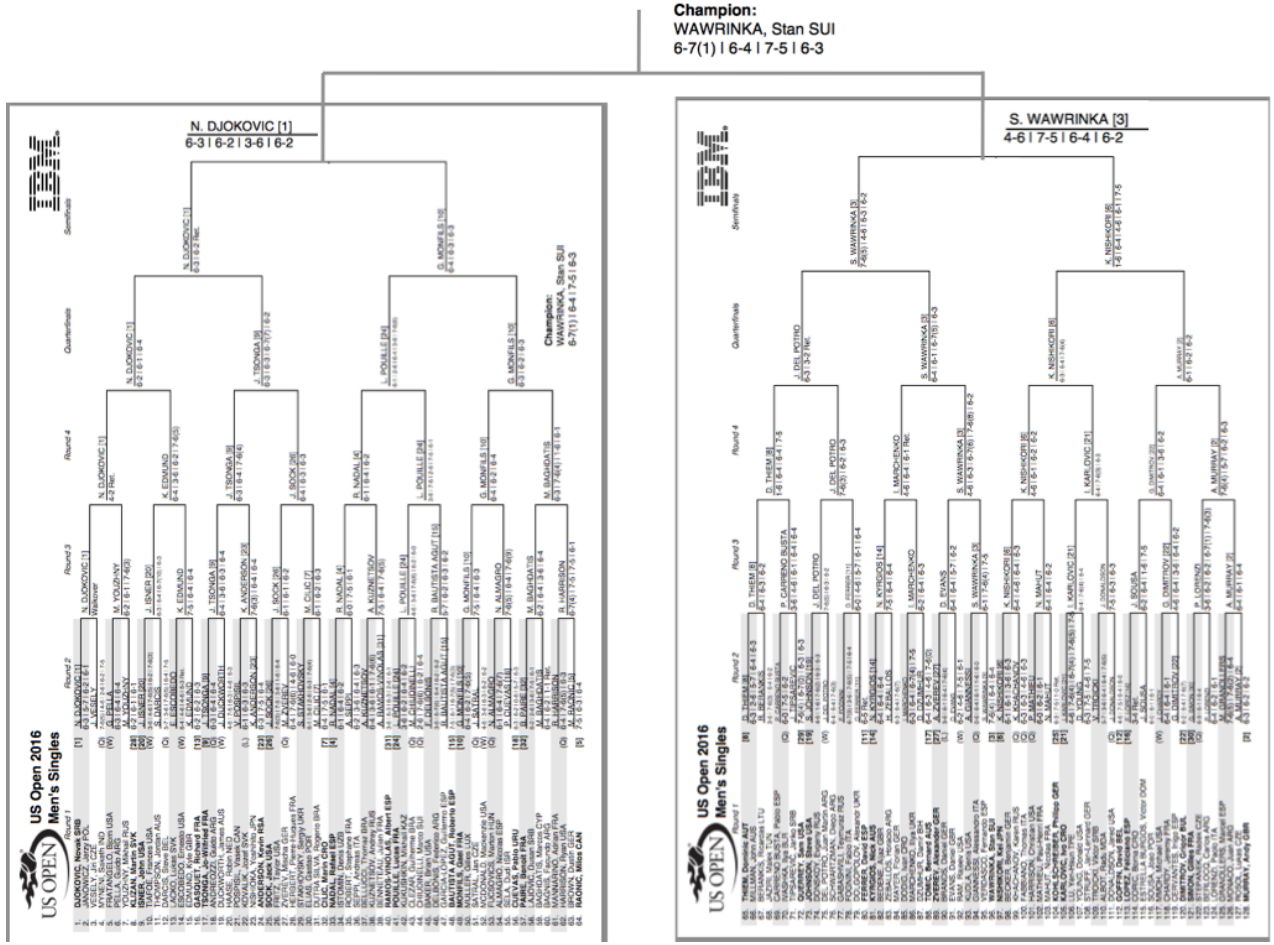
An Introduction to the Introduction



An Introduction to the Introduction

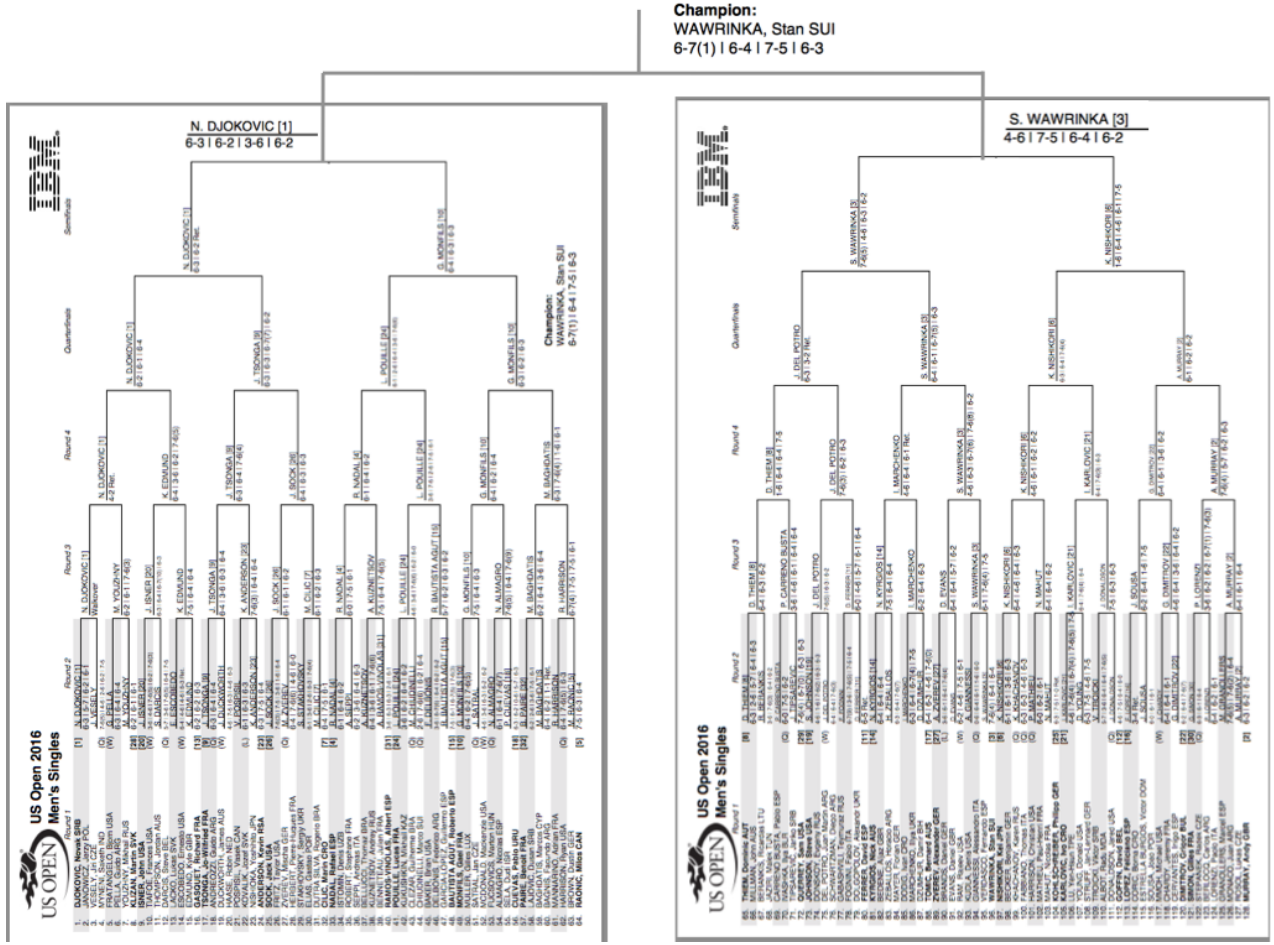


An Introduction to the Introduction



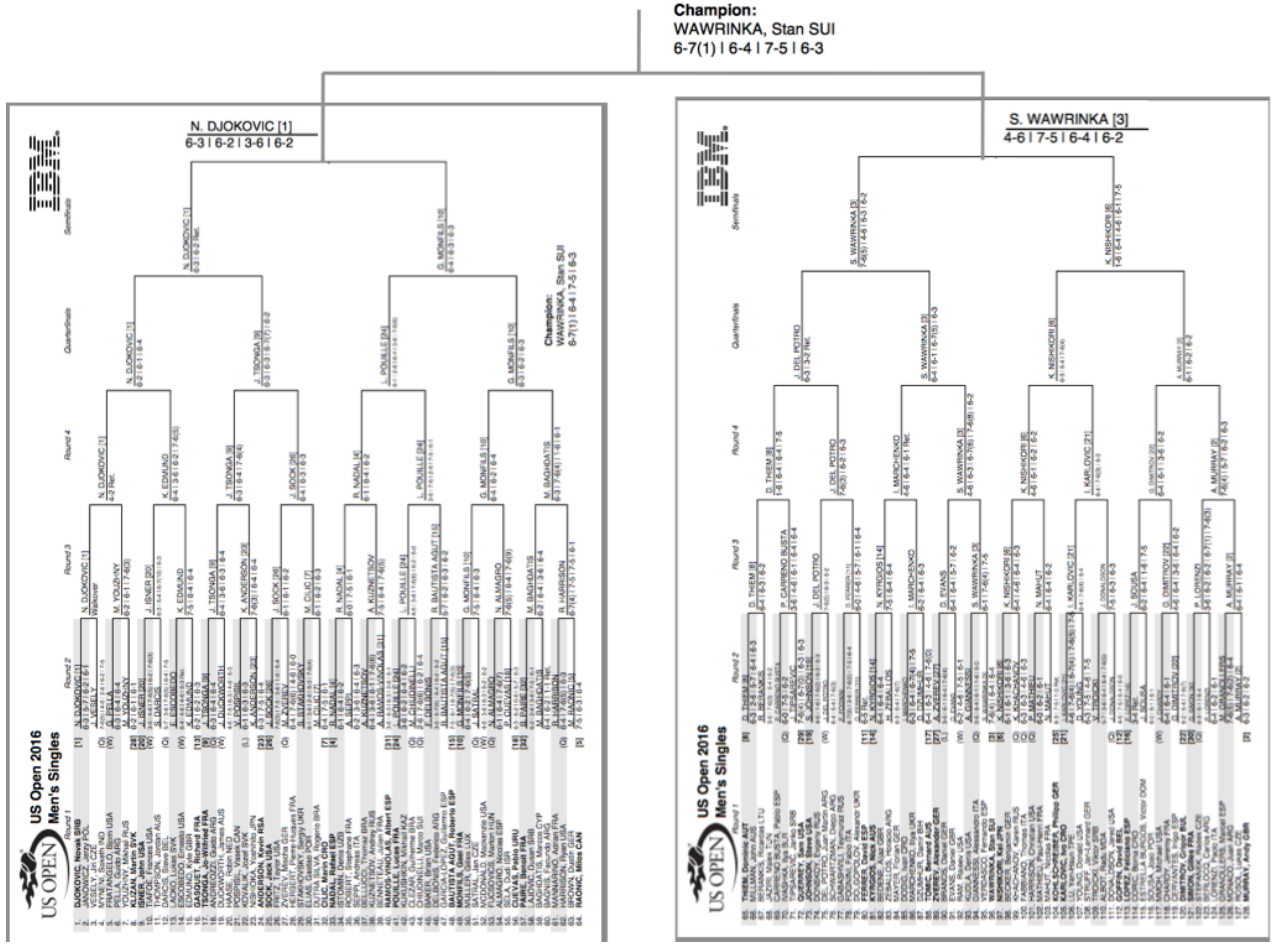
- 128 players in total;

An Introduction to the Introduction



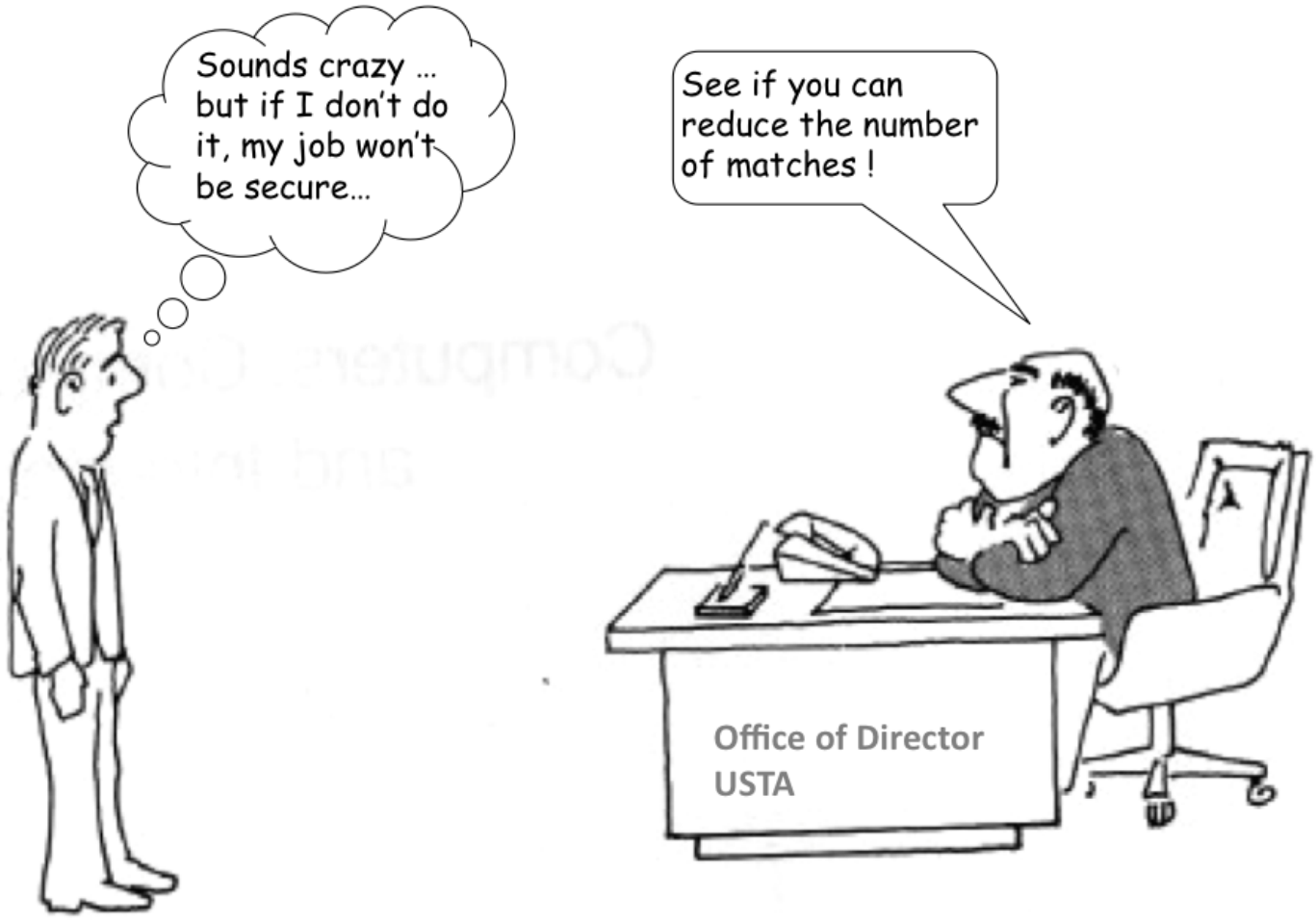
- 128 players in total;
- 127 matches were played to determine who was the champion.

An Introduction to the Introduction

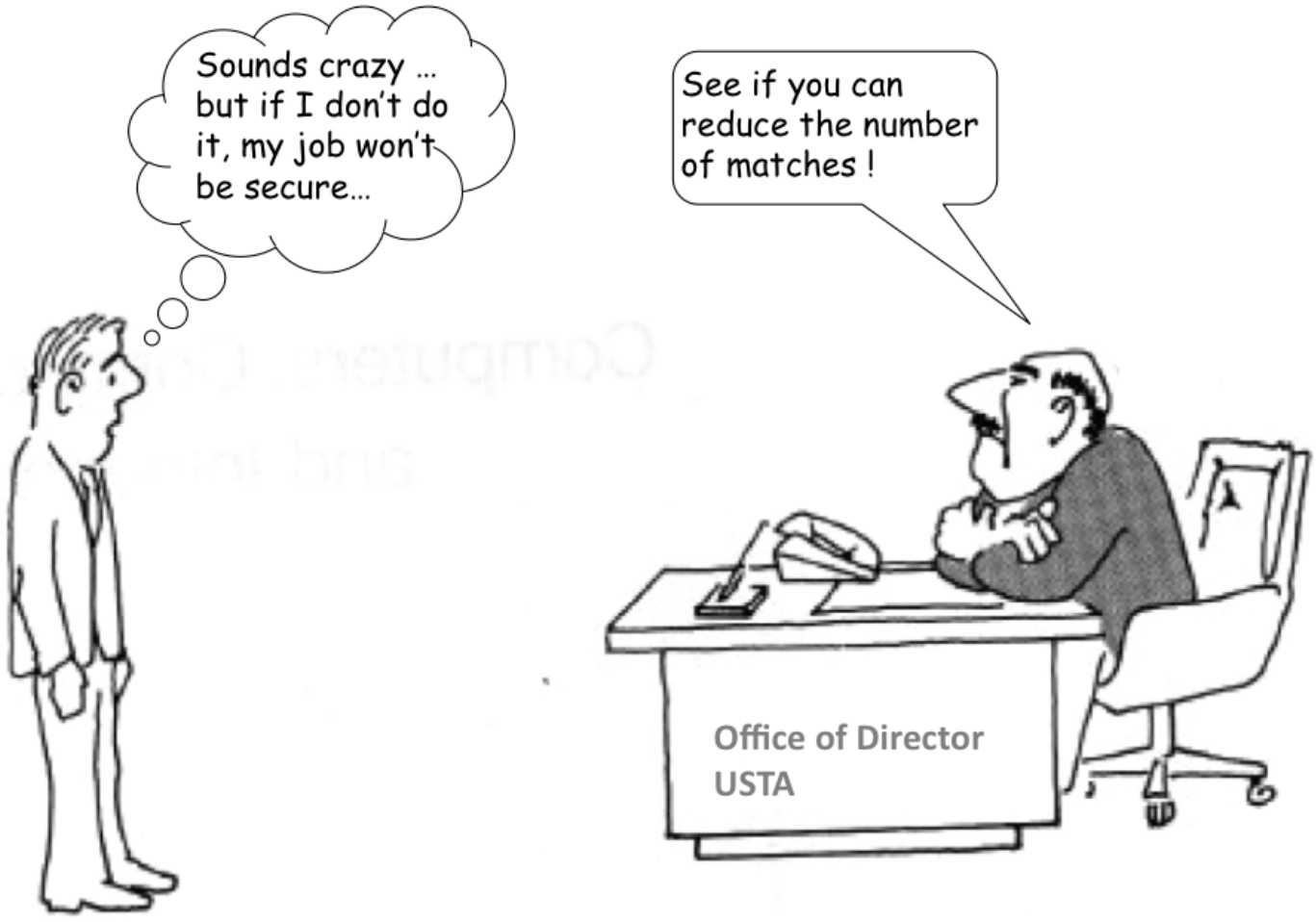


- 128 players in total;
- 127 matches were played to determine who was the champion.
- **Is it possible to just play fewer matches?**

An Introduction to the Introduction



An Introduction to the Introduction



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

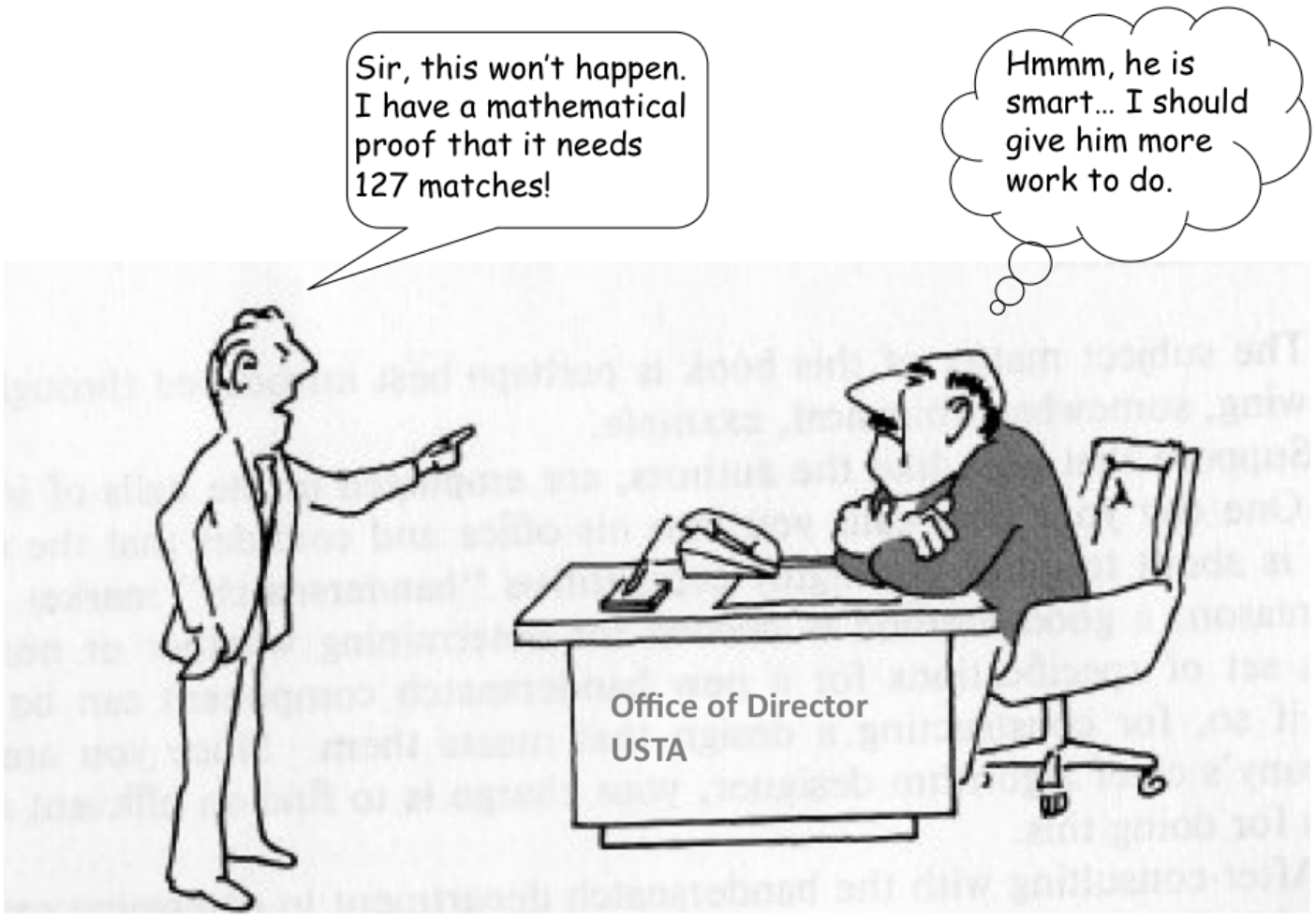
- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were necessary,

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.

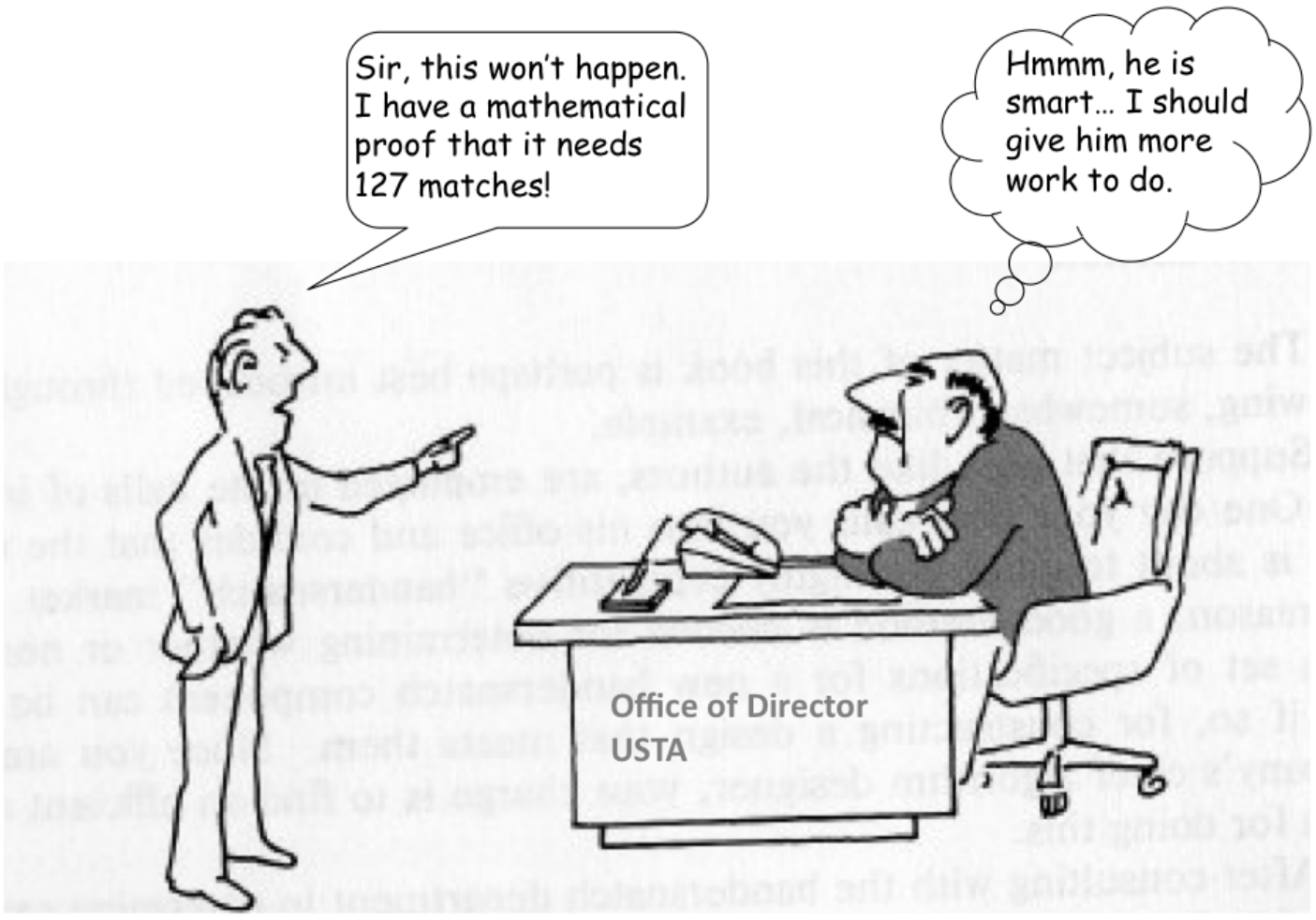
An Introduction to the Introduction

- 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.



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	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

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:

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;

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;

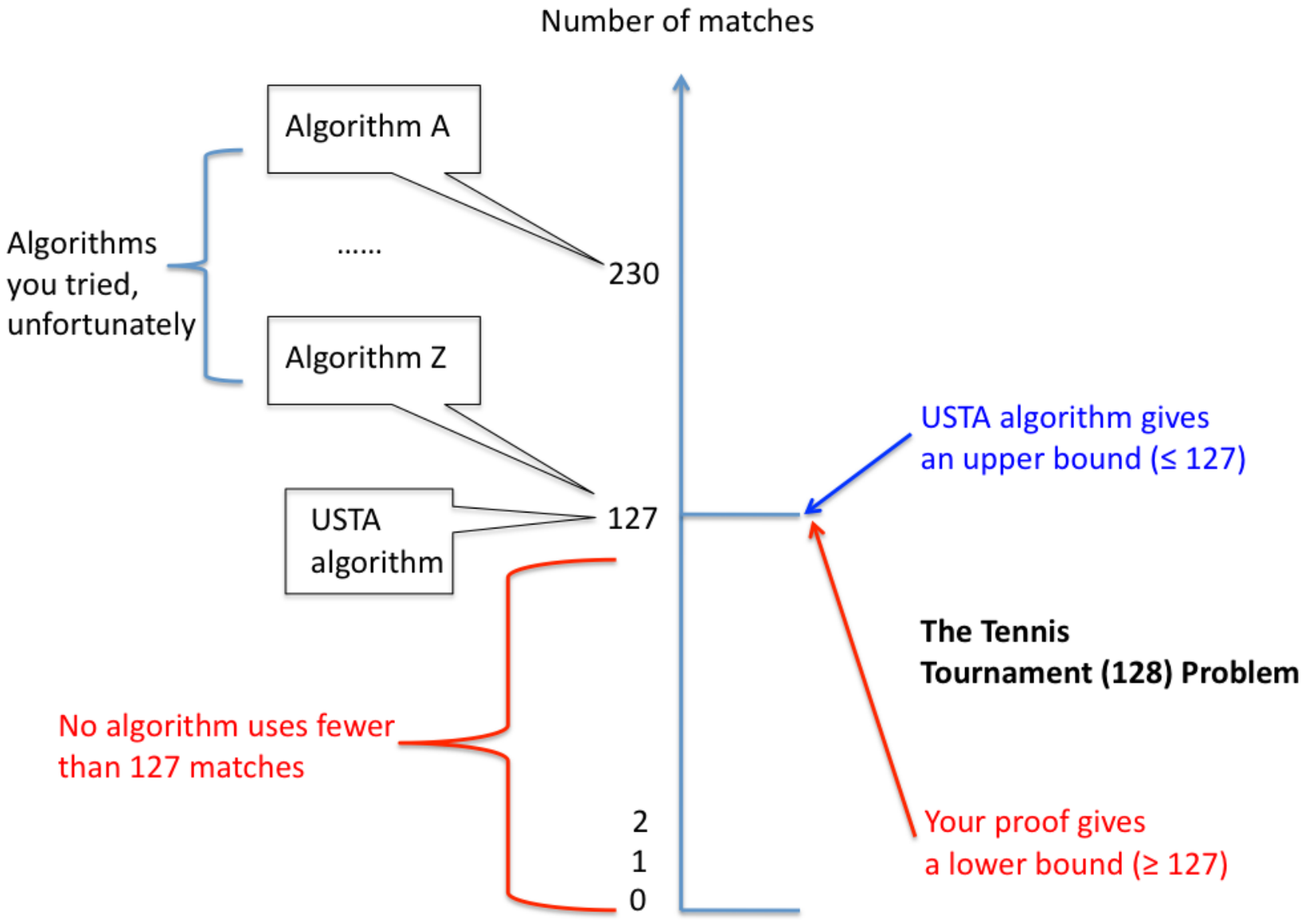
An Introduction to the Introduction

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.

An Introduction to the Introduction



An Introduction to the Introduction

In general, for a given problem Π,

An Introduction to the Introduction

In general, for a given problem Π,

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

An Introduction to the Introduction

In general, for a given problem Π,

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

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

An Introduction to the Introduction

In general, for a given problem Π,

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

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

- a proof that Π cannot have algorithms faster than time T,

An Introduction to the Introduction

In general, for a given problem Π,

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

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

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

An Introduction to the Introduction

In general, for a given problem Π,

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

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

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

  a proof that Π cannot have algorithms faster than time S, with $S > T$,

An Introduction to the Introduction

In general, for a given problem Π,

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

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

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

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

An Introduction to the Introduction

In general, for a given problem Π,

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

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

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

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

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

An Introduction to the Introduction

In general, for a given problem Π,

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

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

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

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

- for problem Π, 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**.

An Introduction to the Introduction

In general, for a given problem Π,

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

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

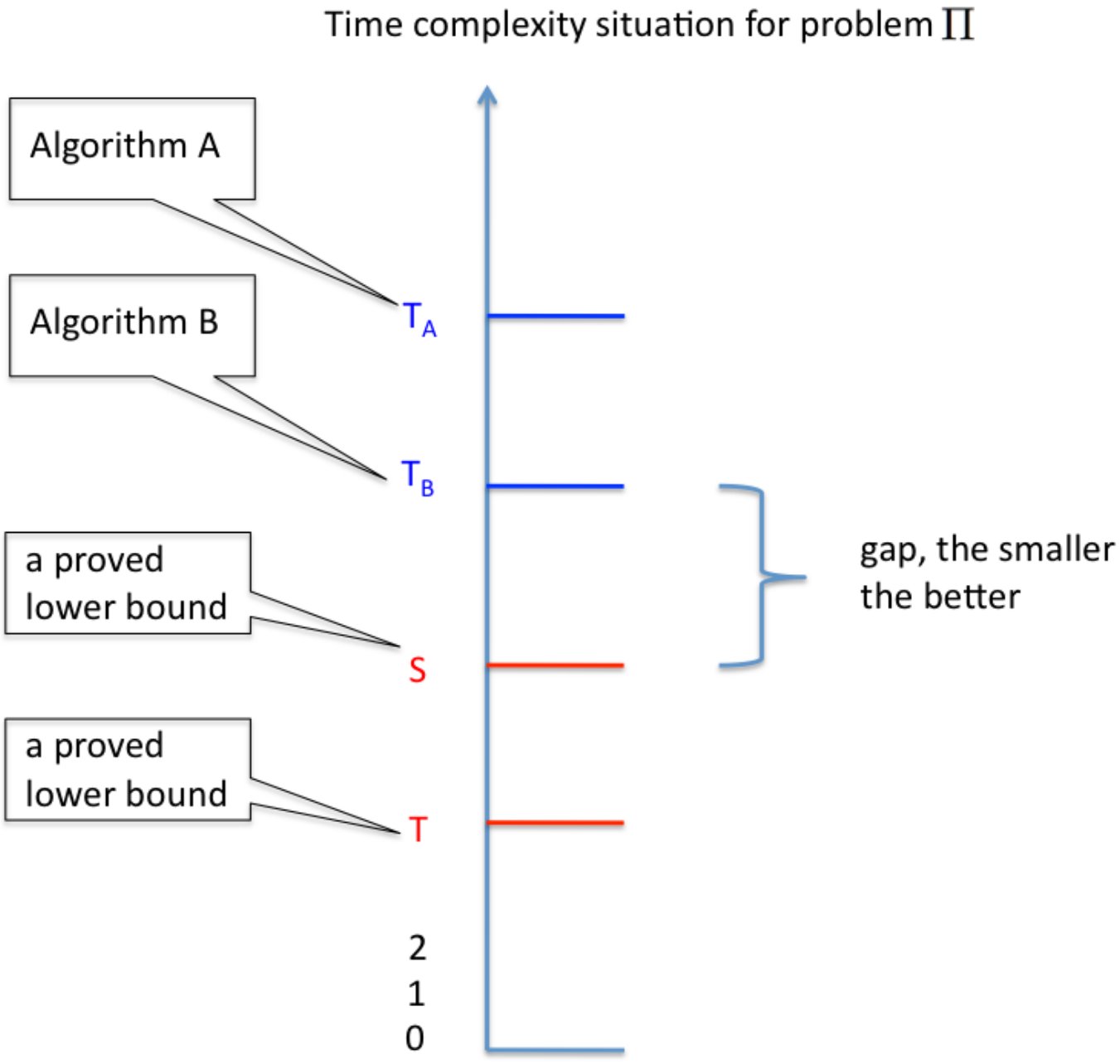
- a proof that Π cannot have algorithms faster than time T, gives a lower bound T for Π;

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

- for problem Π, 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**.
  Also the corresponding algorithms (that offer the upper bound) are called **optimal** for Π.

An Introduction to the Introduction



An Introduction to the Introduction

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

An Introduction to the Introduction

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;

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;

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

The Introduction

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

The Introduction

- What is this course about (and why is it needed)?
  - about basic yet indispensable skills for problem solving

The Introduction

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

The Introduction

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

    i.e., creative thinking without programming languages

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented

The Introduction

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

    i.e., creative thinking without programming languages

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

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)
- Goals to achieve

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)
- Goals to achieve
  - to learn to measure performance of algorithms

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)
- Goals to achieve
  - to learn to measure performance of algorithms
  - to master some fundamental algorithmic techniques

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)
- Goals to achieve
  - to learn to measure performance of algorithms
  - to master some fundamental algorithmic techniques
  - to study advanced algorithmic skills

The Introduction

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

    i.e., creative thinking without programming languages
- How different is this course from other algorithm courses?
  - design technique-oriented, not application-oriented
  - emphasis on guaranteed performance (typically in efficiency)
- Goals to achieve
  - to learn to measure performance of algorithms
  - to master some fundamental algorithmic techniques
  - to study advanced algorithmic skills
  - to understand computational intractability

Part I. Foundations

Part I. Foundations

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

Part I. Foundations

The theme of the course

Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms

Part I. Foundations

The theme of the course

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

Part I. Foundations

The theme of the course

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

Design and analysis of algorithms are closely related.

Part I. Foundations

Example: the Fibonacci sequence.

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

Part I. Foundations

Example: the Fibonacci sequence.

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

That is:

n	:	1	2	3	4	5	6	7	8	9	...
$f(n)$	:	1	1	2	3	5	8	13	21	34	...

Part I. Foundations

Problem 1: Computing the nth Fibonacci number:

Part I. Foundations

Problem 1: Computing the nth Fibonacci number:

INPUT: $n \geq 1$;
OUTPUT: the nth number in the Fibonacci sequence.

Part I. Foundations

Problem 1: Computing the nth Fibonacci number:

INPUT: $n \geq 1$;
OUTPUT: the nth number in the Fibonacci sequence.

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

Part I. Foundations

Problem 1: Computing the nth Fibonacci number:

INPUT: $n \geq 1$;
OUTPUT: the nth number in the Fibonacci sequence.

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

- recursive: task decomposition, top-down, recursive calls;

Part I. Foundations

Problem 1: Computing the nth Fibonacci number:

INPUT: $n \geq 1$;
OUTPUT: the nth number in the Fibonacci sequence.

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

- recursive: task decomposition, top-down, recursive calls;
- iterative: more tightly coupled tasks, bottom-up approaches;

Part I. Foundations

REC-FIBONACCI(n)

Part I. Foundations

REC-FIBONACCI(n)

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

Part I. Foundations

REC-FIBONACCI(n)

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

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ or $n = 2$, **return** (1);
else
$T_1 =$ REC-FIBONACCI$(n - 1)$;

Part I. Foundations

REC-FIBONACCI(n)

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

Part I. Foundations

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

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?

Part I. Foundations

REC-FIBONACCI(n)

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

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

its execution is via a *run-time stack*

Part I. Foundations

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

REC-FIBONACCI(n)

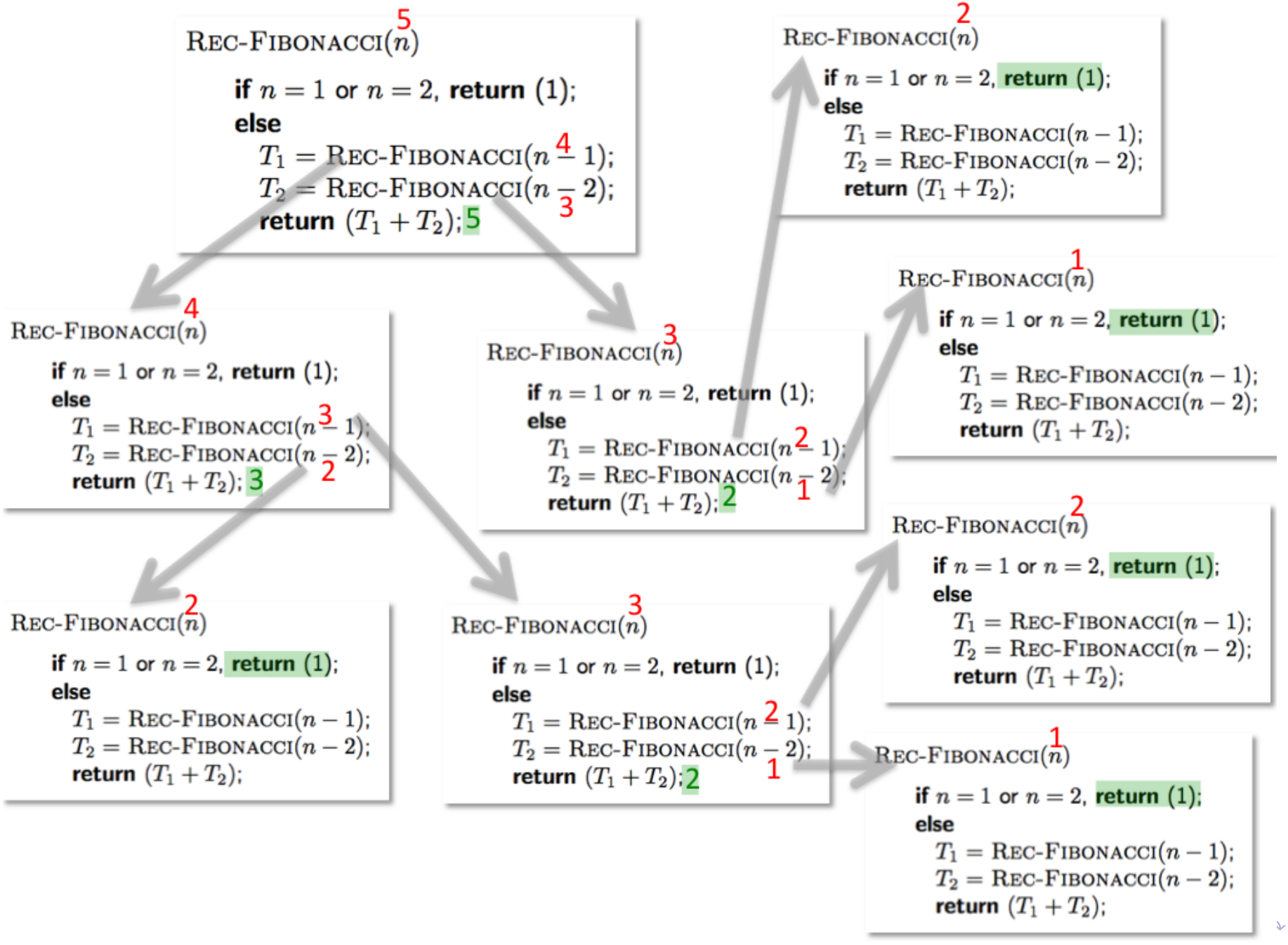
if $n = 1$ or $n = 2$, **return** (1);
else
$T_1 = \text{REC-FIBONACCI}(n-1)$;
$T_2 = \text{REC-FIBONACCI}(n-2)$;
return $(T_1 + T_2)$;

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

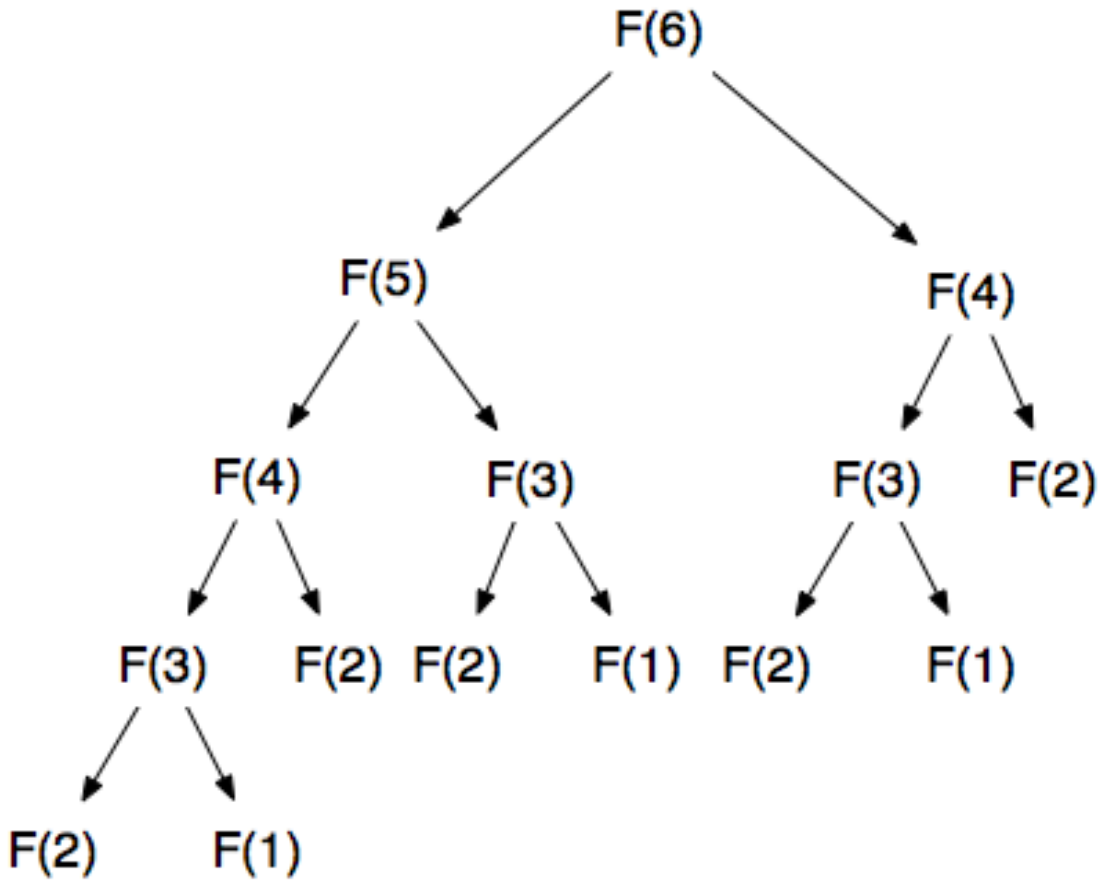
its execution is via a *run-time stack*

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

Part I. Foundations

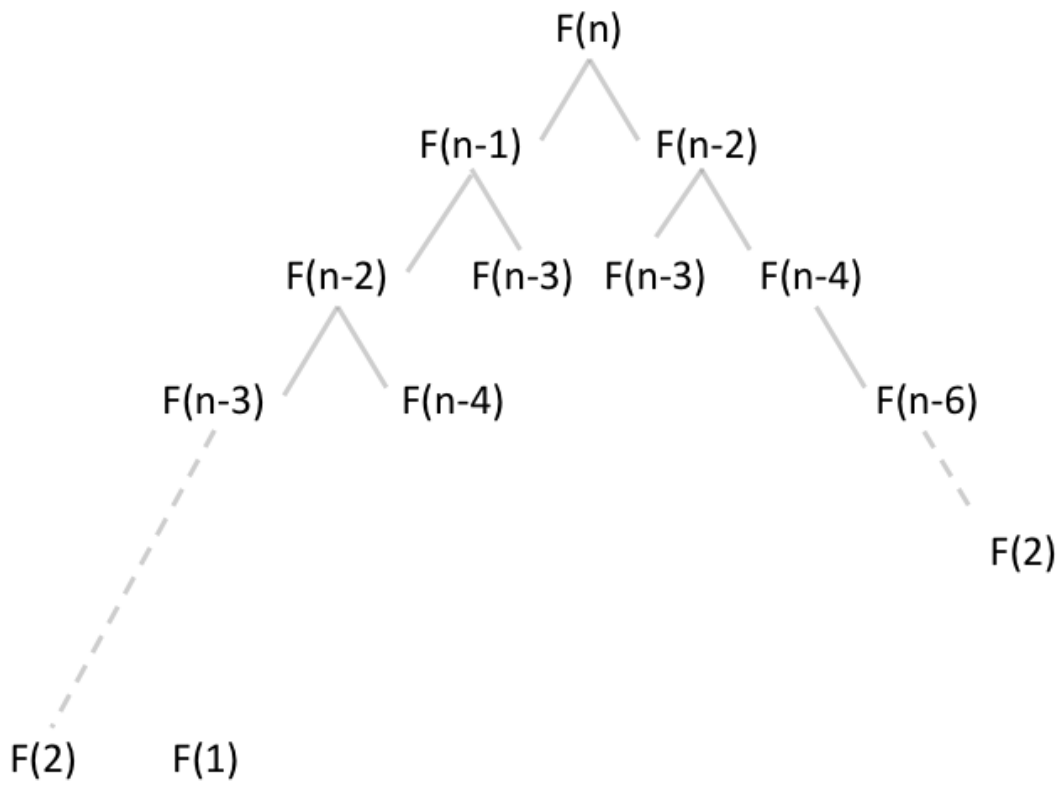


Part I. Foundations



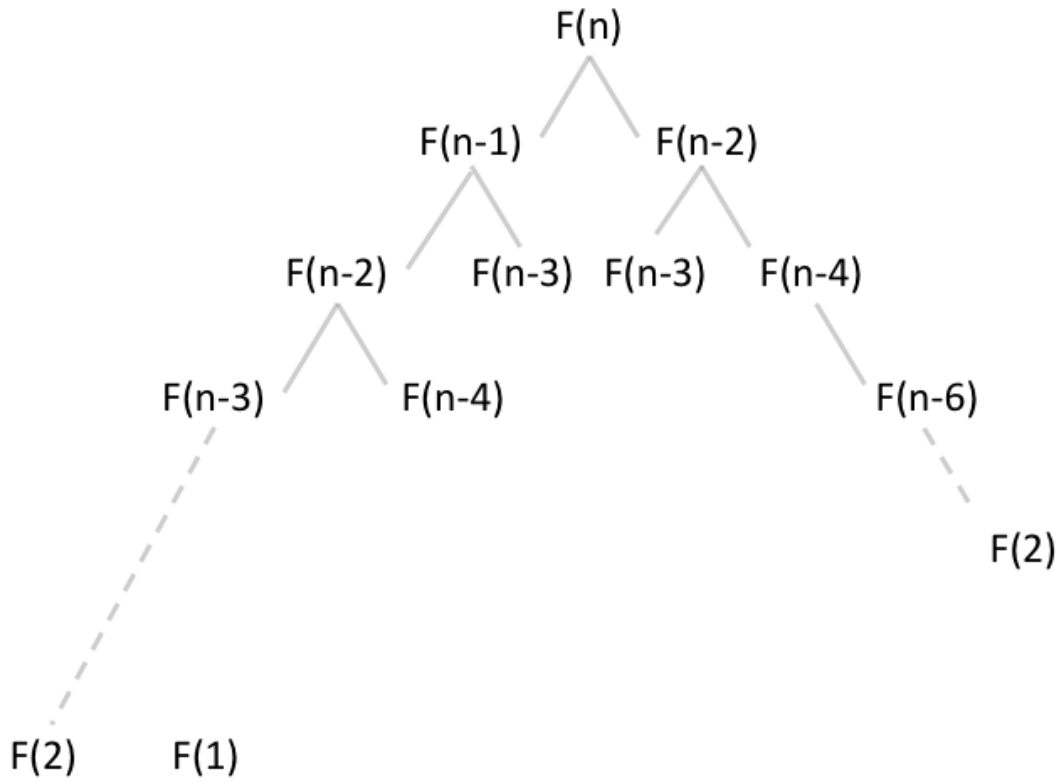
Repeated computations everywhere!

Part I. Foundations



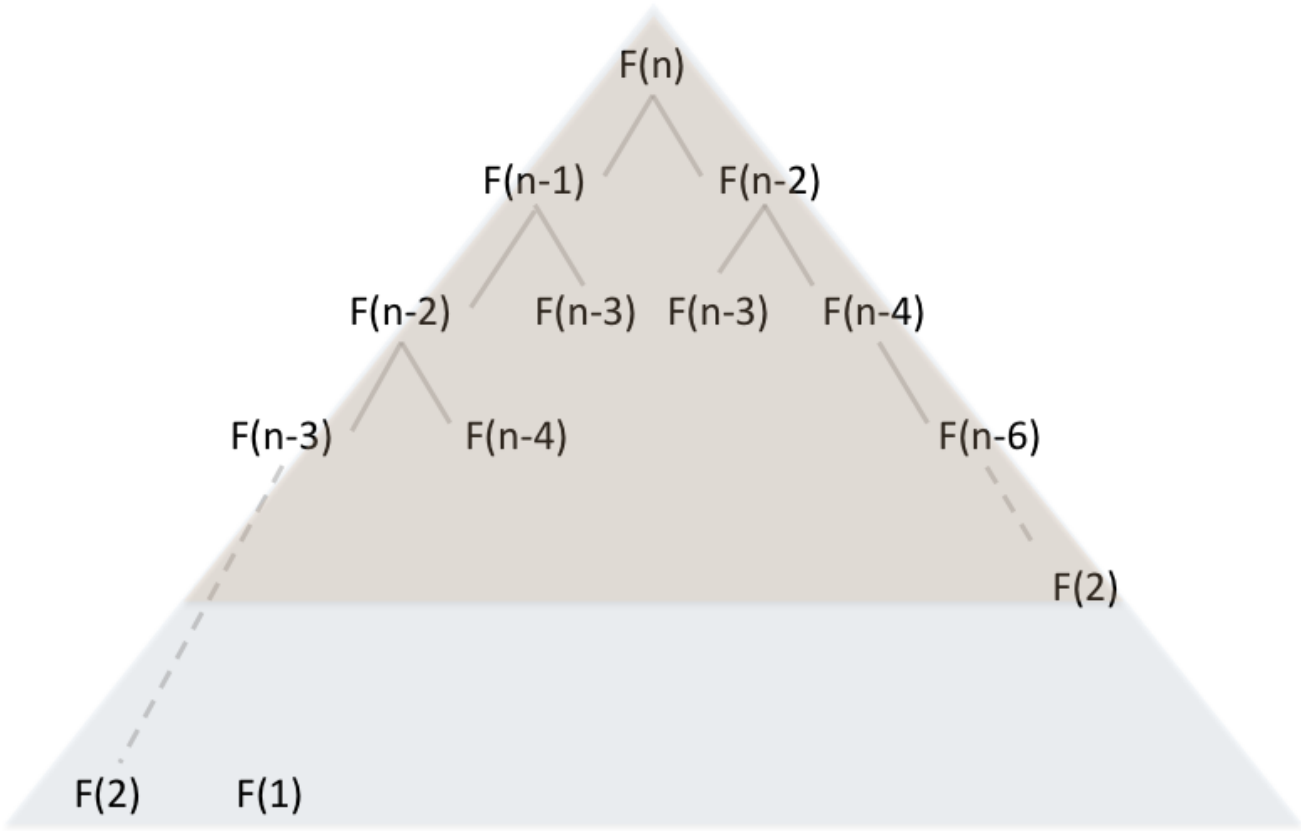
The size of tree is the number of recursive calls;

Part I. Foundations

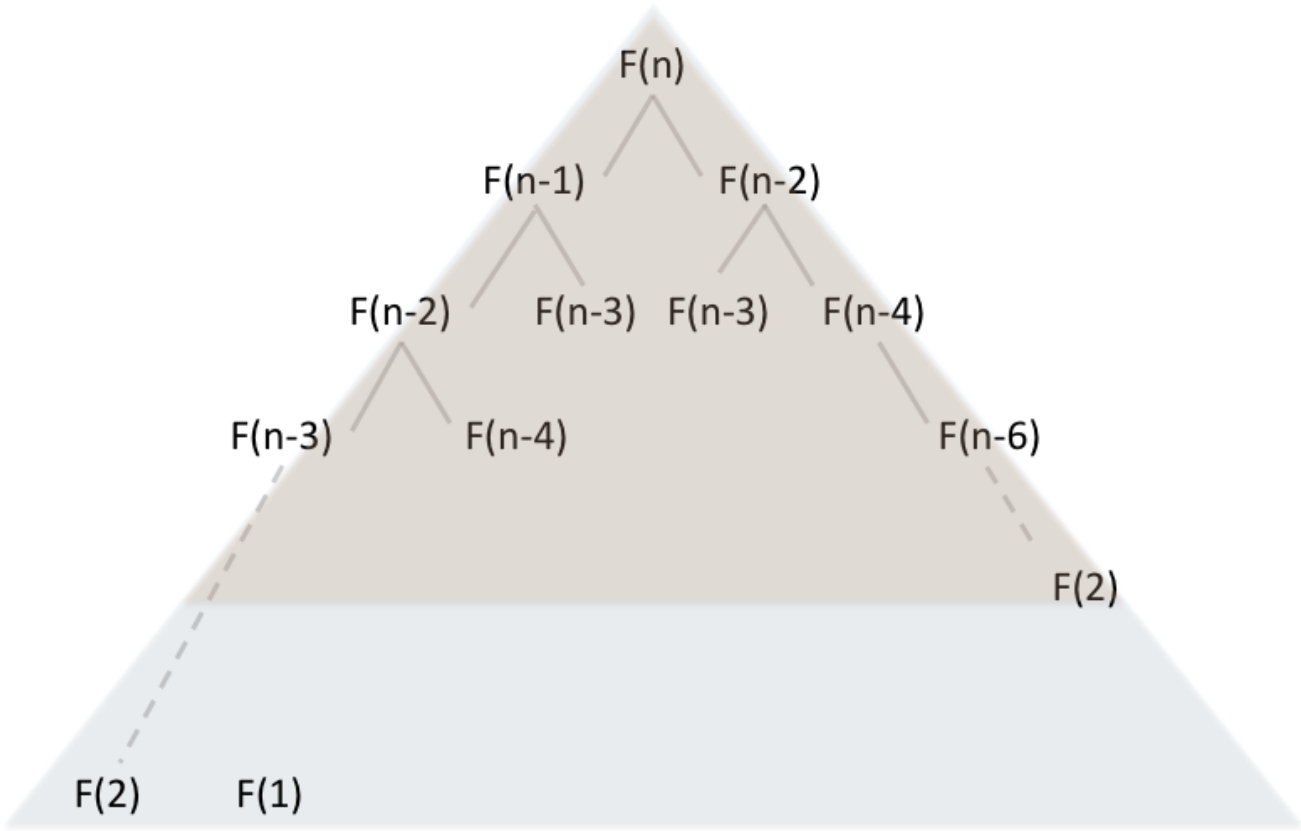


The size of tree is the number of recursive calls;
How big is it?

Part I. Foundations

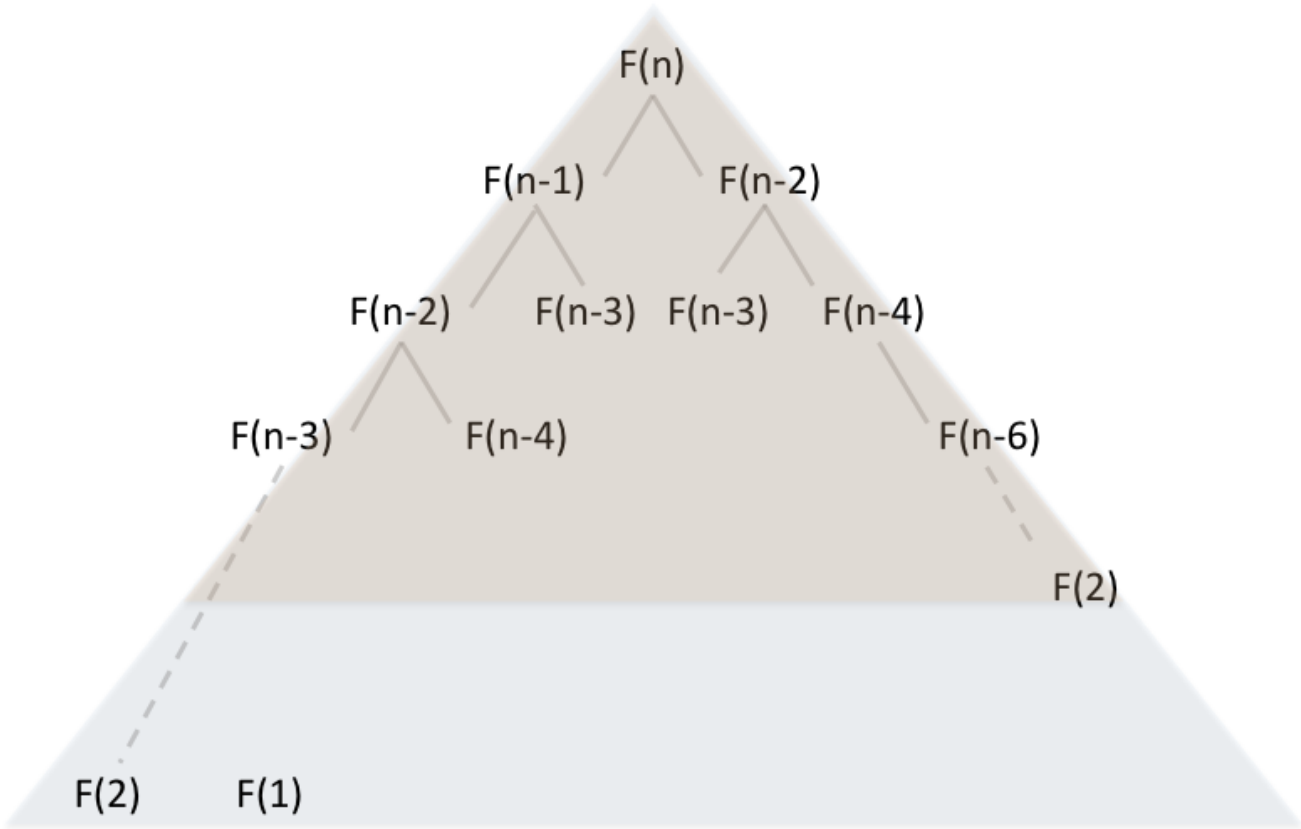


Part I. Foundations



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

Part I. Foundations



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

roughly: $2^{\frac{n}{2}} \leq$ size of tree $\leq 2^n$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

Part I. Foundations

ITERATIVE-FIBONACCI(n)

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

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ or $n = 2$ **return** (1);
else
$M[1] = 1$, $M[2] = 1$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

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

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

where $T_{if} = c_1$,

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for}$

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

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

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

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

$$T_{total} \leq c_1$$

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

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

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

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**
    $M[i] = M[i-1] + M[i-2]$
  return $(M[n])$

How fast is it?

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

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

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

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**
    $M[i] = M[i-1] + M[i-2]$
  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 ITERATIVE-FIBONACCI(n) does is:

Part I. Foundations

Actually, all the algorithm ITERATIVE-FIBONACCI(n) does is:

To fill out a table of size n, with

Part I. Foundations

Actually, all the algorithm 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 ITERATIVE-FIBONACCI(n) does is:

To fill out a table of size n, with

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

Part I. Foundations

Actually, all the algorithm 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 ITERATIVE-FIBONACCI(n) uses is

Part I. Foundations

Actually, all the algorithm 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 ITERATIVE-FIBONACCI(n) uses is

$$T(n) = c \times n$$

Chapter 1. The role of algorithms in computing

Chapter 1. The role of algorithms in computing

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

Chapter 1. The role of algorithms in computing

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

Example 2: An algorithm skeleton;

Algorithm MAXIMUM;

INPUT: list $X = \{a_1, \cdots, a_n\}$;
Body that is a series of instructions;
OUTPUT: y, the maximum of $a_1, \cdots, a_n$.

Chapter 1. The Role of Algorithms in Computing

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

Chapter 1. The Role of Algorithms in Computing

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

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

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

.

Chapter 1. The Role of Algorithms in Computing

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

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

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

.

For some problems, the functions computed are predicates, i.e., output $y \in \{\texttt{TRUE}, \texttt{FALSE}\}$

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 $\langle a_1, \cdots, a_n \rangle$;

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

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 INSERTION-SORT(A)

1 **for** $j = 2$ **to** $length[A]$ **do**

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1    for j = 2 to length[A] do
2        key = A[j]
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1   for j = 2 to length[A] do
2       key = A[j]
3       {Insert A[j] into sorted A[1..j − 1]}
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1   for j = 2 to length[A] do
2       key = A[j]
3       {Insert A[j] into sorted A[1..j − 1]}
4       i = j − 1
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1   for j = 2 to length[A] do
2       key = A[j]
3       {Insert A[j] into sorted A[1..j − 1]}
4       i = j − 1
5       while i > 0 and A[i] > key
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

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

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

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

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

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

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

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

Analysis of the algorithm:

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

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

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

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

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

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

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

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Finding loop invariant becomes necessary and sufficient.

The most difficult task is to do this for a loop statement.

Finding loop invariant becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

The most difficult task is to do this for a loop statement.

Finding loop invariant becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

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

The most difficult task is to do this for a loop statement.

Finding loop invariant becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

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

However, finding loop invariants is difficult!

Chapter 2. Getting Started

Efficiency analysis: This is to show that

Chapter 2. Getting Started

Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.

Chapter 2. Getting Started

Efficiency analysis: This is to show that

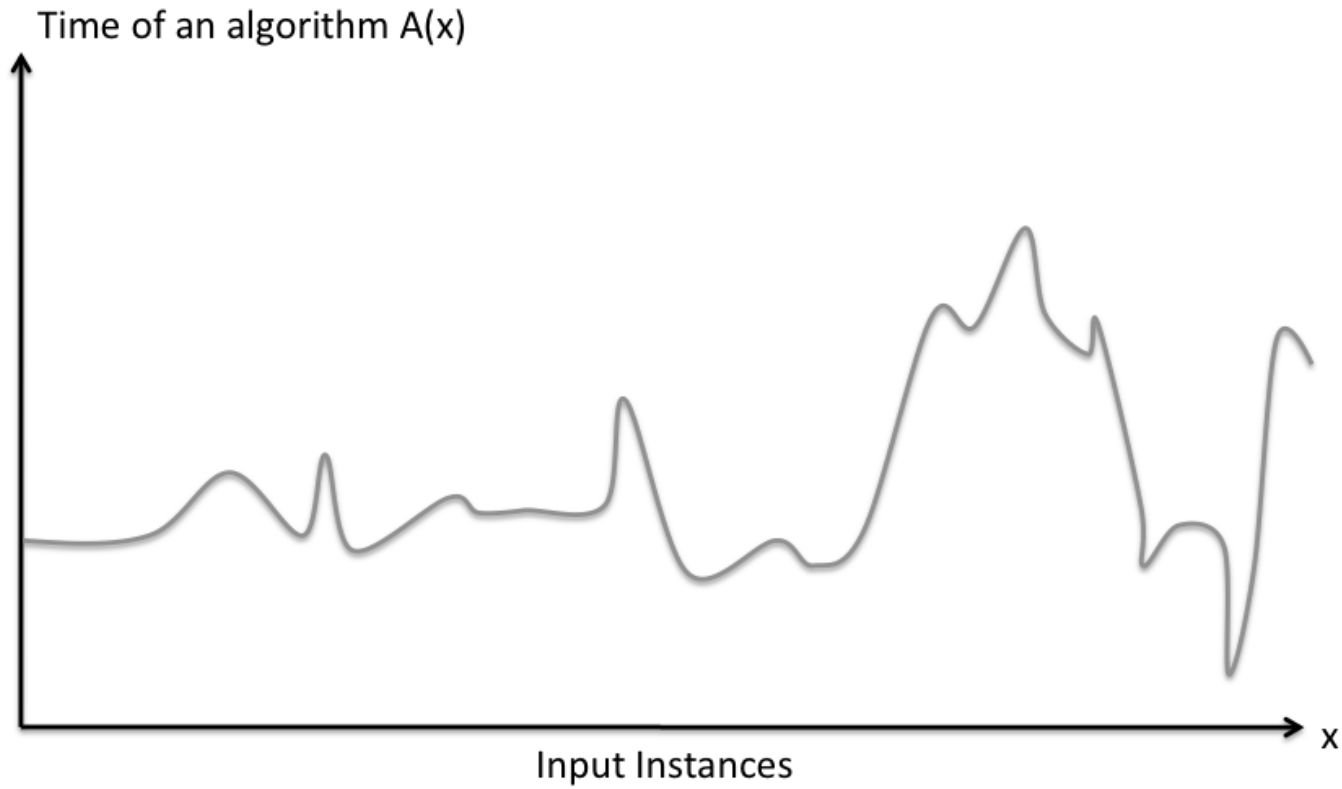
- For all cases of input, the needed computation resources for the algorithm.
- resources can be CPU time and memory space used in the computation.

Chapter 2. Getting Started

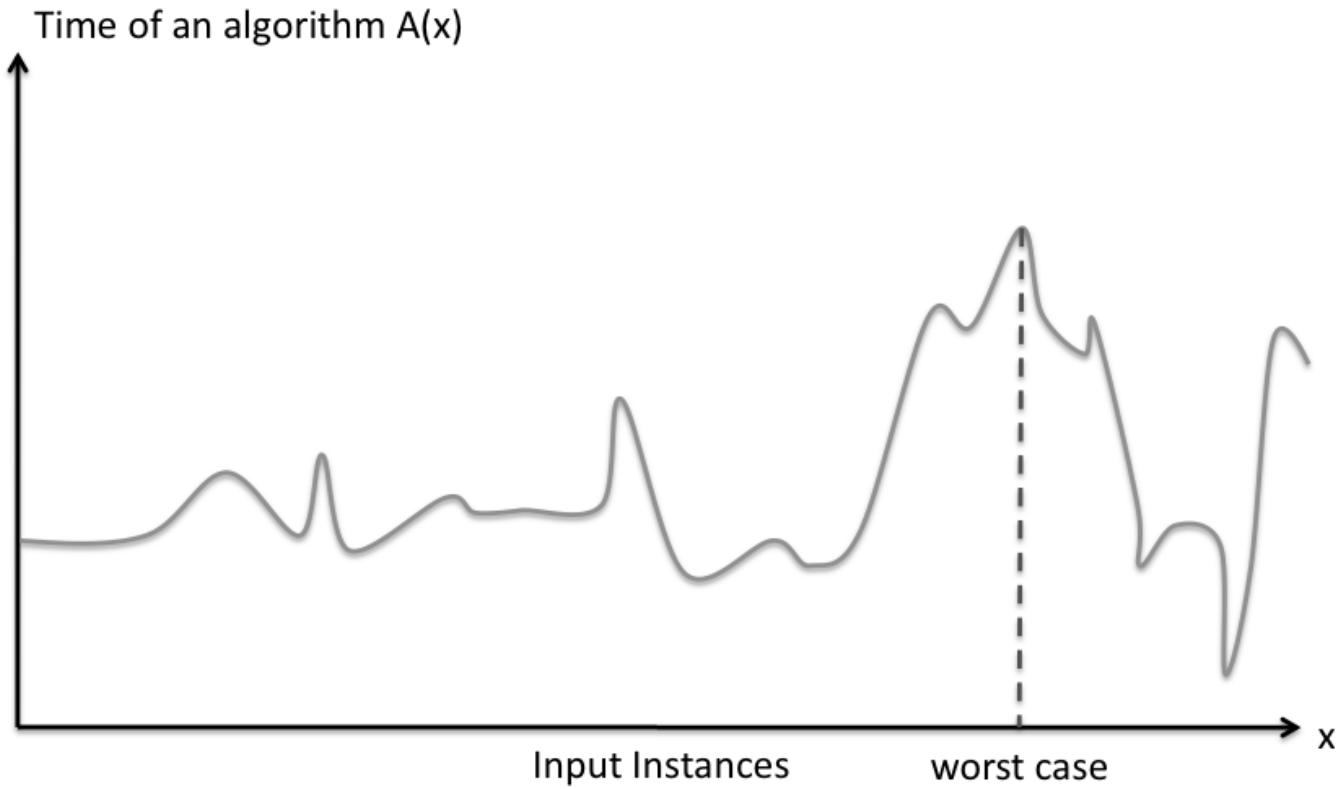
Efficiency analysis: This is to show that

- 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.

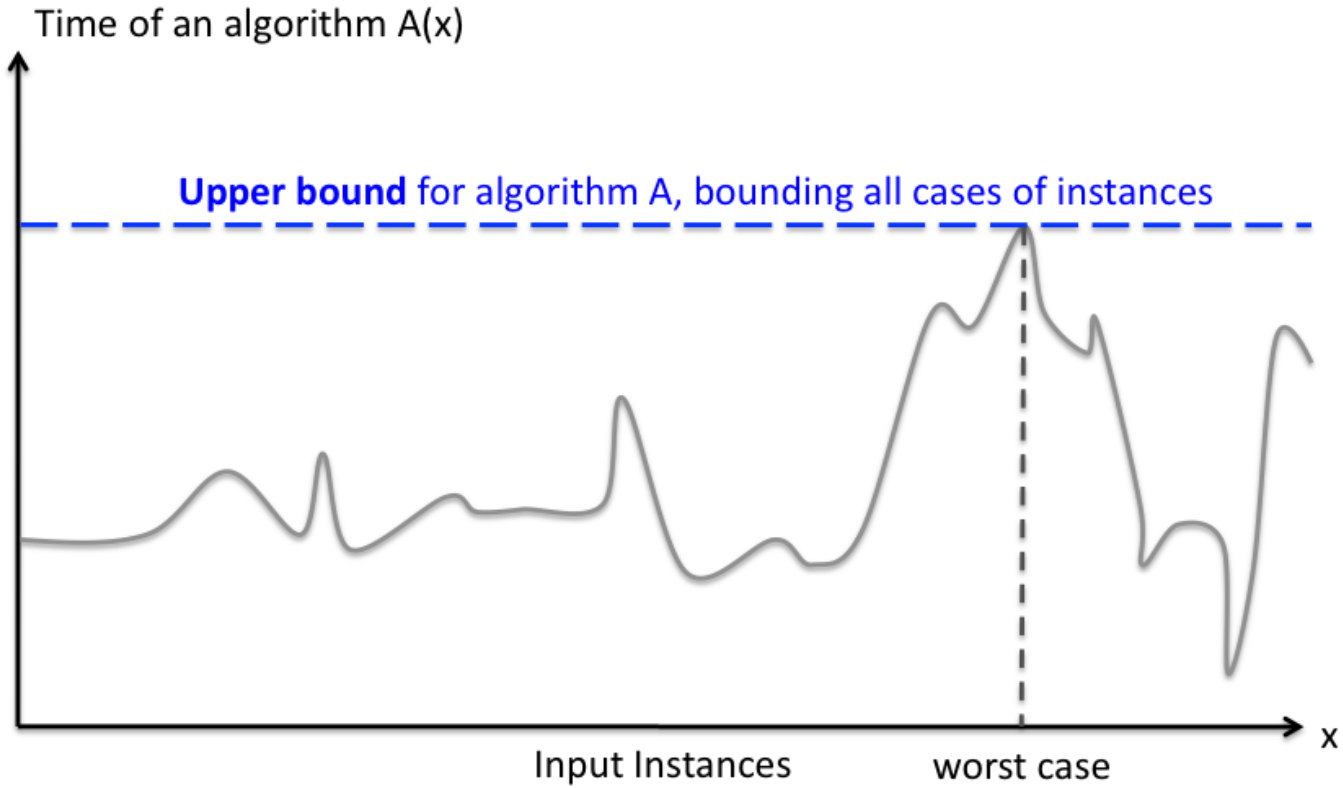
Chapter 2. Getting Started



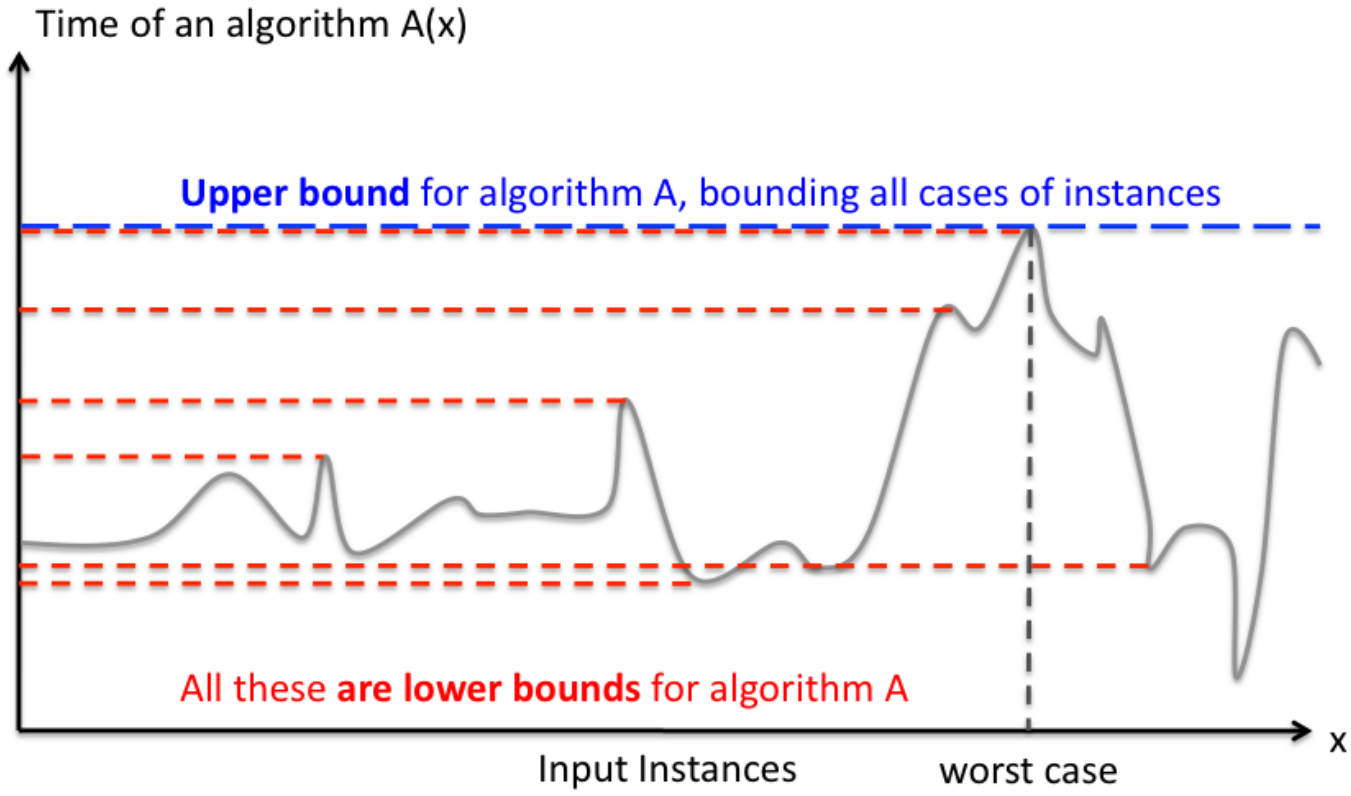
Chapter 2. Getting Started



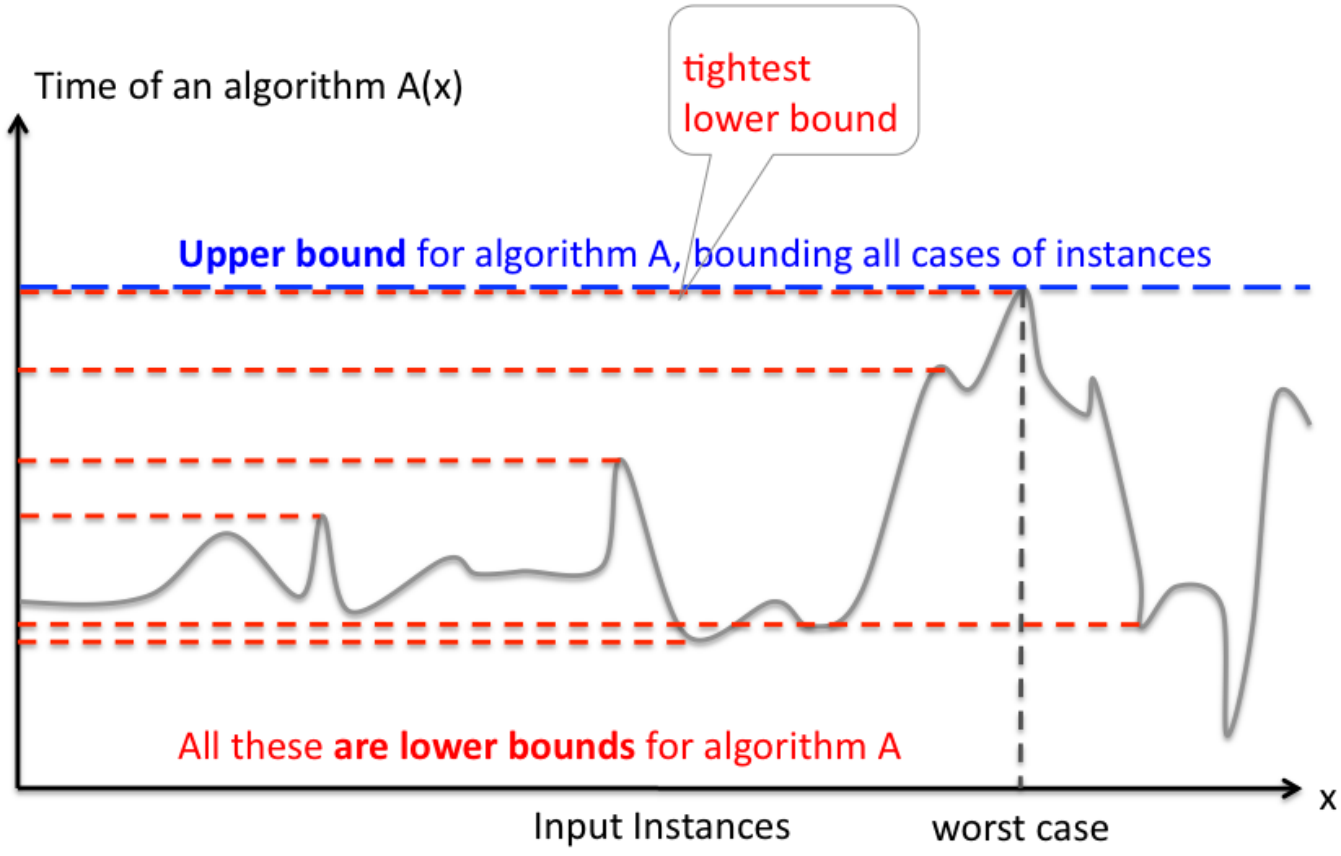
Chapter 2. Getting Started



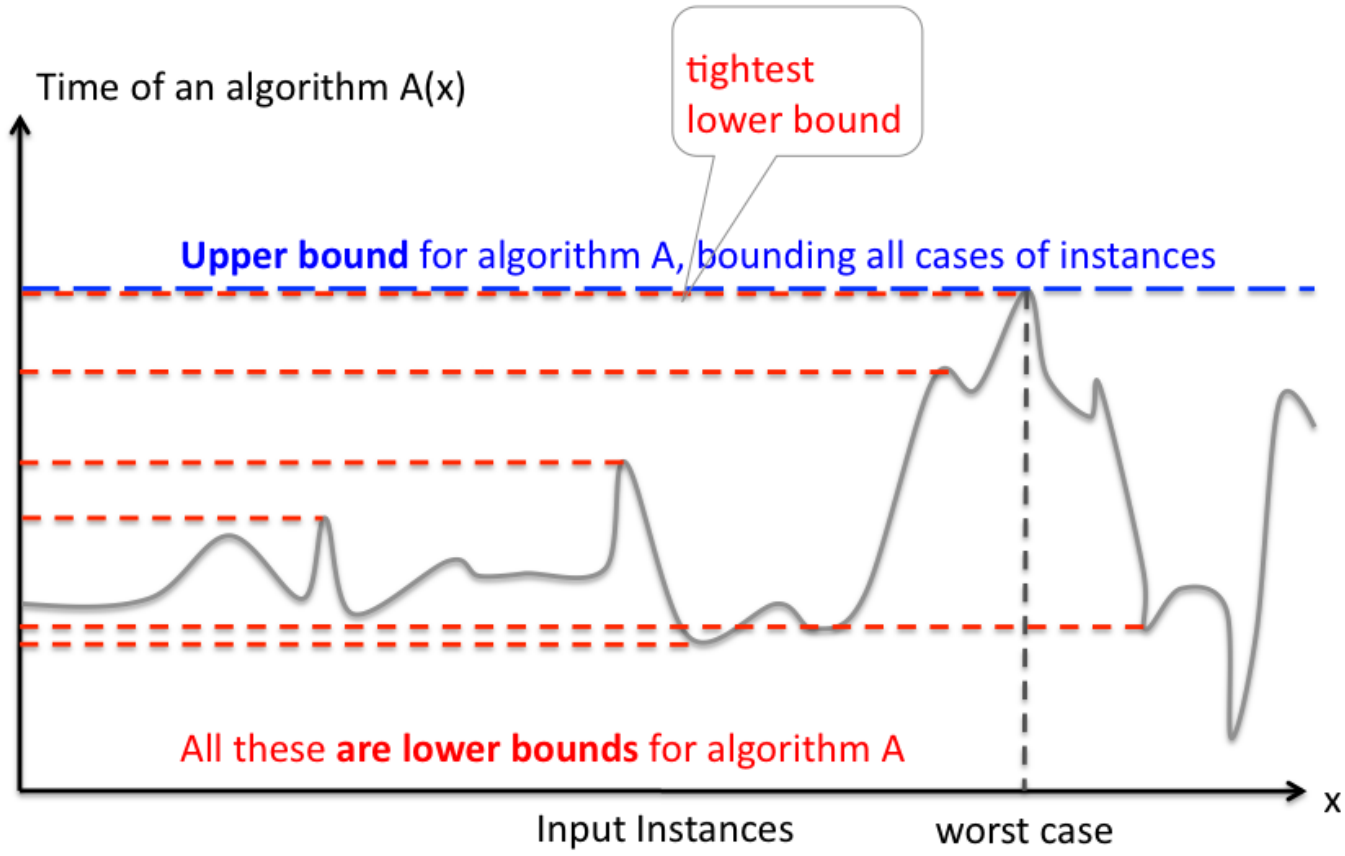
Chapter 2. Getting Started



Chapter 2. Getting Started



Chapter 2. Getting Started



lower bounds ≤ worst case time ≤ upper bounds

Chapter 2. Getting Started

Resource measurement based on

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
- counting primitive operations: addition, substraction, floor, ceiling, multiplication, jump, memory movement,

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
- counting primitive operations: addition, substraction, 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, substraction, floor, ceiling, multiplication, jump, memory movement,

  these operations differs in time by a constant multiplicative factor.
- speed between different machines: a constant multiplicative factor.

Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

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

Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

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

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

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

Chapter 2. Getting Started

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

Chapter 2. Getting Started

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

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

for some constants a, b, c,

Chapter 2. Getting Started

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

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

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

Chapter 2. Getting Started

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

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

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

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

Chapter 2. Getting Started

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

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

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

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

$$T(n) \leq a\frac{n}{2}(n+1) + bn + c - a$$

Chapter 2. Getting Started

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

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

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

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

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

for some constants x, y, z.

Chapter 2. Getting Started

So we have proved:

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

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?)

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

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.,

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

$\leq$ means at least one case for which $\leq$ 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.,

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

$\leq$ means at least one case for which $\leq$ holds;

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

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.,

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

$\leq$ 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:

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

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;

$xn^2 + yn + z$ is a complexity upper bound for $T(n)$

Chapter 2. Getting Started

Chapter 2. Getting Started

Important complexity issues:

Chapter 2. Getting Started

Important complexity issues:

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

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.

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.

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.

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.

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

Chapter 2. Getting Started

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

Chapter 2. Getting Started

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.

Chapter 2. Getting Started

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

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

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

Chapter 3. Growth of Functions

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

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

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

Chapter 3. Growth of Functions

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

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

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

In general,

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

Chapter 3. Growth of Functions

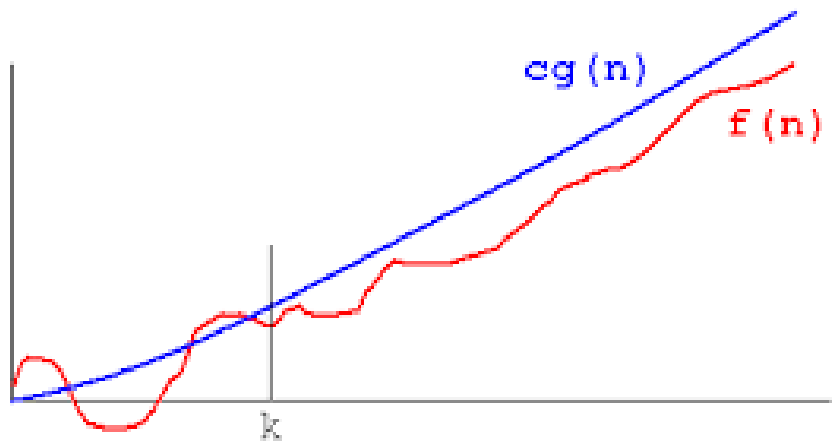
Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

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

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

In general,

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



Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

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

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:

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:

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

Chapter 3. Growth of Functions

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 $O(n^2)$.

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

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 $O(n)$.

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!

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

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 $O(n^2)$.

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 $O(n^2/(log_2^n)^2)$.

Chapter 3. Growth of Functions

Could you also correct the time complexities for INSERTION SORT and REC-FIBONACCI measured by the input size?

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in 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$(A.p, q, r)$

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

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 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

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

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?)

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?)

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

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$T(\frac{n}{2}) \leq 2T(\frac{n}{2^2}) + \frac{n}{2} + c$
$T(\frac{n}{2^2}) \leq 2T(\frac{n}{2^3}) + \frac{n}{2^2} + c$
$\ldots$
$T(\frac{n}{2^h}) \leq 2T(\frac{n}{2^{h+1}}) + \frac{n}{2^h} + c$ where $\frac{n}{2^{h+1}} = 1$

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

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

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

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$T(\frac{n}{2}) \leq 2T(\frac{n}{2^2}) + \frac{n}{2} + c$
$T(\frac{n}{2^2}) \leq 2T(\frac{n}{2^3}) + \frac{n}{2^2} + c$
$\ldots$
$T(\frac{n}{2^h}) \leq 2T(\frac{n}{2^{h+1}}) + \frac{n}{2^h} + c$ where $\frac{n}{2^{h+1}} = 1$

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

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

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$\begin{aligned}
&T(n) \leq 2T(\tfrac{n}{2}) + n + c \\
&T(\tfrac{n}{2}) \leq 2T(\tfrac{n}{2^2}) + \tfrac{n}{2} + c \\
&T(\tfrac{n}{2^2}) \leq 2T(\tfrac{n}{2^3}) + \tfrac{n}{2^2} + c \\
&\cdots \\
&T(\tfrac{n}{2^h}) \leq 2T(\tfrac{n}{2^{h+1}}) + \tfrac{n}{2^h} + c \qquad \text{where } \tfrac{n}{2^{h+1}} = 1
\end{aligned}$$

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

$$\begin{aligned}
&T(n) \leq 2T(\tfrac{n}{2}) + n + c \\
&2T(\tfrac{n}{2}) \leq 2^2 T(\tfrac{n}{2^2}) + 2 \times \tfrac{n}{2} + 2c \\
&2^2 T(\tfrac{n}{2^2}) \leq 2^3 T(\tfrac{n}{2^3}) + 2^2 \times \tfrac{n}{2^2} + 2^2 c
\end{aligned}$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$T(\frac{n}{2}) \leq 2T(\frac{n}{2^2}) + \frac{n}{2} + c$
$T(\frac{n}{2^2}) \leq 2T(\frac{n}{2^3}) + \frac{n}{2^2} + c$
$\ldots$
$T(\frac{n}{2^h}) \leq 2T(\frac{n}{2^{h+1}}) + \frac{n}{2^h} + c$ where $\frac{n}{2^{h+1}} = 1$

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

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$2T(\frac{n}{2}) \leq 2^2T(\frac{n}{2^2}) + 2 \times \frac{n}{2} + 2c$
$2^2T(\frac{n}{2^2}) \leq 2^3T(\frac{n}{2^3}) + 2^2 \times \frac{n}{2^2} + 2^2c$
$\ldots$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$T(\frac{n}{2}) \leq 2T(\frac{n}{2^2}) + \frac{n}{2} + c$
$T(\frac{n}{2^2}) \leq 2T(\frac{n}{2^3}) + \frac{n}{2^2} + c$
$\ldots$
$T(\frac{n}{2^h}) \leq 2T(\frac{n}{2^{h+1}}) + \frac{n}{2^h} + c$ where $\frac{n}{2^{h+1}} = 1$

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

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$2T(\frac{n}{2}) \leq 2^2T(\frac{n}{2^2}) + 2 \times \frac{n}{2} + 2c$
$2^2T(\frac{n}{2^2}) \leq 2^3T(\frac{n}{2^3}) + 2^2 \times \frac{n}{2^2} + 2^2c$
$\ldots$
$2^hT(\frac{n}{2^h}) \leq 2^{h+1}T(\frac{n}{2^{h+1}}) + 2^h \times \frac{n}{2^h} + 2^hc$ where $\frac{n}{2^{h+1}} = 1$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$T(\frac{n}{2}) \leq 2T(\frac{n}{2^2}) + \frac{n}{2} + c$
$T(\frac{n}{2^2}) \leq 2T(\frac{n}{2^3}) + \frac{n}{2^2} + c$
$\ldots$
$T(\frac{n}{2^h}) \leq 2T(\frac{n}{2^{h+1}}) + \frac{n}{2^h} + c$ where $\frac{n}{2^{h+1}} = 1$

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

$T(n) \leq 2T(\frac{n}{2}) + n + c$
$2T(\frac{n}{2}) \leq 2^2T(\frac{n}{2^2}) + 2 \times \frac{n}{2} + 2c$
$2^2T(\frac{n}{2^2}) \leq 2^3T(\frac{n}{2^3}) + 2^2 \times \frac{n}{2^2} + 2^2c$
$\ldots$
$2^hT(\frac{n}{2^h}) \leq 2^{h+1}T(\frac{n}{2^{h+1}}) + 2^h \times \frac{n}{2^h} + 2^hc$ where $\frac{n}{2^{h+1}} = 1$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T(\frac{n}{2}) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$\begin{aligned}
&T(n) \leq 2T(\tfrac{n}{2}) + n + c \\
&T(\tfrac{n}{2}) \leq 2T(\tfrac{n}{2^2}) + \tfrac{n}{2} + c \\
&T(\tfrac{n}{2^2}) \leq 2T(\tfrac{n}{2^3}) + \tfrac{n}{2^2} + c \\
&\cdots \\
&T(\tfrac{n}{2^h}) \leq 2T(\tfrac{n}{2^{h+1}}) + \tfrac{n}{2^h} + c \qquad \text{where } \tfrac{n}{2^{h+1}} = 1
\end{aligned}$$

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

$$\begin{aligned}
&T(n) \leq 2T(\tfrac{n}{2}) + n + c \\
&2T(\tfrac{n}{2}) \leq 2^2T(\tfrac{n}{2^2}) + 2 \times \tfrac{n}{2} + 2c \\
&2^2T(\tfrac{n}{2^2}) \leq 2^3T(\tfrac{n}{2^3}) + 2^2 \times \tfrac{n}{2^2} + 2^2c \\
&\cdots \\
&2^hT(\tfrac{n}{2^h}) \leq 2^{h+1}T(\tfrac{n}{2^{h+1}}) + 2^h \times \tfrac{n}{2^h} + 2^hc \qquad \text{where } \tfrac{n}{2^{h+1}} = 1 \\
+ \;&\text{---} \\
&T(n) \leq 2^{h+1}T(\tfrac{n}{2^{h+1}}) + (h+1)n + c\sum_{i=0}^{h} 2^i
\end{aligned}$$

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

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

Chapter 4. Solving Recurrences

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

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

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

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

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

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

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

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n\log_2 n + c(n-1) \end{aligned}$$

Chapter 4. Solving Recurrences

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

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

$$\begin{aligned} T(n) &\leq 2^{h+1} T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n-1) \\ &= n \log_2 n + 2cn - c \end{aligned}$$

Chapter 4. Solving Recurrences

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

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

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n\log_2 n + c(n-1) \\ &= n\log_2 n + 2cn - c \\ &= O(n\log_2 n) \end{aligned}$$

Chapter 4. Solving Recurrences

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

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

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n\log_2 n + c(n-1) \\ &= n\log_2 n + 2cn - c \\ &= O(n\log_2 n) \end{aligned}$$

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

$$n\log_2 n + 2cn - c \leq an\log_2 n \qquad (1)$$

when $n > k$.

Chapter 4. Solving Recurrences

$$T(n) \le 2^{h+1}T(\tfrac{n}{2^{h+1}}) + (h+1)n + c\sum_{i=0}^{h} 2^i$$

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

$$\begin{aligned} T(n) &\le 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\le cn + n\log_2 n + c(n-1) \\ &= n\log_2 n + 2cn - c \\ &= O(n\log_2 n) \end{aligned}$$

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

$$n\log_2 n + 2cn - c \le an\log_2 n \qquad (1)$$

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

$$T(n) \le 2^{h+1}T(\tfrac{n}{2^{h+1}}) + (h+1)n + c\sum_{i=0}^{h} 2^i$$

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

$$\begin{aligned} T(n) &\le 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\le cn + n\log_2 n + c(n-1) \\ &= n\log_2 n + 2cn - c \\ &= O(n\log_2 n) \end{aligned}$$

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

$$n\log_2 n + 2cn - c \le an\log_2 n \qquad (1)$$

when $n > k$.

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

That is, $n\log_2 n + 2cn - c \le 2n\log_2 n$ when $n > k = 2^{2c}$.

Chapter 4. Solving Recurrences

$$T(n) \le 2^{h+1}T(\tfrac{n}{2^{h+1}}) + (h+1)n + c\sum_{i=0}^{h} 2^i$$

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

$$\begin{aligned} T(n) &\le 2^{h+1}T(1) + n\log_2 n + c(2^{h+1} - 1) \\ &\le cn + n\log_2 n + c(n-1) \\ &= n\log_2 n + 2cn - c \\ &= O(n\log_2 n) \end{aligned}$$

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

$$n\log_2 n + 2cn - c \le an\log_2 n \qquad (1)$$

when $n > k$.

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

That is, $n\log_2 n + 2cn - c \le 2n\log_2 n$ when $n > k = 2^{2c}$.
So

$$T(n) = O(n\log_2 n)$$

.

Chapter 4. Solving Recurrences

Methods for solving recurrences

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

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

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

Chapter 4. Solving Recurrences

Methods for solving recurrences

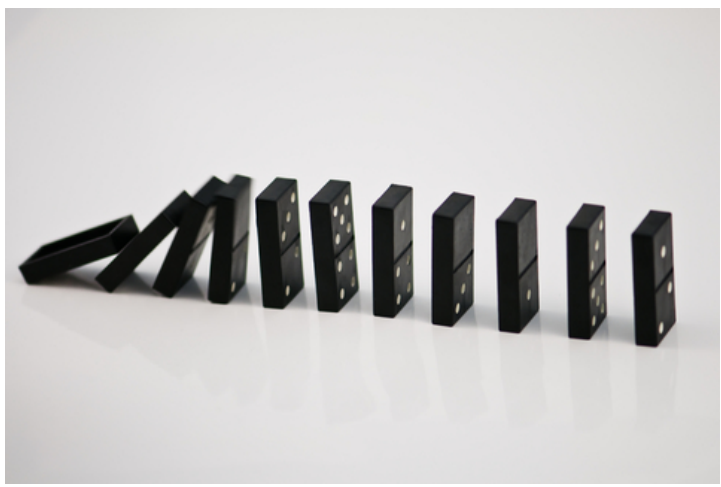
1. Substitution method (based on math induction)

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

"The principle for dominos to fall".



Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;

Chapter 4. Solving Recurrences

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: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

Chapter 4. Solving Recurrences

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: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

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

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

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

Chapter 4. Solving Recurrences

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: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

- other variants: e.g., $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k)$
  but we need to make sure all n's beyond the base cases are covered.

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

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

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

- other variants: e.g., $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k)$
  but we need to make sure all n's beyond the base cases are covered.

  $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k) \wedge \mathcal{P}(2k+1)$

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)

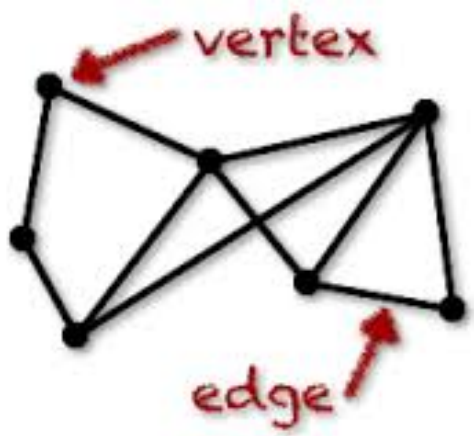
- Observation: every number should be compared *directly* or *indirectly* with the maximum.
- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.

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.

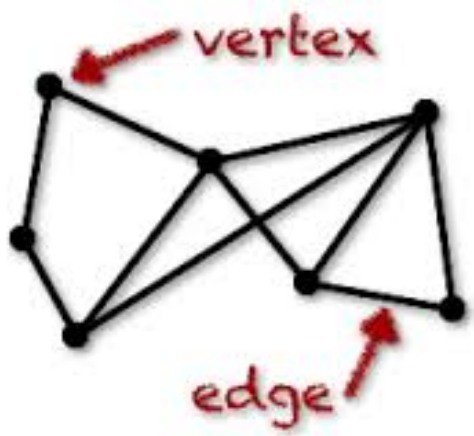


Chapter 4. Solving Recurrences

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.

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.

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

$$\geq \sum_{i=1}^{t}(m_i - 1) + r = \sum_{i=1}^{t} m_i - \sum_{i=1}^{t} 1 + r = \sum_{i=1}^{t} m_i + r - t \geq \sum_{i=1}^{t} m_i = k$$

Chapter 4. Solving Recurrences

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 $(A, key, k+1, j)$

Chapter 4. Solving Recurrences

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,

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

where $c > 0, d > 0$ are constants.

Chapter 4. Solving Recurrences

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

Chapter 4. Solving Recurrences

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

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

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

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

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

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

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

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

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

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

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

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

- 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

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

- 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

$$T(n) \le T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

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

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

$$T(n) \le a \log_2 n \quad \text{when } n \ge k \qquad (2)$$

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

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

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

Chapter 4. Solving Recurrences

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

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

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

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

- 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

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

- 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

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

- 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

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

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

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

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

- 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 $\lfloor \frac{n}{2} \rfloor$)

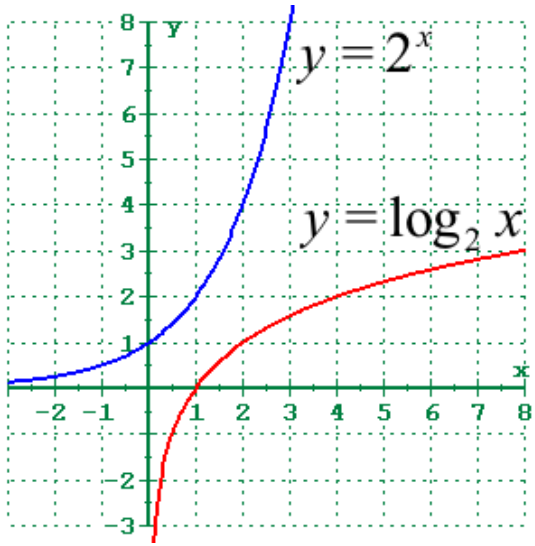
- Then we need to find the k such that (2) holds for all $n \geq k$.

  $k = 1$ 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 $T(2) \leq a \log_2 2$.

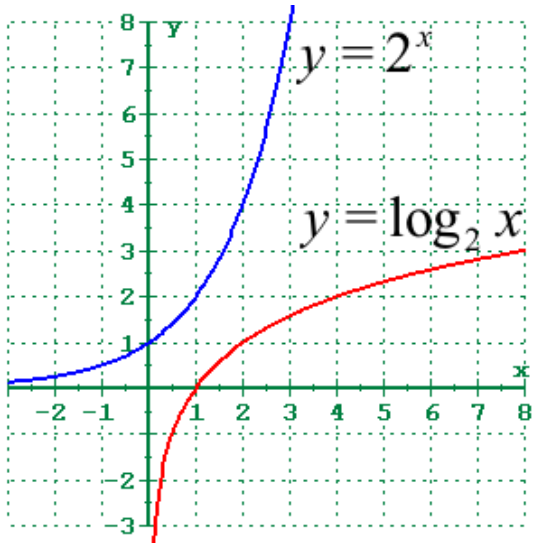
Chapter 4. Solving Recurrences

A little review on logarithm functions:



Chapter 4. Solving Recurrences

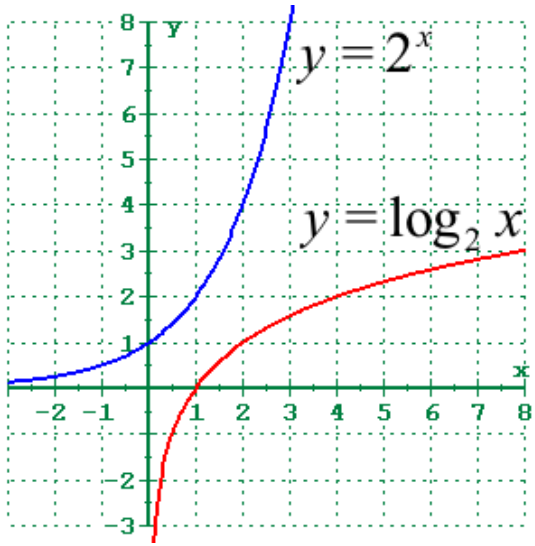
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;

Chapter 4. Solving Recurrences

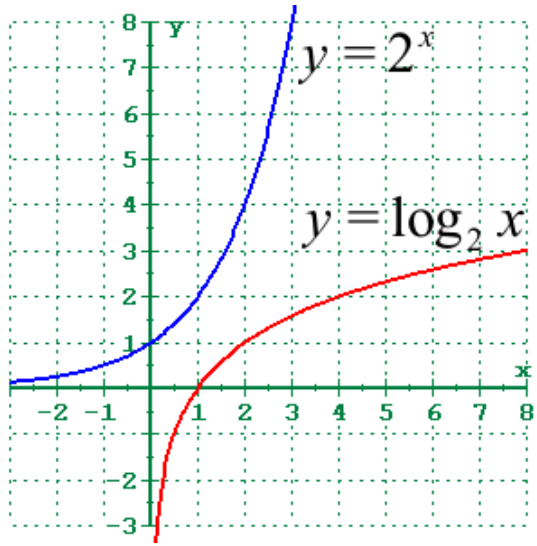
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;

Chapter 4. Solving Recurrences

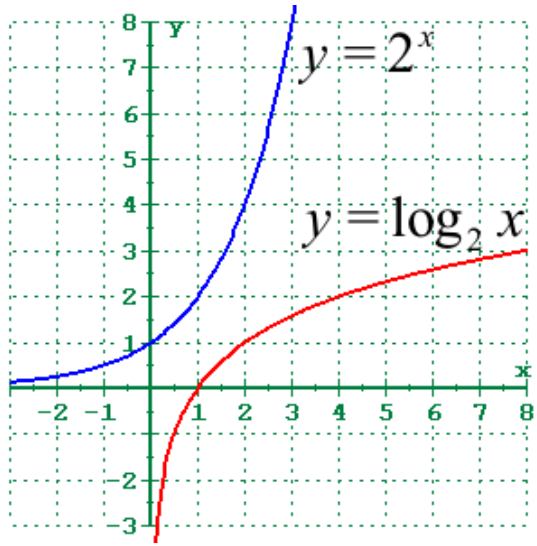
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;

Chapter 4. Solving Recurrences

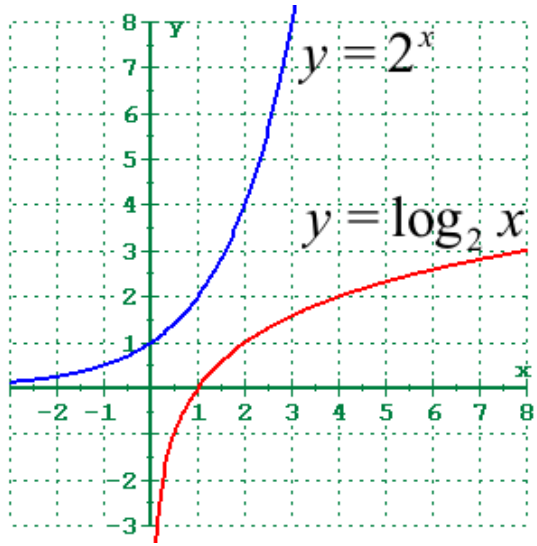
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;
- $\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n$;

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)

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

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

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

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

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

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

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

Verify:

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.

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)

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

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

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 $n = 1$.

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.,

$$T(\lfloor \tfrac{2n}{3} \rfloor) \leq c\lfloor \tfrac{2n}{3} \rfloor \log_2 \lfloor \tfrac{2n}{3} \rfloor + 2$$

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.,

$$T(\lfloor\tfrac{2n}{3}\rfloor) \leq c\lfloor\tfrac{2n}{3}\rfloor\log_2\lfloor\tfrac{2n}{3}\rfloor + 2$$

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)

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

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.,

$$T(\lfloor\tfrac{2n}{3}\rfloor) \leq c\lfloor\tfrac{2n}{3}\rfloor\log_2\lfloor\tfrac{2n}{3}\rfloor + 2$$

substitute it in the recurrence, we get

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

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.,

$$T(\lfloor\tfrac{2n}{3}\rfloor) \leq c\lfloor\tfrac{2n}{3}\rfloor\log_2\lfloor\tfrac{2n}{3}\rfloor + 2$$

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.,

$$T(\lfloor\tfrac{2n}{3}\rfloor) \leq c\lfloor\tfrac{2n}{3}\rfloor\log_2\lfloor\tfrac{2n}{3}\rfloor + 2$$

substitute it in the recurrence, we get

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

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

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.,

$$T(\lfloor\tfrac{2n}{3}\rfloor) \leq c\lfloor\tfrac{2n}{3}\rfloor\log_2\lfloor\tfrac{2n}{3}\rfloor + 2$$

substitute it in the recurrence, we get

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$

$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$

$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

To allow that the last term $\leq cn\log_2 n + 2$, the assumed upper bound,

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$
$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$
$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$
$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$
$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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?

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

$$= cn(\log_2 n + \log_2\tfrac{2}{3}) + 3 + n = cn\log_2 n + 3 + n - cn\log_2\tfrac{3}{2}$$
$$=\leq cn\log_2 n + 2 + 1 + n - cn\log_2\tfrac{3}{2}$$

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?

Does this also imply

$$T(n) = O(n\log_2 n) \text{ or } T(n) = O(n\log_2 n + 2)\,?$$

Chapter 4. Solving Recurrences

2. Changing variables

Chapter 4. Solving Recurrences

2. Changing variables

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

Chapter 4. Solving Recurrences

2. Changing variables

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

Define $m = \log_2 n$,

Chapter 4. Solving Recurrences

2. Changing variables

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

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

Chapter 4. Solving Recurrences

2. Changing variables

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

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

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

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: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

Chapter 4. Solving Recurrences

2. Changing variables

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

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

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

rename the function: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

solve it, we have $S(m) = O(m \log m)$

Chapter 4. Solving Recurrences

2. Changing variables

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

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

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

rename the function: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

solve it, we have $S(m) = O(m \log m)$

so $T(n) = T(2^m) = O(m \log m) = O(\log n \log \log n)$.

Chapter 4. Solving Recurrences

3. Recursive tree method

Chapter 4. Solving Recurrences

3. Recursive tree method

By unfolding the recurrence to make a recursive-tree.

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.

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

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) 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 $T(1) = 1$

Chapter 4. Solving Recurrences

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

l_0: $T(n)$

Chapter 4. Solving Recurrences

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

l_0:		$T(n)$		n^2
l_1:	$T(n/4)$	$T(n/4)$	$T(n/4)$	

Chapter 4. Solving Recurrences

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

$$
\begin{array}{lccccccc}
l_0: & & & T(n) & & & & n^2 \\
l_1: & & T(n/4) & & T(n/4) & & T(n/4) & 3(\frac{n}{4})^2 \\
l_2: & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & &
\end{array}
$$

Chapter 4. Solving Recurrences

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

l_0:		$T(n)$		n^2
l_1:	$T(n/4)$	$T(n/4)$	$T(n/4)$	$3(\frac{n}{4})^2$
l_2:	$T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$	$T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$	$T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$	$3^2(\frac{n}{4^2})^2$
l_3:	$\ldots\ldots$			

Chapter 4. Solving Recurrences

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

l_0:		$T(n)$		n^2
l_1:	$T(n/4)$	$T(n/4)$	$T(n/4)$	$3(\frac{n}{4})^2$
l_2:	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$3^2(\frac{n}{4^2})^2$
l_3:	.....			$3^3(\frac{n}{4^3})^2$
l_4:	.....			

Chapter 4. Solving Recurrences

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

l_0:		$T(n)$		n^2
l_1:	$T(n/4)$	$T(n/4)$	$T(n/4)$	$3(\frac{n}{4})^2$
l_2:	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$	$3^2(\frac{n}{4^2})^2$
l_3:	.....			$3^3(\frac{n}{4^3})^2$
l_4:	.....			
l_{m-1}:	.....			

Chapter 4. Solving Recurrences

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

$$
\begin{array}{lccccccc}
l_0: & & & T(n) & & & & n^2 \\
l_1: & & T(n/4) & & T(n/4) & & T(n/4) & 3(\frac{n}{4})^2 \\
l_2: & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & & T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2}) & & 3^2(\frac{n}{4^2})^2 \\
l_3: & \ldots\ldots & & & & & & 3^3(\frac{n}{4^3})^2 \\
l_4: & \ldots\ldots & & & & & & \\
l_{m-1}: & \ldots\ldots & & & & & & 3^{m-1}(\frac{n}{4^{m-1}})^2
\end{array}
$$

Chapter 4. Solving Recurrences

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

l_0: $T(n)$ n^2

l_1: $T(n/4)$ $T(n/4)$ $T(n/4)$ $3(\frac{n}{4})^2$

l_2: $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $3^2(\frac{n}{4^2})^2$

l_3: ..... $3^3(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... $3^{m-1}(\frac{n}{4^{m-1}})^2$

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

Chapter 4. Solving Recurrences

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

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

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

Chapter 4. Solving Recurrences

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

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

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

Then $T(n)$ is the sum

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

Chapter 4. Solving Recurrences

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

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

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

Then $T(n)$ is the sum

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

Chapter 4. Solving Recurrences

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

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

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

Then $T(n)$ is the sum

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

Chapter 4. Solving Recurrences

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

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

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

Then $T(n)$ is the sum

$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m T(1)$$
$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m \times 1$$
$$= n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{n}{4^{m-1}})^2] + 3^m(\tfrac{n}{4^m})^2$$
$$= n^2[1 + \tfrac{3}{16} + (\tfrac{3}{16})^2 + (\tfrac{3}{16})^3 + \cdots + (\tfrac{3}{16})^m]$$

Chapter 4. Solving Recurrences

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

l_0: $T(n)$ n^2

l_1: $T(n/4)$ $T(n/4)$ $T(n/4)$ $3(\frac{n}{4})^2$

l_2: $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $3^2(\frac{n}{4^2})^2$

l_3: ..... $3^3(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... $3^{m-1}(\frac{n}{4^{m-1}})^2$

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

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

Then $T(n)$ is the sum

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

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

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

$$= n^2[1 + \tfrac{3}{16} + (\tfrac{3}{16})^2 + (\tfrac{3}{16})^3 + \cdots + (\tfrac{3}{16})^m]$$

$$= n^2(\frac{1-(\frac{3}{16})^{m+1}}{1-\frac{3}{16}})$$

Chapter 4. Solving Recurrences

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

l_0: $T(n)$ n^2

l_1: $T(n/4)$ $T(n/4)$ $T(n/4)$ $3(\frac{n}{4})^2$

l_2: $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $T(\frac{n}{4^2})$ $3^2(\frac{n}{4^2})^2$

l_3: ..... $3^3(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... $3^{m-1}(\frac{n}{4^{m-1}})^2$

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

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

Then $T(n)$ is the sum

$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m T(1)$$
$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m \times 1$$
$$= n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{n}{4^{m-1}})^2] + 3^m(\tfrac{n}{4^m})^2$$
$$= n^2[1 + \tfrac{3}{16} + (\tfrac{3}{16})^2 + (\tfrac{3}{16})^3 + \cdots + (\tfrac{3}{16})^m]$$
$$= n^2(\frac{1-(\frac{3}{16})^{m+1}}{1-\frac{3}{16}})$$
$$\leq n^2(\frac{1}{1-\frac{3}{16}})$$

Chapter 4. Solving Recurrences

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

l_0: $T(n)$ n^2

l_1: $T(n/4)$ $T(n/4)$ $T(n/4)$ $3(\frac{n}{4})^2$

l_2: $T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$ $T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$ $T(\frac{n}{4^2})\ T(\frac{n}{4^2})\ T(\frac{n}{4^2})$ $3^2(\frac{n}{4^2})^2$

l_3: ..... $3^3(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... $3^{m-1}(\frac{n}{4^{m-1}})^2$

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

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

Then $T(n)$ is the sum

$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m T(1)$$
$$T(n) = n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{1}{4^{m-1}})^2] + 3^m \times 1$$
$$= n^2[1 + 3(\tfrac{1}{4})^2 + 3^2(\tfrac{1}{4^2})^2 + 3^3(\tfrac{1}{4^3})^2 + \cdots + 3^{m-1}(\tfrac{n}{4^{m-1}})^2] + 3^m(\tfrac{n}{4^m})^2$$
$$= n^2[1 + \tfrac{3}{16} + (\tfrac{3}{16})^2 + (\tfrac{3}{16})^3 + \cdots + (\tfrac{3}{16})^m]$$
$$= n^2(\frac{1-(\frac{3}{16})^{m+1}}{1-\frac{3}{16}})$$
$$\leq n^2(\frac{1}{1-\frac{3}{16}})$$
$$= \tfrac{16}{13}n^2$$

for all $n > 0$.

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

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

l_0: n^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: $(n/4)^2$ $(n/4)^2$ $(n/4)^2$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

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

l_0: n^2

l_1: $(n/4)^2$ $(n/4)^2$ $(n/4)^2$

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

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

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: .....

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

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: ……

l_4: ……

3^3 nodes of $(\frac{n}{4^3})^2$

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

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: ……

3^3 nodes of $(\frac{n}{4^3})^2$

l_4: ……

l_{m-1}: ……

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

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: ..... 3^3 nodes of $(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... 3^{m-1} of $(\frac{n}{4^{m-1}})^2$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

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

l_0: n^2

l_1: $(n/4)^2$ $(n/4)^2$ $(n/4)^2$

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: ..... 3^3 nodes of $(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... 3^{m-1} of $(\frac{n}{4^{m-1}})^2$

l_m: $T(1), T(1), T(1), T(1), T(1), \ldots,$ 3^m nodes of $T(1)$

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

l_2: $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$ $(\frac{n}{4^2})^2$

l_3: ..... 3^3 nodes of $(\frac{n}{4^3})^2$

l_4: .....

l_{m-1}: ..... 3^{m-1} of $(\frac{n}{4^{m-1}})^2$

l_m: $T(1), T(1), T(1), T(1), T(1), \ldots,$ 3^m nodes of $T(1)$

Chapter 5. Probabilistic analysis and randomized algorithms

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is "average" cases, not the worst case;

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is "average" cases, not the worst case;
- With assumption that input data are in a probabilistic distribution

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is "average" cases, not the worst case;
- With assumption that input data are in a probabilistic distribution
- Close relationship with randomized algorithms

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

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 $a > 0$.

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 $b > 0$.

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;

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.

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.

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.

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

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

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

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

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)

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)

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms

  - always gives answer correctly;

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

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, $Prob(T(n) \leq cn \log n) \geq 0.75$

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

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;

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;
    $Prob(\text{ to answer } 'NO' \text{ on } 'NO' \text{ instance}) = 1$

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, $Prob(T(n) \leq cn \log n) \geq 0.75$

- Monte Carlo algorithms

  - on 'NO' instances, 100% accuracy;
    $Prob(\text{ to answer } 'NO' \text{ on } 'NO' \text{ instance}) = 1$

  - on 'YES' instances, $\geq 75\%$ accuracy;

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;
    $Prob(\text{ to answer } 'NO' \text{ on } 'NO' \text{ instance}) = 1$

  - on 'YES' instances, $\geq 75\%$ accuracy;
    $Prob(\text{ to answer } 'YES' \text{ on } 'YES' \text{ instance}) \geq 0.75$

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, $Prob(T(n) \leq cn \log n) \geq 0.75$

- Monte Carlo algorithms

  - on 'NO' instances, 100% accuracy;
    $Prob(\text{ to answer } 'NO' \text{ on } 'NO' \text{ instance}) = 1$

  - on 'YES' instances, $\geq 75\%$ accuracy;
    $Prob(\text{ to answer } 'YES' \text{ on } 'YES' \text{ instance}) \geq 0.75$

    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, $Prob(T(n) \leq cn \log n) \geq 0.75$

- Monte Carlo algorithms

  - on 'NO' instances, 100% accuracy;
    $Prob(\text{ to answer } 'NO' \text{ on } 'NO' \text{ instance}) = 1$

  - on 'YES' instances, $\geq 75\%$ accuracy;
    $Prob(\text{ to answer } 'YES' \text{ on } 'YES' \text{ instance}) \geq 0.75$

    Accuracy 75% can be improved to 99.99% with multiple trials.

- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.