Design and Analysis of Algorithms

Faculty of Information Technology - MTA

Summer 2017

Outline for Today

- Course Information
- Algorithmic Analysis
  - Proving correctness with induction
  - Proving runtime with asymptotic (tiệm cận) analysis
- Divide and Conquer I
  - Mergesort

Course Information

Prof. David. and I

- I think algorithms are cool and useful.
- I want to convince of you that algorithms are cool and useful.

Course staff

- …

Course Information

Office: S1 1901

…

Prerequisites

CS 103 (Mathmetical Foundations of Computing)
CS 109 (Probability for Computer Scientists)

CS 106A (Programming Methodology)
CS 106B (Programming Abstractions)
CS 106X (Programming Abstractions (Accelerated))

Course Information

Lectures

I will provide a high-level overview of the concepts.

~~You’re welcome to download the PDFs, but Google Drive links will be most up-to-date.~~

Homework

Assignments will reinforce concepts from lecture in detail.

~~Submit on Gradescope (Entry code: 9GNVBM).~~

~~You must type or LaTeX your submission (no written homeworks accepted).~~

Course Information

Grades: 6 Homeworks and 1 Final

- 6 Homeworks (60%, 10% each)
  - You will have 3 late days (24 hours each), 2 max per homework.
  - ~~Each homework will be released on Friday and due the following Friday at 11:59 p.m.~~
  - We will try to release grades one week from the late deadline.
- 1 Final (40%)

Course Information

Syllabus

We'll cover a new topic each week; there will be a few standalone topics.

Topics include

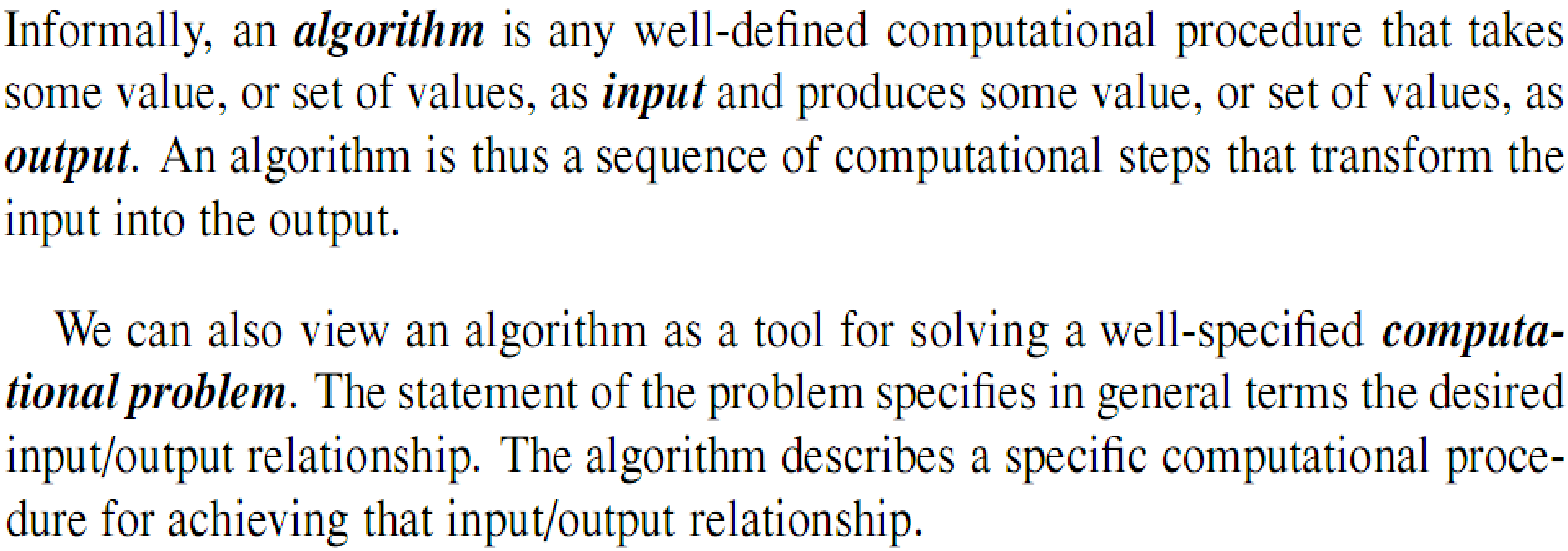
Divide and Conquer,
Randomized Algorithms,
Graph Algorithms,
Greedy Algorithms,
Dynamic Programming.

~~See the website for additional details.~~

Algorithmic Analysis

Summer 2017 • Lecture 01

Algorithms

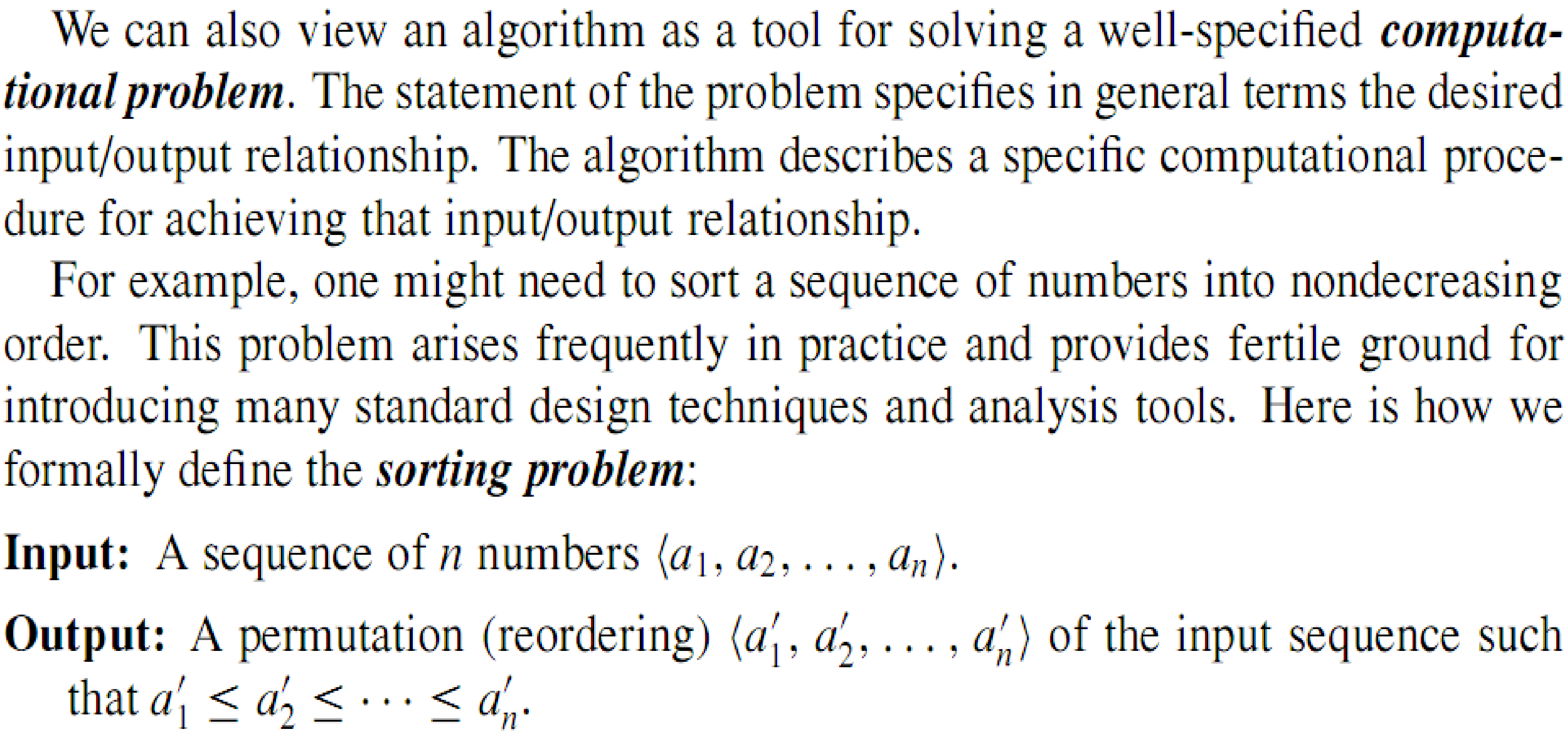


Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Introduction to Algorithms, Chapter 1 - The Role of Algorithms in Computing

Algorithms



We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

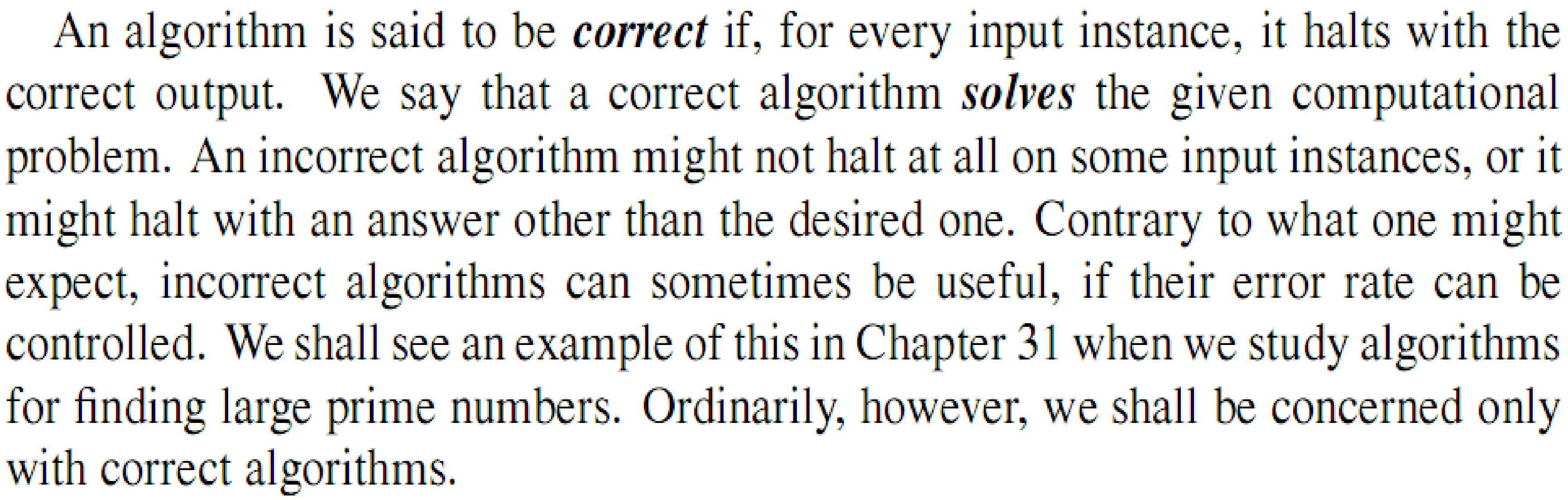
For example, one might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

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

Introduction to Algorithms, Chapter 1 - The Role of Algorithms in Computing

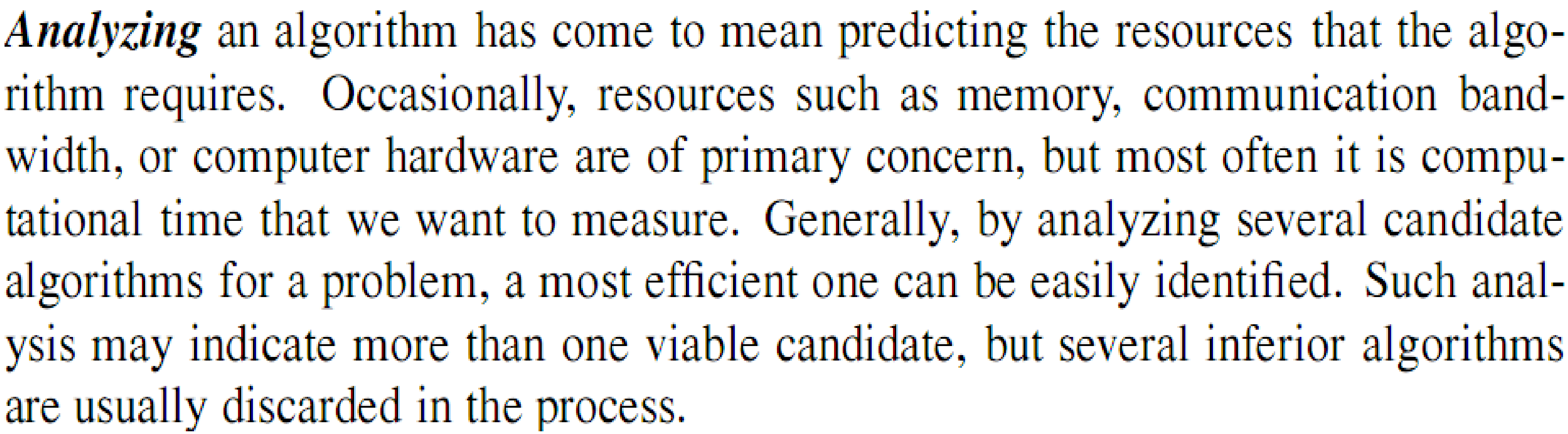
Algorithms



An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output. We say that a correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one. Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled. We shall see an example of this in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we shall be concerned only with correct algorithms.

Introduction to Algorithms, Chapter 1 - The Role of Algorithms in Computing

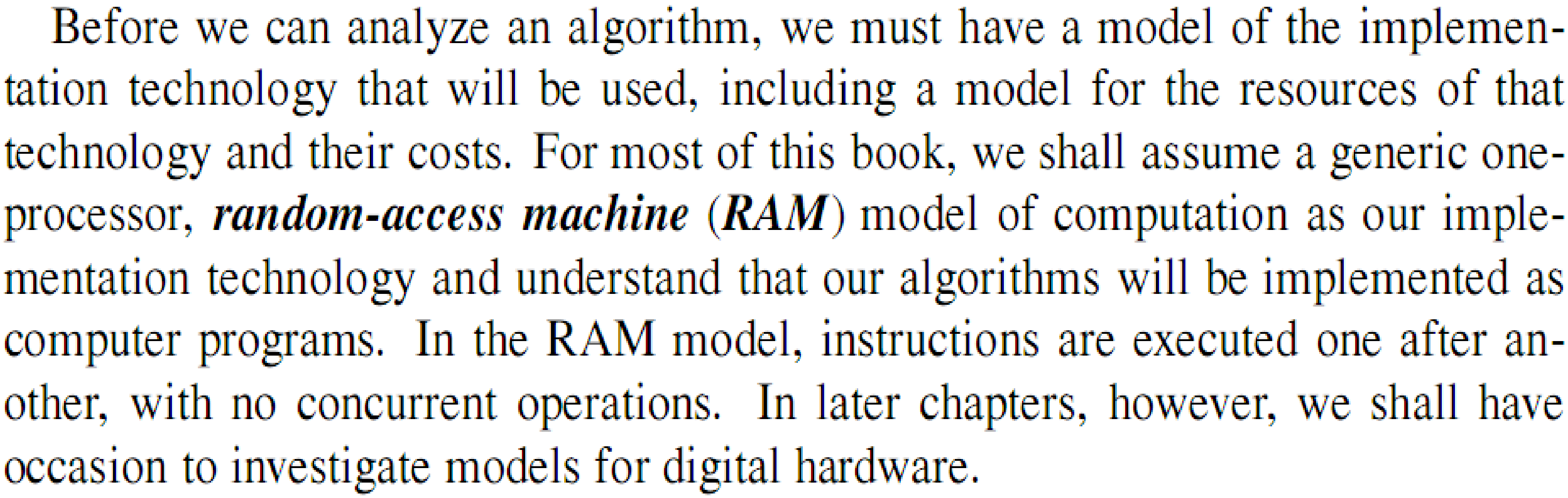
Analyzing algorithms



Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, a most efficient one can be easily identified. Such analysis may indicate more than one viable candidate, but several inferior algorithms are usually discarded in the process.

Introduction to Algorithms, *Chapter 2 - Getting Started*

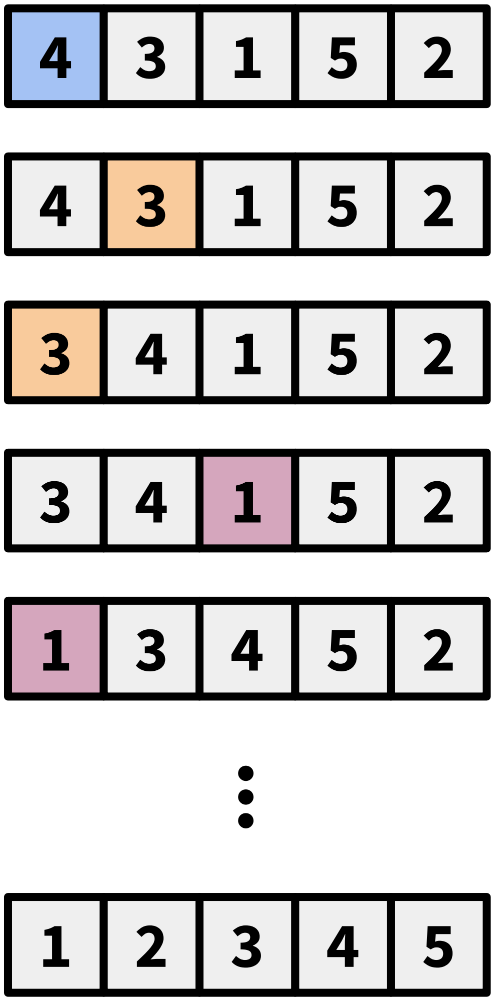
Analyzing algorithms



Before we can analyze an algorithm, we must have a model of the implementation technology that will be used, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine*** (***RAM***) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations. In later chapters, however, we shall have occasion to investigate models for digital hardware.

Introduction to Algorithms, *Chapter 2 - Getting Started*

Insertion sort



4	3	1	5	2	Let’s sort an unsorted list of numbers **`A`**. The sublist **`A[0:0]`** is trivially sorted.
4	3	1	5	2	Look at the second element, **`A[1]`**.
3	4	1	5	2	Insert the element into a new position such that the sublist **`A[0:1]`** is sorted.
3	4	1	5	2	Now look at the third element, **`A[2]`**.
1	3	4	5	2	Insert it such that the sublist **`A[0:2]`** is sorted.
⋮					
1	2	3	4	5	The entire array **`A[0:4]`** is sorted.

Insertion sort

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Insertion sort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Proving Correctness

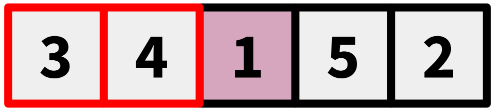
- Algorithms often initialize, modify, or delete new data.
  - In the case of insertion sort, it might be challenging for an untrained observer to formalize the notion of correctness since the manner in which the algorithm behaves depends on the input list.
  - Is there a way to prove the algorithm works, without checking it for all (infinitely many) input lists?
- To reason about the behavior of algorithms, it often helps to look for things that *don't* change.
  - Notice that insertion sort maintains a sorted sublist, the length of which grows each iteration.
- This unchanging property is called an **invariant**.

Proving Correctness

For example, an **invariant** of the outer for-loop of insertion sort: At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

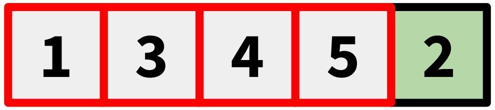
Sanity checks:

At the start of the third iteration (i.e. the iteration when i = 2), the first 2 elements of the list are sorted. True.



3	4	1	5	2

At the start of the fifth iteration (i.e. the iteration when i = 4), the first 4 elements of the list are sorted. True.



1	3	4	5	2

Proving Correctness

Less formally (explain it to your co-worker) ...

At the start of the first iteration, the first element of the array is sorted.

By construction, the ith iteration puts element **`A[i]`** in the right place.

At the start of the i = length(A)th iteration (aka the end of the algorithm), the first length(A) elements are sorted.

Yay for hand wavy-ness!

Proving Correctness

More formally (rigorously) …

Outer invariant (for-loop): At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Inner invariant (while-loop): At the start of iteration j of the inner while-loop, **`A[0:j,j+2:i]`** contains the same elements as the original sublist **`A[0:i-1]`**, still sorted, such that all of the values in the right sublist **`A[j+2:i]`** are greater than **`cur_value`**.

Proving Correctness

More formally (rigorously) ...

Lemma: If **`A[0:i]`** is sorted at the start of iteration i = x-1 of the loop, then **`A[0:i]`** will be sorted at the start of iteration i = x of the loop.

Proof:

To prove this statement, we first prove the inner loop invariant.

The invariant holds at the start of the iteration j = i-1 of the inner while-loop. To see why, notice that **`A[0:j,j+2:i]`** describes the same sublist as **`A[0:i-1,i+1:i]`**(since we initialized j to i-1), which trivially contains the same elements as the original sublist **`A[0:i-1]`**, still sorted, since the right sublist **`A[i+1:i]`** is empty.

Furthermore, since the right sublist is empty, all of its values are all vacuously greater than **`cur_value`**.

Proving Correctness

Proof of lemma, cont.:

Now, we will prove the inductive step. Suppose that the invariant holds at the start of an arbitrary iteration j = y (inductive hypothesis). We prove that it still holds at the start of iteration j = y-1. There are two cases of the while-loop condition to consider:

The condition returns True. First, **`A[j]`** is copied to **`A[j+1]`**. Since **`A[j]`** = **`A[j+1]`** and **`A[0:j,j+2:i]`** satisfies the invariant for j = y (by the inductive hypothesis), now **`A[0:j-1,j+1:i]`** also satisfies the invariant for j = y. Then, j is decremented by 1 to y-1, so **`A[0:j,j+2:i]`** now satisfies the invariant for j = y-1, maintaining the invariant for the next iteration.

The condition returns False. The loop terminates. Since either (1) j is -1 or (2) **`cur_value`** is greater than **`A[j]`**, then **`A[0:j],cur_value`** must be sorted (recall the invariant guarantees that **`A[0:j]`** is sorted).

Furthermore,

since all of the values in the right sublist **`A[j+2:i]`** are sorted and greater than **`cur_value`**, then **`A[0:j],cur_value,A[j+2:i]`** must be sorted. Thus, at the termination of the loop, **`A[0:i]`** (the first i+1 elements) is sorted. □

Proving Correctness

Theorem: Insertion sort sorts the input list.

Proof:

At the start of the first iteration of the outer for-loop, **`A[0:-1]`** (an empty sublist) is trivially sorted.

By our lemma, if **`A[0:x-1]`** is sorted at the start of iteration i = x of the loop, then **`A[0:x]`** will be sorted at the start of iteration i = x+1 of the loop.

The loop terminates at the start of iteration **`length(A)`**, which implies that **`A[0:length(A)-1]`** is sorted when the loop ends, which proves the theorem. □

Proving Correctness

Both the lemma and theorem follow a consistent format:

Initialization: The loop invariant starts out as true.

Maintenance: If the loop invariant is true at step i, then it’s true at step i+1.

Termination: If the loop invariant is true at the end of the algorithm, this tells you something about what you’re trying to prove.

Insertion sort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Analyzing Runtime

```
algorithm insertion_sort(list A):
  for i = 1 to length(A):
    let cur_value = A[i]
    let j = i - 1
    while j > 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

O(n) work per iteration

O(n) iterations

Total work: $O(n^2)$

Big-O Notation

Big-O notation is a mathematical notation for upper-bounding a function's rate of growth.

- Informally, it can be determined by ignoring constants and non-dominant growth terms.

Examples

- $n + 137 = O(n)$
- $3n + 42 = O(n)$
- $n^2 + 3n - 2 = O(n^2)$
- $n^3 + 10n^2 \log n - 15n = O(n^3)$
- $2^n + n^2 = O(2^n)$

A Brainteaser

- Can you think of non-decreasing functions f and g such that neither $f = O(g)$ nor $g = O(f)$?

Big-O Notation

Formally speaking, let f, g: N → N.

Then f(n) = O(g(n)) iff

$\exists n_0 \in N, c \in R.$

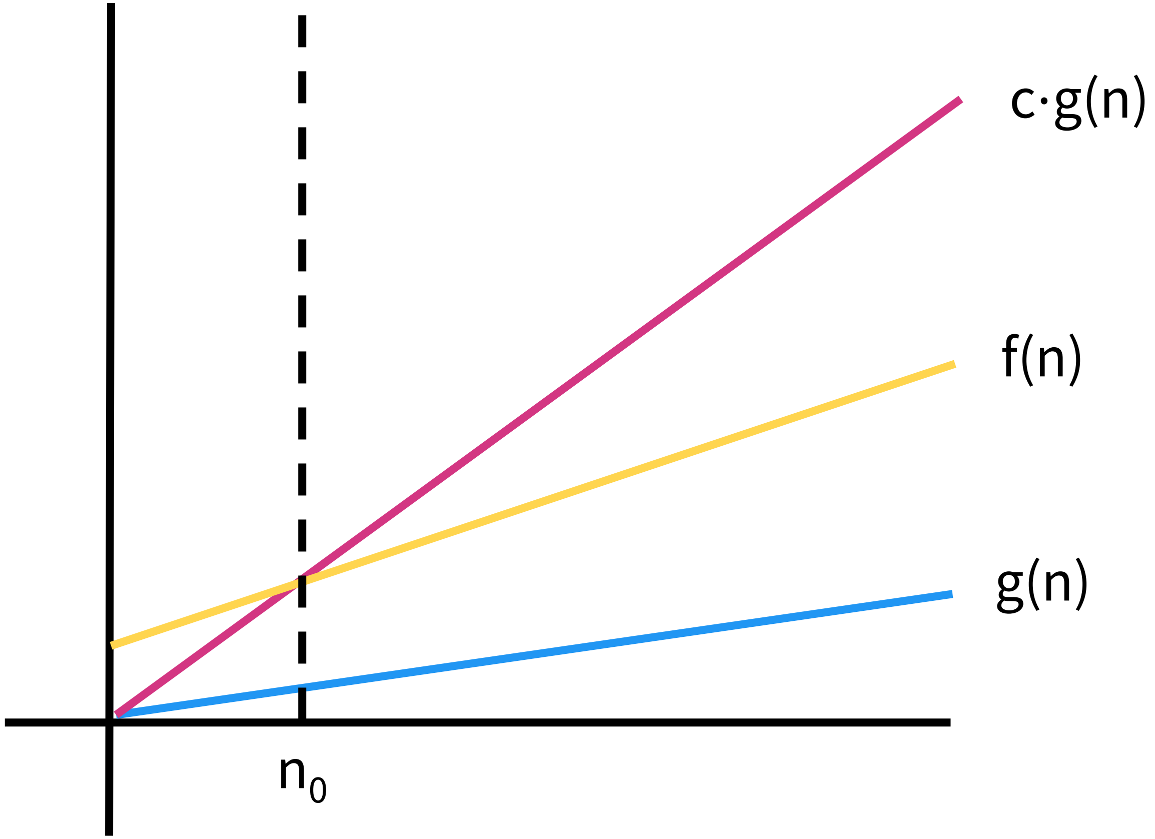
$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$

Intuitively, this means that f(n) is upper-bounded by g(n) aka f(n) is “at most as big as” g(n).

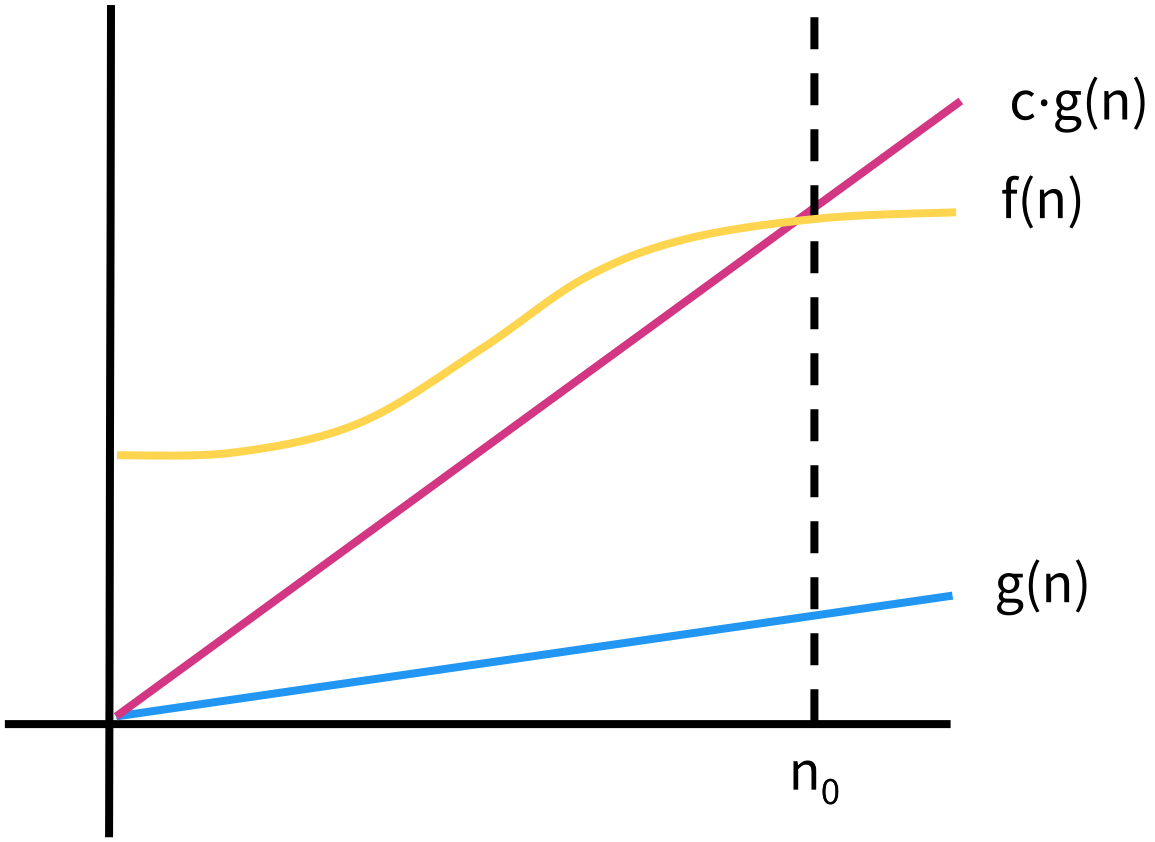
Big-O Notation

$f(n) = O(g(n))$ iff $\exists n_0 \in N, c \in R. \ \forall n \in N. \ (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



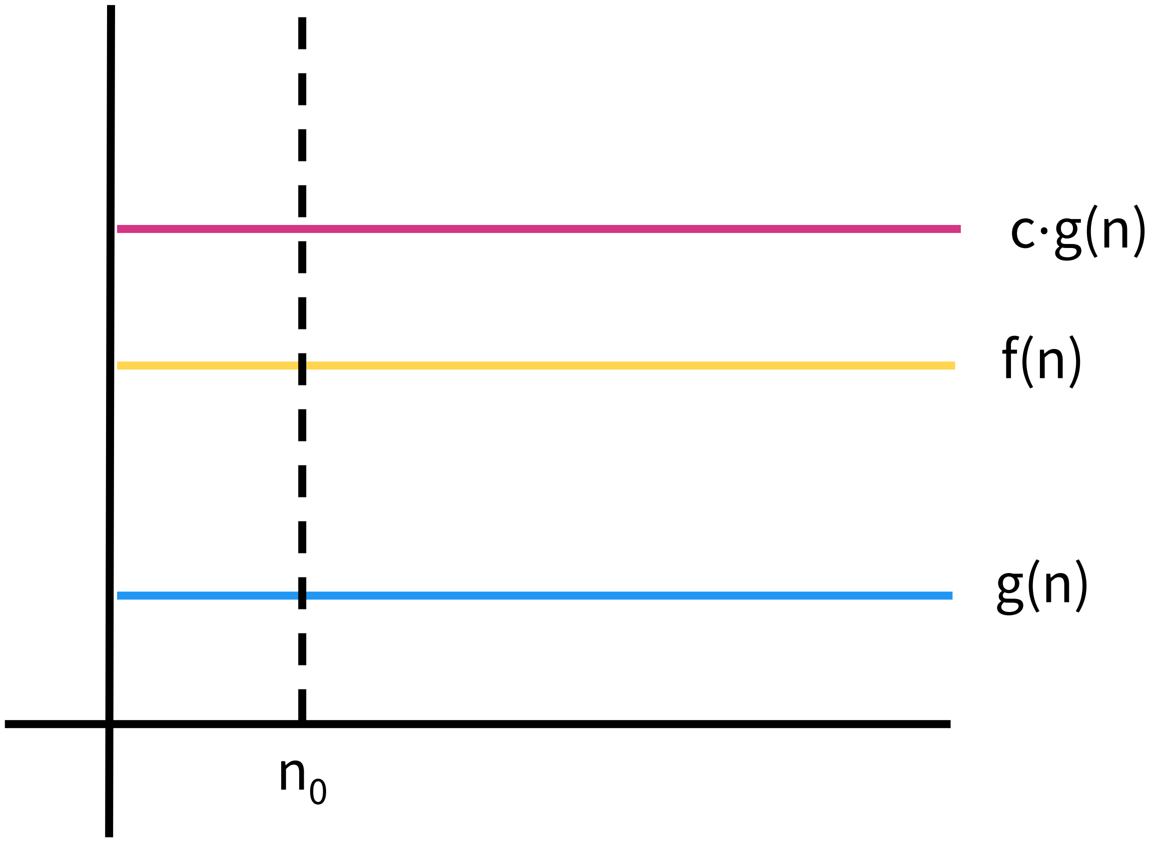
Big-O Notation

$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



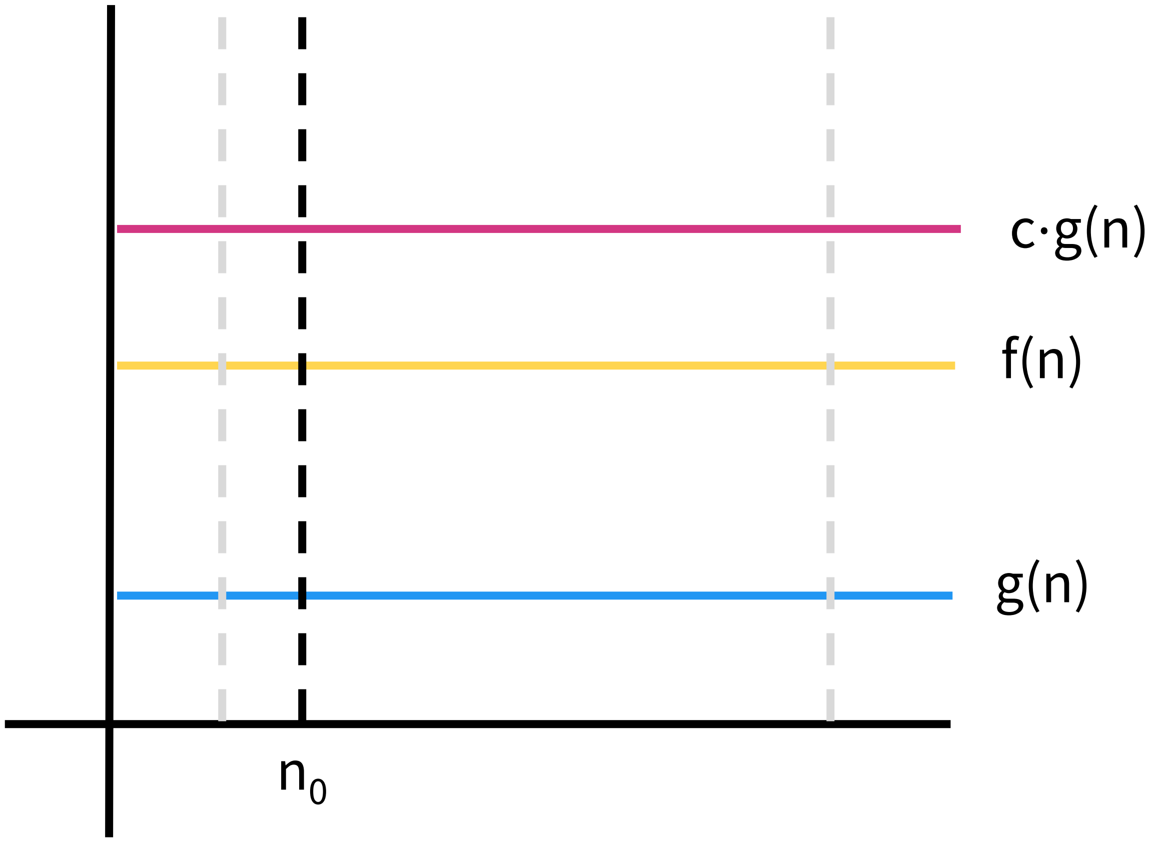
Big-O Notation

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$



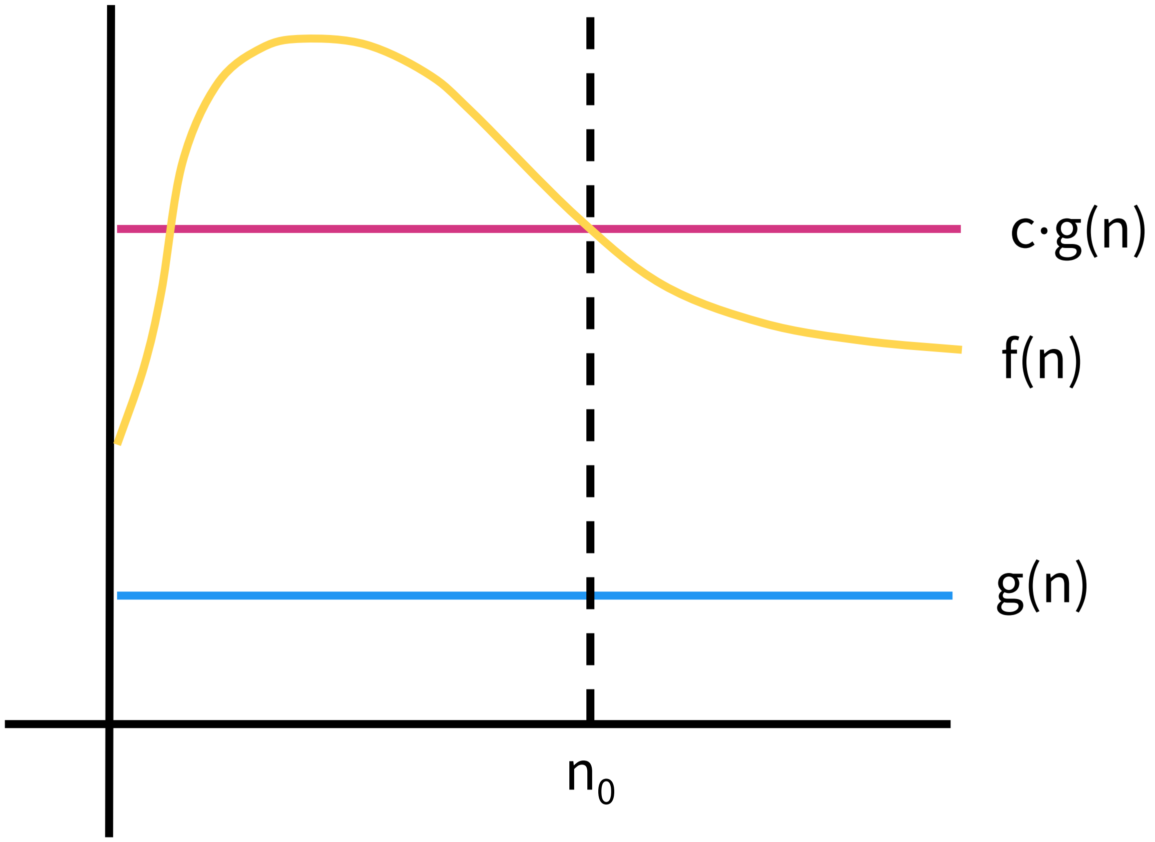
Big-O Notation

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in \mathbb{N}, c \in \mathbb{R}. \quad \forall n \in \mathbb{N}. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$



Big-O Notation

$f(n) = O(g(n))$ iff $\exists n_0 \in N, c \in R.\ \forall n \in N.\ (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



Big-O Notation

To prove $f(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

- Suppose $f(n) = n$ and $g(n) = n \log n$. We prove that $f(n) = O(g(n))$.
- Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq cn \log n$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

To prove $f(n) \neq O(g(n))$, proceed by contradiction.

- Suppose $f(n) = n^2$ and $g(n) = n$. We prove that $f(n) \neq O(g(n))$.
- Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq cn$. Consider $n = \max\{c, n_0\} + 1$. Then $n \geq n_0$, but we have $n > c$, which implies that $n^2 > cn$. Contradiction!

Big-Ω Notation

Let f, g: N → N.

Then f(n) = Ω(g(n)) iff

$\exists n_0 \in N, c \in R.$

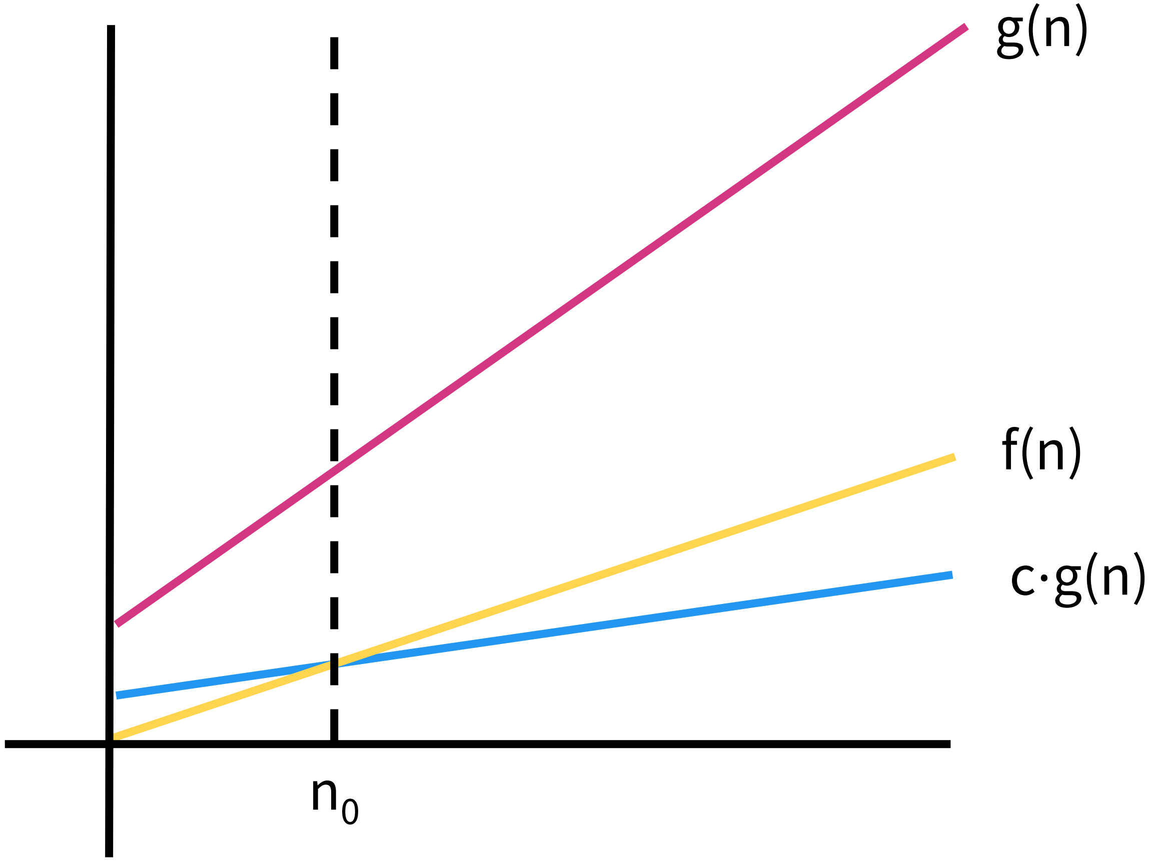
$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$

Intuitively, this means that f(n) is lower-bounded by g(n) aka f(n) is “at least as big as” g(n).

Big-Ω Notation

$f(n) = \Omega(g(n))$ iff $\exists n_0 \in N, c \in R.\ \ \forall n \in N.\ \ (n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$



Big-Θ Notation

f(n) = Θ(g(n)) iff both f(n) = O(g(n)) and f(n) = Ω(g(n)).

More verbosely, let f, g: N → N.

Then f(n) = Θ(g(n)) iff

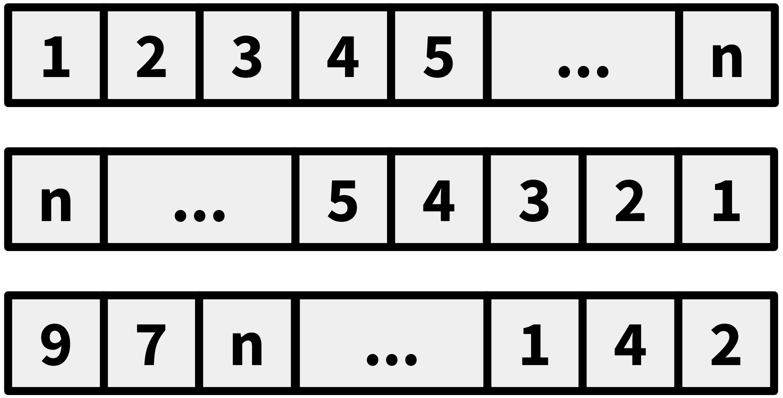
$\exists n_0 \in N$, c_1 and $c_2 \in R$.

$\forall n \in N$.

$(n \geq n_0 \rightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$

Intuitively, this means that f(n) is lower and upper-bounded by g(n) aka f(n) is “the same as” g(n).

Best case vs. Worst case



1 2 3 4 5 ... n

n ... 5 4 3 2 1

9 7 n ... 1 4 2

Total work: $O(n)$ or $O(n^2)$ or $\Omega(n)$ or $\Omega(n^2)$?

Best case vs. Worst case

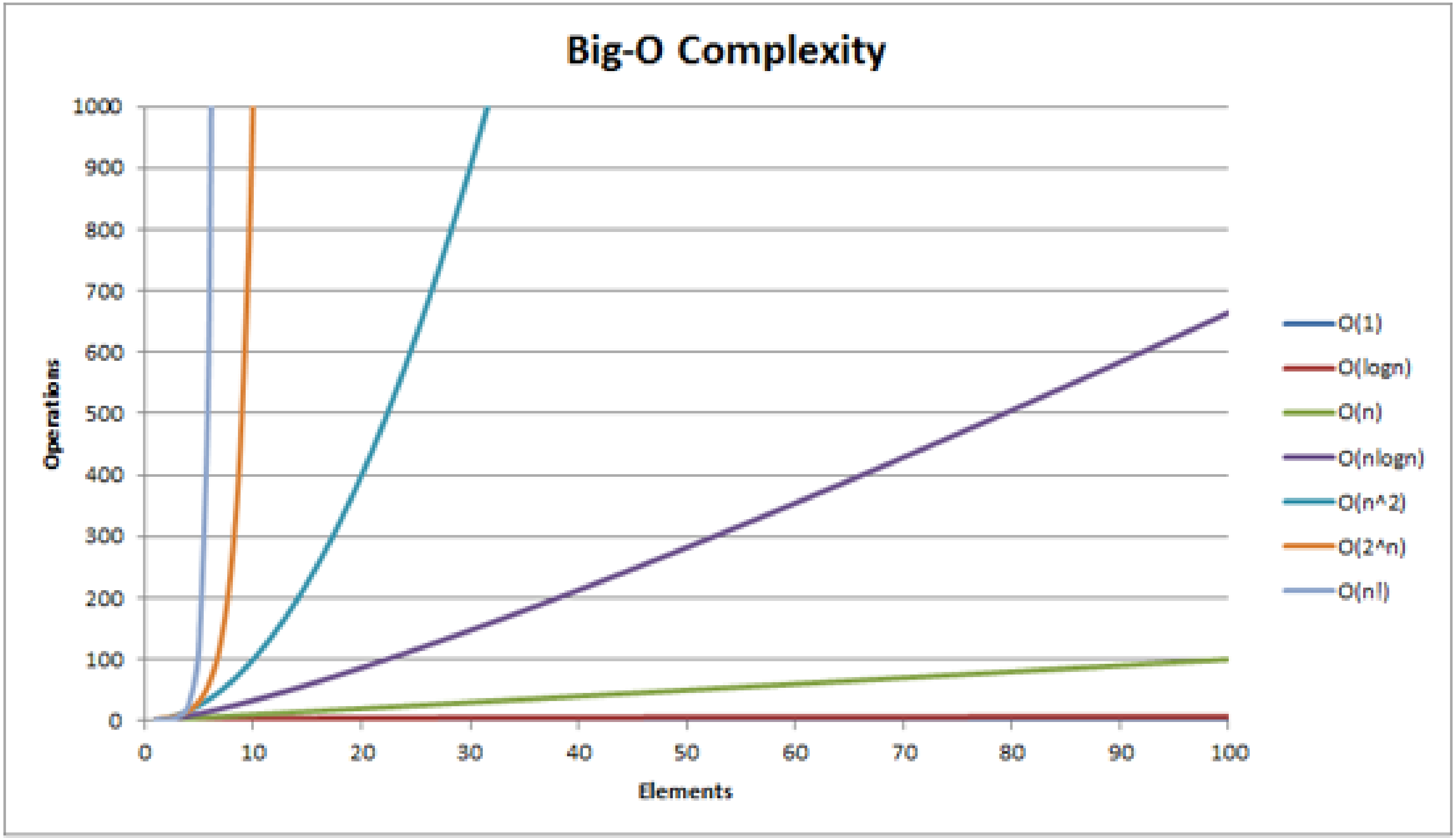
The worst-case runtime of insertion sort is $\Theta(n^2)$.

The best-case runtime of insertion sort is $\Theta(n)$.

Usually, we care more about the worst-case time.

It’s acceptable, albeit not entirely precise (Mặc dù không hoàn toàn chính xác - google translate), to say the runtime of insertion sort is $\Theta(n^2)$.

Big-O Complexity



Hard problem

Hard problem (P, NP class)

- Một bài toán được gọi là có độ phức tạp đa thức, hay còn gọi là có thời gian đa thức, nếu số các phép tính cần thiết khi thực hiện thuật toán không vượt quá O(n^k), với k nguyên dương nào đó, còn n là kích thước của dữ liệu đầu vào.

- Các thuật toán với O(b^n), trong đó n là kích thước dữ liệu đầu vào, còn b là một số nguyên dương nào đó gọi là các thuật toán có độ phức tạp hàm mũ hoặc thời gian mũ.

Hard problem (P, NP class)

- P (Polynomial time) là lớp các bài toán giải được với thời gian đa thức.

- NP (Nondeterministic Polynomial time) là lớp các bài toán mà lời giải của nó có thể kiểm tra với thời gian đa thức.

P = NP ?

- Bài toán **P so với NP** là một bài toán mở quan trọng trong lĩnh vực khoa học máy tính.

- Nó đặt ra vấn đề là: bất kì bài toán nào có lời giải có thể được kiểm chứng với thời gian đa thức (lớp NP) cũng có thể được giải quyết với thời gian đa thức (lớp P) hay không? hay

P = NP ???

P = NP ?

- Được Stephen Cook đưa ra năm 1971 trong bài báo nổi tiếng "The complexity of theorem proving procedures”

- Là một trong số bảy bài toán của giải thiên niên kỷ được chọn bởi Viện Toán học Clay.

- Mỗi bài trong số bảy bài này có giải thưởng

  US$1,000,000

  cho lời giải đúng đầu tiên.

An example of NP problem

- Cho một tập hợp các số nguyên, tìm một tập hợp con khác rỗng có tổng bằng 0. Ví dụ, có tập hợp con nào của {−2, −3, 15, 14, 7, −10} có tổng bằng 0?

- Lời giải "có, vì {−2, −3, −10, 15} có tổng bằng 0" có thể được kiểm chứng dễ dàng bằng cách cộng các số đó lại.

- Tuy nhiên, hiện chưa có thuật toán nào để tìm ra một tập hợp như thế trong thời gian đa thức..

An example of NP problem

- Cho một tập hợp các số nguyên, tìm một tập hợp con khác rỗng có tổng bằng 0. Ví dụ, có tập hợp con nào của {−2, −3, 15, 14, 7, −10} có tổng bằng 0?

- Có một thuật toán đơn giản thực thi trong thời gian hàm mũ là kiểm tra tất cả 2^n-1 tập hợp con khác rỗng.

- Bài toán này nằm trong NP (kiểm chứng nhanh chóng) nhưng chưa biết có nằm trong P (giải nhanh chóng) hay không

NP-Complete

- Bài toán A $\in$ NP được gọi là NP-đầy đủ (NP-Complete) nếu thỏa mãn tính chất: A giải được với thời gian đa thức thì mọi bài toán khác trong NP giải được bằng thời gian đa thức.
- Khi đó ta có **P = NP**

Some NP-Complete problems

- Boolean satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem (decision version)
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

https://en.wikipedia.org/wiki/NP-completeness

5 min break

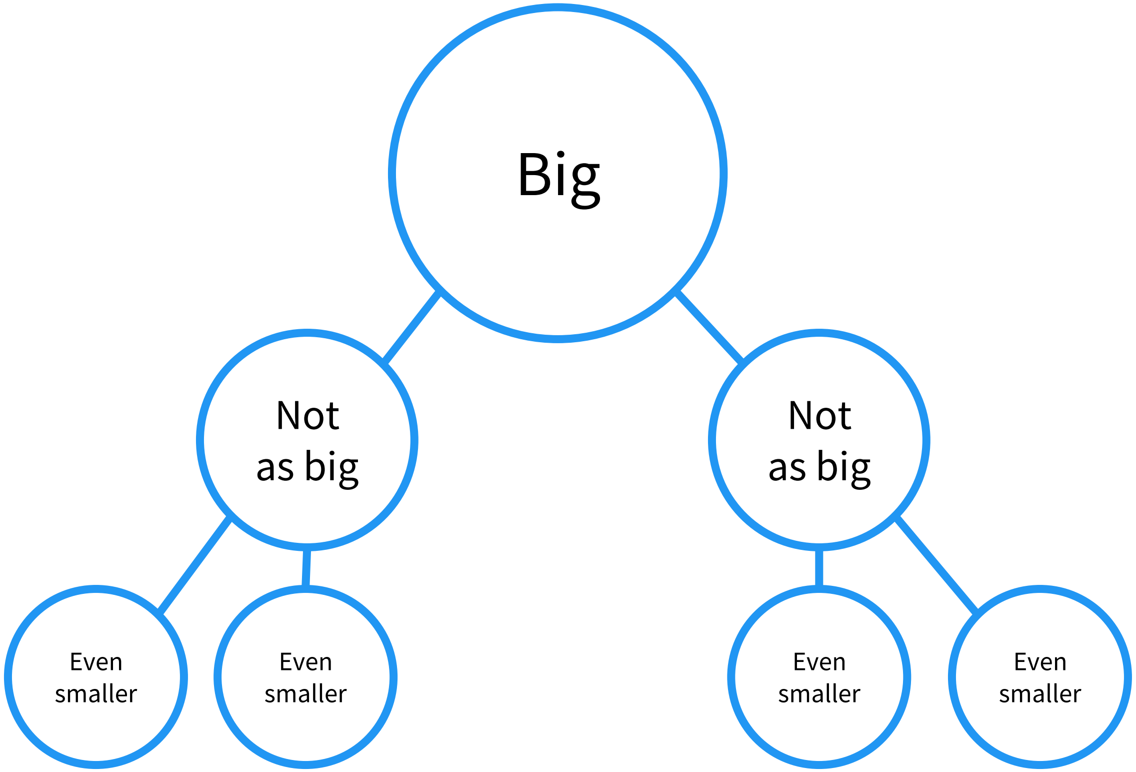
Divide and Conquer I

Summer 2017 • Lecture 1

Divide and Conquer

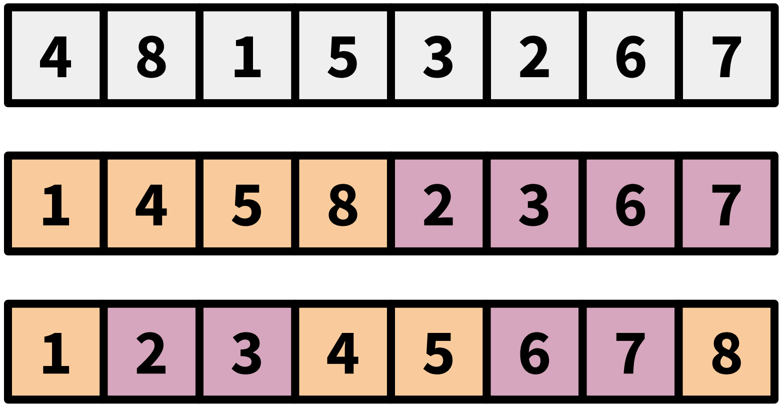
Divide: break current problem into smaller problems.

Conquer: solve the smaller problems and collate the results to solve the current problem.



Mergesort

Let’s use divide and conquer to improve upon insertion sort!



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

Let’s sort an unsorted list of numbers **`A`**.

Recursively sort each half, **`A[0:3]`** and **`A[4:7]`**, separately.

Merge the results from each half together.

Mergesort

```
algorithm mergesort(list A):
  if length(A) ≤ 1:
    return A
  let left = first half of A
  let right = second half of A
  return merge(
    mergesort(left),
    mergesort(right)
  )
```

Total work: O(???)

Mergesort

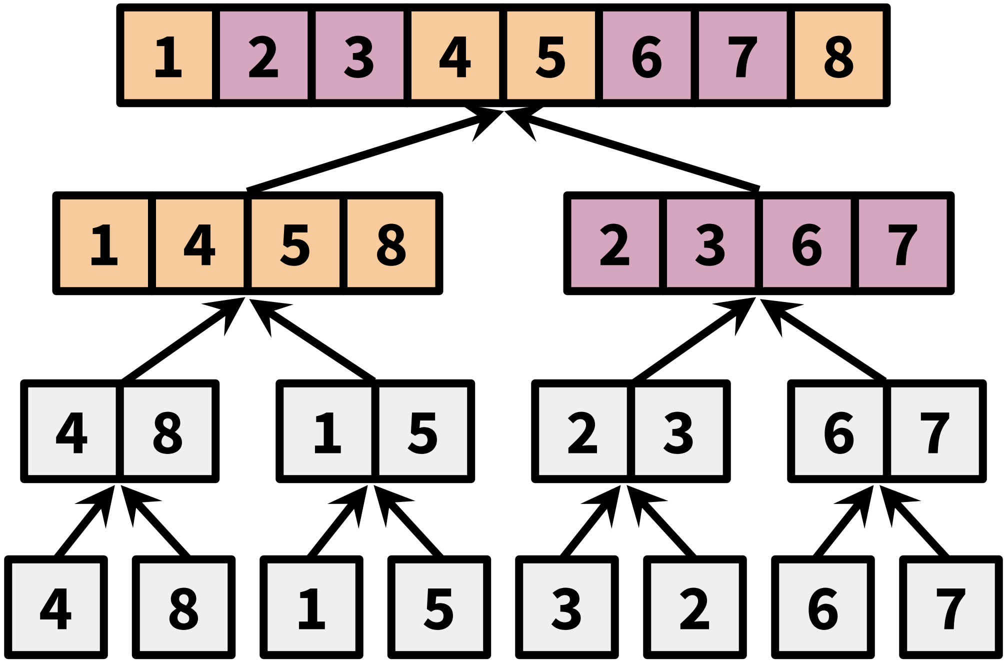
```
algorithm merge(list A, list B):
  let result = []
  while both A and B are nonempty:
    if head(A) < head(B):
      append head(A) to result
      pop head(A) from A
    else:
      append head(B) to result
      pop head(B) from B
  append remaining elements in A to result
  append remaining elements in B to result
  return result
```

Total work: $\Theta(a+b)$, where a and b are the lengths of lists A and B.

Mergesort

Tracing the recursive calls . . .

Sorted list



Original list

Mergesort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Proving Correctness

Less formally (explain it to your co-worker) ...

Consider a list of length k. If n is `0` or `1`, `mergesort` correctly sorts the list since an empty or single-element list is already sorted (base case).

Now suppose `mergesort` correctly sorts lists of length `1` to k-`1`. Since `left` and `right` must have lengths `1` to k-`1`, `mergesort` correctly sorts these lists. By construction, `merge` joins the elements from the two sorted lists into a single sorted list of length k, which it returns. Thus, `mergesort` returns the elements of the original list, but in sorted order (inductive case).

In the top recursive call, `mergesort` sorts the original array of length n (conclusion).

Analyzing Runtime

Let T(n) represent the runtime of `mergesort` on a list of length n.

- T(n/2) is the runtime of `mergesort` on a list of length n/2. T(6881441) is the runtime of `mergesort` on a list of length 6,881,441. T(⌈n/17⌉) is the runtime of `mergesort` on a list of length ⌈n/17⌉.
- Recall that `mergesort` on a list of length n calls `mergesort` once for `left` and once for `right`, which costs **T(⌈n/2⌉) + T(⌊n/2⌋)**.
- After that, it calls `merge` on the two sublists, which costs **Θ(n)**.

Here's our first **recurrence relation**,

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

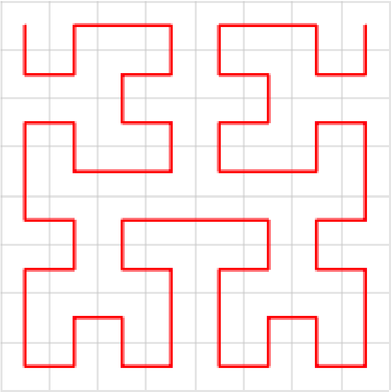
Analyzing Runtime

A **recurrence relation** is a function or sequence whose values are defined in terms of earlier values.

Here, we've written a recurrence relation for the runtime of mergesort. But we could have just as easily written one to describe something else recursive.

e.g. The Fibonacci sequence can be defined by its recurrence relation $T(n) = T(n-1) + T(n-2)$, where $T(n)$ represents the nth element of the sequence.

e.g. The length of the Hilbert curve fractal can be written as its recurrence Relation $H(n) = 3\cdot(1/(2^n-1)) + 4\cdot(2^{n-1}-1)H(n) / (2^n-1)$, where $H(n)$ is the length of the curve.



Analyzing Runtime

How do we solve our recurrence relation?

Assumption 1: First, it's helpful to assume that n is a power of two.

~~$T(0) = \Theta(1)$~~

$$T(1) = \Theta(1) = c_1$$

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2T(n/2) + c_2 n \end{aligned}$$

Assumption 2: Let $c = \max\{c_1, c_2\}$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Analyzing Runtime

How do we solve our new recurrence relation?

$$T(1) \leq c$$

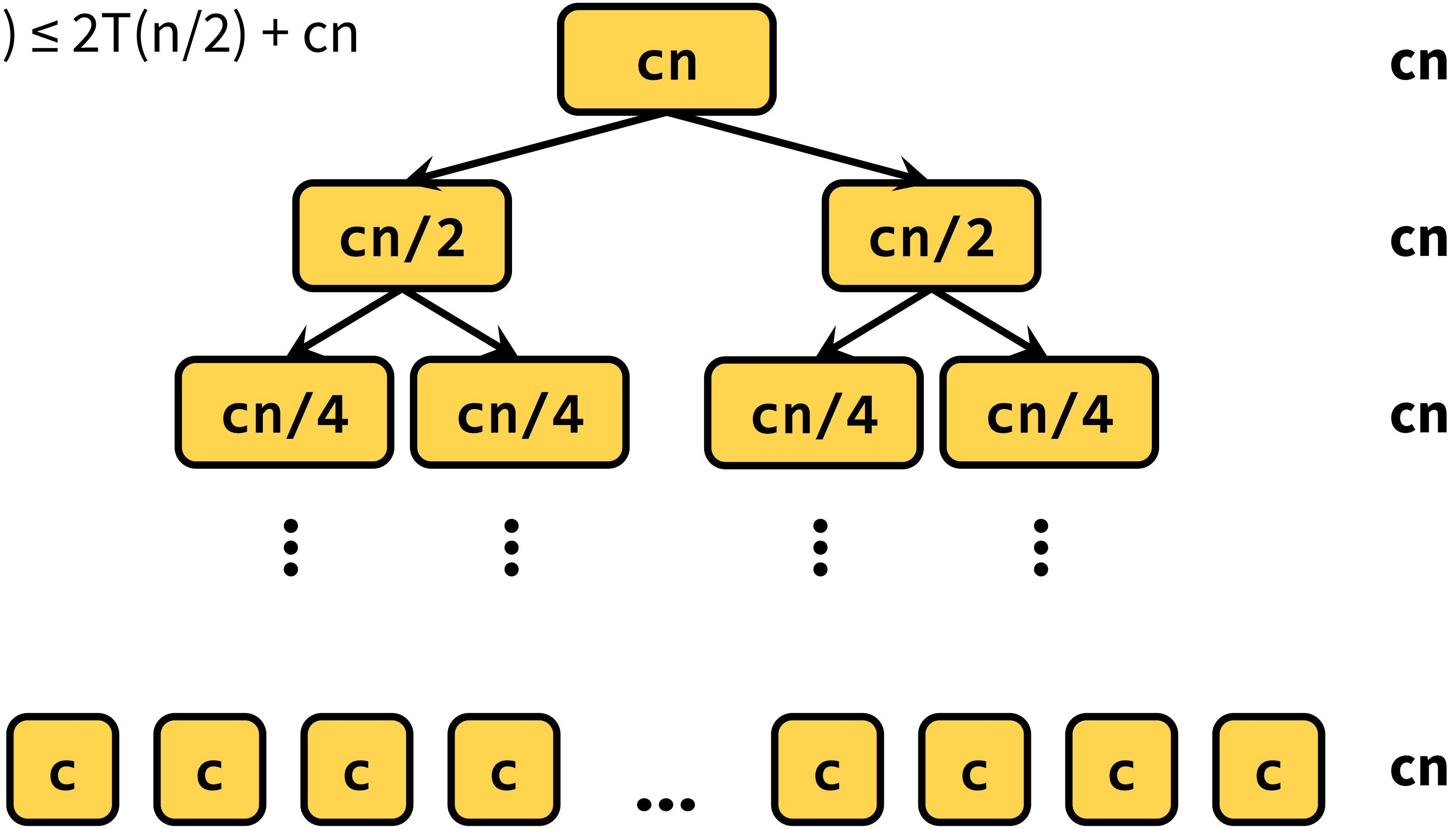
$$T(n) \leq 2T(n/2) + cn$$

We'll use the **recursion tree method**.

Recursion Tree Method

$T(1) \leq c$

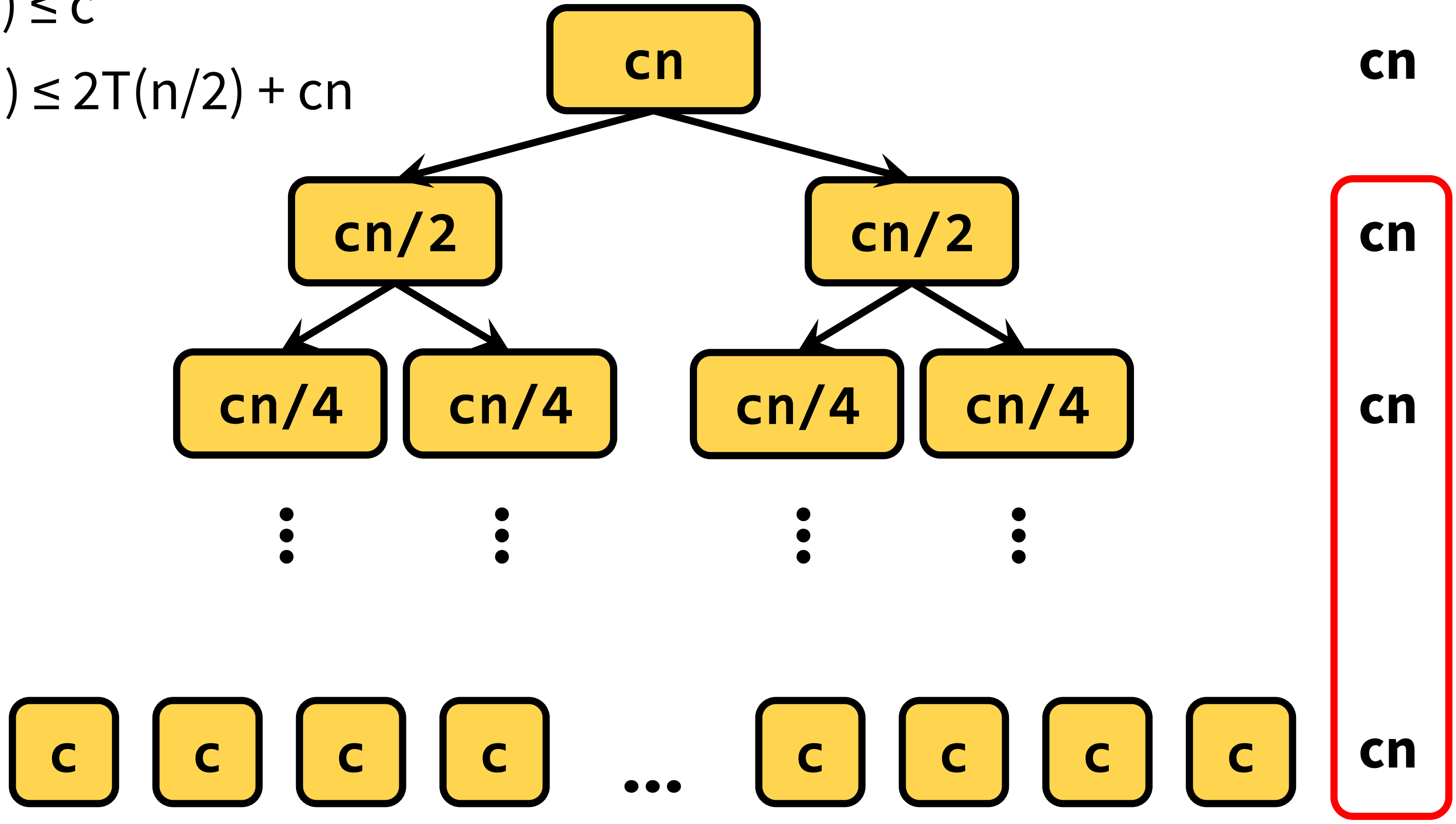
$T(n) \leq 2T(n/2) + cn$



Recursion Tree Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$



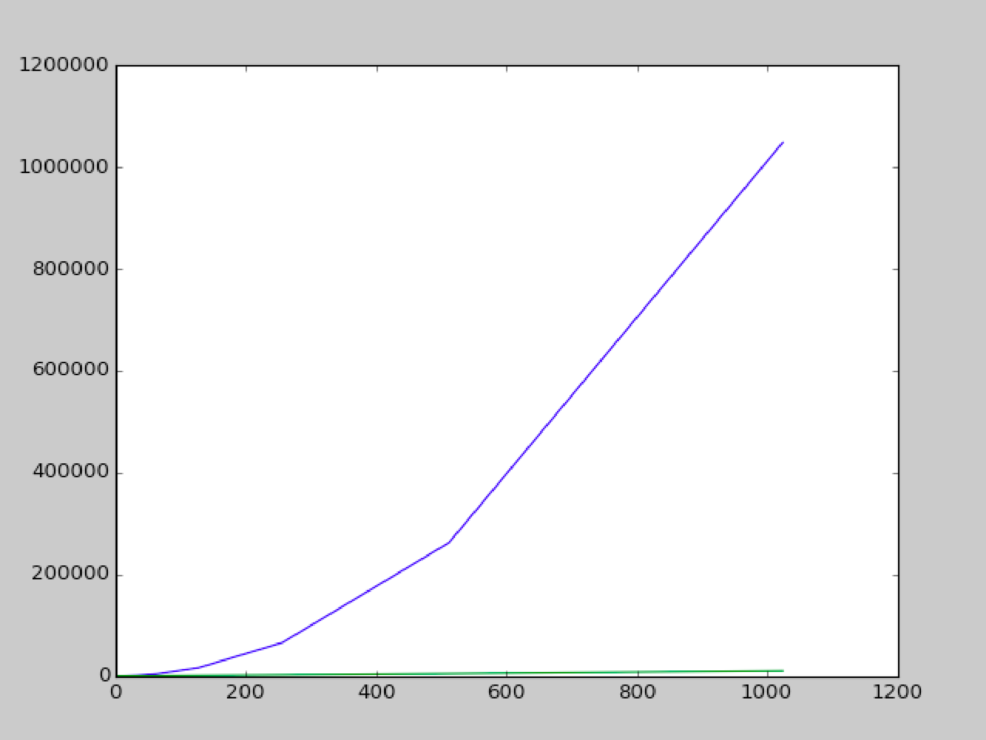
Total work: $cn \log_2 n + cn$

Analyzing Runtime

The best and worst-case runtime of `mergesort` is Θ(n log n).

The worst-case runtime of `insertion_sort` was Θ(n^2).

THIS IS A HUGE IMPROVEMENT!!


1200000
1000000
800000
600000
400000
200000
0
0
200
400
600
800
1000
1200