Design and Analysis of Algorithms

CS3230

Week 2

Basics of Algorithm Analysis

Arnab Bhattacharyya

Rahul Jain

Tentative Schedule of Lectures

Date	Topic
15/08	Stable matching
22/08	Basics of Algorithm Analysis
29/08	Graphs
05/09	Greedy algorithms
12/09	Divide & Conquer
19/09	Dynamic programming
03/10	Max flow
10/10	Midterm
17/10	Intractability
24/10	NP-completeness
31/10	Approximation
07/11	Local search
14/11	Randomized algorithms

Online Portal

- Canvas (https://canvas.nus.edu.sg/courses/45767)
  - Home
  - Announcements
  - Assignments
  - Discussions
  - Syllabus
  - Grades
  - Files
  - Visit Canvas frequently

Assessment

- Midterm Assessment: 20% (**10th October 2023, 10 am-12 pm**)
- Endterm Assessment: 40% (**27th November 2023, 9-11:30 am**)
- Weekly problem sets: 20%
- Tutorial participation: 10%
- Programming assignments: 10%

Problem Sets

- There will be 12 written assignments, released 1 week after the corresponding lecture.
  - Problem set released by 6 pm on Monday, due by 6 pm on the the following Monday
  - Exception for Problem Set 6; see schedule on Canvas.

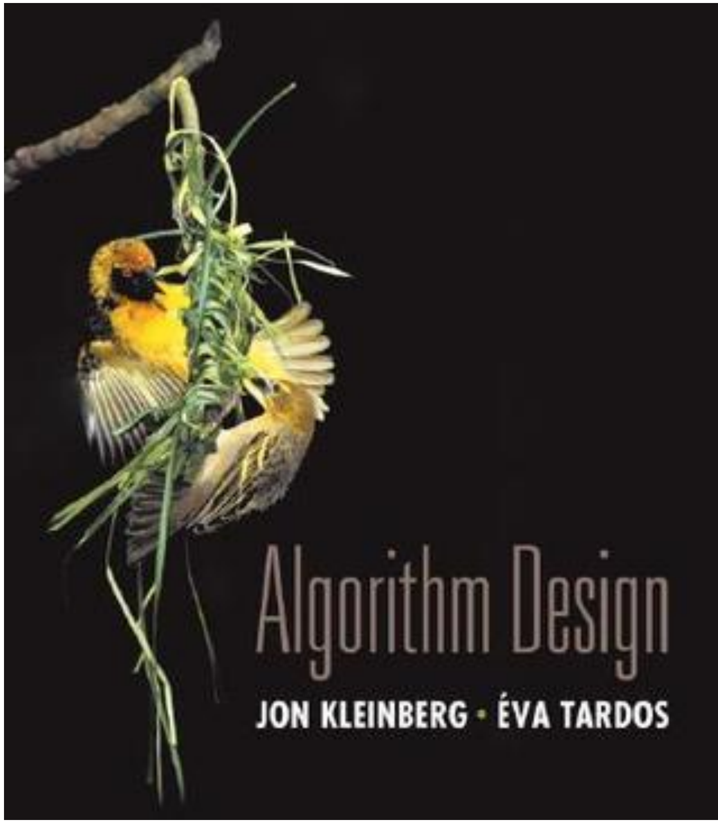
- Submit on Canvas

- Graded by your TA.

References

- Slides and Lecture Notes

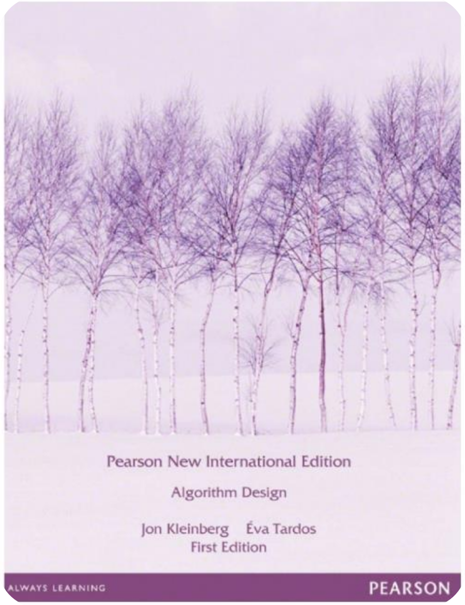
- Textbook: **Algorithm Design**,
  by Kleinberg & Tardos



- Read online: https://ebookcentral.proquest.com/lib/nus/detail.action?docID=5173481

Purchase your module's chosen Pearson textbook

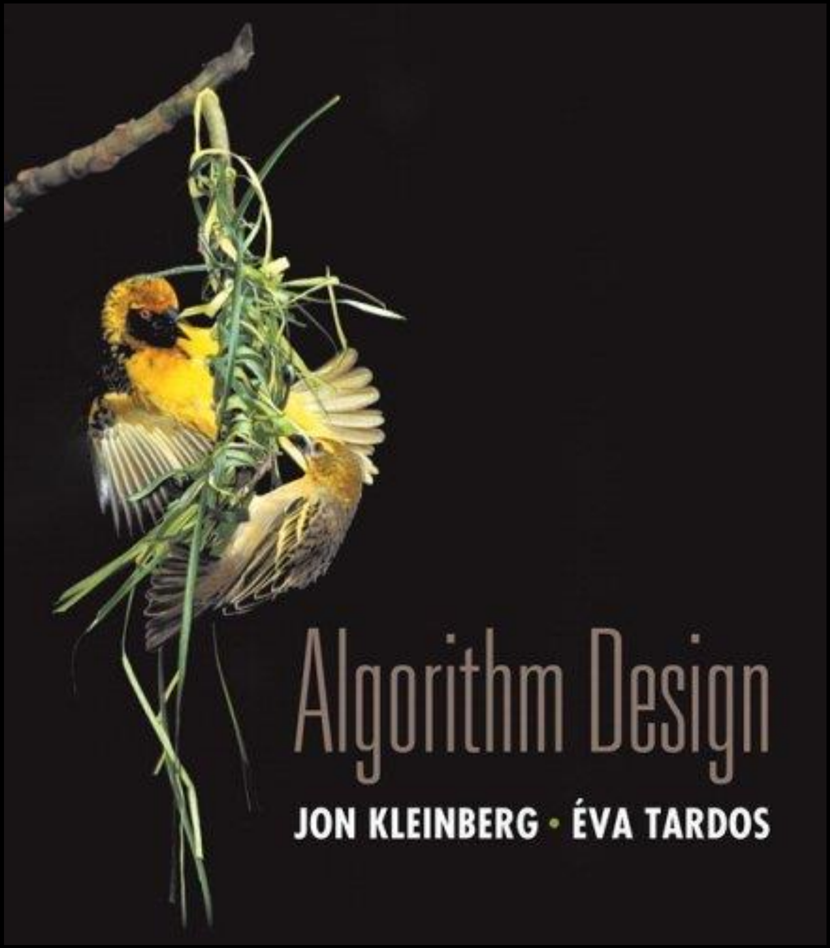
Module	CS3230 Design and Analysis of Algorithms
Title	Algorithm Design: Pearson New International Edition 1st Edition
eBook	https://shopee.sg/product/849371650/22352080786



Available at Shopee

Academic Policy (on Plagiarism)

- Do your work YOURSELF
- If you are REALLY stuck,
  - Approach instructor/tutor for help
- If you want to discuss with fellow students
  - Discuss general approach (not detailed answers)
  - You MUST write up YOUR OWN answers
  - You MUST write down names of collaborators
- Do NOT copy/compare answers!
  - If you do so, you will get direct F (due to our school policy).
  - My personal opinion does not matter.
- Please do not post assignment questions and put your code in public repositories
  - For example, should NOT post anything on stackoverflow,….
  - You can ask question on Canvas
- Resource: https://www.comp.nus.edu.sg/cug/plagiarism/



Chapter 2

Basics of Algorithm Analysis

Slides by Kevin Wayne.

Introduction

What is an algorithm?

A finite sequence of “well-defined” instructions to solve a given computational problem

A prime objective of the course: Design of **efficient** algorithms

- Focus on running time

How to analyze running time?

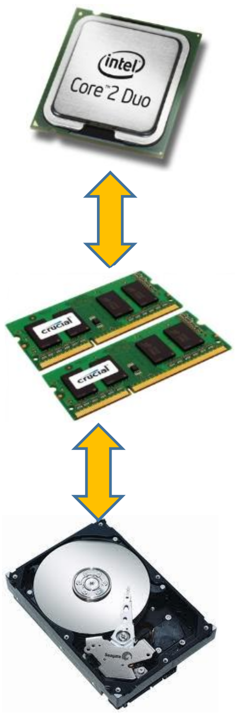
- **Simulation:** Run the algorithm many times and measure the running time

Machine dependent

Input dependent

- **Mathematical analysis**

Let us open a desktop/laptop



A processor (CPU)

speed = few GHz
(a few **nanoseconds** to execute an instruction)

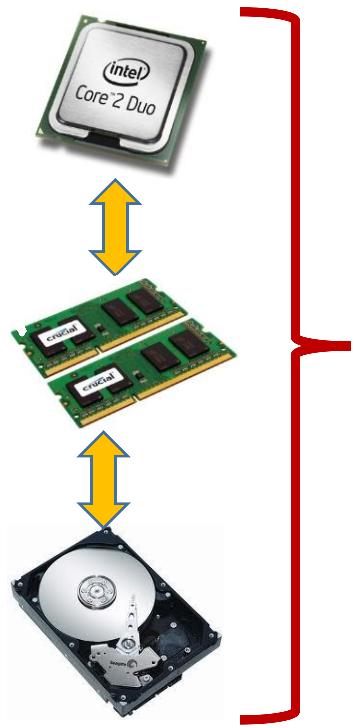
Internal memory (RAM)

size = a few GB (Stores a billion bytes/words)
speed = a few GHz(a few **nanoseconds** to read a byte/word)

External Memory (Hard Disk Drive)

size = a few tera bytes
speed : seek time = **miliseconds**
transfer rate= around **billion** bits per second

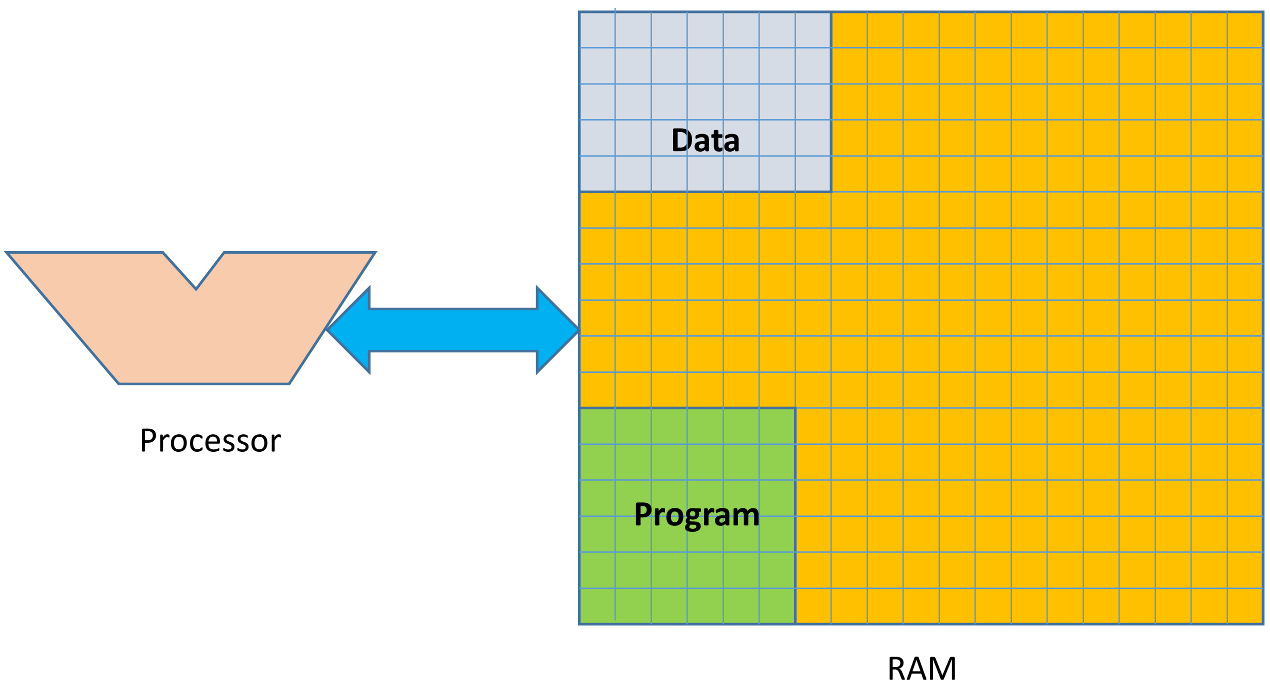
Model of Computation



(Mathematical) Model of Computation

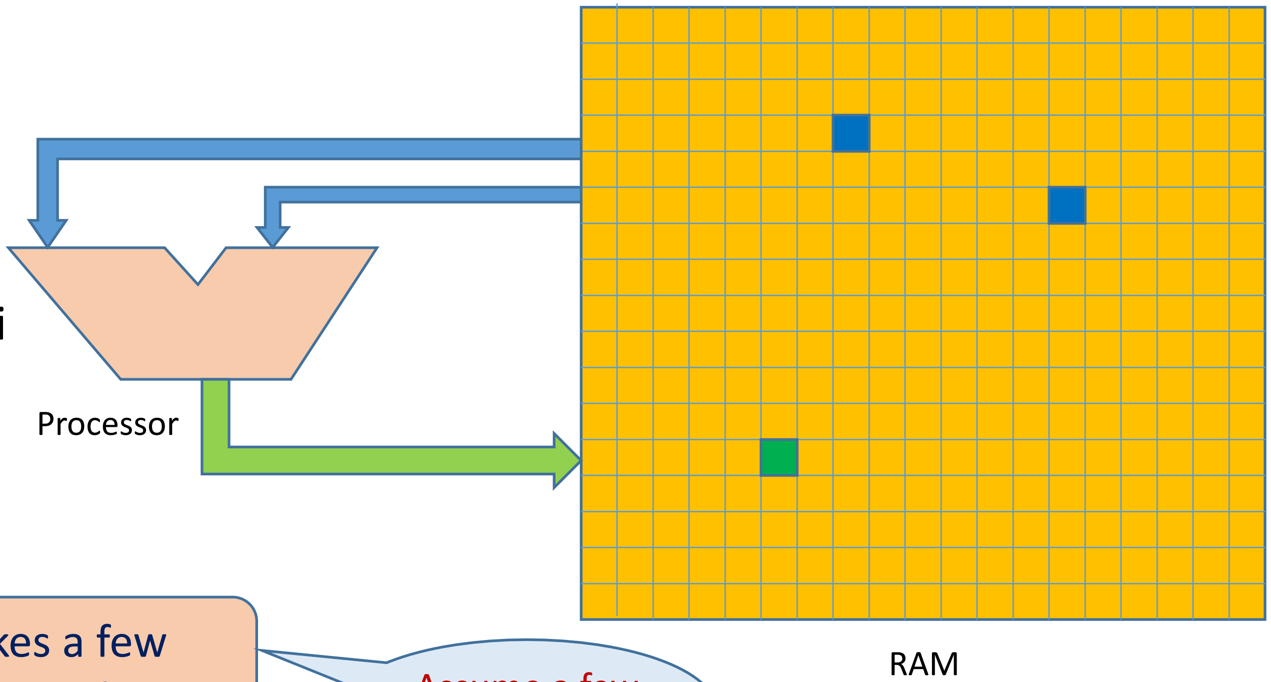
1. Simple
2. Close to real computers

Word-RAM model



How does an instruction execute?

1. Decoding instruction
2. Fetching operands
3. Performing arithmetic/logical operations
4. Storing the results back into RAM



Each instruction takes a few cycles to get executed

Assume a few nanoseconds

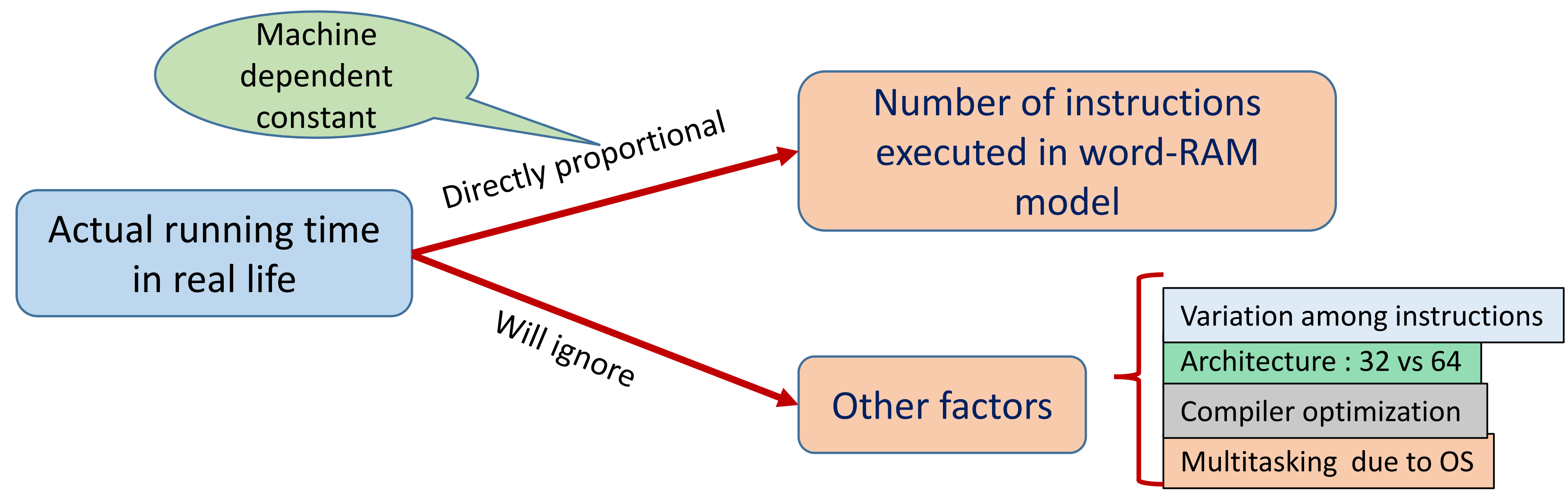
Word-RAM model

- Word is the **basic storage** unit of RAM. Word is a collection of few bytes.
- Each input item (number, name) is stored in **binary format**.
- RAM can be viewed as a huge array of words.
- Any arbitrary location of RAM can be **accessed** in the same time **irrespective** of the location.
- Data as well as Program **reside fully** in RAM.
- Each arithmetic or logical operation (+, -, *, /, OR, AND, NOT) involving a **constant number** of words takes **constant number of cycles (steps)** by the CPU.

How to measure Running Time?

This is what we will count (mathematically)

- Number of instructions taken in word-RAM model



Example - Fibonacci Number F(n)

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) for n>1

Problem 1: Given n, m, compute $F(n) mod\ m$

- Recursive algorithm
- Iterative algorithm

Two algorithms for Fibonacci numbers (mod m)

Recursive Algorithm

```
RFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1) + RFIB(n-2)) mod m);
}b
```

Iterative Algorithm

```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else {
                a ← 0;   b← 1;
                For(i=2 to n) do
                {
                temp ← b;
                b← (a+b) mod m;
                a← temp; }
        }
        return b;}
```

Analyze algorithms for F(n) mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

Let $R(n)$ be the number of instructions executed by RFIB(n,m)

$R(0) = 2$

Analyze algorithms for F(n) mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

Let $R(n)$ be the number of instructions executed by **RFIB**(n,m)

$R(0) = 2$

$R(1) = 3$

Analyze algorithms for F(n) mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

No. of instructions $\geq \mathbf{2^{(n-2)/2}}$

Let R(n) be the number of instructions executed by **RFIB**(n,m)

R(0) = 2

R(1) = 3

R(n) = 6 + R(n-1) + R(n-2) for n>1

Exercise: Use induction to show, for all $\boldsymbol{n \geq 4}$

1. $R(n) \geq F(n)$
2. $F(n) \geq 2^{(n-2)/2}$

Analyze algorithms for F(n) mod m

Now analyze the Iterative Algorithm

No. of instructions =

```
IFIB(n,m) {
          if n=0 return 0;
          else if n=1 return 1;
          else {      a ← 0;   b← 1;
                For(i=2 to n) do
                {     temp ← b;
                        b← (a+b) mod m ;
                        a← temp; }
             }
  return b;}
```

Example: Analyze algorithms for F(n) mod m

Now analyze the Iterative Algorithm

```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else {     a ← 0;   b← 1;
              For(i=2 to n) do
              {    temp ← b;
                     b← (a+b) mod m ;
                     a← temp; }
           }
   return b;}
```

No. of instructions $\leq 4+5(n-1)+1$
$= 5n$

Worst-case time

4 instructions

n-1 iterations

5 instructions

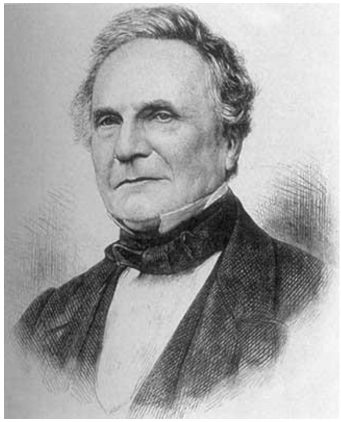
The final instruction

2.1 Computational Tractability

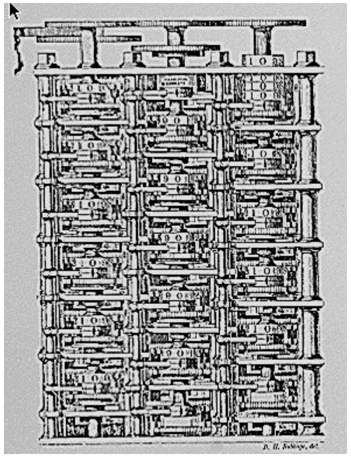
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)



Analytic Engine (schematic)

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching
with n men and n women

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method
Unix grep

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

2.2 Asymptotic Order of Growth

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

Algorithm 2
$T(n) = n^2 + 1000$

Which one is more efficient?

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

Algorithm 2
$T(n) = n^2 + 1000$

Which one is more efficient?

Algorithm 2 when $n < 10$
Algorithm 1 when $n > 10$

Time complexity really matters only for large-sized input

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

is more efficient than

Algorithm 2
$T(n) = n^2 + 1000$

Only compare for asymptotically large values of input size

Asymptotic analysis for running time

- Different machines have different running time.
- We do not measure *actual run-time*.
- We estimate the rate-of-growth of running time by asymptotic analysis.
  - Example: $0.01n^3$ grows faster than $1000n^2$!

Machine Independent

- To compare running time of two different algorithms we see which is more efficient (or fast) for large inputs in the **worst case**.

Central Mantra of Asymptotic Analysis

Suppress constant factors and lower-order terms

Asymptotic Order of Growth

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Asymmetric:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."
- Use Ω for lower bounds.

Example

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- **Claim:** $2n^2 = O(n^3)$
- **Proof:** Let $f(n)=2n^2$.
  - Note that $f(n)=2n^2 \leq n^3$ when $n \geq 2$.
  - Set $c=1$ and $n_0=2$.
  - We have $f(n)=2n^2 \leq c \cdot n^3$ for $n \geq n_0$.
  - By definition $2n^2 = O(n^3)$.

Proof Practice #1

- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- We want to prove that $f(n) = O(g(n))$ by showing that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- What should be c and n_0? (There may be more than one correct answer.)

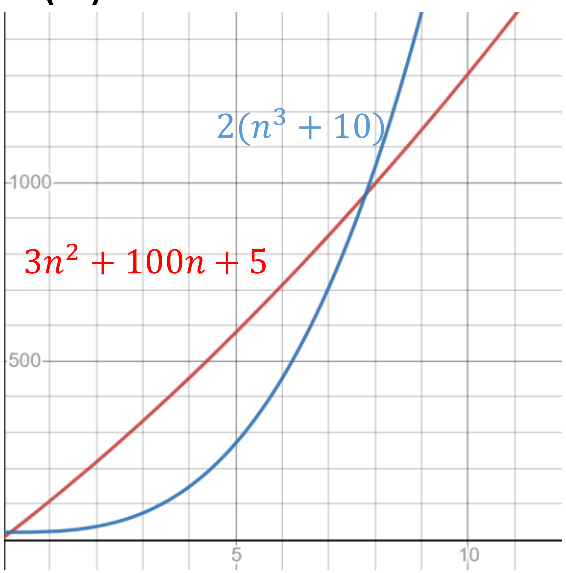
- (A) $c=2$, $n_0=10$
- (B) $c=1$, $n_0=12$
- (C) $c=5$, $n_0=2$
- (D) $c=1$, $n_0=10$

Solution for Proof Practice #1

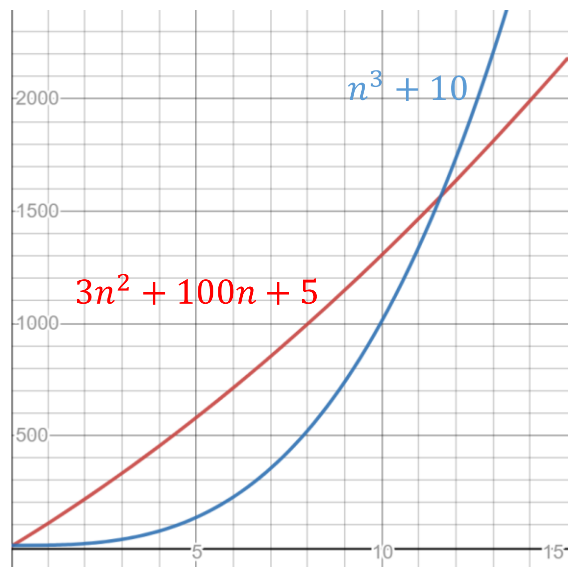
- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- (C) is false
  - When $c=5$, $n_0=2$,
  - $f(2)=3\cdot 2^2+100(2)+5=217$ and $g(2)=2^3+10=18$
  - Hence, $f(n) > c\ g(n)$ when $n=n_0=2$ and $c = 2$.
- (D) is false
  - When $c=1$, $n_0=10$,
  - $f(10)=3\cdot 10^2+100(10)+5=1305$ and $g(10)=10^3+10=1010$
  - Hence, $f(n) > c\ g(n)$ when $n=n_0=10$ and $c = 1$.

Solution for Proof Practice #1

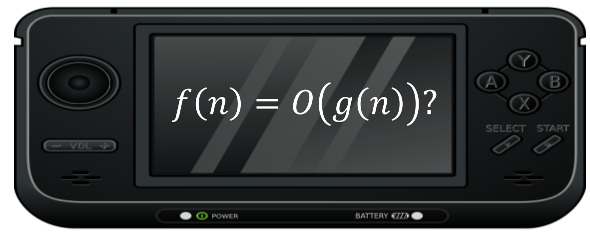
- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- (A) is true



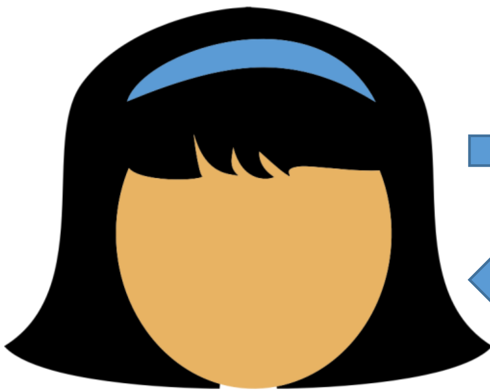
- (B) is true



Think of it like a game!



You



Numbers c and n_0

Number n bigger than n_0

Challenger

Win if $f(n) \leq c \cdot g(n)$

Win if $f(n) > c \cdot g(n)$

$f(n) = O(g(n))$ iff you have a winning strategy for above game.

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

can avoid specifying the base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

every exponential grows faster than every polynomial

Common confusions

- $2^{n+5} = O(2^n)$, because $2^{n+5} = 32 \cdot 2^n$.
- But, 2^{5n} is not $O(2^n)$. Check this!
- $\max\big(f(n), g(n)\big) = \Theta\big(f(n) + g(n)\big)$.

Proof Practice #2

Suppose $f(n) = O(g(n))$. Is $2^{f(n)} = O\left(2^{g(n)}\right)$? Select all options that hold.

- (A) Yes, for all such f and g.
- (B) Never, no matter what f and g are.
- (C) Sometimes yes, sometimes no, depending on the functions f and g.
- (D) Yes, whenever $f(n) \leq g(n)$ for all sufficiently large n.

Solution for Proof Practice #2

Suppose $f(n) = O(g(n))$. Is $2^{f(n)} = O\left(2^{g(n)}\right)$? Select all options that hold.

- (A) Yes, for all such f and g.
- (B) Never, no matter what f and g are.
- (C) Sometimes yes, sometimes no, depending on the functions f and g.
- (D) Yes, whenever $f(n) \leq g(n)$ for all sufficiently large n.

Solution for Proof Practice #2

- $f(n) = 5n, g(n) = n$ violates A, and $f(n) = g(n) = n$ violates B. So, (C) holds.

- (D) also holds, because $f(n) \leq g(n)$ implies $2^{f(n)} \leq 2^{g(n)}$.
  - But it's not necessary. $n + 5 > n$ for all n, but $2^{n+5} = O(2^n)$.

2.4 A Survey of Common Running Times

Linear Time: O(n)

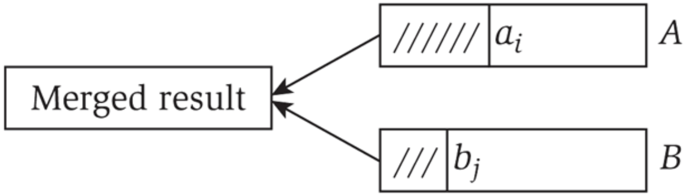
Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else         append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^2
      if (d < min)
         min ← d
   }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$ Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```