Models of Computing

General purpose model [can do anything a real computer can]

I. pseudo-code, each line takes 1 time step

careful: e.g. initializing an array might be 1 line but should cost n time steps (n = length of array).

With this caution, this model reflects reality — so long as the problem fits in main memory and the numbers involved fit in one word.

Dealing with large numbers:

example function Fibonacci (n)

```
    i ← 0, j ← 1
    for k ← 1 to n
        j ← i+j
        i ← j-i   (old value of j)
    return j
```

2n steps

$0, 1, 1, 2, \underbrace{3, 5}_{+}, 8, \ldots$

But the numbers grow so quickly that 2n steps does not reflect reality: $n = 47$ causes overflow on 32 bit words.

II. Pseudo-code, take word size into account

Either limit word size (constant or $\log n$) OR count cost of arithmetic on large words e.g. $a * b$

size of number a = # bits in binary representation

$\sim \log a \qquad \lfloor \log a + 1 \rfloor$

$a * b$ takes $O((\log a)(\log b)^{.59})$ using efficient multiplication.

[we'll see the alg. later on]

school method takes $O((\log a)(\log b))$

We can be more formal by defining a simple assembly language and counting steps in that:

III RAM = Random Access Machine — abstracts assembly lang.

- "random access" means we can access memory location i at unit cost (not like tape or Turing machine.

word RAM — each memory location holds one word and assume # bits in word $\geq \log n$, n = input size

IV Circuit family — abstracts hardware circuitry

V Turing machine — abstracts human computer working with pencil and paper

Note: time to access memory location i is proportional to i.

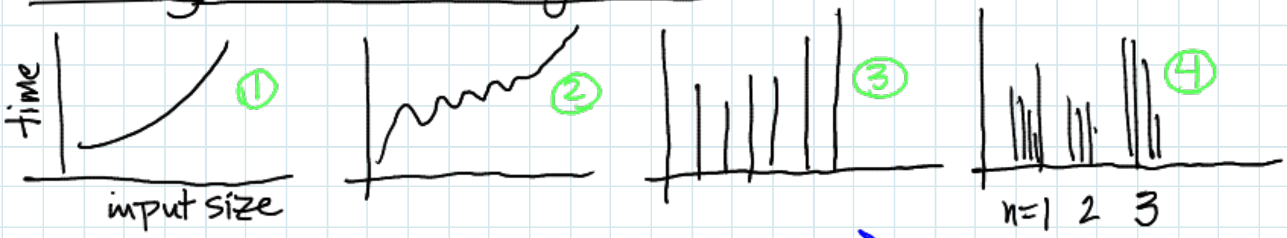
Special purpose or "structured" models of computing

- comparison-based model for sorting $\Omega(n \log n)$ lower bound.
- arithmetic model

Our model of Computing

- pseudo-code and try to be realistic about what are elementary operations
- for graph algorithms, we'll assume $\log n$ bits per word and constant time for arithmetic on words.

Running Time of an Algorithm



we think of this   more accurately →   we'll start here and abstract back

running time depends on input

$T_{\mathcal{A}}(I)$ – running time of algorithm $\mathcal{A}$ on input I ④

We expect running time to increase as size of input increases

Simplify by expressing run-time as a fn of input size ③

[Note: model of computing must say how to count input size]

For given size n, there are various inputs (a finite number).

How do we combine running-times to one number?

Worst case running time

[the standard unless otherwise specified]

$T_{\mathcal{A}}(n) = \max\{T_{\mathcal{A}}(I) : I \text{ an input of size } n\}$

[leave off $\mathcal{A}$ if it's understood]

Alternative: average case running time – replace Max by Average

or use a more general probability distribution on inputs (avg. assumes uniform distribution).

When an algorithm uses random numbers (e.g. quicksort) we use <u>expected run time</u>

[discuss later]

$T_A^E(I)$ = expected run time on input I
(depends on random nos. used by A)

Still use worst case over inputs

$T_A^E(n) = \text{Max}\{T_A^E(I) : I \text{ input of size } n\}$

<u>Challenge</u> If you have a RAM model and allow unlimited # bits for each memory location and charge 1 for arithmetic and shifts, how can you "cheat" and sort in $O(n)$ steps

Hint: Do all n^2 pairwise comparisons in 1 subtraction

Asymptotic Analysis of Algorithms

Recall running time of algorithm as fn of input size n

$T(n) = \max\{T(I) : I \text{ an input of size } n\}$

T(I) and size(I) measured according to model of computation

We want T(n) to be

- simple to express, e.g. n^2
- machine independent, thus ignore multiplicative factors (one machine might be twice as fast), thus ignore lower order terms ($n+5 \leq 2n$, $n \geq 5$)

Definition Big Oh notation

Let $f(n), g(n)$ be functions from $\mathbb{N}$ to $\mathbb{R}^{\geq 0}$

$f(n)$ is $O(g(n))$ "order $g(n)$", "big oh of $g(n)$"

if $\exists$ constants $c > 0$ and n_0 s.t.

$f(n) \leq c \cdot g(n)$ for all $n > n_0$.

(we say f is bounded by a constant times g for n sufficiently large – this is what asymptotic means)

Notation $f(n) \in O(g(n))$ [CLRS writes $f(n) = O(g(n))$]

Big oh gives an upper bound. Examples:

- $T(n) = 5n^2 + 3n + 25$ is $O(n^2)$
- $10^{100} \cdot n$ is $O(n)$
- $\log n$ is $O(n)$ but n is not $O(\log n)$
- 2^{n+1} is $O(2^n)$
- ? $(n+1)!$ is $O(n!)$? NO.

Properties of Big oh

- max rule: $O(f(n)+g(n))$ is $O(\max\{f(n), g(n)\})$
- transitivity: $f(n) \in O(g(n))$, $g(n) \in O(h(n))$
  $\Rightarrow f(n) \in O(h(n))$

<u>Further Definitions</u>

- $f(n)$ is $\Omega(g(n))$ "omega" if $\exists$ constants c, n_0 s.t.
  $f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$
- $f(n)$ is $\Theta(g(n))$ "theta" if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$
  – gives an <u>exact</u> (asymptotic) bound.
- $f(n)$ is $o(g(n))$ "little oh" if for any constant $c>0$, there exists a constant n_0 s.t.
  $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$
  equivalently $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ (for $f, g : \mathbb{N} \to \mathbb{R}^+$)

<u>Comparing Algorithms using Asymptotic Analyis</u>

Suppose the worst case run time of
algorithm A is $O(n^2)$
algorithm B is $O(n \log n)$

Which is better? – We can't know!

This is like $x \leq 5$ $y \leq 10$, which is smaller?

To compare algorithms, we need tight bounds

$\Theta(n^2)$ vs $\underline{\Theta(n \log n)}$ better.

O is like $\leq$

o . . . $<$

θ . . . $=$

One difference: we can compare any 2 numbers

$x \leq y$ or $y \leq x$

But there are functions f, g s.t. $f(n)$ is not $O(g(n))$ and $g(n)$ is not $O(f(n))$. Find some.

<u>Challenge</u>

Can you find such fns using just $+, *$, exponentiation, log?

- not allowed to use sin, $\lfloor \rfloor$, $\lceil \rceil$
- not allowed to say $f(n) = \begin{cases} \text{—} & n \text{ even} \\ \text{—} & n \text{ odd} \end{cases}$

Ref. G.H. Hardy. See Concrete Math, Graham, Knuth, Patashnik.

Typical run-times and how they compare

$\log n$ - binary search
$n \log n$ - sorting
n - find max
n^2 - insertion sort
n^3 — multiplying two $n \times n$ matrices
$n!$ — try all orderings of a set (e.g. Travelling Salesman)
2^n — try all subsets

$\sqrt{n}$, $\log \log n$, $\log^2 n$

Ordering, where $f(n) \ll g(n)$ means $f(n) \in o(g(n))$

$$1 \ll \log\log n \ll \log n \ll \log^2 n \ll \sqrt{n} \ll n$$
$$\ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n!$$

Also $n^a \in o(n^b)$ $b > a > 0$

$\log^a n \in o(n^b)$ $b > 0$

$n^a \in o(2^n)$

Skiena Ch. 2 is a good reference

Examples using above.

- $n \log n$ vs $n\sqrt{n}/2$

  from above $\log n \in o(\sqrt{n})$

  so $n \log n \in o(n\sqrt{n}/2)$

- $\log(\sqrt{n})$ vs $\log n$

  $\log(\sqrt{n}) = \log n^{\frac{1}{2}} = \frac{1}{2} \log n \in \Theta(\log n)$

Sometimes we will analyze an algorithm's run-time in terms of several parameters.

EX1. graph with n vertices and m edges
m is $O(n^2)$ but $O(n+m)$ can be better than $O(n^2)$

EX2. input and output size
e.g. Jarvis' March $O(n \cdot h)$ n = input size, h = output size.

Defn $f, g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$
$f(n,m)$ is $O(g(n,m))$ if $\exists$ constants c, n_0, m_0 s.t.
$f(n,m) \leq c \cdot g(n,m)$ for all $n \geq n_0$, $m \geq m_0$

Summary We analyze algorithms by analyzing the asymptotic rate of growth of the worst case run time.

Does it really matter?
Show charts from Garey & Johnston, Skiena

from Skiena's book

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

Size of Largest Problem Instance
Solvable in 1 Hour

	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Figure 1.3 from *Computers and Intractability: A Guide to the Theory of NP-Completeness,* by Garey and Johnson: Effect of improved technology on several polynomial and exponential time algorithms.