Objectives

- Analyzing algorithms
- Asymptotic running times

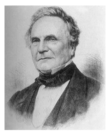
Our Process

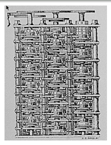
1. Understand/identify problem
   - Simplify as appropriate
2. Design a solution
3. Analyze
   - Correctness, efficiency
   - May need to go back to step 2 and try again
4. Implement (On Monday)
   - Within bounds shown in analysis

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)

http://plan28.org/

TODAY'S GOAL:
DEFINE ALGORITHM EFFICIENCY

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 "Exponential"
  - Unacceptable in practice

Example: How many possible solutions are there in the stable matching problem?

In other words, how many possible *perfect* matchings are there? For each perfect match, we'll check if it's stable.

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 "Exponential"
  - Unacceptable in practice
- Example: Stable matching: n! with n men and n women
  - If n increases by 1, what happens to the running time?

How Do We Measure Runtime?

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

What are alternatives to worst-case analysis?

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

Towards a Definition of Efficient…

- Desirable scaling property: When input size doubles, algorithm should only slow down by some constant factor C
  - Doesn't grow multiplicatively

Polynomial-Time

Defn. 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.

✓Desirable scaling property: When input size doubles, algorithm should only slow down by some constant factor C
  - What happens if we double N?

- Defn. An algorithm is *polynomial time* (or *polytime)* if the above scaling property holds.

Algorithm Efficiency

- **Defn.** An algorithm is ***efficient*** if its running time is *polynomial*
- Justification: It really works in practice!
  - In practice, 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 ($6.02 \times 10^{23} \times N^{20}$) and are useless in practice
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare

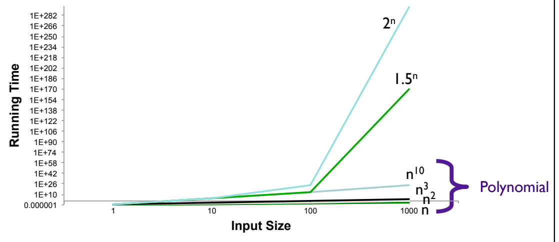
Running Times

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.

Input Size	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

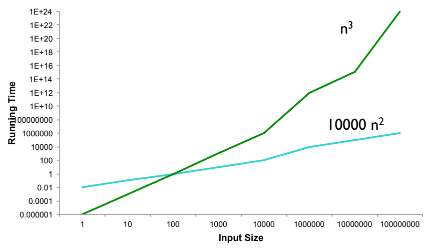
Polynomial

Visualizing Running Times



- Huge difference from polynomial to not polynomial
- Differences in runtime matter more as input size increases

Comparing 10000 n^2 and n^3



As input size increases, n^3 dominates large constant * n^2
➔Care about running time as input size approaches infinity
➔Only care about highest-order term

Asymptotic Order of Growth: Upper Bounds

- **T(n)** is the worst case running time of an algorithm
- We say that 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 $T(n) \leq c \cdot f(n)$

"order f(n)"

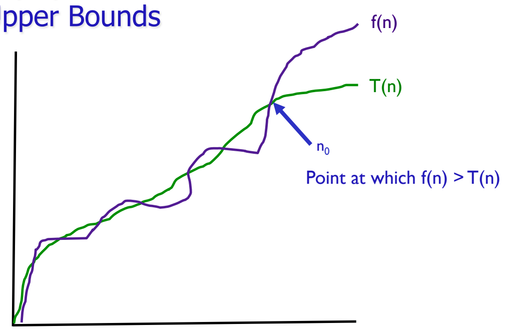
c cannot depend on n

sufficiently large n

T(n) is bounded above by a constant multiple of f(n)

➔T is ***asymptotically upperbounded*** by f

Asymptotic Order of Growth: Upper Bounds



Upper Bounds Example

- Find an upperbound for
  $T(n) = pn^2 + qn + r$
  - p, q, r are positive constants

Idea: Let's inflate the terms in the equation so that all terms are n^2

Upper Bounds Example

- $T(n) = pn^2 + qn + r$
  - p, q, r are positive constants
- For all $n \geq 1$,

$$\begin{aligned} T(n) &= pn^2 + qn + r \\ &\leq pn^2 + qn^2 + rn^2 \\ &= (p+q+r)\, n^2 \\ &= c\, n^2 \end{aligned}$$

➔ $T(n) \leq cn^2$, where $c = p+q+r$

➔ $T(n) = O(n^2)$

- Also correct to say that $T(n) = O(n^3)$

Notation

- $T(n) = O(f(n))$ is a slight abuse of notation
  - 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
  - Use Ω for lower bounds

Asymptotic Order of Growth: Lower Bounds

- Complementary to upper bound

ε cannot depend on n

- T(n) is **$\Omega(f(n))$** if there exist constants $\varepsilon > 0$

sufficiently large n

and $n_0 \geq 0$ such that for all $n \geq n_0$, we have

$T(n) \geq \varepsilon \cdot f(n)$

T(n) is bounded below by a constant multiple of f(n)

➔T is ***asymptotically lowerbounded*** by f

Example: Lower Bound

- $T(n) = pn^2 + qn + r$
  - p, q, r are positive constants
- Idea: *Deflate* terms rather than inflate
- For all $n \geq 0$,

$T(n) = pn^2 + qn + r \geq pn^2$

➔ $T(n) \geq \varepsilon n^2$, where $\varepsilon = p > 0$

➔ $T(n) = \Omega(n^2)$

- Also correct to say that $T(n) = \Omega(n)$

Tight bounds

T(n) is **$\Theta(f(n))$** if T(n) is both $O(f(n))$ and $\Omega(f(n))$

- The "right" bound

A Fashion Analogy

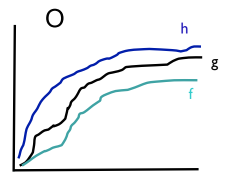
- O == Hammer pants
  - Loose and baggy with plenty of room for the pants to shrink or the body to grow
- Ω == The pants you plan to fit in this summer after working off the snacks from Christmas
- Θ == Katy Perry's skin tight jeans in a teenage dream
  - Can't make them any smaller, and no extra room to even fit a cell phone in the pocket

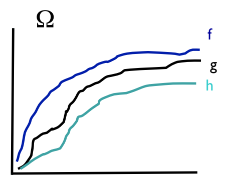
Courtesy Andy Danner, Swarthmore

Property: Transitivity

- If f = O(g) and g = O(h) then f = O(h)
- If f = Ω(g) and g = Ω(h) then f = Ω(h)
- If f = Θ(g) and g = Θ(h) then f = Θ(h)

Proofs in book





Property: Additivity

- If f = O(h) and g = O(h) then f + g = O(h)
- If f = Ω(h) and g = Ω(h) then f + g = Ω(h)
- If f = Θ(h) and g = Θ(h) then f + g = Θ(h)

Proofs in book

Sketch proof for O:
By defn, f ≤ c · h
By defn, g ≤ d · h
f + g ≤ c · h + d · h = (c + d) h = c' · h
→ f + g is O(h)

Practice: Asymptotic Order of Growth

What are the upper bounds, lower bounds, and tight bound on T(n)?

- $T(n) = 32n^2 + 17n + 32$

Practice: Asymptotic Order of Growth

- $T(n) = 32n^2 + 17n + 32$
  - 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 $\Theta(n^3)$

ASYMPTOTIC BOUNDS FOR CLASSES OF ALGORITHMS

Asymptotic Bounds for Polynomials

- $a_0 + a_1 n + \ldots + a_d n^d \in \Theta(n^d)$ if $a_d > 0$

→ Runtime determined by higher-order term

- Polynomial time. Running time is $O(n^d)$ for some constant *d* that is independent of the input size *n*
- Other examples of polynomial times:
  - $O(n^{1/2})$
  - $O(n^{1.58})$
  - $O(n \log n) \leq O(n^2)$

Asymptotic Bounds for Logarithms

- Logarithms. $\log_b n = x$, where $b^x = n$
  - Approximate: To represent *n* in base-*b,* need *x*+1 digits

N	b	x
100	10	
1000	10	
100	2	
1000	2	

Asymptotic Bounds for Logarithms

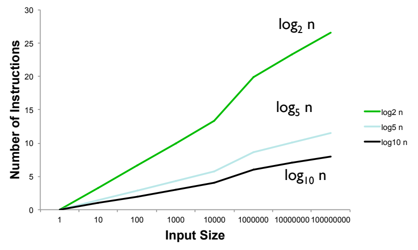
- Logarithms. $\log_b n = x$, where $b^x = n$
  - Approximate: To represent *n* in base-*b,* need *x*+1 digits

N	b	x
100	10	2
1000	10	3
100	2	6.64
1000	2	9.92

Describe the running time of an O(log n) algorithm as the input size grows. Compare with polynomials.

Asymptotic Bounds for Logarithms

- Logarithms. $\log_b n = x$, where $b^x = n$
  - x is number of digits to represent n in base-b representation



Asymptotic Bounds for Logarithms

- Logarithms. $\log_b n = x$, where $b^x = n$

→ Slowly growing functions

- Identity: $\log_a n = \log_b n / \log_b a$
  - Means that

$\log_a n = 1/\log_b a * \log_b n$

Constant!

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

Asymptotic Bounds for Logarithms

- Logarithms. $\log_b n = x$, where $b^x = n$

→ Slowly growing functions

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

→ Don't need to specify the base

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

→ Log grows slower than every polynomial

Asymptotic Bounds for Exponentials

- Exponentials: functions of the form $f(n) = r^n$ for constant base *r*
  - Faster growth rates as *n* increases
- For every r > 1 and every d > 0, $n^d = O(r^n)$

→ Every exponential grows faster than every polynomial

Summary of Asymptotic Bounds

- In terms of growth rates ….

Logarithms < Polynomials < Exponentials

A SURVEY OF COMMON RUNNING TIMES

Linear Time: O(n)

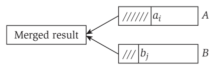
- Running time is at most a ***constant*** factor times the size of the input
- Example. Computing the maximum: Compute maximum of n numbers $a_1, \ldots, a_n$

```
max = $a_1$
for i = 2 to n
   if ($a_i$ > max)
      max = $a_i$
```

Constant work for each input (does not depend on n)

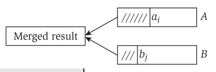
Example Linear Time: O(n)

- 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



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

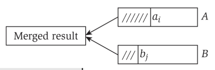


```
i = 1, j = 1
while (both lists are nonempty)
   if ($a_i \le b_j$)
      append $a_i$ to output list and increment i
   else ($a_i \le b_j$)
      append $b_j$ to output list and increment j

append remainder of nonempty list to output list
```

Example 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
- Claim. Merging two lists of size n takes O(n) time
- Proof. After each comparison, the length of output list increases by 1



```
i = 1, j = 1
while (both lists are nonempty)
   if ($a_i \le b_j$)
      append $a_i$ to output list and increment i
   else ($a_i \le b_j$)
      append $b_j$ to output list and increment j

append remainder of nonempty list to output list
```

O(n log n) Time

- Also referred to as ***linearithmic*** time
- Arises in divide-and-conquer algorithms
  - Splitting input into equal pieces, solve recursively, combine solutions in linear time

What well-known set of algorithms has an O(n logn) running time?

O(n log n) Time Example

- Sorting: Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons
- Mergesort
  1. Break input into equal-sized pieces
  2. Sorts each half recursively
  3. Merges sorted halves into a sorted list

Talk about the bound on running time later…

O(n log n) Time Example

- Largest empty interval. Given *n* (not necessarily ordered) time-stamps $x_1, \dots, x_n$ at 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) Time Example

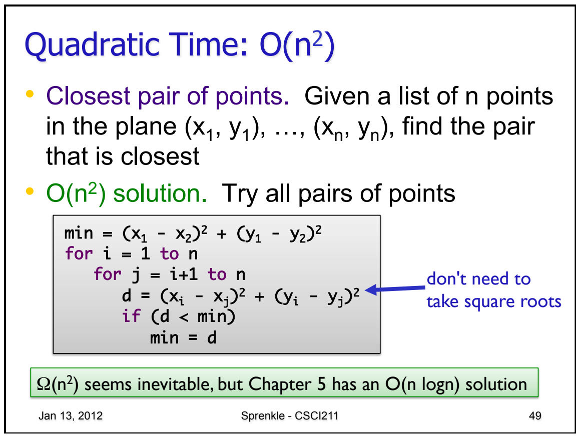
- Largest empty interval. Given *n* (not necessarily ordered) time-stamps $x_1, \dots, x_n$ at which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?
- O(n log n) solution
  1. Sort time-stamps
  2. Scan sorted list in order, identifying the maximum gap between successive time-stamps

Quadratic Time: $O(n^2)$

- Examples?

Quadratic Time: $O(n^2)$

- Examples:
  - Enumerate all pairs of elements
  - Often involves nested loops (n iterations)



Quadratic Time: O(n^2)

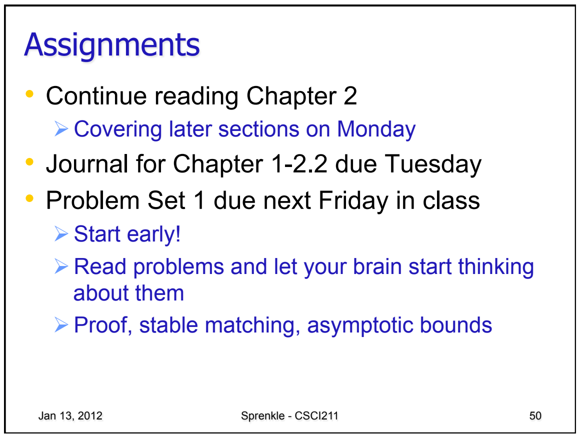
- Closest pair of points. Given a list of n points in the plane (x_1, y_1), …, (x_n, y_n), find the pair that is closest
- O(n^2) solution. Try all pairs of points

```
min = $(x_1 - x_2)^2 + (y_1 - y_2)^2$
for i = 1 to n
   for j = i+1 to n
      d = $(x_i - x_j)^2 + (y_i - y_j)^2$
      if (d < min)
         min = d
```

don't need to take square roots

$\Omega(n^2)$ seems inevitable, but Chapter 5 has an O(n logn) solution

Jan 13, 2012 Sprenkle - CSCI211 49



Assignments

- Continue reading Chapter 2
  - Covering later sections on Monday
- Journal for Chapter 1-2.2 due Tuesday
- Problem Set 1 due next Friday in class
  - Start early!
  - Read problems and let your brain start thinking about them
  - Proof, stable matching, asymptotic bounds

Jan 13, 2012 Sprenkle - CSCI211 50