Implications for Algorithm Design

Θ	running time on fast computer	characteristics of typical tasks with the specified running time
1	n is irrelevant	examine only a fixed number of things regardless of input size
log n	any n is fine	repeatedly eliminate a fraction of the search space
n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	examine all pairs insertion sort, selection sort
n^3		examine all triples
2^n	impractical for n > 40	enumerate all subsets
n!	useless for n ≥ 20	enumerate all permutations

Big-Oh From Algorithms

use the table on the previous slide

An array contains each of the numbers 1..n plus one duplicate value. Which value is duplicated?

- Algorithm A uses quicksort or mergsort to sort all of the numbers, then makes one pass through the array looking for adjacent slots with the same value.
- Algorithm B makes one pass through the array to sum the numbers, then uses the formula $\frac{n(n-1)}{2}$ to calculate the sum of the numbers 1..n and subtracts that from the sum of the array's value.
- Algorithm C makes one pass through the array and for each value, makes a pass through the rest of the array to see if another copy of that value is found i.e. each value in the array is compared to each other value to find the duplicate.

sort, then examine each object a fixed number of times → Θ(n log n) + Θ(n) = Θ(n log n)

examine each object a fixed number of times, then examine only a fixed number of things → Θ(n) + Θ(1) = Θ(n)

for each object, examine each object a fixed number of times → Θ(n) x Θ(n) = Θ(n^2)

B A C

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		

suitability for n = 25, 2500, 250,000, 250,000,000

Questions

How do you choose between multiple algorithms with suitable big-Ohs?

n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	examine all pairs insertion sort, selection sort

- if n = 1,000, all three of these are potentially suitable
- consider other factors
  - is there already a library implementation?
  - if you have to implement something, which is simpler to implement (and implement correctly)?
  - are there significant differences in memory usage?

Questions

O(n log n) is pretty practical – why couldn't you just use mergesort or quicksort for a very large array?

n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort

- real systems have only a limited amount of memory
  - if the array is too large to fit into memory, it is kept on disk and parts are swapped into memory when needed
- if successive accesses are scattered throughout the array, the system spends all of its CPU time swapping things in and out of memory instead of actually sorting
  - the assumption that each memory access is one time step also breaks down
- need algorithms exhibiting *locality of access* to minimize swaps

Key Points

- the running time of a series of simple operations is Θ(1)
- the running time of a loop is the sum of the time taken by each iteration
  - if the time is the same for each iteration, the total time reduces to the number of repetitions times the time per iteration
- the running time of a recursive function is expressed with a recurrence relation
- logs and exponents come into play when something is repeatedly divided or multiplied

The following table outlines the few easy rules with which you will be able to compute $\Theta(\sum_{i=1}^{n} f(i))$ for functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$. (We consider more general functions at the end of this section.)

b^a	d	e	Type of Sum	$\sum_{i=1}^{n} f(i)$	Examples	
> 1	Any	Any	**Geometric Increase (dominated by last term)**	$\Theta(f(n))$	$\sum_{i=0}^{n} 2^{2^i}$	$\approx 1 \cdot 2^{2^n}$
					$\sum_{i=0}^{n} b^i$	$= \Theta(b^n)$
					$\sum_{i=0}^{n} 2^i$	$= \Theta(2^n)$
$= 1$	> -1	Any	**Arithmetic-like (half of terms approximately equal)**	$\Theta(n \cdot f(n))$	$\sum_{i=1}^{n} i^d$	$= \Theta(n \cdot n^d) = \Theta(n^{d+1})$
					$\sum_{i=1}^{n} i^2$	$= \Theta(n \cdot n^2) = \Theta(n^3)$
					$\sum_{i=1}^{n} i$	$= \Theta(n \cdot n) = \Theta(n^2)$
					$\sum_{i=1}^{n} 1$	$= \Theta(n \cdot 1) = \Theta(n)$
					$\sum_{i=1}^{n} \frac{1}{i^{0.99}}$	$= \Theta(n \cdot \frac{1}{n^{0.99}}) = \Theta(n^{0.01})$
	$= -1$	$=0$	**Harmonic**	$\Theta(\ln n)$	$\sum_{i=1}^{n} \frac{1}{i}$	$= \log_e(n) + \Theta(1)$
	< -1	Any	**Bounded tail (dominated by first term)**	$\Theta(1)$	$\sum_{i=1}^{n} \frac{1}{i^{1.001}}$	$= \Theta(1)$
					$\sum_{i=1}^{n} \frac{1}{i^2}$	$= \Theta(1)$
< 1	Any	Any			$\sum_{i=1}^{n} (\frac{1}{2})^i$	$= \Theta(1)$
					$\sum_{i=0}^{n} b^{-i}$	$= \Theta(1)$

from Jeff Edmonds, *How to Think About Algorithms*

Big-Oh for Sums

Use the big-Oh for sums table to find the Θ approximation for the sum $\sum_{i=1}^{n} i \log i$.

2. [W] Give the Θ approximation for each of the following sums. Use the big-Oh for sums table.

   a. $\Sigma_{i=1..n}$ (log i)
   b. $\Sigma_{i=1..n}$ $(1/2^i)$
   c. $\Sigma_{i=1..\log n}$ $(n\, i^2)$
   d. $\Sigma_{i=1..n}$ $\Sigma_{j=1..i^2}$ (ij log i)

Big-Oh From Algorithms

An array contains each of the numbers 1..n plus one duplic value. Which value is duplicated?

- Algorithm A uses quicksort or mergsort to sort all of the numbers, then makes one pass through the array looking for adjacent slots with the same value.
- Algorithm B makes one pass through the array to sum the numbers, then uses the formula $\frac{n(n-1)}{2}$ to calculate the sum of the numbers 1..n and subtracts that from the sum of the array's value.
- Algorithm C makes one pass through the array and for each value, makes a pass through the rest of the array to see if another copy of that value is found i.e. each value in the array is compared to each other value to find the duplicate.

```
sort(arr)
for i ← 0..n-2
  if arr[i] == arr[i+1]
    dup ← arr[i]
    break
```

```
sum ← 0
for i ← 0..n-1
  sum += arr[i]
dup ← sum-n(n-1)/2
```

```
for i ← 0..n-1
  for j ← i+1..n-1
    if arr[i] == arr[j]
      dup ← arr[j]
      break
```


Exponent Rules

Assume that a and b are nonzero real numbers, and m and n are any integers.

1) Zero Property of Exponent

$$b^0 = 1$$

2) Negative Property of Exponent

$$b^{-n} = \frac{1}{b^n} \quad \text{OR} \quad \frac{1}{b^{-n}} = b^n$$

3) Product Property of Exponent

$$\left(b^m\right)\left(b^n\right) = b^{m+n} \qquad b^{1/2} = \sqrt{b}$$

4) Quotient Property of Exponent

$$\frac{b^m}{b^n} = b^{m-n}$$

5) Power of a Power Property of Exponent

$$\left(b^m\right)^n = b^{mn}$$

6) Power of a Product Property of Exponent

$$(ab)^m = a^m b^m$$

7) Power of a Quotient Property of Exponent

$$\left(\frac{a}{b}\right)^m = \frac{a^m}{b^m}$$

Log Rules

definition of log:
if x = $\log_b(n)$ then n = b^x

Rule 1: $\log_b(M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b\left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b\left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b(1) = 0$

Rule 5: $\log_b(b) = 1$

Rule 6: $\log_b\left(b^k\right) = k$

Rule 7: $b^{\log_b(k)} = k$

Where: $b > 1$, and M, N and k can be any real numbers but M and N must be positive!

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)} \qquad d^{c\log_2(n)} = n^{c\log_2(d)}$$

Logarithms and Exponents

For the following pairs of functions, indicate whether f=O(g), f=Ω(g), or f=Θ(g).

- $f(n) = \log n^2, g(n) = 2^{\log n}$ [pairA]
- $f(n) = \log_{10} n, g(n) = 10n$ [pairB]
- $f(n) = \log_{10} n, g(n) = \log_2 2n$ [pairC]

- tips
  - know the growth rate ordering of common functions: 1, log n, n, n log n, n^2, 2^n, n!
  - simplify other functions to make them more familiar

Solving Recurrence Relations

T(n) = a T(n-b) + f(n) where f(n) = $\Theta(n^c \log^d n)$

Cases are based on the number of subproblems and f(n).

a	f(n)	behavior	solution
> 1	any	base case dominates (too many leaves)	T(n) = $\Theta(a^{n/b})$
1	≥ 1	all levels are important	T(n) = Θ(n f(n))

Solving Recurrence Relations

$T(n) = a\,T(n/b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$

Cases are based on the relationship between the number of subproblems, the problem size, and f(n).

(log a)/(log b) vs c	d	behavior	solution
<	any	top level dominates – more work splitting/combining than in subproblems (root too expensive)	$T(n) = \Theta(f(n))$
=	> -1	all levels are important – log n steps to get to base case, and roughly same amount of work in each level	$T(n) = \Theta(f(n) \log n)$
=	< -1	base cases dominate – so many subproblems that taking care of all the base cases is more work than splitting/combining (too many leaves)	$T(n) = \Theta(n^{(\log a)/(\log b)})$
>	any		

Big-Oh for Recurrence Relations

Use the big-Oh for recurrence relations tables to find the Θ approximation for the recurrence relation
$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n)$.

- $T(n) = 2T(n/2) + \Theta(\log n)$
- $T(n) = 3T(n/9) + \Theta(n)$
- $T(n) = 8T(n/2) + \Theta(n^2)$
- $T(n) = T(n-1) + \Theta(1)$

The Limits of Asymptotic Complexity

- big-Oh provides a useful but big picture view
  - allows comparing algorithms rather than programs
  - can determine if an algorithm is fast enough to be practical
- big-Oh is not suitable for “which is faster?” comparisons between algorithms whose running times belong to the same growth rate class
  - specific implementation details, constant factors, and lower-order terms matter
  - ways in which real systems differ from the RAM model matter
  - actual performance depends on the specific inputs typical for the application