CSC 321: Data Structures

Fall 2013

Algorithm analysis, searching and sorting

- best vs. average vs. worst case analysis
- big-Oh analysis (intuitively)
- analyzing searches & sorts
- general rules for analyzing algorithms
- analyzing recursion recurrence relations
- specialized sorts
- big-Oh analysis (formally), big-Omega, big-Theta

1

Algorithm efficiency

when we want to classify the efficiency of an algorithm, we must first identify the costs to be measured

- memory used? sometimes relevant, but not usually driving force
- execution time? dependent on various factors, including computer specs
- # of steps somewhat generic definition, but most useful

to classify an algorithm's efficiency, first identify the steps that are to be measured

e.g., for searching: # of inspections, …
for sorting: # of inspections, # of swaps, # of inspections + swaps, …

must focus on key steps (that capture the behavior of the algorithm)

- e.g., for searching: there is overhead, but the work done by the algorithm is dominated by the number of inspections

2

Best vs. average vs. worst case

when measuring efficiency, you need to decide what case you care about

- best case: usually not of much practical use
  - the best case scenario may be rare, certainly not guaranteed
- average case: can be useful to know
  - on average, how would you expect the algorithm to perform
  - can be difficult to analyze – must consider all possible inputs and calculate the average performance across all inputs
- worst case: most commonly used measure of performance
  - provides upper-bound on performance, guaranteed to do no worse

sequential search:	best?	average?	worst?
binary search:	best?	average?	worst?

note: best ≠ small, worst ≠ big   best/worst case are relative to arbitrary size N

3

Big-Oh (intuitively)

intuitively: an algorithm is O(f(N)) if the # of *steps* involved in solving a problem of size N has f(N) as the dominant term

O(N):	5N	3N + 2	N/2 – 20
$O(N^2)$:	N^2	$N^2 + 100$	$10N^2 – 5N + 100$

…

why aren't the smaller terms important?

- big-Oh is a "long-term" measure
- when N is sufficiently large, the largest term dominates

consider $f_1(N) = 300*N$ (a very steep line) & $f_2(N) = ½*N^2$ (a very gradual quadratic)

in the short run (i.e., for small values of N), $f_1(N) > f_2(N)$
  e.g., $f_1(10) = 300*10 = 3,000 > 50 = ½*10^2 = f_2(10)$
in the long run (i.e., for large values of N), $f_1(N) < f_2(N)$
  e.g., $f_1(1,000) = 300*1,000 = 300,000 < 500,000 = ½*1,000^2 = f_2(1,000)$

4

Big-Oh and rate-of-growth

big-Oh classifications capture rate of growth

- for an O(N) algorithm, doubling the problem size doubles the amount of work

  e.g., suppose Cost(N) = 5N – 3
  - Cost(s) = 5s – 3
  - Cost(2s) = 5(2s) – 3 = 10s - 3

- for an O(N log N) algorithm, doubling the problem size more than doubles the amount of work

  e.g., suppose Cost(N) = 5N log N + N
  - Cost(s) = 5s log s + s
  - Cost(2s) = 5(2s) log (2s) + 2s = 10s(log(s)+1) + 2s = 10s log s + 12s

- for an $O(N^2)$ algorithm, doubling the problem size quadruples the amount of work

  e.g., suppose Cost(N) = $5N^2 – 3N + 10$
  - Cost(s) = $5s^2 – 3s + 10$
  - Cost(2s) = $5(2s)^2 – 3(2s) + 10 = 5(4s^2) – 6s + 10 = 20s^2 – 6s + 10$

5

Big-Oh of searching/sorting

sequential search: worst case cost of finding an item in a list of size N

- may have to inspect every item in the list

Cost(N) = N inspections + overhead
→ O(N)

selection sort: cost of sorting a list of N items

- make N-1 passes through the list, comparing all elements and performing one swap

Cost(N) = (1 + 2 + 3 + … + N-1) comparisons + N-1 swaps + overhead
= N*(N-1)/2 comparisons + N-1 swaps + overhead
= ½ N^2 – ½ N comparisons + N-1 swaps + overhead
→ $O(N^2)$

6

General rules for analyzing algorithms

1. for loops: the running time of a for loop is at most
   running time of statements in loop × number of loop iterations

```
for (int i = 0; i < N; i++) {
    sum += nums[i];
}
```

2. nested loops: the running time of a statement in nested loops is
   running time of statement in loop × product of sizes of the loops

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        nums1[i] += nums2[j] + i;
    }
}
```

7

General rules for analyzing algorithms

3. consecutive statements: the running time of consecutive statements is
   sum of their individual running times

```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += nums[i];
}
double avg = (double)sum/N;
```

4. if-else: the running time of an if-else statement is at most
   running time of the test + maximum running time of the if and else cases

```
if (isSorted(nums)) {
    index = binarySearch(nums, desired);
}
else {
    index = sequentialSearch(nums, desired);
}
```

8

EXAMPLE: finding all anagrams of a word (approach 1)

```
for each possible permutation of the word
    • generate the next permutation
    • test to see if contained in the dictionary
    • if so, add to the list of anagrams
```

efficiency of this approach, where L is word length & D is dictionary size?

```
for each possible permutation of the word
    • generate the next permutation
```
→ O(L), *assuming a smart encoding*
```
    • test to see if contained in the dictionary
```
→ O(D), *assuming sequential search*
```
    • if so, add to the list of anagrams
```
→O(1)

since L! different permutations, will loop L! times

→ $O(L! \times (L + D + 1))$ → $O(L! \times D)$

note: 6! = 720, 7! = 5,040, 8! = 40,320, 9! = 362,880, 10! = 3,628,800, 11! = 39,916,800

9

EXAMPLE: finding all anagrams of a word (approach 2)

```
sort letters of given word
traverse the entire dictionary, word by word
    • sort the next dictionary word
    • test to see if identical to sorted given word
    • if so, add to the list of anagrams
```

efficiency of this approach, where L is word length & D is dictionary size?

```
sort letters of given word
```
→ O(L log L), *assuming an efficient sort*
```
traverse the entire dictionary, word by word
    • sort the next dictionary word
```
→O(L log L), *assuming an efficient sort*
```
    • test to see if identical to sorted given word
```
→ O(L)
```
    • if so, add to the list of anagrams
```
→ O(1)

since dictionary is size D, will loop D times

→ $O(L \log L + (D \times (L \log L + L + 1)))$ → $O(L \log L \times D)$

10

Approach 1 vs. approach 2

clearly, approach 2 will be faster $O(L \log L \times D)$ vs. $O(L! \times D)$

- for a 5-letter word:

  5 log 5 × 117,000 ≈ 12 × 117,000 = 1,404,000

  5! × 117,000 = 120 × 117,000 = 14,040,000

- for a 10-letter word:

  10 log 10 × 117,000 ≈ 33 × 117,000 = 3,861,000

  10! × 117,000 = 3,628,800 × 117,000 = 424,569,600,000

approach 3: instead of sorting the letters in a word, count the number of a's, b's, c's, … and compare with counts from the other word   EFFICIENCY?

11

Analyzing recursive algorithms

recursive algorithms can be analyzed by defining a *recurrence relation*:

cost of searching N items using binary search =
cost of comparing middle element + cost of searching correct half (N/2 items)

more succinctly: Cost(N) = Cost(N/2) + C

Cost(N) = Cost(N/2) + C	can unwind Cost(N/2)
= (Cost(N/4) + C) + C	
= Cost(N/4) + 2C	can unwind Cost(N/4)
= (Cost(N/8) + C) + 2C	
= Cost(N/8) + 3C	can continue unwinding
= …	(a total of $\log_2 N$ times)
= Cost(1) + $(\log_2 N)$*C	
= C $\log_2 N$ + C'	where C' = Cost(1)
➔ O(log N)	

12

Analyzing merge sort

cost of sorting N items using merge sort =
cost of sorting left half (N/2 items) + cost of sorting right half (N/2 items) +
cost of merging (N items)

more succinctly: $Cost(N) = 2Cost(N/2) + C_1N + C_2$

$Cost(N) = 2Cost(N/2) + C_1*N + C_2$	can unwind Cost(N/2)
$= 2(2Cost(N/4) + C_1N/2 + C_2) + C_1N + C_2$	
$= 4Cost(N/4) + 2C_1N + 3C_2$	can unwind Cost(N/4)
$= 4(2Cost(N/8) + C_1N/4 + C_2) + 2C_1N + 3C_2$	
$= 8Cost(N/8) + 3C_1N + 7C_2$	can continue unwinding
= …	(a total of $\log_2 N$ times)
$= NCost(1) + (\log_2 N)C_1N + (N-1) C_2$	
$= C_1N \log_2 N + (C'+C_2)N - C_2$	where C' = Cost(1)

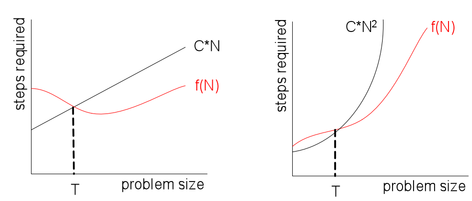
➔ O(N log N)

13

Big-Oh (slightly more formally)

more formally: an algorithm is O(f(N)) if, *after some point*, the # of steps can be bounded from above by a scaled f(N) function

O(N): if number of steps can eventually be bounded by a line
$O(N^2)$: if number of steps can eventually be bounded by a quadratic
…



"after some point" captures the fact that we only care about the long run

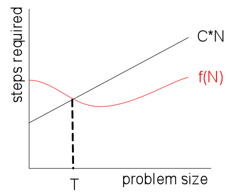
- for small values of N, the constants can make an O(N) algorithm do more work than an $O(N^2)$ algorithm
- but beyond some threshold size, the $O(N^2)$ will always do more work

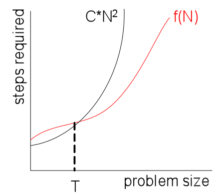
e.g., $f_1(N) = 300N$ & $f_2(N) = ½ N^2$ what threshold forces $f_1(N) \le f_2(N)$?

14

Big-Oh (formally)

an algorithm is O(f(N)) if there exists a positive constant C & non-negative integer T such that for all N ≥ T, # of steps required ≤ C*f(N)





for example, selection sort:

N(N-1)/2 inspections + N-1 swaps = (N^2/2 + N/2 -1) steps

if we consider C = 1 and T = 1, then

$N^2/2 + N/2 - 1$	$\leq N^2/2 + N/2$	*since added 1 to rhs*
	$\leq N^2/2 + N(N/2)$	*since 1 ≤ N at T and beyond*
	$= N^2/2 + N^2/2$	
	$= 1N^2$	➔ O(N^2)

in general, can use C = sum of positive terms, T = 1 (but other constants work too)

15

Exercises

consider an algorithm whose cost function is

Cost(N) = $3N^2 - 12N + 5$

intuitively, we know this is O(N^2)

formally, what are values of C and T that meet the definition?

- an algorithm is O(N^2) if there exists a positive constant C & non-negative integer T such that for all N ≥ T, # of steps required ≤ C*N^2

consider an algorithm whose cost function is

Cost(N) = $12N^3 - 5N^2 + N - 300$

intuitively, we know this is O(N^3)

formally, what are values of C and T that meet the definition?

- an algorithm is O(N^3) if there exists a positive constant C & non-negative integer T such that for all N ≥ T, # of steps required ≤ C*N^3

16

Exercise

consider a merge-3 sort algorithm

1. if the list contains 0 or 1 items, then done
2. otherwise, divide the list into thirds and recursively sort each third
3. then, merge the sorted thirds into a single sorted list

what is the recurrence relation for this algorithm?

closed (polynomial) form?

Big-Oh?

Specialized sorts

for general-purpose, comparable data, O(N log N) is optimal

- i.e., it is proven that there is no sorting algorithm better than O(N log N) for sorting arbitrary lists of elements (using only data comparisons)
- proof later

interestingly, you can do better *in special cases*

- if the range of potential data values is limited → frequency list
- if the data values can be compared lexicographically → radix sort

Frequency lists

suppose there is a fixed, reasonably-sized range of values

- such as years in the range 1900-2006

1975	2002	2006	2002	2005	1999	1950	1903	2006	2001	2006	1975	2003	1900	1980	1900

- construct a frequency array with |range| counters, initialized to 0

2	0	0	1	. . .	1	2	1	0	1	3
1900	1901	1902	1903	. . .	2001	2002	2003	2004	2005	2006

- then traverse and copy the appropriate values back to the list

1900	1900	1903	1950	1975	1975	1980	1999	2001	2002	2002	2003	2005	2006	2006	2006

big-Oh analysis?

19

Radix sort

suppose the values can be compared lexicographically (either character-by-character or digit-by-digit)

radix sort:

1. take the least significant char/digit of each value
2. sort the list based on that char/digit, but keep the order of values with the same char/digit
3. repeat the sort with each more significant char/digit

"ace"	"baa"	"cad"	"bee"	"bad"	"ebb"

most often implemented using a "bucket list"

- here, need one bucket for each possible letter
- copy all of the words ending in "a" in the 1st bucket, "b" in the 2nd bucket, …

"baa"	"ebb"		"cad" "bad"	"ace" "bee"	
"a"	"b"	"c"	"d"	"e"	. . .

20

Radix sort (cont.)

- copy the words from the bucket list back to the list, preserving order
- results in a list with words sorted by last letter

"a"	"b"	"c"	"d"	"e"	. . .
"baa"	"ebb"		"cad"	"ace"	
			"bad"	"bee"	

→

"baa"	"ebb"	"cad"	"bad"	"ace"	"bee"

- repeat, but now place words into buckets based on next-to-last letter
- results in a list with words sorted by last two letters

"a"	"b"	"c"	"d"	"e"	. . .
"baa"	"ebb"	"ace"		"bee"	
"cad"					
"bad"					

→

"baa"	"cad"	"bad"	"ebb"	"ace"	"bee"

- repeat, but now place words into buckets based on first letter
- results in a sorted list

"a"	"b"	"c"	"d"	"e"	. . .
"ace"	"baa"	"cad"		"ebb"	
	"bad"				
	"bee"				

→

"ace"	"baa"	"bad"	"bee"	"cad"	"ebb"

big-Oh analysis?

21

Big-Omega & Big-Theta

Big-Oh represents an asymptotic upper bound on algorithm cost

- but not necessarily a "tight" bound
- if an algorithm is O(N), then it is also $O(N^2)$

$f(N) = 5N - 2 < 5N \leq 5N^2$ (when $N \geq 1$)

to really capture rate of growth, we must prove a tight bound on cost

Big-Omega is an asymptotic lower bound

- an algorithm is $\Omega(f(N))$ if there exists a positive constant C & non-negative integer T such that for all $N \geq T$, # of steps required $\geq C*f(N)$

Big-Theta is a tight asymptotic bound (both lower and upper)

- an algorithm is $\theta(f(N))$ if it is $O(f(N))$ and $\Omega(f(N))$

22

Proving a tight bound

to formally prove rate-of-growth, must show Big-Theta

- $f(N) = N^2 + 5N - 2 \leq N^2 + 5N \leq N^2 + 5N^2$ (when $N \geq 1$) $= 6N^2$ → $O(N^2)$
- $f(N) = N^2 + 5N - 2 > N^2 + 4N$ (when $N \geq 1$) $> 1N^2$ → $\Omega(N^2)$

→ $\theta(N^2)$

as long as we are conservative in proving the upper-bound, the corresponding lower-bound usually follows easily

- so, usually algorithm analysis is stated in terms of Big-Oh (even though Big-Theta is implied)

Alternative definition of Big-Oh

an algorithm is O(f(N)) if $\lim_{N \to \infty} Cost(N)/f(N) < \infty$

EXAMPLE: $Cost(N) = 5N^2 - 3N + 1$

$\lim_{N \to \infty} (5N^2 - 3N + 1)/N^2 = 5 < \infty$ → $O(N^2)$

$\lim_{N \to \infty} (5N^2 - 3N + 1)/N^3 = 0 < \infty$ → $O(N^3)$

$\lim_{N \to \infty} (5N^2 - 3N + 1)/N = \infty$ → not $O(N)$

O vs. Ω

since O represents an upper bound and Ω represents a lower bound, there is an inverse relationship

THEOREM: f(N) is O(g(N)) if and only if g(N) is Ω(f(N)).

PROOF:

f(N) is O(g(N)) ←→ f(N) ≤ Cg(N) for N ≥ T

←→ g(N) ≥ (1/C)f(N) for N ≥ T

←→ g(N) is Ω(f(N))

EXAMPLE: $f(N) = 3N^2 + 2$ $g(N) = N^2$

$f(N) = 3N^2 + 2 \leq 5N^2$ when $N \geq 1$ → $O(N^2)$

$g(N) = N^2 = 1/5\ (5N^2) \geq 1/5\ (3N^2 + 2)$ when $N \geq 1$ → $\Omega(N^2)$

25

A log is a log

mathematically, $x = \log_b y$ ←→ $y = b^x$

e.g., $10 = \log_2 1024$, since $1024 = 2^{10}$

properties of logarithms

$\log_b (nm) = \log_b n + \log_b m$ $\log_b (n/m) = \log_b n - \log_b m$

$\log_b (n^r) = r \log_b n$ $\log_a n = \log_b n / \log_b a$

this last property is why we don't care about the log base for Big-Oh

$f(N)$ is $O(\log_a N)$ ←→ $f(N) <= C \log_a N$ for $N \geq T$

←→ $f(N) <= C \log_a N = C (\log_b N / \log_b a) = (C/\log_b a) \log_b N$ for $N \geq T$

←→ $f(N)$ is $O(\log_b N)$

26

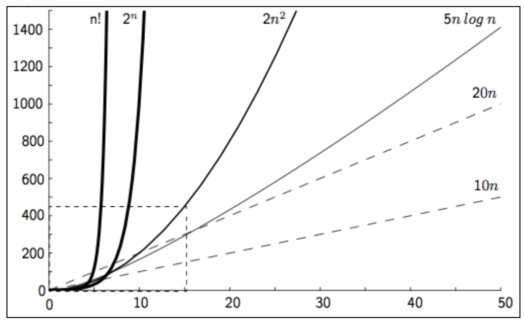
How bad is O(N!)?

recall the first approach to generating anagrams O(L! x D)

Stirling's formula: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\epsilon(n)}$ where $\frac{1}{12n+1} \leq \epsilon(n) \leq \frac{1}{12n}$.

- as n gets large, ε(n) approaches 0, so $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

➔ O(N!) ~ O(N^N)



27