

# CSC 427: Data Structures and Algorithm Analysis

Fall 2011

## Algorithm analysis, searching and sorting

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

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?

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) &  $f_2(N) = \frac{1}{2}*N^2$  (a very gradual quadratic)

in the short run (i.e., for small values of  $N$ ),  $f_1(N) > f_2(N)$

e.g.,  $f_1(10) = 300*10 = 3,000 > 50 = \frac{1}{2}*10^2 = f_2(10)$

in the long run (i.e., for large values of  $N$ ),  $f_1(N) < f_2(N)$

e.g.,  $f_1(1,000) = 300*1,000 = 300,000 < 500,000 = \frac{1}{2}*1,000^2 = f_2(1,000)$

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  $\text{Cost}(N) = 5N - 3$ 
  - $\text{Cost}(S) = 5S - 3$
  - $\text{Cost}(2S) = 5(2S) - 3 = 10S - 3$
- for an  $O(N \log N)$  algorithm, doubling the problem size more than doubles the amount of work  
e.g., suppose  $\text{Cost}(N) = 5N \log N + N$ 
  - $\text{Cost}(S) = 5S \log S + S$
  - $\text{Cost}(2S) = 5(2S) \log(2S) + 2S = 10S(\log(S)+1) + 2S = 10S \log S + 12S$
- for an  $O(N^2)$  algorithm, doubling the problem size quadruples the amount of work  
e.g., suppose  $\text{Cost}(N) = 5N^2 - 3N + 10$ 
  - $\text{Cost}(S) = 5S^2 - 3S + 10$
  - $\text{Cost}(2S) = 5(2S)^2 - 3(2S) + 10 = 5(4S^2) - 6S + 10 = 20S^2 - 6S + 10$

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

$$\begin{aligned}\text{Cost}(N) &= N \text{ inspections} + \text{overhead} \\ &\rightarrow O(N)\end{aligned}$$

**selection sort:** cost of sorting a list of  $N$  items

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

$$\begin{aligned}\text{Cost}(N) &= (1 + 2 + 3 + \dots + N-1) \text{ comparisons} + N-1 \text{ swaps} + \text{overhead} \\ &= N(N-1)/2 \text{ comparisons} + N-1 \text{ swaps} + \text{overhead} \\ &= \frac{1}{2} N^2 - \frac{1}{2} N \text{ comparisons} + N-1 \text{ swaps} + \text{overhead} \\ &\rightarrow O(N^2)\end{aligned}$$

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  $\times$  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  $\times$  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! × (L + D + 1)) → O(L! × D) note: 6! = 720      9! = 362,880  
7! = 5,040      10! = 3,628,800  
8! = 40,320      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 × (L log L + L + 1))) → O(L log L × 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:

$$5 \log 5 \times 117,000 \approx 12 \times 117,000 = 1,404,000$$

$$5! \times 117,000 = 120 \times 117,000 = 14,040,000$$

- for a 10-letter word:

$$10 \log 10 \times 117,000 \approx 33 \times 117,000 = 3,861,000$$

$$10! \times 117,000 = 3,628,800 \times 117,000 = 424,569,600,000$$

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:  $\text{Cost}(N) = \text{Cost}(N/2) + C$

$$\text{Cost}(N) = \text{Cost}(N/2) + C$$

can unwind  $\text{Cost}(N/2)$

$$= (\text{Cost}(N/4) + C) + C$$

can unwind  $\text{Cost}(N/4)$

$$= \text{Cost}(N/4) + 2C$$

$$= (\text{Cost}(N/8) + C) + 2C$$

$$= \text{Cost}(N/8) + 3C$$

$$= \dots$$

can continue unwinding  
(a total of  $\log_2 N$  times)

$$= \text{Cost}(1) + (\log_2 N) \cdot C$$

$$= C \log_2 N + C'$$

where  $C' = \text{Cost}(1)$

$$\rightarrow O(\log N)$$

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:  $\text{Cost}(N) = 2\text{Cost}(N/2) + C_1N + C_2$

$$\begin{aligned}
 \text{Cost}(N) &= 2\text{Cost}(N/2) + C_1N + C_2 && \text{can unwind Cost}(N/2) \\
 &= 2(2\text{Cost}(N/4) + C_1N/2 + C_2) + C_1N + C_2 \\
 &= 4\text{Cost}(N/4) + 2C_1N + 3C_2 && \text{can unwind Cost}(N/4) \\
 &= 4(2\text{Cost}(N/8) + C_1N/4 + C_2) + 2C_1N + 3C_2 \\
 &= 8\text{Cost}(N/8) + 3C_1N + 7C_2 && \text{can continue unwinding} \\
 &= \dots && \text{(a total of } \log_2 N \text{ times)} \\
 &= N\text{Cost}(1) + (\log_2 N)C_1N + (N-1)C_2 \\
 &= C_1N \log_2 N + (C_1 + C_2)N - C_2 && \text{where } C' = \text{Cost}(1) \\
 &\rightarrow O(N \log N)
 \end{aligned}$$

13

## Recurrence relations (in general)

if  $\text{Cost}(N) = a\text{Cost}(N/b) + O(1)$

$$\begin{array}{lll}
 a = 1 & \rightarrow & O(\log N) \\
 a < b & \rightarrow & O(\log_a b \sqrt{N}) \\
 a = b & \rightarrow & O(N) \\
 a > b & \rightarrow & O(N^{\log_b a})
 \end{array}$$

if  $\text{Cost}(N) = a\text{Cost}(N/b) + O(N)$

$$\begin{array}{lll}
 a < b & \rightarrow & O(N) \\
 a = b & \rightarrow & O(N \log N) \\
 a > b & \rightarrow & O(N^{\log_b a})
 \end{array}$$

14

## Iteration vs. Recursion

consider the task of calculating  $a^b$

could do it iteratively using a loop

$$a^b = \underbrace{a * a * a * \dots * a}_{b \text{ times}}$$

could do it recursively

$$\begin{aligned} a^0 &= 1 \\ a^b &= a^{b/2} * a^{b/2} && \text{if } b \text{ is even} \\ a^b &= a * a^{b/2} * a^{b/2} && \text{if } b \text{ is odd} \end{aligned}$$

efficiency tradeoff?

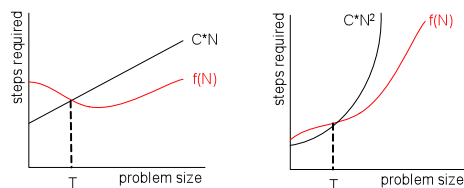
15

## 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.,  $f_1(N) = 300N$  &  $f_2(N) = \frac{1}{2}N^2$

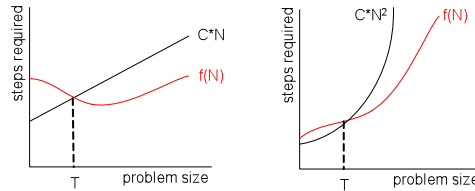
what threshold forces  $f_1(N) \leq f_2(N)$  ?

16



## Big-Oh (formally)

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



for example, selection sort:

$$N(N-1)/2 \text{ inspections} + N-1 \text{ swaps} = (N^2/2 + N/2 - 1) \text{ steps}$$

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

$$\begin{aligned} N^2/2 + N/2 - 1 &\leq N^2/2 + N/2 && \text{since added 1 to rhs} \\ &\leq N^2/2 + N(N/2) && \text{since } 1 \leq N \text{ at } T \text{ and beyond} \\ &= N^2/2 + N^2/2 \\ &= 1N^2 && \rightarrow O(N^2) \end{aligned}$$

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

17

## Exercises

consider an algorithm whose cost function is

$$\text{Cost}(N) = 12N^3 - 5N^2 + N - 300$$

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  $N \geq T$ , # of steps required  $\leq C \cdot N^3$

consider "merge3-sort"

- If the range to be sorted is size 0 or 1, then DONE.
- Otherwise, calculate the indices  $1/3$  and  $2/3$  of the way through the list.
- Recursively sort each third of the list.
- Merge the three sorted sublists together.

what is the recurrence relation that defines the cost of this algorithm?

what is its big-Oh classification?

18

## Specialized sorts

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

- for special cases, you can actually do better

e.g., 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  $|\text{range}|$  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"

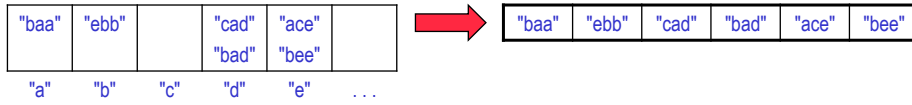
- here, need one bucket for each possible letter
- copy all of the words ending in "a" in the 1st bucket, "b" in the 2<sup>nd</sup> bucket, ...

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

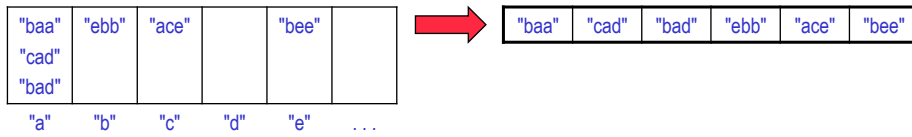
20

## Radix sort (cont.)

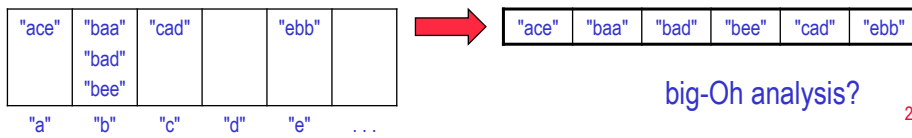
- copy the words from the bucket list back to the list, preserving order
- results in a list with words sorted by last letter



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



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



big-Oh analysis?