CSC 427: Data Structures and Algorithm Analysis

Fall 2006

Algorithm analysis, searching and sorting

- big-Oh analysis (more formally)
- analyzing searches & sorts
- recurrence relations
- 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

Big-Oh revisited

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., $f_1(1,000) = 300*1,000 = 300,000 < 500,000 = ½*1,000^2 = f_2(1,000)$

3

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 $Cost(N) = 5N^2 – 3N + 10$
  - $Cost(S) = 5S^2 – 3S + 10$
  - $Cost(2S) = 5(2S)^2 – 3(2S) + 10 = 5(4S^2) – 6S + 10 = 20S^2 – 6S + 10$

4

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
→ $O(N^2)$

5

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
= …	
= Cost(1) + ($\log_2$N)*C	
= C $\log_2$ N + C'	where C' = Cost(1)
→ O(log N)	

6

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: $Cost(N) = 2*Cost(N/2) + C_1*N + C_2$

$Cost(N) = 2*Cost(N/2) + C_1*N + C_2$ can unwind Cost(N/2)
$= 2*(2*Cost(N/4) + C_1*N/2 + C_2) + C_1*N + C_2$
$= 4*Cost(N/4) + 2C_1*N + 3C_2$ can unwind Cost(N/4)
$= 4*(2*Cost(N/8) + C_1*N/4 + C_2) + 2C_1*N + 3C_2$
$= 8*Cost(N/8) + 3C_1*N + 7C_2$ can continue unwinding
= …
$= N*Cost(1) + (\log_2 N)*C_1*N + (N-1) C_2$

$= C_1*N \log_2 N + (C'+C_2)*N - C_2$ where C' = Cost(1)

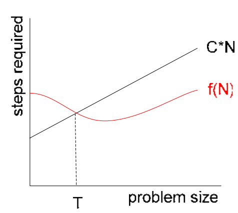
➔ O(N log N)

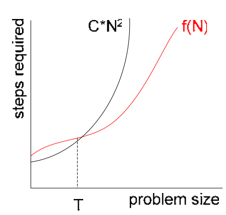
7

Big-Oh revisited

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

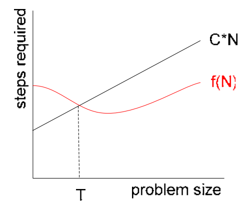
- for small values of N, the constants can make an O(N) algorithm do more work than an $O(N^2)$ algorithm
- but beyond some threshold size, the $O(N^2)$ will always do more work

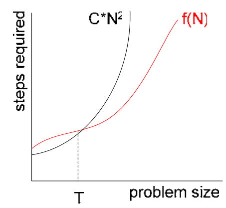
e.g., $f_1(N) = 300*N$ & $f_2(N) = ½ N^2$ what threshold forces $f_1(N) \leq f_2(N)$?

8

Technically speaking…

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 + overhead = $(N^2/2 + N/2 -1 + X)$ steps

if we consider C = 2 and T = max(X, 1), then

$(N^2/2 + N/2 - 1 + X) \leq (N^2/2 + N^2/2 + N^2) = 2N^2$ ➔ $O(N^2)$

9

Exercises

consider an algorithm whose cost function is

$Cost(N) = 12N^3 – 5N^2 + N – 300$

intutitively, 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 ≤ $C*N^3$

consider "merge3-sort"

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

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

what is it's big-Oh classification?

10

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 |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?

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"

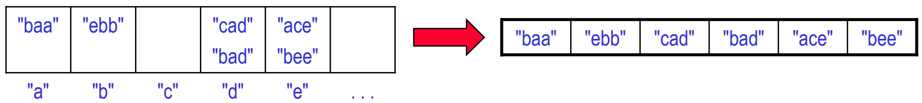
most often implemented using a "bucket list"

- 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, …

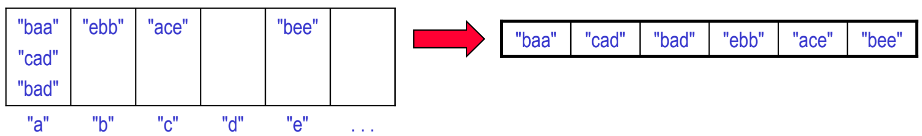
"baa"	"ebb"		"cad" "bad"	"ace" "bee"	
"a"	"b"	"c"	"d"	"e"	. . .

Radix sort (cont.)

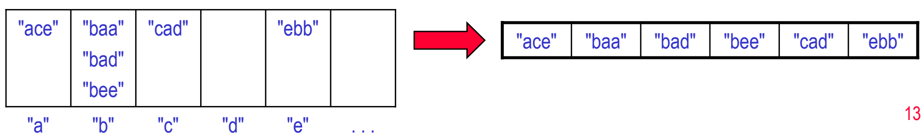
- copy the words from the bucket list back to the list, preserving order
- results in a list with words sorted by last letter



- 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



HW2

big-Oh analysis of radix sort?

for HW2:

- implement generic `BucketList<E>` class
- augment the provided code to allow for variable-length strings
- time executions to verify big-Oh performance

```
public class Sorts {
    public static final int NUM_LETTERS = 26;
    public static final int STR_LENGTH = 3;

    public static void radixSort(ArrayList<String> list) {
        BucketList<String> buckets =
            new BucketList<String>(NUM_LETTERS+1);

        ArrayList<String> tempList = list;
        for (int i = STR_LENGTH; i > 0; i--) {
            for (String str : tempList) {
                buckets.add(str.charAt(i-1)-'a', str);
            }
            tempList = buckets.asList();
            buckets.clear();
        }

        for (int i = 0; i < list.size(); i++) {
            list.set(i, tempList.get(i));
        }
    }
}
```

create bucket list

<u>repeatedly:</u> copy to buckets

copy back to list & clear buckets

place sorted words back in list

QUESTION: why do we need `tempList`?