CS472 Module 4 Part C - Divide and Conquer - Quicksort

Athens State University

February 5, 2016

Outline

Contents

1 A review of Partitioning and the Selection Problem

The Selection Problem

- Find the k-th smallest element in a list of n numbers
- Find the *median* $k = \lceil n/2 \rceil$ in a list of n numbers

The Selection Problem: Partitioning

Consider how one might *partition* an array A[0..(n-1)]

	s	
all are $\leq$ A[s]	A[s]	all are $\geq$ A[s]

Repeat until $s = k - 1$:

- If $s = k - 1$ then we have a solution.
- If $s > k - 1$ then look for the k-th smallest element to the left.
- If $s < k - 1$ then look for the $k - s$-th smallest element to the right

The Selection Problem: How do we partition?

One approach: Lomuto's Partitioning Algorithm

- Scan the array left to right, keeping the array in three sections
  - A set less than some value p, where p is the *pivot* of the partition. Put the pivot at the start of the array

  - A set greater than or equal to p
  - A set of values where we are uncertain if they are less than or greater than or equal to the pivot

- On each iteration, decrease the size of the unknown section by one element until it is empty. Achieve a partition by exchanging the pivot with the element in the split position s

The Selection Problem: Quickselect

Algorithm 1: Hoare's quickselect algorithm

```
Input: A list, an index left, an index right, and list length n
Output: The n-th smallest element within left and right range
if the list contains only one element then
    return that element;
Select a pivot between left and right;
pivot ← partition (list, left, right, pivot);
if pivot = n then
    return list;
else if n < pivot then
    return quickselect (list,list, pivot-1, n);
else
    return quickselect (list, pivotIndex + 1, right, n);
```

2 Quicksort: Fun with partitioning

Quicksort: Summary

- Let's pivot an array on the array's first element

p	A[i]<=p	A[i]>=p

- Now exchange the pivot with the last element in the first partition
  - We have placed the pivot in its correct spot
- Sort the two partitions recursively

Quicksort: A better partitioning algorithm

Algorithm 2: Hoare's Partitioning Algorithm

```
Input: A subarray A[l,r] of a larger array of n-1 elements
Output: A partition of A[l,r] with a return value of s, the split location
p ← A[l];
i ← l;
j ← r + 1;
repeat
    repeat
        Increment i;
    until A[i] ≥ p;
    repeat
        Decrement j;
    until A[j]<p;
    swap (A[i],A[j]);
until i ≥ j;
swap (A[i],A[j]);
swap (A[l],A[j]);
return j;
```

Quicksort: Analysis

- Best case: Split in the middle - $\Theta(n * lg(n))$
- Worst case: Sorted Array! - $\Theta(n^2)$
- Average case: random arrays - $\Theta(n * lg(n))$
- Considered the method of choice for internal sorting of large files ($n \geq 1000$)
- Implementation provided in most class libraries

Quicksort: Analysis

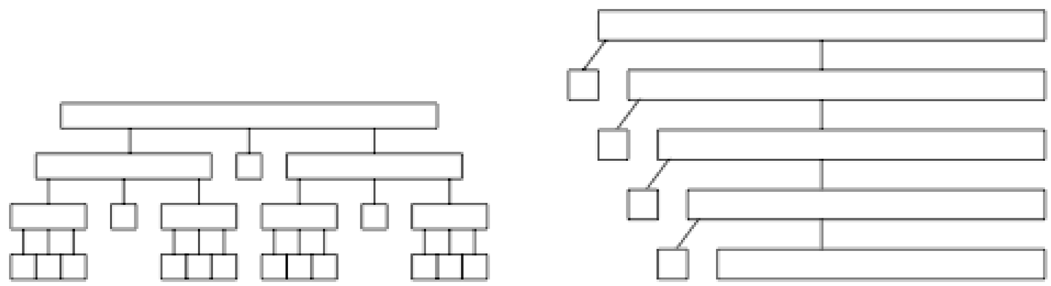

Figure 4.6: The best-case (l) and worst-case (r) recursion trees for quicksort

3 Can we make quicksort quicker?

Can we make quicksort quicker?

- Notice our claim about the average case:

  Quicksort runs $\Theta(n * lg(n))$ *time, with high probability, if you give me randomly ordered data to start.*
- So, suppose we randomly permute the order of the elements of the array before we sort said array?
  - Permutation can be done in $O(n)$ time
  - Provides a level of guarantee that algorithm will run in $\Theta(n * lg(n))$ time!
- Can get similar result if we randomly pick an element as the pivot on each step of the algorithm.

Can we make quicksort quicker?

A common interview question deals with the implications of choosing n_0 in the definition of the "Big-Oh" function. The correct response to this question gives a way to get faster performance out of quicksort

- For some small values of n, a sort with quadratic run-time like insertion sort will run better than a log-linear sort like quicksort
- So, do a combination... when the recursion process makes n small enough, sort the remainder of the array using insertion sort

Can we make quicksort quicker?

- Our presentation of quicksort uses recursion
- Just like with quickselect, we can get rid of this tail recursion and rewrite the sort as an iterative algorithm

Just how much faster?

- Recall the definition of $O(f(n))$. We say a function $g(n)$ is $O(f(n))$ if $g(n) \leq c * f(n)$ for all values of $n \geq n_0$
- Our three tricks for making quicksort faster is just playing with the value of the constant c.
- But, our little tricks improve that value by about 20-25 percent.

And an observation about comparing sorts using $O(g(n))$

- One can obviously see the difference between sorts that are $O(n^2)$ and $O(n * lg(n))$
- But what about comparing the performance of sort algorithms that are all $O(n * lg(n))$?
- Again, you are playing with the value of c in the definition of $O(g(n))$
  - Experimentation may be required to make a decision!

4 Quicksort as example of divide-and-conquer

Quicksort as example of divide-and-conquer

- Quicksort is an example of a divide-and-conquer algorithm
- The act of partitioning divides the problem into smaller problems, to which we apply the quicksort algorithm
- Finally, we combine the answers to the smaller problem to get a solution to the larger problem

5 Key Points

Key Points

- What is partitioning of an array
- The quicksort algorithm
- Analysis of the quicksort algorithm
- How can we make quicksort quicker?
  - And how does that effort tie back into rate-of-growth functions?