Data Structures and Algorithms

Spring 2009-2010

Outline

1. Sorting Algorithms (contd.)
   - Quicksort (contd.)
   - QuickSelect
   - Lower Bounds on Sorting
   - $o(n \log n)$-Sorting ??

Outline

Analysis of Quicksort

- Time to sort array of length 0, 1 is 1: $T(0) = T(1) = 1$
- $T(n) = T(i) + T(n - i - 1) + cn$ where $i = |S_1|$
- Three cases to consider: worst case, best case and average case

Best-case analysis

Pivot will split S exactly in half

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= cn \log n + n = O(n \log n) \end{aligned}$$

Analysis of Quicksort (contd.)

Worst-case analysis

Pivot is smallest every time: $i = 0$

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
T(n-1) &= T(n-2) + c(n-1) \\
&\vdots \\
T(2) &= T(1) + 2c \\
T(n) &= T(1) + c\sum_{i=2}^{n} i = O(n^2)
\end{aligned}$$

Analysis of Quicksort (contd.)

Average-case analysis

Let $T(n)$ be average time to sort n values:

- Assume that every element has equal likelihood of being the pivot element
- This is only valid if every sub-array is random
- If the pivot element is jth largest ($1 \leq j \leq n$), which occurs with prob. $1/n$, and cn is work done partitioning the set, then

$$\begin{aligned} T(n) &= cn + \frac{1}{n}\sum_{j=1}^{n}(T(j-1) + T(n-j)) \\ &= cn + \frac{1}{n}\sum_{j=0}^{n-1}(T(j) + T(n-j-1)) \end{aligned}$$

Analysis of Quicksort (contd.)

Therefore,

$$\begin{aligned} T(n) &= \frac{2}{n}\left[\sum_{j=0}^{n-1} T(j)\right] + cn \\ nT(n) &= 2\left[\sum_{j=0}^{n-1} T(j)\right] + cn^2 \end{aligned}$$

and

$$(n-1)T(n-1) = 2\left[\sum_{j=0}^{n-2} T(j)\right] + c(n-1)^2$$

By subtracting the previous two equations, we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

Analysis of Quicksort (contd.)

By dropping $-c$ term to simplify the expression and by dividing by $n(n+1)$ we have a sum that can "telescope"

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2c}{n+1} \\
\frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1} + \frac{2c}{n} \\
&\vdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3}
\end{aligned}$$

Analysis of Quicksort (contd.)

Thus, by adding all terms

$$\begin{aligned} \frac{T(n)}{n+1} &= T(1) + 2c\sum_{i=3}^{n+1}\frac{1}{i} \\ &\approx 2c\log_e(n+1) \\ &\approx \frac{2c}{\log e}\log n \end{aligned}$$

And so,

$$T(n) = O(n \log n)$$

Thus `quicksort()` runs in $O(n \log n)$ *average* time.

Outline

1. Sorting Algorithms (contd.)
   - Quicksort (contd.)
   - QuickSelect
   - Lower Bounds on Sorting
   - $o(n \log n)$-Sorting ??

QuickSelect: Finding the kth Largest Element

- Using priority queues we can find the kth largest element in time $O(n + k \log n)$
- ✗ So finding the *median* element will take us (worst-case) $O(n + \frac{n}{2} \log n) = O(n \log n)$
- In each call to `quicksort()` the set is partitioned about pivot element, whose final resting place we know
- If we're looking for the kth largest (smallest) element, from the position of the pivot, since we know their sizes, we know which set, S_1 or S_2, we should be looking for it in
- ✓ So we only need to make *one* recursive call at each step
- ✓ From our analysis of "General Solutions to Divide and Conquer" when $1 = a < c = 2$ the best-case running time for the algorithm must be $O(n)$
- ✓ We can show also that the average-case running-time is $O(n)$

Outline

Best Possible Performance

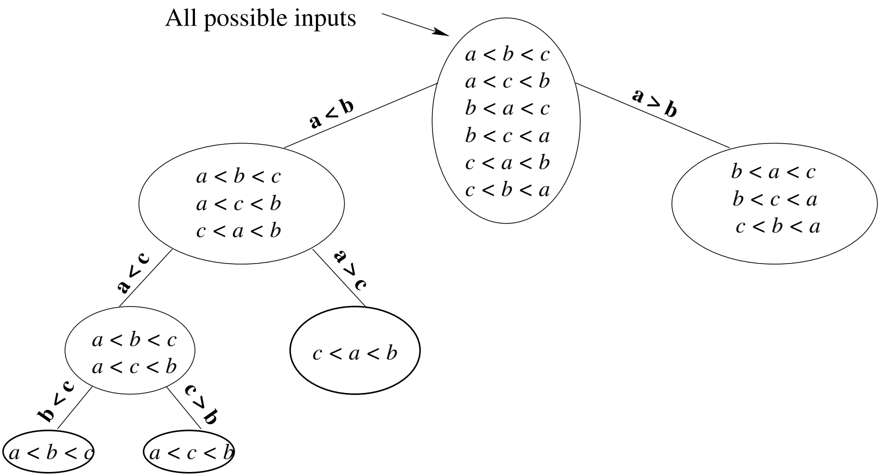
- Is $O(n \log n)$-time as good as we can do from a sorting algorithm?
- Yes, provided we are limited to two-way comparisons of elements in unit time
- In these cases we show that in the worst case: any algorithm that uses only two-way comparisons requires $\lceil \log n! \rceil$ comparisons in the worst case and $\log n!$ comparisons in the average case

The Tree of All Orderings

Idea:

- Given n items, correct ordering is just one of $n!$ possible orderings of items
- We can think of all the possible orderings as being the leaves of a tree
- The "branches" of the internal nodes are decision points *i.e.,* "if $a < b$ branch left, o/w branch right"

The Tree of All Orderings (contd.)



The Tree of All Orderings (contd.)

- Easy to show by induction that a binary tree of depth d has at most 2^d leaves
- Therefore, a binary tree with L leaves will have depth at least $\lceil \log L \rceil$
- Now, since any decision tree on n elements must have $n!$ leaves – a unique leaf for each possible ordering – any sorting algorithm using only 2-way comparisons between elements needs at least $\lceil \log L \rceil$ comparisons in worst case

$$
\begin{aligned}
\log n! &= \log(n \times n-1 \times \cdots 1) \\
&\geq \log n + \log(n-1) + \cdots + \log \frac{n}{2} \\
&\geq \frac{n}{2} \log \frac{n}{2}, \qquad \text{and, since } \log \frac{a}{b} = \log a - \log b, \\
&\geq \frac{n}{2} \log n - \frac{n}{2} = \Omega(n \log n)
\end{aligned}
$$

Outline

Bucket Sort Revisited

- We saw *radix* sort could sort data into buckets in time $O(p(n+b))$, p is number of passes and $p = f(b)$
- This is *linear* in n
- However, when we decide what bucket to put an element in to, we are doing a b-way comparison, hence this sort does not fit our previous model