Algorithms

Linear-Time Sorting Continued

Medians and Order Statistics

Review: Comparison Sorts

- Comparison sorts: O(n lg n) at best
  - Model sort with decision tree
  - Path down tree = execution trace of algorithm
  - Leaves of tree = possible permutations of input
  - Tree must have n! leaves, so O(n lg n) height

Review: Counting Sort

- Counting sort:
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element i
    - Use that number to place i in position k of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place:
    - O(n) array to hold sorted output
    - O(k) array for scratch storage

Review: Counting Sort

```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Review: Radix Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Review: Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time
- *How many bits in a computer word?*

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical $O(n \lg n)$ comparison sort
  - Requires approx $\lg n = 20$ operations per number being sorted
- *So why would we ever use anything but radix sort?*

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Summary: Radix Sort

- Radix sort:
  - Assumption: input has d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast! Stable! Simple!
  - Doesn't sort in place

Bucket Sort

- Bucket sort
  - Assumption: input is *n* reals from [0, 1)
  - Basic idea:
    - Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when we study *hash tables*

Order Statistics

- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the nth order statistic
- The *median* is the $n/2$ order statistic
  - If n is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes:
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
  - Total cost: 3 comparisons per 2 elements = O(3n/2)

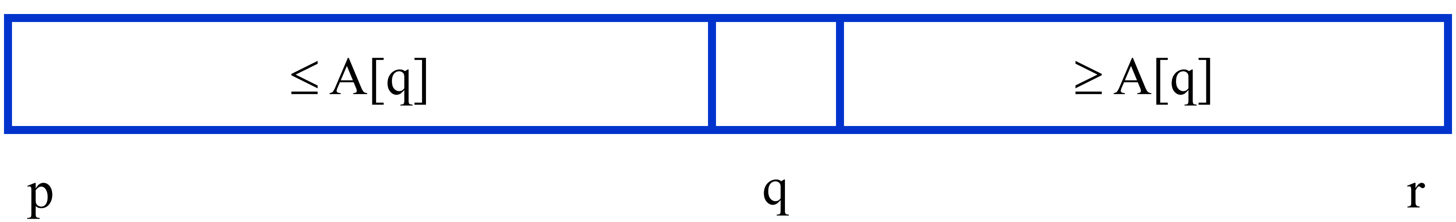
Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

Randomized Selection

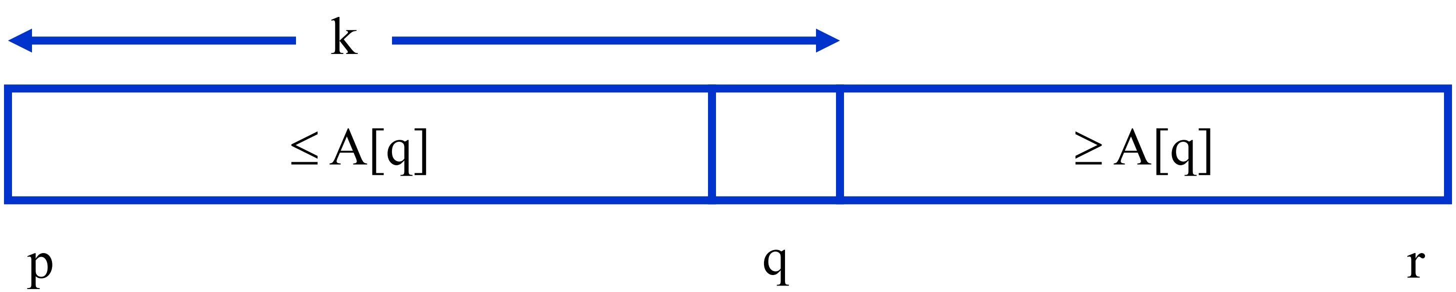
- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a slightly different partition than the book:

  q = RandomizedPartition(A, p, r)



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];    // not in book
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Randomized Selection

- Analyzing **`RandomizedSelect()`**
  - Worst case: partition always 0:n-1

    $T(n) = T(n-1) + O(n)$ = ???

    $= O(n^2)$ (arithmetic series)

    - No better than sorting!
  - "Best" case: suppose a 9:1 partition

    $T(n) = T(9n/10) + O(n)$ = ???

    $= O(n)$ (Master Theorem, case 3)

    - Better than sorting!
    - *What if this had been a 99:1 split?*

Randomized Selection

- Average case
  - For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

What happened here?

  - Let's show that T(n) = O(n) by substitution

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$ *The recurrence we started with*

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$ *Substitute T(n) ≤ cn for T(k)*

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$ *"Split" the recurrence*

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$ *Expand arithmetic series*

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ *Multiply it out*

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ *The recurrence so far*

$$= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$ *Multiply it out*

$$= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$$ *Subtract c/2*

$$= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right)$$ *Rearrange the arithmetic*

$$\le cn \quad \text{(if c is big enough)}$$ *What we set out to prove*

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element x

Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide n elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median x of the $\lfloor n/5 \rfloor$ medians
  4. Partition the n elements around x. Let k = rank(x)
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find ith smallest element in first partition

     else (i > k) use Select() recursively to find (i-k)th smallest element in last partition

Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are $\leq x$?*
  - At least 1/2 of the medians = $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are $\leq x$?*
  - At least $3 \lfloor n/10 \rfloor$ elements
- For large n, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements $\leq x$
- Similarly: at least $n/4$ elements $\geq x$

Worst-Case Linear-Time Selection

- Thus after partitioning around x, step 5 will call Select() on at most $3n/4$ elements
- The recurrence is therefore:

$$
\begin{aligned}
T(n) &\leq T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) \\
&\leq T(n/5) + T(3n/4) + \Theta(n) && \lfloor n/5 \rfloor \leq n/5 \\
&\leq cn/5 + 3cn/4 + \Theta(n) && \textit{Substitute } T(n) = cn \\
&= 19cn/20 + \Theta(n) && \textit{Combine fractions} \\
&= cn - (cn/20 - \Theta(n)) && \textit{Express in desired form} \\
&\leq cn \quad \text{if } c \text{ is big enough} && \textit{What we set out to prove}
\end{aligned}
$$

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

Linear-Time Median Selection

- Given a “black box” O(n) median algorithm, what can we do?
  - ith order statistic:
    - Find median x
    - Partition input around x
    - if ($i \leq (n+1)/2$) recursively find ith element of first half
    - else find ($i - (n+1)/2$)th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$
  - *Can you think of an application to sorting?*

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - T(n) = 2T(n/2) + O(n) = O(n lg n)