Medians and Order Statistics Ch. 9

Let A be a set containing *n* distinct unordered elements:

Definition: The ith order statistic is the ith smallest element, e.g.,

- minimum = 1st order statistic
- maximum = nth order statistic
- median(s) = $\lfloor(n+1)/2\rfloor$ and $\lceil(n+1)/2\rceil$

Selection Problem: Find the ith order statistic for a given i
input: Set *A* of *n* (***distinct***) numbers, and a number *i*, $1 \le i \le n$
output: The element $x \in A$ that is larger than exactly (i - 1) elements of *A*

Medians and Order Statistics

Given a set of "n" numbers we can say that,

- Mean: Average of the "n" numbers
- Median: Having sorted the "n" numbers, the value which lies in the middle of the list such that half the numbers are higher than it and half the numbers are lower than it.

The problem of finding the median can be generalized to finding the kth smallest number where k = n/2.

Medians and Order Statistics

kth smallest number can be found using:

- Scan Approach with a time complexity T(n) = kn
- Sort Approach with a time complexity T(n) = nlogn

O(nlgn) solution to selection problem

Selection Problem: Find the ith order statistic for a given i
input: Set *A* of *n* (distinct) numbers, and a number *i*, $1 \le i \le n$
output: The element $x \in A$ that is larger than exactly (i - 1) elements of *A*

NaiveSelection(*A, i*)
1. *A'* = FavoriteCBSort(*A*)
2. **return** *A'[i]*

Running Time:
*O(n*lg*n)* for comparison-based sorting.
Can we do better???

Idea: Use an O(nlgn) comparison-based sorting algorithm, such as heapsort or mergesort. Then return the ith element in the sorted array.

Any ideas for an algorithm to find the minimum?

Finding Minimum (or Maximum)

Running Time:
- just scan input array
- exactly n-1 comparisons

Minimum(A)
1. *lowest* = A[1]
2. **for** *i* = 2 to *n*
3. *lowest* = min(*lowest*, A[*i*])

Is this the best possible time for finding the minimum?

Yes!

Why are n - 1 comparisons necessary?

- Any algorithm that finds the minimum must compare all elements with the "leader" (think of a tournament).
- so...there must be at least n – 1 losers (and each loss requires a comparison)
- We must look at every key, otherwise the missed one may be the minimum. Each look (except the first) requires a comparison.

Finding Minimum & Maximum

What if we want to find *both* the minimum and maximum elements in a set?

How many comparisons are necessary?

- Plan A: find the minimum and maximum separately using n – 1 comparisons for min and n – 2 for max = 2n – 3 comparisons
  Is it possible to do better? Yes!
- Plan B: Process elements in pairs. Compare pairs of elements from the input first with each other and then compare the smaller to the current min and the larger to the current max, changing current values of max and/or min if necessary.
  Cost = at most 3 compares for every 2 elements.
  Total cost = $3\lfloor n/2 \rfloor$.

FindMin&Max(A)
if length[A] % 2 == 0
    if A[1] > A[2]
        min = A[2]
        max = A[1]
    else
        min = A[1]
        max = A[2]
else // n % 2 == 1
    min=max=A[1]

Compare the rest of the elements in pairs, comparing only the maximum element of each pair with max and the minimum element of each pair with min

- If n is even, there is 1 initial compare and then 3(n-2)/2 compares = 3n/2 – 2
- If n is odd, there are 3(n-1)/2 compares
- In either case, the maximum number of compares is ≤ $3\lfloor n/2 \rfloor$

Selection of ith-order statistic in (Expected) Linear Time

- Randomized-Partition first swaps A[r] with a random element of A and then proceeds as in Partition.

Randomized-Partition(A, p, r)
1. j ← Random(p, r)
2. swap A[r] ↔ A[j]
3. **return** Partition(A, p, r)

Selection of ith-order statistic in (Expected) Linear Time

- Randomized-Select returns the ith smallest element of A.
  - like Randomized-QuickSort, except we only need to make one of the recursive calls. Why?

Randomized-Partition(A, p, r)
1. j ← Random(p, r)
2. swap A[r] ↔ A[j]
3. **return** Partition(A, p, r)

Randomized-Select(A, p, r, i)
1. **if** p == r **return** A[p]
2. q = Randomized-Partition(A, p, r)
3. k = q – p + 1
4. **if** i == k **return** A[q]
5. **else if** i < k **return** Randomized-Select(A, p, q-1, i) \\ lower half
6. **else return** Randomized-Select(A, q+1, r, i - k) \\ upper half

Running Time of Randomized-Select

- Worst-case : unlucky with bad 0 : n - 1 partitions.
  $T(n) = T(n - 1) + \theta(n) = \theta(n^2)$
  (same as for worst-case of QuickSort)
- Best-case : really lucky and quickly reduce subarrays
  $T(n) = T(n/2) + \theta(n)$ (what is running time if we use the Master Theorem?)
- Average-case : like Quick-Sort, will be asymptotically close to best-case.

Selection in worst-case linear time

Key: Guarantee a "good" split when array is partitioned - will yield an algorithm that always runs in linear time.

Select(A, i)   /* i is the ith order statistic. */
1. divide input array A into groups of size 5 per group
2. sort the groups of 5, picking the middle elements of each group of 5 and putting each middle element into an array A'.
3. call Select(A', i) to find m, the median of the $\lceil n/5 \rceil$ medians.
4. partition A around m, splitting it into two arrays A[p, q-1] and A[q+1, r] and returning q, the index of the split point
5. if (i == q) return m
   else if (i < q) call Select on the part of A < q
   else if (i > q) call Select on the part of A > q

Selection in Linear Worst-Case Time

Modified version of Partition in line 4 of Select that takes as an extra input parameter, the value of the element to partition around, m.

Partition(A, p, r, m)
1. i = p - 1
2. for j = p to r – 1
3.   if A[j] ≤ m
4.   i = i + 1
5.   swap A[i] and A[j]
6. swap A[i+1] and A[m]
7. return i + 1

Selection in Linear Worst-Case Time

Main idea: this algorithm guarantees that Partition causes a "good" split, with at least a constant fraction of the n elements <= x and a constant fraction > x.

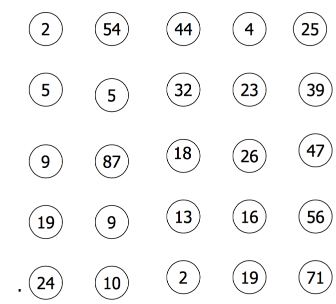
Start the analysis by getting a lower bound on the number of elements that are greater or less than x, the median of medians.

Example:

(2,5,9,19,24,54,5,87,9,10,44,32,18,13,2,4,23, 26,16,19,25,39,47,56,71) is a set of “n” numbers

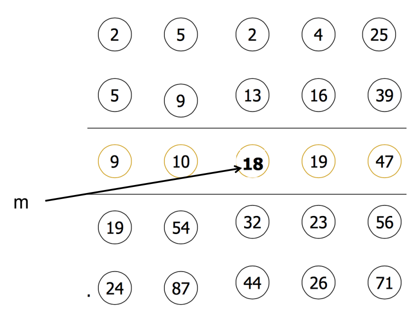
Selection in Linear Worst-Case Time

Step 1: Group numbers in sets of 5 (shown vertically)



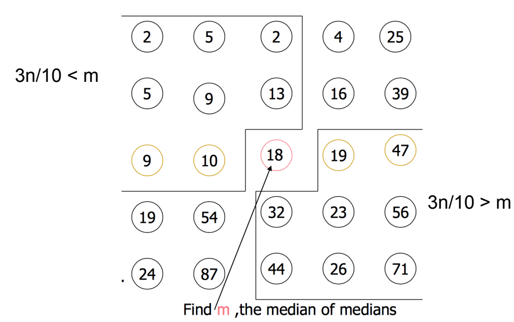
Selection in Linear Worst-Case Time

Step 2: Find Median of each group



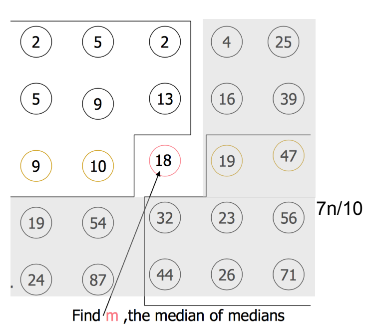
Selection in Linear Worst-Case Time

Step 3: Partition around m



Selection in Linear Worst-Case Time

Step 4: Call Select(A, i) recursively



Selection in Linear Worst-Case Time

Main idea: this algorithm guarantees that Partition causes a "good" split, with at least a constant fraction of the n elements <= m and a constant fraction > m.

Start the analysis by getting a lower bound on the number of elements that are greater than m, the median of medians.

Note:
- At least 1/2 of the medians found in step 2 are greater than the median of medians, m.
- Look at the groups containing medians greater than m. Each contributes 3 elements that are > m (the median of the group and the 2 elements in the group greater than the group's median), except for 2 of the groups: the group containing m (which has only 2 elements > m) and the group with < 5 elements.

Selection in Linear Worst-Case Time

- Thus, we know that at least

$$3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

elements are > m (Symmetrically, the number of elements that are < m is at least 3n/10 - 6).

Therefore, when we call Select recursively in step 5, it is on at most (7n/10) + 6 elements. Find this value by using

$$10n/10 - (3n/10 - 6) = (7n/10) + 6$$

Running Time of Select

Running Time (each step):

1. O(n) (break into groups of 5)
2. O(n) (sorting 5 numbers and finding median is O(1) time)
3. $T(\lceil n/5 \rceil)$ (recursive call to find median of medians)
4. O(n) (partition is linear time)
5. T(7n/10 + 6) (maximum size of subproblem)

Therefore, we get the recurrence

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

Running Time of Select

Solve this recurrence using a good guess. Guess $T(n) \leq cn$

$$\begin{aligned} T(n) &= T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \\ &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\ &\leq c((n/5) + 1) + 7cn/10 + 6c + O(n) \\ &= cn - (cn/10 - 7c) + O(n) \\ &\leq cn \end{aligned}$$

When n >= 80 (cn/10 -7c) is positive

Choosing big enough c makes O(n) + (cn/10 -7c) positive, so last line holds. (Try c = 200)