Week 6 Lab - with solutions

Collaboration level 0 (no restrictions). Open notes.

Topics: asymptotic analysis, comparison-based sorting, sorting lower bound, linear-time sorting, heaps, selection.

As you solve each problem below, write down your solution and ask for feedback on the writing. One of the skills you'll develop in this class is how to communicate technical material clearly. Your solutions need to be clear and easily human-readable.

1. Let A be a list of n (not necessarily distinct) integers. Describe an $O(n)$-algorithm to test whether any item occurs more than $\lceil n/4 \rceil$ times in A.

   We expect: (1) pseudocode and an English description of your algorithm; (2) why is it correct; (3) analysis of its running time.

   Solution: If an element occurs more than $\lceil n/2 \rceil$ times in a A then it must be the median of A. However, the reverse is not true, so once the median is found, you must check to see how many times it occurs in A. The algorithm takes $O(n)$ time provided you use linear selection; it uses no additional space, except $\Theta(1)$ for some variables.

   TEST(A, n)

   1 Use linear selection SELECT(n/2) to find the median m of A.

   2 Do one more pass through A and count the number of occurences of m.

   - if m occurs more than $\lceil n/2 \rceil$ times then return YES;
   - otherwise return NO.

2. (C-4.22) Let A and B be two sequences of n integers each. Given an integer x, describe an $O(n \lg n)$ algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

   We expect: (1) pseudocode and an English description of your algorithm; (2) analysis of its running time.

Answer:

(a) Algorithm 1:

- for i=0 to n-1: search in A for an element of value k-A[i]

This is correct because for every element A[i], it checks to see if there exists another element that could pair with it to add to k. If a pair exists, it will be found.

A linear search takes linear time, and this runs in quadratic time worst case.

(b) Algorithm 2: Another idea is to enumerate all possible pairs:

```
 for i=0 to n-1  {
       for j= 0  to n-1 {
             if (A[i]+ A[j] == k) return TRUE
       } //for j
} //for i
//if we are here, no pair mathched
return FALSE
```

This is correct because it enumerates all possible pairs with one element in A and another one in B.

Running time: quadratic.

(c) Algorithm 3: Sort the array (using an $O(n \lg n)$ sorting algorithm, and implement Algorithm 1, but with binary search instead of linear search. This runs in $O(n \lg n)$ time.

(d) Algorithm 4: Once both arrays are sorted, the problem can be solved in linear time. This does not change th asymptotic running time, but it's a neat idea: start with a left pointer $l = 0$ on teh first element in array A, and right pointer $r = n - 1$ on teh last element in array B. Check if $A[l] + B[r] < x$: if yes, then advance left $l++$, otherwise decrement right.

```
sort array A and array B

l=0
r=len(B)-1

while (l< len(A) && r >= 0)
     if A[l]+B[r] == x: you got it! return TRUE
```

```
if A[l]+B[r] < x: l++
          //A[l] paired with the current element in B is too small,
          //so A[l] cannot pair with any remaining element in B  to give x
          //==> move to the next element in A

if A[l]+B[r] > x: r--
          //B[r] paired with the current element in A is too large,
          //so B[r] cannot pair with any subsequent element in A to give x
          // ==> move to the previous element in B
```

(e) Algorithm 5: Use hashing: If the elements are integers, a better solution is possible with hashing. Insert all elements in a hash table, and for every A[i], search the hash table for value k-A[i]. Integer hashing can be done in O(1) worst-case, so this can all work in linear time.

If the keys are not integers, hashing may give good average case running time, but can be slow in the worst case (in this case: solution could be quadratic worst-case with hashing). Summary: hashing is something to be tried in practice, but beware of its worst-case bounds.

(b) Generalize to 3-sum: Find if there exist 3 elements in the array whose sum is k, or report that no such subset exists. Analyze running time.

We expect: (1) pseudocode and an English description of your algorithm; (2) analysis of its running time.

Answer:

(a) Algorithm 1: Enumerate all triplets, check if their sum ==k. This runs in cubic time.

(b) Algorithm 2: sort the array. Then

```
for i=0 to n-1
   for j=0 to n-1
         binary search for an erray element of value k-A[i]-A[j]
```

This runs in $O(n^2 \lg n)$.

(c) Algorithm 3: Sort the arrays. Then, for every element $a[i]$, in order, use $O(n)$ algorithm from part (a) to find if there are two elements that sum to $x - a[i]$.

This runs in $O(n^2)$ time.

(d) Algorithm 4: If elements are integers, can use hashing to get running time down to $O(n^2)$.

3. (adapted form GT C-4.27, CLRS 9.3-6) Given an unsorted sequence S of n elements, and an integer k, we want to find $O(k)$ elements that have rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$, and so on.

   (a) Describe the "naive" algorithm that works by repeated selection, and analyze its running time function of n and k (do not assume k to be a constant).

   (b) Describe an improved algorithm that runs in $O(n \lg k)$ time. You may assume that k is a power of 2. After you describe it, argue why its running time is $O(n \lg k)$.

   We expect: pseudocode and an English description of your algorithm, and analysis of its running time.

   Solution:

   (a)
   - Algorithm: For $i = 1$ to $k - 1$: SELECT $(A, \lceil i \cdot n/k \rceil)$
   - Analysis: This runs in $(k-1) \cdot O(n) = O(nk)$

   (b) For simplicity, assume that k is a power of 2. For e.g. consider k=8; then we want to find the elements with ranks $n/8, n/4, 3n/8, n/2, 5n/8, 3n/4, 7n/8$; that's a total of 7 elements.

   The idea is to think of this recursively: first find the median $n/2$, then find the medians of the two halves, $n/4$ and $3n/4$, then the 4 medians in each of the four quarters $n/8, 3n/8, 5n/8, 7n/8$, and so on. Recursive pseudocode could look as below:

   //find k elements with rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$ ... in the range of the array $A[p..r]$
   kPARTITION(A, p, r, k)
   - if $k \leq 1$ return //base case: if $k \leq 1$, we're done
   - else

     $q = SELECT(A, (p+r)/2)$ //find median and output it
     kPARTITION$(A, p, (p+r)/2, k/2)$
     kPARTITION$(A, (p+r)/2 + 1, r, k/2)$

   Analysis: First let's think of the recursion depth: First time you find the median, then two medians, then 4, and so on. At every level of the recursion the number of medians you find doubles. After i levels of recursion you found a total of $1+2+4+...+2^i = 2^{i+1}-1$ medians. To get a total of k medians it takes it takes $\lg k - 1 = \Theta(\lg k)$ levels (this is the same as saying that $1 + 2 + 4 + ...2^{\lg k - 1} = 2^{\lg k} - 1 = k - 1$). On each level we call SELECT on different portions of the array, such that the total number of elements in all calls to SELECT on one level is n, so each level takes $O(n)$ time. Overall this is $\Theta(n \lg k)$ time.

   Another way to analyze is to write a recurrence for the running time: $T(n, k) = 2T(n/2, k/2) + \Theta(n)$, and $T(n/k, 1) = 1$ has solution $T(n) = \Theta(n \lg k)$.

4. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x \ \leq A[x+1]$. For example. there are six local minima in the following array:

$$A = [9, 7, 7, 2, 1, 3, 7, 5, 4, 7, 3, 3, 4, 8, 6, 9]$$

.

We can obviously find a a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\lg n)$ time. (*Hint: with the given boundary conditions, the array must have at least one local minimum. Why?*)

We expect: (1) pseudocode and an English description of your algorithm; (2) why is it correct; (3) analysis of its running time.

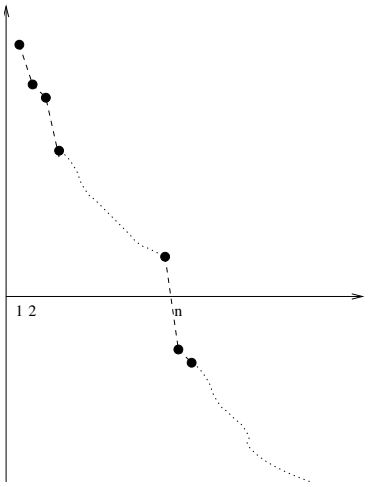
SOLUTION: This works in a manner similar to binary search; the difference is that it is not enough to look only at the element in the middle to decide whether to recurse left or right; we need to look at the elements left and right of the middle as well. Let x be the index in the middle of the range in the current subproblem (initially the range of the problem is $1..n$ and $x = \lfloor n/2 \rfloor$). We distinguish the following case:

- $A[x-1] \leq A[x]$ and $A[x] \geq A[x+1]$: there is a local minimum on either side, so pick one and recurse.
- $A[x-1] \leq A[x]$ and $A[x] \leq A[x+1]$: there is a local minimum on the left side, so recurse on the left half.
- $A[x-1] \geq A[x]$ and $A[x] \geq A[x+1]$: there is a local minimum on the right side, so recurse on the right half.
- $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$: x is a local minimum, so return it.

The running time is given by the recurrence $T(n) = T(n/2) + \Theta(1)$, which solves to $T(n) = \Theta(\lg n)$

Optional problems

1. Consider a *monotonically decreasing* function $f : N \rightarrow Z$ (that is, a function defined on the natural numbers taking integer values, such that $f(i) > f(i+1)$). Assuming we can evaluate f at any i in constant time, we want to find $n = \min\{i \in N | f(i) \leq 0\}$ (that is, we want to find the value where f becomes negative).



We can obviously solve the problem in $O(n)$ time by evaluating $f(1), f(2), f(3), \ldots f(n)$. Describe an $O(\log n)$ algorithm.

(*Hint*: Evaluate f on $O(\log n)$ carefully chosen values between 1 and $2n$ - but remember that you do not know n initially).

2. You accepted an internship as consultant for a green company, which is planning a large underground water pipeline running east to west through a field of n water wells.The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (that is, either north or south, perpendicular to the main pipe). The the x- and y-coordinates of the wells are given. For a certain location of the main pipeline, the total length of the spur for that location is defined as the sum of the lengths of the spurs of each well to the main pipeline. The goal is to pick the optimal location of the main pipeline that minimizes the total length of the spur.

   Describe how you will pick the optimal location of the main pipeline, and argue why it's optimal. Describe your argument for the case n is odd; then extend it to when n is even.

   Solution: The problem specifies that the pipe has to be horizontal. The optimal location is through the y-value that corresponds to the median of the y-ccordinates of the wells. Call this median y. If n is odd, this median is unique. If n is even, the pipeline's y-coordinate can be anywhere between the lower and the upper median. We are going to argue why this is correct.

   For simplicity, we'll assume n is odd, so the median is unique.

   The cost is the sum of the distances of the wells to the pipeline. Because the pipeline goes through the median, half of the wells are below the pipeline and half of the wells are above the pipeline; one well, the one corresponding to the median , will be right on the pipeline. If there is more than one well with the same y-coordinate as the median, then there'll be several wells right on the pipe. In general let's denote a the number of wells strictly above thepipeline; b the number of wells strictly below the pipeline; and c the number of wells right on the pipeline. If all y-coords are distinct we know that $a = b = \lfloor n/2 \rfloor$ and $c = 1$. Otherwise we know that $c > 1$ and $a + c > b$ and $b + c > a$ (we'll need these below).

   Let's see what happens if we move the pipeline *up* by ϵ (It's best to draw a figure to visualize): all the wells that were above the pipe are now closer by ϵ to the pipe. All the wells that were on or below the pipe are now further away by ϵ. Overall the cost decreases by $a \cdot \epsilon$ and increases by $(b + c) \cdot \epsilon$. Since $b + c > a$ this implies that the cost increases.

   The situation where we move the pipeline down by ϵ is symmetrical.

   So when we move the pipeline upward from the median, the cost increases; When we move the pipeline down from the median, the cost increases. It means the cost is optimal when the pipeline passes through the median.

   The case when n is even is very similar. Draw four wells and convince yourself that the pipeline can be situated anywhere between the 2nd and the 3rd smallest wells.