CS5050 Advanced Algorithms

Spring Semester, 2021 Assignment 3: Prune and Search

**Due Date: 11:59 p.m.**, Tuesday, March 2, 2021

As for Question 4 of Assignment 1, for each of the algorithm designproblems in all assignments of this class, you are required to clearly describe the main idea of youralgorithm. Although it is not required, you are encouraged to give the pseudo-code (unless youfeel it is really not necessary). You also need to briefly explain why your algorithm is correct if thecorrectness is not that obvious. Finally, please analyze the running time of your algorithm.

1. **(20 points)** Suppose you are given an array *A*[1··· *n*] with *n* entries, with each entry holding a distinct number (i.e., no two numbers of *A* are equal). You are told that the sequence of values *A*[1]*,A*[2]*,... ,A*[*n*] is *unimodal:* For some index *p* between 1 and *n*, the values in the array entries increase up to position *p* in *A* and then decrease the remainder of the way until position *n*. (So if you were to draw a plot with the array position *j* on the *x*-axis and the value of the entry *A*[*j*] on the *y*-axis, the plotted points would rise until *x*-value *p*, where they would achieve their maximum, and then fall from there on.)

You would like to find the “peak entry” *p* without having to read the entire array – in fact, by reading as few entries of *A* as possible. Show how to find the entry *p* by reading at most *O*(log*n*) entries of *A*. In other words, design an *O*(log*n*) time algorithm to find the peak entry *p*.

For example, let *A* = {1*,*4*,*6*,*8*,*11*,*12*,*10*,*9*,*7}, which is be a unimodal array. The peak entry of *A* is 12. So the output of your algorithm may be either 12 or the index of 12 in *A*.

**Main Idea:**

Suppose the first and last elements of the array are A[start] and A[end]. Take the middle element of Array A at index p. If A[p] > A[p-1] and A[p] < A{p+1] then you can ignore A[start..p-1]. If A[p] < A[p-1] and A[p] > A[p+1]then we have passed the peak and we can ignore A[p..end] and we recurse on A[start..p-1]. At each step, we omit half the array, but in the process the peak is never omitted. Eventually we will be left with the peak element which is returned.

**Pseudo Code:**

int findPeak(A[], start , end) {

int p = (start + end)/2;

if (A[p] > A[p-1] && A[p] < A[p+1]) {

return findPeak(A[], p+1, end);

}

else if (A[p] < A[p-1] && A[p] > A[p+1]) {

return findPeakEntry(A, start, p-1);

}

Else{

return p;}

**Time:**

Since this is similar to the binary search, the algorithm runs in O(logn).

1. **(20 points)** In the SELECTION algorithm we studied in class, the input numbers are divided into groups of five. Will the algorithm still work in linear time if they are divided into groups of seven? Please justify your answer.

The algorithm would still work in linear time because you can arbitrarily choose how many elements to divide the groups into. This means that if you partition into groups of 7 instead of 5 you would still have to search each element, which gives O(n) time independent of the number you choose. As long as you divide the input size each time you call your function, linear time will still be achievable.

1. **(20 points)** Suppose you are consulting for an oil company, which is planning a large pipeline (called the *main pipeline*) running horizontally from east to west through an oil field of *n* wells. From each well, a *spur pipeline* is to be connected directly to the main pipeline along a shortest path (going to either the north or the south), as shown in Figure 1.

Suppose there are *n* wells, represented by *n* points *p*1*,p*2*,...,pn* in the plane. We are given the *x*- and *y*-coordinates of the *n* wells *pi* = (*xi,yi*) for *i* = 1*,*2*,... ,n*. Note that the wells are not given in any sorted order. Our goal is to pick an optimal location for the main pipeline (i.e., find the *y*-coordinate of the main pipeline) such that the **total sum of the lengths** of the spur pipelines is minimized. For simplicity, we assume that no two wells have the same *x*-coordinate or *y*-coordinate.

Design an *O*(*n*) time algorithm to compute an optimal location for the main pipeline.

**Main Idea:**

Since our goal is to find the optimal y-coordinate for the main pipeline we can split up this problem into two different scenarios. The first scenario being n is even and the other n is odd. In the scenario where n is even, we can choose any y location in between the upper and lower median and it will be optimal. In the case that n is odd we will set the y coordinate of the main pipeline to be equal to the median of all the y coordinates of the wells.

**Correctness:**

When n is odd there are n/2 wells above and below the median. If you were to move the pipelines y coordinate in either direction, you would lose an equal amount distance on either side.

**Time:**

1. **(30 points)** Here is a generalized version of the selection problem, called *multiple selection*. Let *A*[1···*n*] be an array of *n* numbers. Given a sequence of *m* sorted integers *k*1*,k*2*,...,km*, with 1 ≤ *k*1 *< k*2 *<* ··· *< km* ≤ *n*, the *multiple selection problem* is to find the *ki*-th smallest

1

wells

spur pipelines

main pipeline

Figure 1: Illustrating Problem 3: the horizontal solid line is the main pipeline and the dashed vertical segments are the spur pipelines.

number in *A* for all *i* = 1*,*2*,... ,m*. For simplicity, we assume that no two numbers of *A* are equal.

For example, let *A* = {1*,*5*,*9*,*3*,*7*,*12*,*15*,*8*,*21}, and *m* = 3 with *k*1 = 2, *k*2 = 5, and *k*3 = 7. Hence, the goal is to find the 2nd, the 5-th, and the 7-th smallest numbers of *A*, which are 3, 8, and 12, respectively.

* 1. Design an *O*(*n*log*n*) time algorithm for the problem. **(5 points)**

**Main Idea:**

If we sort the array first, we can achieve an O(nlogn) time for this algorithm.

* 1. Design an *O*(*nm*) time algorithm for the problem. Note that this is better than the

*O*(*n*log*n*) time algorithm if *m <* log*n*. **(5 points)**

**Main Idea:**

We can achieve a O(nm) time by maintaining the m smallest elements and considering every element of A[].

* 1. Improve your algorithm to *O*(*n*log*m*) time, which is better than both the *O*(*n*log*n*) time and the *O*(*nm*) time algorithms. **(20 points)**

**Main Idea:**

The main idea of this algorithm is to partition both the array(A[]) and the sequence M in order to create the desired O(nlogm) time. To start we create an empty array to store the set of desired values. Then we partition the array A into two smaller arrays. We then create an index I and check to see if we need to add a value to the Results array. Then partition the M set into two different arrays and recall them recursively.

**Pseudo Code:**

selectMulti(A[],M){

Results[];

If (M is empty)

{return Results}

P = A[n/2];

Partition A into sets A0<p and A1>p.

i = A0.size + 1

if (M contains i){

then remove i from M and add (i=>p) to the result array.}

Partition M into sets M0<i and M1>i;

add selectMulti(A0,M0) to the result set

subtract i from each k in M1

call selectMulti(A1,M1), add i to each index of the output, and add this to the result set

return Results[]

}

**Time:**

This algorithm chooses the middle value as the pivot which means M gets split in half each time, and so the number of recursive calls will be exactly log m, and since quickselect runs in linear time, the result will still be O(n log m).

**Total Points:** 90

2