Daniel Oliveros A02093272
February 22, 2018

Assignment 3: Prune and Search

Problem #1 Peak entry

1. **Algorithm Description:** This algorithm follows a prune and search approach. It grabs the element in the middle index and determines if there is positive growth going on behind it, if so, it uses that element and the ones to its right to check for the peak, otherwise, it uses the elements to its left to find it.

2. **Pseudocode:**

```
def findPeak(A, i, j):
    if(i==j):
        return i
    mid = (i+j)//2 + 1
    if(A[mid]>A[mid-1]):
        return findPeak(A, mid, j)
    else:
        return findPeak(A, i, mid-1)

a = [1, 4, 6, 8, 11, 12, 10, 9, 7]
index = findPeak(a, 0, len(a)-1)
print("for array ", a, " the peak item is ", a[index])
```

3. **Correctness:** In the array $a = [1, 4, 6, 8, 11, 12, 10, 9, 7]$ there are 9 elements. When we perform the first check we look at array at location 4, In this location, we can see that 11 is greater than 8, so we use the section of the array $a_1 = [11, 12, 10, 9, 7]$. In this section, we look at element 10 and see that it's smaller than 12, so we continue to work with the array $a_2 = [11, 12]$. Then, we see that 12 is greater than 11, so we are left with array $a_3 = [12]$. We then return this element.

4. **Time Analysis:** This array follows a similar approach to binary search, it performs $\log n$ number of operations.

Problem #2 SELECT with groups of seven.
SELECT will still have linear time even if the arrays are made up of seven elements. This can be explained solving the following recursion:

$$T(n) = T(\frac{n}{7}) + T(max\{|A_1|, |A_2|\}) + n$$

In order to determine what the highest possible value for $max\{|A_1|, |A_2|\}$ is, we first should think of the smallest possible either can take. This is

$$|A_1| = \frac{n}{7} * \frac{1}{2} * 4 = \frac{2}{7}n$$

So, this makes the highest possible value for $|A_1| = \frac{5}{7}n$. We move onto putting this in our equation:

$$T(n) = T(\frac{n}{7}) + T(\frac{5}{7}n) + n$$

To prove this recurrence has $O(n)$, we can use the guessing method. So:

$$
\begin{aligned}
\text{Guess:} \quad & T(n) = O(n) \\
\text{Assume:} \quad & T(\tfrac{n}{7}) \leq c\tfrac{n}{7}, T(\tfrac{5}{7}n) \leq c\tfrac{5}{7}n \\
\text{Then:} \quad & T(\tfrac{n}{7}) + T(\tfrac{5}{7}n) + n \leq & c\tfrac{n}{7} + c\tfrac{5}{7}n + n \leq^{?} & \quad cn \\
& & 1 \leq^{?} & \quad \tfrac{1}{7}c \\
& & 7 \leq & \quad c
\end{aligned}
$$

So, since we can find a set of conditions under which $O(n)$ is an upper bound for our formula, then the formula's runtime is still $O(n)$ regardless of the change in list size.

Problem #3 Main pipeline.

To solve this problem, we determined the best point to pick in the pipeline was one with the same height as the median of all the wells in the set. This is due to the fact that this location minimizes the total variance throughout the array, making it the best choice. In order to find the median easily with an $O(n)$ runtime, we can simply use the SELECT algorithm outlined in class. The solution to this problem would, therefore, be SELECT(y-values, n/2). This would return the median of the array y-values, which is just an array comprised of all the heights of our set of oil wells.

Problem #4 Multiple selection.

(a) $O(n \log n)$ approach:

In order to solve this problem with an $n \log n$ approach, we can simply sort the array A, which takes $n \log n$ operations, and then for each element in the k sequence of m integers, simply go through the sorted version of A and print $A[k_i]$. This second step should only take m operations, so the final complexity of this algorithm is $O(n \log n)$.

(b) $O(nm)$ approach:

For this method, we can use our SELECT algorithm to do most of the work for us. So, simply iterate through every k element and perform the operation SELECT$(A, n - k_i)$. Since the computation time for each SELECT call is n, and we are performing it m times, our total complexity amounts to $O(nm)$

(c) $O(n \log m)$ Approach:

1. **Algorithm Description:** This algorithm follows a prune and search approach. It first sets up a variety of arrays based on the original set of k elements. We traverse this set of arrays by performing a binary approach on the set of k elements. This binary approach works similarly to search, except that instead of looking for a particular element, our logic centers on finding arrays where the element we're looking to insert can be placed into. We look leftward to determine if our element can be placed into a given array, if so, we place it into it. If the array is full, but our element still belongs in it, then we place the element in that array and push the other one onto the next array.
   Once all elements have been placed into the arrays, we traverse our tree in order and print the last element in each of the smaller arrays.

2. **Pseudocode:**

```
def multipleSel(A, k):

    create diff array k' from k
    create an array B of arrays where b[i] = []*k'[i]

    for i in A: #O(n)
        Perform a binary lookup on k (O(log m)) where we look at its
            ↪ related B array in search of optimal arrays to insert
            ↪ into.
        Insert into array at B, if array is full shift the biggest
            ↪ element from array inserted into to next array in order.

    go through each array in B and print out the last element in each
        ↪ list.
```

3. **Correctness:** To show how this method works, let's look at an example. Suppose $A = [4, 6, 7, 3, 5]$, $m = 3$, $k_1 = 1$, $k_2 = 2$, and $k_3 = 4$. We create the k' array $k' = [1, 1, 2]$. And from here we make the list $B = [[], [], [,]]$.

   Once these are all built, we proceed to iterate through each element in A to see where we want to insert them in B. We perform our binary lookup, looking first at $B[1]$, since there is a list to its left ($B[1/2]$ or $B[0]$) that's currently empty, then we go check that list. List $B[0]$ is empty and there are no lists to its left, so we insert 4 into it. B is now $[[4], [], [,]]$.
   From here, we look at 6. We check $B[1]$, see that the array to its left is full but still look at $B[0]$ to check if 6 would be a better fit for this position. Since 6 is greater than 4 then we know it doesn't belong in this position, so we go back to $B[1]$ and insert 6 into it. B is now $[[4], [6], [,]]$.
   We look at 7 now, since $B[1]$ is full and $7 > 6$ then we move onto the element to its right, $B[1 * 2]$. Since this list is empty and there isn't a list to its left, then we insert 7 into it. B is now $[[4], [6], [7,]]$
   Now, we look at 3. Since 3 is smaller than 6, we check the list at location B[0], and since 3 is also smaller than 4, then we insert 3 into this location. This means that we have to shift every element to the next "position" in the sequence. After this step, $B = [[3], [4], [6, 7]]$.
   Finally, we look to insert 5. Since it's greater than 4 we look at $B[2]$, and since $B[2]$ is full and $5 < 6$ then we look to insert 5 into this array. This shifts 7 out of our set of elements, as there is no place where it needs to be held anyway. After all these insertions are done $B = [[3], [4], [5, 6]]$.

   To finish off, we go through each array in B and print out its last element, in this case that output would be $3, 4, 6$, which correspond with the 1st, 2nd, and 4th smallest elements in A.

4. **Time Analysis:** The lookups for which array to insert into take $\log m$ time, and we perform each of these operations n times. Every other operation takes either m or n time, so our asymptotic runtime for this algorithm is $O(n \log m)$.

People I worked with for this assignment:

- Chris Darais
- Andrew Aposhion