CS5050 Advanced Algorithms

Spring Semester, 2018

Assignment 3: Prune and Search

Philip Nelson

Worked with: Ammon Hepworth, Raul Ramirez, Hailey Maxwell, and Jack Kiefer

1. **(20 points)** Peak of a unimodal array.

   1. **Algorithm Description** The algorithm *peak* finds the peak of a unimodal array, A, in $O(\log n)$ making use of a prune and search algorithm similar to binary search. The algorithm begins by calculating the midpoint of the array, It checks if the element to the right is less than it and the element to the left is greater. If this is the case, then the midpoint is the peak. If, however, this is not the case, then it will determine whether the slope is positive or negative. Imagine if the value of the elements of A were plotted against their indices. If the current element is not the peak, then the peak must be "uphill" from it and it can ignore the points "downhill". The next point to be checked is halfway between the current midpoint and the "uphill" side, until the peak is found.

   2. **Pseudocode** *peak*

```
template <typename T>
int peak(std::vector< T > A, int start, int end)
{
  // no peak exists
  if (start > end)
    return -1;

  auto mid = (start + end) / 2;

  // peak located
  if (A[mid - 1] < A[mid] && A[mid] > A[mid + 1])
    return mid;

  // positive slope
  if (A[mid - 1] < A[mid] && A[mid] < A[mid + 1])
    return peak(A, mid + 1, end);

  // negative slope
  if (A[mid - 1] > A[mid] && A[mid] > A[mid + 1])
    return peak(A, start, mid - 1);
}
```

3. Time Analysis The algorithm *peak* finds the peak of a unimodal array, A, in $O(\log n)$. It is capable of doing this because each time a point is examined, half of the remaining elements are pruned. This means that the recurrence is $T(n) = T(\frac{n}{2}) \Rightarrow T(n) = \log(n)$.

2. **(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 selection algorithm partitions a set A into two sets A_1 and A_2. In order to pick a pivot, selection uses a method of choosing a median of medians. If groups of seven are used, instead of five, the median of medians will still guarantee that a constant portion of the elements of A will go in sets A_1 and A_2. As shown below in Figure 1, we can see that both A_1 and A_2 will have at least $\frac{n}{7} \cdot \frac{1}{2} \cdot 4 = \frac{4}{14}n$ elements and at most $n - \frac{4}{14}n = \frac{10}{14}n$. Therefore, in the worst case, one partition of A will have at least $\frac{4}{14}n$ elements. Since $\frac{4}{14}n$ is a constant percentage of n, not just a constant number, the selection algorithm will still have a time complexity of $O(n)$.
This idea can be further expanded to any number of elements e. The smallest that a partition of A could be is $\frac{n}{e} \cdot \frac{1}{2} \cdot \lceil \frac{e}{2} \rceil = \frac{1}{2e} \lceil \frac{e}{2} \rceil n$ elements. In like fashion, this is a constant percentage of n, therefore the algorithm can run in linear time.

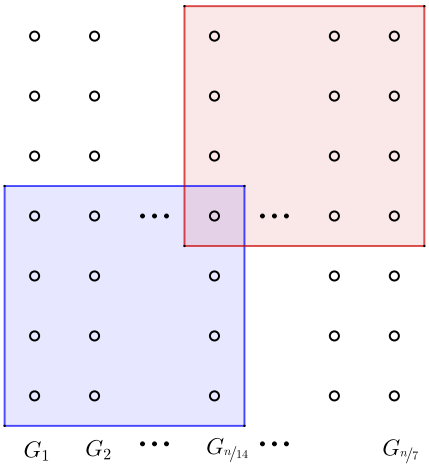

Figure 1: Groups of 7

3. **(20 points)** Design an $O(n)$ time algorithm to compute an optimal location for the main pipeline.

   1. **Algorithm Description** The algorithm *layPipe* finds the optimal location to place the main pipeline given an array, A, of well locations. It will accomplish this by selecting the median y value from all the wells.

   2. **Pseudocode** *layPipe*

```
template < typename T >
int layPipe(std::vector< T > A)
{
  // collect all the y coordinates
  std::vector< T > ys;
  for(auto && p : A)
    ys.push_back(p.y);

  // use selection to find the median y value
  auto y = selection(ys, ys.size()/2);

  // place pipeline at the median y value
  return y;
}
```

   3. **Time Analysis** The algorithm *layPipe* finds the optimal location to place the main pipeline given an array, A, of well locations. The algorithm begins by doing one linear scan to separate the y values into an array which takes $O(n)$ time. Then it uses the *selection* algorithm to choose the median y value, which also takes $O(n)$ time. Since *layPipe* does $2n$ work to find the location of the main pipeline, it's time complexity is $O(n)$.

   4. **Correctness** The main pipeline can be laid exactly on the y value of the median well because the optimal location for the pipe will place the same number of wells on either side of the main pipeline.

      For the case that there are an odd number of wells, the main pipeline will run directly over the well with the median y value. If you were to shift the main pipeline up one unit, for example, you would be moving the pipeline closer to $\lfloor \frac{n}{2} \rfloor$ wells but further from $\lceil \frac{n}{2} \rceil$ wells. Since n is odd, this means the pipeline is moving away from one more well than it is moving closer to, therefore, for every unit it moves, it adds one more unit of total spur pipeline.

      For the case that there are an even number of wells, there will not be one single median well but two. Lets call the y values for these two median wells y_1 and y_2. As long as the pipeline is placed at a y_p such that $y_1 \leq y_p \leq y_2$ the total spur pipeline is minimized. If the main pipeline is moved around within the range $[y_1, y_2]$ there are an equal number of wells on either side so the change in total spur pipeline is not affected.

4. **(30 points)** Here is a generalized version of the selection problem, called *multiple selection*. Let $A[1 \cdots n]$ be an array of n numbers. Given a sequence of m sorted integers $k_1, k_2, \ldots, k_m$, with $1 \leq k_1 < k_2 < \cdots < k_m \leq n$, the *multiple selection problem* is to find the k_i-th smallest number in A for all $i = 1, 2, \ldots, m$. 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.

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

   The trivial solution is to sort the array and do a single linear scan to find the k-smallest elements. Sorting can be done in $O(n \log n)$ and the linear scan will take an additional n work. Since $n \log n$ is the dominating term, the algorithm's time complexity is $O(n \log n)$.

   (b) Design an $O(nm)$ time algorithm for the problem. **(5 points)**

   The *selection* algorithm can find the k^{th} smallest element in $O(n)$ time. If there are m elements to locate, we can simple employ the *selection* algorithm for each one. The algorithm will do n work m times which is time complexity $O(nm)$.

   (c) Improve your algorithm to $O(n \log m)$ time. **(20 points)**

   In order to achieve multiple selection in $O(n \log m)$ the algorithm will begin by finding the k_{mid} element. Because all the k's are sorted, once we have found the k_{mid} we will have the following situation,

   $$A_1 < k_{mid} < A_2$$

   where the $\{k_1, \ldots, k_{mid-1}\} \in A_1$ and $\{k_{mid+1}, \ldots, k_m\} \in A_2$. The algorithm then calls the function again with A_1 and $\{k_1, \ldots, k_{mid-1}\}$, and A_2 and $\{k_{mid+1}, \ldots, k_m\}$. It will need to divide the array of k's $\log m$ times and at each level, there will be n work done. For this reason, the time complexity will be $O(n \log m)$.

Total Points: 90