Steven Proctor
A01514516

#1 Unimodal Max

Find the max value in an unimodal array in O(logn) time.

PSEUDO CODE:

```
public int findUnimodalMax(int[] A, int start, int end) {
        if (end == start) { return A[start]; }

        if (end == start+1) {
                if (A[start] >= A[end]) {return A[start];}
                else { return A[end]; }
        }

        int mid = (start + end) / 2;

        if (A[mid] > A[mid+1] && A[mid] < A[mid-1]) {
                return findUnimodalMax(A, start, mid-1);
        } else {
                return findUnimodalMax(A, mid+1, end);
        }
}
```

DESCRIPTION:

This algorithm uses the prune and search method, similar to a binary search. By cutting the array in half and discarding the unnecessary half each time, you will only have to traverse a maximum of log(n) times to find the max element of the array. The comparison done each time through the function are negligible, as they only result in $c * log(n)$, which becomes $O(logn)$.

ANALYSIS:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/4) + 2 \\ &= T(n/8) + 3 \\ &\cdots \\ &= T(n/2^k) + k \end{aligned}$$

$$2^k = n \quad when \quad k = log_2(n)$$

$$T(n) = T(n/2^{log_2(n)}) + log_2(n)$$

$$T(n) = 1 + log_2(n)$$

$$T(n) = O(log_2(n))$$

#2 SELECTION ALGORITHM

Will the Selection Algorithm work in linear time if the numbers are divided into groups of seven, instead of five?

DESCRIPTION:

The traditional Deterministic Selection Algorithm is runs in:

$$T(n) = T(max(left.size, right.size)) + T(n/5) + n$$

For worst case scenarios,

$$T(max(left.size, right.size)) = T(7n/10)$$

which can be used to prove that $T(n) = O(n)$. If divisions of seven are use instead of divisions of five:

$$T(n) = T(max(left.size, right.size)) + T(n/7) + n$$

In this case the $max()$ method does not have a running time of $T(7n/10)$. However, an analysis of the algorithm using divisions of seven shows that the algorithm will still run in $O(n)$ time.

ANALYSIS:

When using divisions of seven, and analyzing the left half of the Array:

$$\begin{aligned} At\ least: &\quad 2n/7 \leq pivot \\ At\ least: &\quad 5n/7 > pivot \\ Therefore: &\quad right.size \leq 5n/7 \end{aligned}$$

The same result can be obtained for the left side using the same analysis. This means that the running time can be re-written as:

$$T(n) = T(5n/7) + T(n/7) + n$$

We will then assume that $T(x) \leq c * x$ in an attempt to prove $T(n) = O(n)$

$$\begin{aligned} c * 5n/7 + c * n/7 + n &\leq c * n \\ c * 6n/7 + n &\leq c * n \\ n &\leq c * n/7 \end{aligned}$$

If $c = 14$ then $T(n) \leq n \leq 2n$. Therefore:

$$T(n) = O(n)$$

#3 MAIN PIPELINE

Find the y-coordinate for a pipeline to minimize the total length of the off-shoots needed to reach each well in $O(n)$ time.

DESCRIPTION:

The desired y-location for the pipeline is the same as the median value of all y-coordinates. Using the median ensures that there are an equal number of wells on either side of the main pipeline. This guarantees a minimum because moving the pipe above or below the median would increase the total length of the off-shoots by a factor of at least $n/2$. Thus, the Deterministic Selection algorithm can be used to obtain the solution.

ANALYSIS:

As shown in the previous problem the selection algorithm runs in O(n) time, thus obtaining a solution in the desired time. To further emphasize the accuracy of the solution, imagine an array $A[1, 4, 8]$. The median is $A[2] = 4$ which yields a total off-shoot distance of 7. Moving the pipeline up to 5 or down to 3 would yield a new distance of 8. This is because the pipeline is no longer balanced, and there are more than half the total number of wells on one side of the main pipeline.

One more example can be show with the array $A[7, 2, 9, 5]$. The median of this array can be either $A[4] = 5$ or $A[1] = 7$. If either median is selected as the y-coordinate of the main pipeline, it yields a total off-shoot distance of 9. Moving the pipeline even one position above 7, or below 5 increases the total off-shoot distance by at least n/2. Thus we can see that the median, which can be found in $O(n)$ time by the selection algorithm, provides the optimal location for the main pipeline.

#4 MULTIPLE SELECTION

Find the k^{th} smallest number in $A[1..n]$ for each element $k[1..m]$ in $O(n * log_2(n))$ time, then $O(n * m)$, and finally $O(n * log_2(m))$.

$O(n * log_2(n))$ ALGORITHM:

This problem can be solved in $O(n * log_2(n))$ by using a sorting algorithm, such as Merge Sort, to sort the array $A[1..n]$. From there array $K[1..m]$ can be iterated through one element at a time using the following line of code:

```
B[i] = A[K[i]];
```

This method takes $T(n) = n * log_2(n) + m$ time to complete, which means the total complexity of the algorithm is $O(n * log_2(n))$.

$O(n * m)$ ALGORITHM:

Alternatively, the Deterministic Selection algorithm can be used to accomplish the same task in $O(n * m)$ time. Use the following line of code while iterating through each element of the array $K[1..m]$:

```
B[i] = selection(A, K[i]);
```

Because the Selection algorithm runs in $O(n)$ this method cuts down on the $O(n * log_2(n))$ time needed to sort the algorithm. The iteration runs m times which results in a total run time of $O(n * m)$.

$O(n * log_2(m))$ ALGORITHM:

```
public int mSelect(int[] A, int[] K) {
        if (!K.isEmpty()) {
                if (K.length == 1) {
                        return selection(A, K[1]);
                } else {
                        int p = K.length;
                        int m = K[p/2];
                        selection(A, m);

                        int[] A1 = everything < S[m];
                        int[] A2 = everything > S[m];
                        int[] K1 = {k[1]...k[p/2-1]};
                        int[] K2 = {k[p/2+1]-m...k[p]-m};
                        mSelect(A1, K1);
                        mSelect(A2, K2);
                }
        }
}
```

The pseudo code above illustrates a method that utilizes the runtime of the Selection algorithm as well as a Prune and Search recursive method. This method cuts down on the number of iterations needed to get through array $K[1..m]$ from $O(m)$ to $O(log_2(m))$. This results in a total runtime of $O(n * log_2(m))$.