# Homework 6

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

15 September 2016

## Problem 5:

Below is the table constructed to show the total weight of the Optimal Binary Search Tree.

Table 1: Weight

|   | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0.5 | 0.6 | 0.85 | 1.4 | 2.1 |
| 2 |  | 0.05 | 0.2 | 0.55 | 1.05 |
| 3 |  |  | 0.1 | 0.4 | 0.85 |
| 4 |  |  |  | 0.2 | 0.65 |
| 5 |  |  |  |  | 0.25 |

Below is the table constructed to show the roots for the Optimal Binary Search Tree.

Table 2: Roots

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| 2 | | 0.05 | 0.1 | 0.2 | 0.25 |
| 3 | | | 0.1 | 0.2 | 0.25 |
| 4 | | | | 0.2 | 0.25 |
| 5 | | | | | 0.25 |

In order to recompute the tree from the table of roots, first you have to start in the upper right corner. The upper right corner will be the root for the entire tree. Now to find the next node, find the next value that not the root in the table. In this case it is 0.25, so that will be a child of 0.5 in the right subtree because the key $k_5 > k_1$. Next, get 0.2 from the left of 0.25 in the table and use that as a child to 0.25 in the left subtree because the key $k_5 > k_4$. Then continue this for the other two keys and the following tree will be constructed.
This tree's weight is equal to $0.5(1) + 0.25(2) + 0.2(3) + 0.1(4) + 0.05(5) = 2.25$. Which is the value in the upper right corner of the weight table above.

$$\begin{tikzcd}
 & & 0.5 \arrow[dr] & \\
 & & & 0.25 \arrow[dl] \\
 & & 0.2 \arrow[dl] & \\
 & 0.1 \arrow[dl] & & \\
0.05 & & &
\end{tikzcd}$$

## Problem 6:

Let $\mathcal{P}$ be an $n$ sided convex polygon with vertex set $\mathcal{V}$. Pick a vertex $v \in \mathcal{V}$ and label it 1. In a clockwise manner starting from the vertex labeled 1, continue labeling the vertices $2, 3, \ldots, n$.

The minimum permimeter $n-2$ triangulation of $\mathcal{P}$ can be define recursively as follows.

```
Function: $N\_2Trig$
Globals: $Polygon\ \mathcal{P}$
Input: $Polygon\ P$
Define: $\mathcal{V} := Numbered\ vertices\ of\ P$
if $|\mathcal{V}| = 3$ then
    return $Perimeter(P)$
else
    Define: $P[\,]$
    for $i \leftarrow 1$ to $|\mathcal{V}| - 1$ do
        /* [a,b] means the line from vertex a to vertex b  */
        Define: $e := [i, i+1]$
        foreach $v$ not $i, i+1$ do
            /* Polygon(a,b) constructs polygon with edge a,b
               embedded on $\mathcal{P}$                        */
            /* Polygon(x,y,z) constructs a polygon with
               vertices x,y,z                                    */
            Define: $P_a := Polygon(i, v)$
            Define: $P_b := Polygon(i+1, v)$
            Define: $P_3 := Polygon(i, i+1, v)$
            $P.append(N2Trig(P1) + N2Trig(P2) + N2Trig(P3) -$
              $dist(i, v) - dist(i+1, v))$
        end
    end
    return $min(P)$
end
```

Although this algorithm produces a minimum perimeter $n-2$ triangulation of polygon $\mathcal{P}$, it runs in exponential time due to the recursive calls placed within 2 nested loops.

The triangulation process can be improved significantly by moving to an array based as follows.

```
**Function:** $Dyn\_N_2Trig$
**Globals:** $Polygon\ \mathcal{P}$
/* Initialization                                              */
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        if $i - j < 2$ then
            $A[i][i][i] = 0$
        end
    end
    if $i - j = 1$ then
        $A[i][j][0] = 0$
    end
end
/* calculation                                                 */
Define: $P[\ ]$
for $i \leftarrow 1$ to $n$ do
    foreach $j$ not $i, i+1$ do
        $A[i][(i+1)\ mod\ n][j] = Perimeter(i, i+1, j)$
    end
end
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i+2$ to $n$ do
        $P[i][j] = P[i][j-1] + A[i][i+1][j]$
    end
end
Rerturn: $min(P[n]$
```

## Problem 7:

Consider the following input array:

$$A = \begin{bmatrix} 1 & 2 & -2 & 10 \end{bmatrix}$$

Let $MCS$ be the function that calculates the maximum consecutive subarray of $A$, $A[0, n]$ where $n$ is a parameter to the function. Obviously, $MCS(3)$ will return 3, taking the values 1 and 2 from index 1 and 2 respectively and not taking the value $-2$. Now, a naive implementation of $MCS(n)$ will have a call site with a call to $MCS(n-1)$, but with the above case, knowing that $MCS(3) = 3$ is not enough to know that $MCS(4)$ must be 9. The naive implementation, which has no knowledge that index $n-1$ was not taken, requiring the recombination step to be slightly more sophisticated that simply adding the value at the current index with the value returned from the recursive call site.

To strengthen the inductive hypothesis, we require that $MCS$ not only return the maximum sum of the contiguous subarray elements, but also the index of the final element selected as a member of the contiguous subarray.

Thus, our function signature is now

$$MCS(n) \rightarrow j, max$$

To implement our algorithm, we first perform a preprocessing step which calculates the sum from an index to the end of the array in linear time.

```
input  : An empty, global array B of length $n$
for $i \leftarrow n$ to $0$ do
    if $i == n$ then
        $B[n] \leftarrow 0$
        continue
    end
    $B[n] \leftarrow B[n-1] + A[n]$
end
```

This allows the $MCS$ algorithm to look up the sum of the values from a single index to the end of the array in constant time. Now, we implement $MCS$ as follows...

```
input  : $n$
output: $j, max$
if $n$ == 0 then
    return 0, 0
end
$k, max = MCS(n-1)$
if $Sum(j+1, n) > 0$ then
    $return\ n, max + Sum(j+1, n)$
end
if $max < A[n]$ then
    $return\ n, n$
end
$return\ j, max$
```

As you can see, this operations acts in linear time, reducing the size of $n$ by 1 until reaching a base case, while only producing a single recursive function call. We still have to define $Sum$, which also acts in linear time. $Sum(i, j)$ is the function that calculates the sum of the values $A[i...j]$ inclusively. To calculate the sum, $Sum$ simply returns the difference between $B[j]$ and $B[i]$, a constant time operation.

Unlike the other problems we did in class, there are no overlapping subproblems that need to be cached for this solution to be efficient. Thus, there is no need to convert the above algorithm to an iterative, array based approach.