CSC373 - Problem Set 1

Authors: Luke Bacchus, Naslin Rahman, Zhuoqian Li

January 20, 2021

Question 1

a. We have n piles of m papers. As merging two sorted piles can be done it time proportional to the size of the resulting pile, following the given algorithm the first merge will take $2m$ time, the second merge will take $3m$ time, etc. The running time of this algorithm can be found:

$$\begin{aligned}
&= 2m + 3m + 4m + ... + nm \\
&= \sum_{k=2}^{n} km \\
&= m(\sum_{k=2}^{n} k) \\
&= m(\sum_{k=1}^{n} k - 1) \\
&= m(\frac{(n+1)(n)}{2} - 1)
\end{aligned}$$

This is $\mathcal{O}(mn^2)$.

b. If you have exactly two piles, merge them as in a. If you have more than two piles, divide into two sets of piles of equal size (we can do this because n is an exact power of two). Then, recursively merge each set of piles until you obtain two piles $p1$ and $p2$. Then merge $p1$ and $p2$, and the run time of this process is proportional to the size of $p1 + p2$. The recurrence relation can be described by:

$$T(mn) = \begin{cases} 0, & \text{n = 0 or m = 0.} \\ 2T(\frac{mn}{2}) + mn, & \text{n > 0 and m > 0.} \end{cases} \tag{1}$$

By the master theorem, where in this case: a = 2, b = 2, d = 1 meaning that a = b. Thus, the (worst-case) running time is $O(mn \log mn)$.

Question 2

a. Divide the array into two subarrays: $s1 = A[0 : \frac{n}{2}]$ and $s2 = A[\frac{n}{2} + 1 : n]$. Recursively find S_{ij} such that $S_{ij} = \sum_{r=i}^{j} A[r]$ is the maximum sum over all possible i and j, $1 \leq i \leq j \leq n$ for both subarrays. Then linearly scan left and right from the midpoint to find the maximum subarray beginning in $s1$ and ending in $s2$. Finally, take the maximum of all three maximum sums.

```
# Assume arrays start at index 1
function func(A, i, j):
    if i == j
        return A[i]
    mid = (i + j)/2
    leftSum = func(A, i, mid)
    rightSum = func(A, mid+1, j)
    crossSum = helper(A, i, j, mid)

    return max(leftSum, rightSum, crossSum)



function helper(A,start,end, mid):
    leftSum = 0
    rightSum = 0
    totalSum  = 0

    for i = mid downto start
        sum += A[i]
        if sum > leftSum
            leftSum = sum

    sum = 0
    for j = mid +1 to end
        sum += A[j]
        if sum > rightSum
            rightSum = sum

    return leftSum + rightSum
```

The recurrence relation is $T(n) = 2T(n/2) + n$ for $n > 0$. By the master theorm this is $O(nlogn)$

b. Divide the array into two subarrays and recursively solve as in a. However, instead of just returning the maximum subarray, when we call our function, it will also return the maximum prefix and maximum suffix of the array. That is, the maximum prefix is k such that the $\sum_{j=1}^{k} A_j$ is maximum; similarly the maximum suffix is k such that the $\sum_{j=k}^{n} A_j$ is maximum. For our combine step, instead of linearly scanning over the array to find the maximum subarray beginning in $s1$ and ending in $s2$, we instead return the union of the maximum suffix of $s1$ and maximum prefix of $s2$. Then, take the maximum of all three maximum sums.
The recurrence relation is $T(n) = 2T(n/2) + c$ for $n > 0$ where n is the size of the array and comparing the sums take constant time. In this case, a = 2, b = 2, d = 0, meaning that $a = 2 > b^d = 1$. By the master's theorem, the (worst case) running time is O(n).

```
# This function returns six values:
# totalSum sum of A from i to j,
# prefixSum value of max of prefix,
# suffixSum value of max suffix,
# subSum value of optimal subsequence in A[i] to A[j],
# prefixEnd index of last element of the maximum prefix,
# suffixStart index of first element of the maximum suffix

function possibleSums(A, i, j):
    if i == j
        return (A[i], A[i], A[i], A[i], i, i)
    mid = floor((i + j)/2)
    left = possibleSums(A, i, mid)
    right = possibleSums(A, mid+1, j)

    # Sum of all elements in the array
    totalSum = left.totalSum + right.totalSum

    # Max possible prefix sum
    prefixSum = max(left.prefixSum, left.totalSum + right.prefixSum)

    # Max possible suffix sum
    suffixSum = max(right.suffixSum, left.suffixSum + right.totalSum)

    # Max subarray sum
    subSum = max(left.subSum, right.subSum,
                 left.suffixSum + right.prefixSum)

    ## Update left suffix and right prefix location
    # Default
    prefixEnd = left.prefixEnd
    suffixStart = right.suffixStart
```

```
if (prefixSum == left.totalSum + right.prefixSum){
    prefixEnd = right.prefixEnd
}
if (suffixSum == left.suffixSum + right.totalSum){
    suffixStart = left.suffixStart
}

return (totalSum, prefixSum, suffixSum, subSum, prefixEnd,
    suffixStart)
```