COMP3821 Assignment 1

May 3, 2021

1.1 **Solution:** We can sort the list S using MergeSort. For each query, we can use a binary search in a loop (with variable j as the loop variable) to find the index of the first element in the sorted list with a value greater than $S[j] + k$, storing the index in a variable x. If $x - j - 1 > 0$, then increment a count variable.

The count will return the number of unordered pairs. The algorithm will run in $O(n^2\, logn)$ overall since each binary search takes $O(logn)$, $O(n)$ for the loop and $O(n\, log\, n)$ for the MergeSort.

1.2 **Solution:** We can have an outer loop that goes through each integer z in S. The inner loop picks all the integers after z in the list and checks if the absolute difference between z and the inner integer is <= k. If true, a counter variable is incremented.

The count will return the number of unordered pairs. The algorithm will run in $O(n^2)$, $O(n)$ for the outer loop and $O(n)$ for the inner loop.

2. **Solution:** We can sort the list of integers in ascending order using MergeSort. We know that n is odd, so the median is located at the $n/2$th index. We have a while loop that decrements k until k is 0.

At every iteration, it checks if the median is less than the number to its immediate right (let's call this variable j). If so, the median is incremented by one until it is equal to the number to its right. If the median is equal to j, the algorithm traverses the array from the median until it finds a number greater than the median. If this number does not exist, the number at the last index is incremented. If the number exists, the number to its immediate left is incremented.

The resulting array will have the maximum possible median. This algorithm runs in $O(n \, log \, n)$ time, because the MergeSort takes $O(n \, log \, n)$ time and the median algorithm takes $O(k + n/2)$ (which is $O(n)$) time.

3.1 **Solution:** To find the largest sum in $O(n^3)$ time, we can use a brute force approach using 3 nested loops. The maximum sum is stored outside the loops with an initial value of 0.

The outermost loop picks a number with index i in the array. The middle loop picks a number with index j and has a temporary sum variable contained in it. The innermost loop will produce all sums between k=i and j. If any of these sums are greater than the maximum sum, the maximum sum is updated.

3.2 **Solution:** To find the largest sum in $O(n^2)$ time, we can use a brute force approach using 2 nested loops. The maximum sum is stored outside the loops with an initial value of 0.

The outer loop picks an initial number with index i, and the inner loop with variable j contains a temporary sum variable. The inner loop calculates all the possible sums from i to j. If any of these sums are greater than the maximum sum, the maximum sum is updated.

3.3 **Solution:** To find the largest sum in$O(n \, log \, n)$ time, we can use a divide-and-conquer approach. We know that the sub array that has the maximum sum can be in one of 3 situations:

- Situated such that the sub array crosses the midpoint
- Situated on the right side of the array, between the midpoint and the end
- Situated on the left side of the array, between the start and the midpoint-1

Using the divide-and-conquer approach:

- Divide the array into two halves
- Recursively calculate the maximum sum for the right sub array
- Recursively calculate the maximum sum for the left sub array
- Calculate the maximum sum of the sub-array that crosses the midpoint. To do this, we can find the sum from the midpoint to a point to the right of the midpoint (rightSum). We can also find the sum from the midpoint-1 to some point left of the midpoint (leftSum). Find and return the maximum

among rightSum, leftSum, and rightSum + leftSum, which we will called crossed.

- Return the maximum sum amongst the right, left and crossed sums. If the sum is negative return 0 (value of the empty subarray).The returned value will be the largest sum over all contiguous sub arrays of S. This algorithm takes $O(n\, log\, n)$ time.

Pseudo-Code:

```
int subarraySum (int low, int high, int array[])
{
  // only one element in the array
  if (high == low)
    return array[low]
  else
     {
      // split the array
      int midpoint = low + (high - low)/2

      // find sum on both halves
      int rightSum = subarraySum (midpoint, high, array[])
      int leftSum = subarraySum (low, midpoint-1, array[])

      // to find the crossed sum
      int crossSum = crossedSum(low, high, midpoint, array[])

      return max (leftSum, rightSum, crossSum)
     }
}


int crossedSum(int left, int right, int midpoint, int array[])
{
  inr rSum = 0
  int rightSum = 0
  for(k = mid to right)
   {
      rSum = rSum + array[k]
      If (rSum > rightSum)
      rightSum = rSum
   }
  int lSum = 0
  int leftSum = 0
  for(k = midpoint-1 to left)
   {
      lSum = lSum + array[k]
      If (lSum > leftsSum)
      leftSum = lSum
   }
// return largest of the three values
return max(leftSum, rightSum, leftSum + rightSum)
}
```

3.4 **Proof:** Let S be a list of n integers. The algorithm as described in 3.3 (lets call it ms(S)), will return the largest contiguous sum in S, and max(0,S) if the sum is negative or S is empty.

Proof by induction on n-1.
Base Case: Let n=1.
When n=1, the list S contains a single number. It will return max (0,S). Therefore, ms(S) works when n=1.

Inductive Hypothesis: Assume that ms(S) gives the correct answer for any list S of length n-1 for some $n > 1$.

Inductive Step: We will show that ms(S) for any list of length n.
Let Left and Right denote the subarrays resulting from splitting the array in half. Furthermore, let the max(leftSum, rightSum, crossSum) <= x (sum of the largest sub array).

If the largest sum subarray lies completely on the left side, then from our hypothesis ms(Left) = x. Therefore, the max(leftSum, rightSum, crossSum) = x. If leftSum > x then there is a larger sum, which contradicts the finding that ms(Left) = x for the entire array. If x > leftSum then ms(Left) did not compute the right answer, as the answer lies entirely in the left side of the array and the recursive calls should have ensured this. ms(Left) must be > ms(Right) and ms(Left) > ms(Crossed), otherwise ms(Left) is not equal to x, contradicting the fact that it is the largest sum.

Similarly we can prove for the right side as it is symmetric to the left.

If the above two cases are not met, then the sub array starts in the Left and ends in Right. Then, crossSum = x. If crossSum > x, then the ms algorithm has found a larger sub array sum, contradicting or findings that crossSum in the largest sum in the array. If x > crossSum, the crossedSum function did not cross the middle and thus did not return the right value. Likewise, we can say if crossSum < leftSum, then there is a larger sum in the array, contradicting the fact that crossSum = x. Similarly we can show the contradiction for rightSum.

For all 3 cases ms(S) finds the largest sum in the subarrays, hence by induction ms(S) is correct.

3.5 **Solution:** According to the Master Theorem and after analysing the algorithm, it is clear that the running time is evenly distributed throughout the recurrence tree so $T(n)$ belongs to $\Theta(n^k * log\, n)$ when the recurrence $T(n) = aT(n/b) + cn^k$.
The recurrence relation formed by the divide and conquer algorithm is : $T(n) = 2 * T(n/2) + O(n) = O(n\, logn)$.

4.1 **Solution:**

$f(n) = \sum(i * 2^{i-1}) = n * 2^n - 2^n + 1$
$g(n) = 3^n$

Let's assume $g(n) > c * f(n)$.

Therefore,
$3^n > c * n * 2^n - 2^n + 1$

Applying log to both sides,

$log(3^n) - log(1) > c * log(2^n n - 2^n)$

$n\, log(3) > c * n\, log(2) + log(n) - n\, log(2)$

Since we know that $n > log(n)$,

Therefore $n\, log(3) > c * log(n)$.

Hence our initial assumption was true. Thus, $f(n) = O(g(n))$.

4.2 **Solution:**

$f(n) = log_2 n^{10}$
$g(n) = log_{1000} n^{0.001}$

Let's assume that $g(n) > c * f(n)$

Therefore,

$log_2 n^{10} > c * log_2 n^{10}$

Take $c = 1$,

$1 > log_2 n^{10} / log_{1000} n^{0.001}$

Applying L'Hˆopital's rule to compute the limit,

$$\lim_{n\to\infty} log_2 n^{10} / log_{1000} n^{0.001}$$

=

$$\lim_{n\to\infty} (log_2 n^{10})' / (log_{1000} n^{0.001})'$$

=

$$\lim_{n\to\infty} (10/n * ln(2)) / (0.001/n * ln(1000))$$

=

$$\lim_{n\to\infty} 10 * ln(1000) / (0.001/n * ln(2)) > 1$$

Thus, our initial assumption was false and in fact, $f(n) > c * g(n)$.

Thus, $f(n) = \Omega(g(n))$.

4.3 **Solution:**

$f(n) = n * (2 + sin(n))$
$g(n) = \sum log_2 n / 2^i$

Now, since sine's maximum and minimum values are -1 and 1 respectively, $n <= f(n) <= 3n$.

$g(n)$ can also be written as: $log_2 n(1 + 1/2 + 1/4 + 1/6 + ...)$

We know that $n * log(n) > n$,

Therefore, $g(n) > c * f(n)$. Thus, $f(n) = O(g(n))$.

End.