CO580 Algorithms Course Work 2

Konstantinos Ntolkeras

March 5, 2019

1 First Question

1.a

A naive algorithm to compute C that breaks when it finds the element of A that it is searching for in B will have upper bound of time complexity O(MxN). In the worst case scenario array A and B will have no common elements. Therefore, the algorithm will have to go through the entire array B for each element in A resulting in MxN iterations. In the best case, all the elements in A are also contained in B. In the case that $M < N$ or $M = N$ then we end up with the following arithmetic series: 1+2+3+...+M which is equal to $\frac{M(M-1)}{2}$ and since M^2 is the dominant term of the polynomial we end up with $\Omega(M^2)$. In the case that $M > N$ then we end up with 1+2+...+N+N*(M-N) which is equal to $MN * \frac{N(N-1)}{2}$. The term MN is asymptotically the biggest term of the polynomial so we end up with $\Omega(MN)$.

1.b

The following algorithm, implemented in Java, has upper bound time complexity O(M+N). The worst case occurs when the 1st element of A is bigger than the last element of B.
The algorithm has lower bound time complexity of $\Omega(M)$. The best case scenario occurs when the 1st element of B is bigger than the last element of A, or when the two arrays have the same elements and the same sizes, or when A is bigger than B and array B has the same elements as A, or when B is bigger than A and A has the same elements as B.

```
public static int[] Difference(int A[], int B[]) {
    int C[] = new int[A.length];
    int count = 0;
    int j = 0;
    int M = A.length;
    int N = B.length;

    for (int i = 0; i < M; i++) {
      while (j < N && A[i] > B[j]) {
        j++;
      }
      if (j >= N) {
        for (int m = i; m < M; m++) {
          C[count] = A[m];
          count++;
        }
        return C;
      }
      if (A[i] < B[j]) {
```

```
            C[count] = A[i];
            count++;
          }
          if (A[i] == B[j]) {
            j++;
          }
        }
        return C;
    }
```

2 Second Question

2.a

The following algorithm, implemented in Java, runs in $O(N^2)$ time and returns the length of the longest strictly increasing sequence within an array A.

```
public static int LONGEST(int A[]) {
    int N = A.length;
    int L[] = new int[N];
    for (int i = 0; i < N; i++) {
      L[i] = 1;
    }
    for (int i = 1; i < N; i++) {
      for (int j = 0; j < i; j++) {
        if (A[i] > A[j] && L[i] < L[j] + 1) {
          L[i] = L[j] + 1;
        }
      }
    }
    int max = 0;
    for (int count = 0; count < N; count++) {
      if (max < L[count]) {
        max = L[count];
      }
    }
    return max;
}
```

2.b

If the solution was implemented recursively, without using dynamic programming, the upper bound time complexity would be $O(2^N)$ where N is the size of the array. The lower bound of the time complexity would be $\Omega(2^N)$ as we would have to go through the entire array in every case in order to find the longest possible subarray. The tree at Figure 1 helps us understand the problem at hand. The time the algorithm takes to return the length of the longest strictly increasing sequence within an array A of size N is 2^{N-1} which is equal to $\frac{2^N}{2}$. Therefore, we end up with $\Omega(2^N)$ and $O(2^N)$ and as a consequence $\Theta(2^N)$.

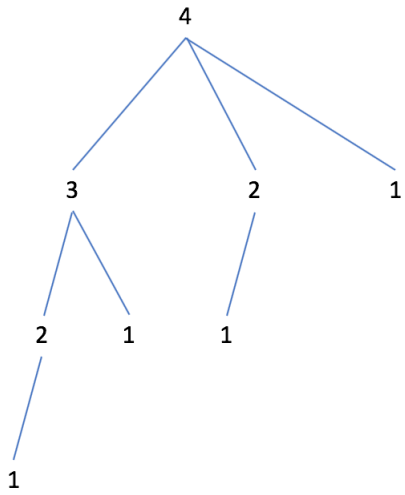

Figure 1: Recursion tree for an array of size 4