CSL351: Analysis and Design of Algorithms
Assignment No. 2

Sahil
2016UCS0008

September 5, 2018

1 Problem 1: Monotonically increasing functions:

It is given that $f(n)$ and $g(n)$ are monotonically increasing integer-valued functions. We need to prove whether $f(n) + g(n)$, $f(n) * g(n)$ and $f \circ g(n)$ are monotonically increasing or not.

Since $f(n)$ and $g(n)$ are monotonically increasing functions, thus

$$\forall n_1, n_2 \;\; s.t. \; n_1 \leq n_2, \;\; f(n_1) \leq f(n_2) \qquad \rightarrow ①$$
$$\forall n_1, n_2 \;\; s.t. \; n_1 \leq n_2, \;\; g(n_1) \leq g(n_2) \qquad \rightarrow ②$$

1.1 $f(n) + g(n)$

From ① and ②,

$$\forall n_1, n_2 \;\; s.t. \; n_1 \leq n_2, \;\; f(n_1) + g(n_1) \leq f(n_2) + g(n_2) \qquad \rightarrow ③$$

From ③, we can conclude that $f(n) + g(n)$ is a monotonically increasing function.

1.2 $f(n) * g(n)$

To check whether $f(n) * g(n)$ is a monotically increasing function, we **can't** directly multiply ① and ② and say that

$$\forall n_1, n_2 \;\; s.t. \; n_1 \leq n_2, \;\; f(n_1) * g(n_1) \leq f(n_2) * g(n_2)$$

This is **not true** because it might be the case that $f(n_1)$ and $g(n_1)$ are both negative, and their product comes out to be larger than $f(n_2) * g(n_2)$. Taking a concrete example:
Let $n_1 \leq n_2$ and $f(n_1) = -1$, $g(n_1) = -10$, $f(n_2) = 1$, $g(n_2) = 5$,
We have $f(n_1) \leq f(n_2)$ and $g(n_1) \leq g(n_2)$, but $f(n_1) * g(n_1) \geq f(n_2) * g(n_2)$.
Thus, **f(n) * g(n)** is not monotonically increasing function. To make it monotonically increasing, we must have $f(n) \geq 0$ and $g(n) \geq 0$.

1.3 $f \circ g(n)$

From ②, we have $g(n_1) \leq g(n_2)$, and since $f(n)$ is monotonically increasing, thus, $f(g(n_1)) \leq f(g(n_2))$. Hence, $f \circ g(n)$ is a monotonically increasing function.

2 Problem 2: Identifying matching pair of gloves:

The smallest no. of gloves required to have atleast one matching pair in the best case is 2 since we might end up picking the pair when we pick up the first two gloves.
In the worst case, we need to pick atleast 10 gloves to ensure that we pick atleast one matching pair. This is because there are 4 pair of red gloves, 4 pairs of yellow and 1 pair of green. So, we might end up picking one glove from each pair during the first 9 times, i.e. we have 9 gloves each from all 9 different pairs, so during the 10th time, we will obviously have 1 glove picked which forms a pair with the already picked ones.

Psuedo code:

```
// Consider red, green, yellow = arrays of size 2 each
// red[0] = true means we have left glove of red color
// red[1] = true means we have right glove of red color
// similarly for each of the three arrays
// Intially, all the elements in the arrays are false
// n is the total number of glove pairs
count ← 0 ;
// count is the number of steps taken to find matching pair
for i ← 1 to 2 * n do
    Pick a random glove ;
    Mark the appropriate element in the correct color array as true ;
    if Opposite glove of same color exists then
        // we have find the matching pair
        return ;
    end
    count ← count + 1 ;
end
```

Algorithm 1: Matching pair of gloves problem

3 Problem 3: Selection sort analysis:

3.1 Pseudocode:

```
Consider a = input array of size n ;
// Assuming array to be 0-indexed
for $i \leftarrow 0$ to $n-2$ do
    $min\_index \leftarrow i$ ;
    // min_index represents the index at which minimum element
    // of sub-array starting from i-th index occurs
    for $j \leftarrow i+1$ to $n-1$ do
        if $a[j] < a[min\_index]$ then
            min_index = j ;
        end
    end
    swap($a[i], a[min\_index]$) ;
end
```

Algorithm 2: Selection sort

3.2 Loop invariant:

During end of each iteration of the outer loop on i, the element at $i-th$ index of the array is occupied by the correct element which would have been at that index when the array was sorted in non-decreasing order.

We can prove this by strong induction on i.
Base case: When $i = 0$, the inner loop on j finds the entire minimum element of the array and at the end, it is swapped with the first index element of the array, thus the first element in the array is the minimum element after the first iteration is over.
Inductive case: We assume that the loop invariant holds for all i from 0 to $k-1$. Thus, the first k positions in the array are occupied by the k smallest numbers in the array.
Now, we need to show that this holds for $i = k$. So, i.e. in the $(k+1)-th$ iteration, we need to show that the correct element will occupy the $k-th$ index of the array as in sorted order.
Clearly, the inner loop finds the minimum value of the array from indices k to $n-1$, which must be the $(k+1)$th smallest element of the array since k

smallest numbers are already present in previous indices as per hypothesis. Thus, this element gets swapped and occupies the correct position.

Hence, our loop invariant is verified.

3.3 Runs for first $n-1$ elements:

It need to run only for the first $n-1$ elements since after the first $n-1$ iterations, the first $n-1$ elements in the array will be correctly occupied by elements as in the sorted array. This holds true by using the loop invariant. So, the last element must obviously be in the correct sorted order as it has no other place left in the array.

3.4 Best-case running time:

In the best case we might end up doing zero swaps. So, this will be the case when the array is already sorted in non-decreasing order. Still, we will be doing $n-i-1$ comparisons in each run of the outer loop of i. Thus, total no. of comparisons =

$$\sum_{i=0}^{n-2} n-i-1 = n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) = \frac{n(n-1)}{2}$$

Thus, best case running time is $\Theta(n^2)$.

3.5 Worst-case running time:

In the worst case we might end up doing some swaps and assignments also. Since we can perform atmost one swap in each iteration over i, thus, we cannot do more than $\Theta(n)$ swaps. Also, we do assignments to the variable **min_index** only when the comparison resulted in true. Since there are $\Theta(n^2)$ comparisons, thus there cannot be more than $\Theta(n^2)$ assignments. Thus, the overall complexity in the worst case is $\Theta(n^2)$.

4 Problem 4: Asymptotical analysis of maximum of two functions:

It is given that $f(n)$ and $g(n)$ are asymptotically non-negative functions. We need to prove that

$$max\,\{f(n), g(n)\} = \Theta(f(n) + g(n))$$

Using the basic definition of Θ notation: Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers, we say that $f(n)$ is $\Theta(g(n))$ **iff** $f(n)$ is $O(g(n))$ and $f(n)$ is also $\Omega(g(n))$. Now, $f(n)$ is $O(g(n))$ **iff**

$$\exists\ c > 0\ \&\ n_0 \geq 1\ s.t.\ \forall n \geq n_0,\ \ f(n) \leq c\ g(n)$$

So, first we show that $max\,\{f(n), g(n)\}$ is $O(f(n) + g(n))$.
Since, $f(n) \leq f(n) + g(n)$ and $g(n) \leq f(n) + g(n)$, thus,

$$max\,\{f(n), g(n)\} \leq f(n) + g(n)\ \forall n$$

This is because $max\,\{f(n), g(n)\}$ has to be either of the two $f(n)$ or $g(n)$ depending on which one is larger.
Thus, $max\,\{f(n), g(n)\}$ is $O(f(n) + g(n))$ where $c = 1$ and $n_0 = 1$.

Now, we show that $max\,\{f(n), g(n)\}$ is $\Omega(f(n) + g(n))$.
Since $f(n) \leq max\,\{f(n), g(n)\}$ and $g(n) \leq max\,\{f(n), g(n)\}$, thus,

$$f(n) + g(n) \leq max\,\{f(n), g(n)\} + max\,\{f(n), g(n)\}$$

$$\Rightarrow f(n) + g(n) \leq 2 * max\,\{f(n), g(n)\}$$

$$\Rightarrow max\,\{f(n), g(n)\} \geq \frac{1}{2}(f(n) + g(n))\ \forall n$$

Thus, $max\,\{f(n), g(n)\}$ is $\Omega(f(n) + g(n))$ where $c = 0.5$ and $n_0 = 1$.

Having proved that $max\,\{f(n), g(n)\}$ is $\Omega(f(n) + g(n))$ and also $O(f(n) + g(n))$, we can now conclude from the basic definition of Θ notation that, $max\,\{f(n), g(n)\}$ is $\Theta(f(n) + g(n))$.

5 Problem 5: Brute force method for maximum subarray problem:

In the maximum subarray problem, we are given an array **A** consisting of **n** integers. We have to find the non-empty, contiguous sub array of **A** which has the largest sum of elements.

So, the brute force algorithm is to check the sum of all the sub arrays and find which one has the largest sum. Let us denote the starting index of a subarray by **start** and ending index of a subarray by **end**. Clearly, there are $\frac{n(n-1)}{2}$ subarrays possible. Thus, if we can find sum of each subarray in $\Theta(1)$, our algorithm would have time complexity of $\Theta(n)$.
To do so, we can keep the **start** variable fixed for a iteration, and move the **end** variable till the size of the array, and keep on incrementing the sum of the subarray from $[start, end]$ where **end** is varying. Now, after **end** cannot vary further, we then increment **start** and again make **end** vary from the position after **start** by initializing the sum to zero and keep on incrementing sum and checking whether it exceeds maximum sum.

The pseudo code for the above algorithm thus is:

```
Consider a = input array of size n ;
// Assuming array to be 0-indexed
$max\_sum \leftarrow 0$ ;
for $start \leftarrow 0$ to $n-1$ do
    $end \leftarrow start$ ;
    $current\_sum \leftarrow 0$ ;
    while $end < n$ do
        $current\_sum \leftarrow current\_sum + a[end]$ ;
        if $current\_sum > max\_sum$ then
            $max\_sum \leftarrow current\_sum$ ;
        end
        $end \leftarrow end + 1$ ;
    end
end
// max_sum variable now contains the maximum sum of subarray
```

Algorithm 3: Maximum sub-array problem

The above algorithm runs in $\Theta(n^2)$ time.