**Computer Algorithms – HW1**

**1. Sort the following functions from asymptotically smallest to asymptotically largest,**

(a) 1 << log(n ^ n) << n choose 2(binomial coefficient) << n^3 + n^2 << n!

(b) 1/n << n^2^2 << n ^ log n << (log n) ^ n << 2^2^n

**Order considering all 10 functions together:**

1/n << 1 << log(n ^ n) << n choose 2(binomial coefficient) << n ^ 3 + n ^ 2 << n ^ 2 ^ 2

<< n ^ log n << (log n) ^ n << n! << 2 ^ 2^ n

**2. (a) Suppose an algorithm A1 divides (in linear time) the input into six equal parts: P1, P2, P3, P4, P5, P6. It then makes the following 10 combinations of these parts: each odd part is combined with each even part: P1 with P2, P4, P6 and so on, and in addition P1 is combined with P3. The algorithm is then run recursively on each of those 10 combinations. The results of the recursive runs are combined in quadratic time to produce the final output.**

* T(n) = a T(n / b) + f(n)
* In this case - b = 3 (Since input is divided into 6 parts and 2 of those parts are combined), a = 10, f(n) is quadratic - (n^2)
  + T(n) = 10 T(n / 3) + (n^2)
* By Master Theorem – if f(n) = O(n ^ k) for some constant k < log(a base b), then
* T(n) = Θ(n ^ log(a base b))
* In our case k = 2, a = 10, b = 3, log (10 base 3) = 2.09 > k
* Therefore Finally, T(n) = Θ(n ^ log(10 base 3))

**(b)** **Another algorithm, A2, divides the input into nine equal parts and runs on a (some number) of them recursively. It then combines the results of the recursive runs in quadratic time to produce the final output.**

By Master Theorem - T(n) = a T(n / b) + f(n)

* In this case - b = 9, a = a, f(n) is quadratic - (n^2) Therefore,
  + T(n) = a T(n / 9) + (n^2)

**3. Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (a) Describe an O(n)-time algorithm for computing a sequence representing the set A ∩ B**

**Brief Description:**

Algorithm starts from the first element of sequences A and compares it with first element of sequence B

1. If values are the same and if the value not equal to lastappended-value, the element is added to the output list and lastappended-value is updated
2. If values are not equal, the next element is traversed in the sequence which had the smaller value in the comparison
3. This iteration is continued till the last element is traversed for any one of the sequences

**Assumption:** A & B are sorted in increasing order

n <- Length of sets A & B, i <- 0, j <- 0

For i < n and j < n

if A[i] = B[j]

if A[i] = lastappended-value

i= i+1, j=j+1

else

append A[i] to output list - A ∩ B

i = i+1, j = j+1

lastappended-value = A[i]

end if

else if A[i] < B[j]

i = i+1

else

j = j+1

end if

end

**Proof of Correctness:**

The algorithm checks for one element in Sequence A with one element in Sequence B and adds it to the output list only if they are equal,

1. Hence there is no case where an element which is not present in both the list will get appended
2. No Duplicate values will be present in the final output list, since the value is checked with the last appended value and since the sequences are sorted, we need not check the entire output list before appending
3. Only way an element which is present in both sequences A and B, but not in output is if the comparison was not executed

* But since the loop runs till the end of any one of the sequence is reached
* And since sequences are sorted and only the sequence which had the lesser value was iterated to the next element, Such a case cannot occur

**Proof of Termination:**

1. For each iteration of the loop, we iterate to the next element for atleast one of the sequences
2. We do not go back to the previous element in any case
3. Algorithm will terminate when the end of any one sequence is reached which can be a maximum of (n + n) sequences

**Asymptotic Notation:**

Since in the algorithm with each run of the for loop, we traverse only forward in either Sequences A or B

The maximum number of times the loop can run is n + n times - hence the time complexity is - O(n)

Similarly maximum space required is n+n for saving two sequences - space complexity – O(n)

**(b) If the sequences A and B are instead unsorted, what is asymptotic time complexity of the most efficient algorithm that you can construct that computes A ∩ B? Describe this algorithm**

**Brief Description:**

Both sequences are sorted using merge sort

Then the same algorithm as above is used to identify the intersection of the two sorted sequences

1. Sort sequence A using merge sort algorithm

/\* Source - http://www.geeksforgeeks.org/merge-sort/ \*/

For sequence A - A[l...r]

MergeSort(A, l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The merge() function is used for merging two halves.

The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and

merges the two sorted sub-arrays into one

2. Similarly sort sequence B using merge sort algorithm

3. Use algoritm in 3(a) to join get A ∩ B

**Proof of correctness and Termination:**

Same as above

**Asymptotic Notation:**

1. Sorting A and B - O(nlog n) /\* Merge sort time complexity \*/
2. Finding A ∩ B - O(n)
3. Total time complexity - O(nlogn)
4. Total Space complexity – O(n)

**4. Graph G has following adjacency lists: 1 − (4;2) 2 − (5;3) 3 − (6;4) 4 − (7;5) 5 − (1;6) 6 − (2;7) 7 − (3;1)**



