**Lab 2**

1. Compute the number of primitive operations for each of the following algorithm fragments.
   1. sum ← 0 1 (assignment operation)

**for** i ← 0 **to** n-1 **do** n + 2 (1 assignment and n + 1 comparison)

sum ← sum + 1 2n (n assignments and n additions)

**T(n) = 3(n + 1)**

* 1. sum ← 0 1

**for** i ← 0 **to** n-1 **do** n + 2

**for** j ← 0 **to** n-1 **do** (n + 2)n

sum ← sum + 1 2n2

**----------------------------------------------------------------------------------------------------------------------------------------------------------------- T(n) = 3(n2 + n + 1)**

1. Determine the asymptotic running time of the following procedure (an exact computation of number of basic operations is not necessary):

int[] arrays(int n) {

int[] arr = new int[n];  **1**

for(int i = 0; i < n; ++i){ **3n + 2** (n + 1 comp, n incr, 1 assig)

arr[i] = 1; **2n** (n array indexing, n assignments)

}

for(int i = 0; i < n; ++i) { **3n + 2** (outer loop runs n times)

for(int j = i; j < n; ++j){ **(3n/2 + 2)n** (inner loop avg run = n/2)

arr[i] += arr[j] + i + j; **(6n2/2)** (3 add, 2 arr ind, 1 assig)

}

}

return arr;  **1**

}

----------------------------------------------------------------------------------------------------------

**T(n) = 9/2n2 + 10n + 6**

Asymptotic analysis of T(n) or representation of T(n) in big-oh notation results in dropping of lower order terms and constant factors. Hence we get:

**T(n) is O(n2)**

In fact, determining order of T(n) could be easily done by using simple rules: for single loop, take O(n), like the first loop, for nested loop like the second part, we have O(n2). For consecutive parts, use addition: which gives us T(n) = O(n) + O(n2), and then dropping the lower order, we can easily arrive at T(n) = O(n2).

1. Algorithm to merge two arrays of integers and then sort the resulting array

**Alogorithm** merge(A, n, B, m)

**Input**: Two arrays A and B, with lengths n and m consecutively

**Output**: Array C, with length n + m and elements sorted in ascending order

C 🡨 combine(A, n, B, m) **1**

sort(C, k) **1**

**Algorithm** combine(A,n, B, m)

**Input**: Two arrays of integers with length n and m

**Output**: An array with includes all the elements of both input arrays

K 🡨 n + m **2**

C 🡨 new Array[k] **1**

**for** i 🡨 0 **to** n **do n + 2**

**C[i] 🡨 A[i] 3n**

**for** i 🡨 0 **to** m **do m + 2**

**C[n + i] 🡨 B[i] 3m**

**return** C **1**

**Algorithm** minSort(C, k)

**Input**: Array of integers with length k

**Output**: Array of integers with elements sorted in ascending order

**for** i 🡨 0 **to** k-2 **do k + 1**

min 🡨 C[i] **k - 1**

**for** j 🡨 i + 1 **to** k-1 **do (k/2)(k-1)**

**if** C[j] < min **then k/2(k-1)**

min 🡨 C[j] **k(k-1)**

C[j] 🡨 C[i] (**3k/2)(k-1)**

C[i] 🡨 min 2**k-2**

**return** C **1**

**c) Java Program implementing the algorithm described above**

**package** lab2;

**import** java.util.Arrays;

**public** **class** ArrayMerger {

**int**[] merge(**int**[] arr1, **int**[] arr2){

**int**[] arr = combine(arr1, arr2);

minSort(arr);

**return** arr;

}

**int**[] combine(**int**[] arr1, **int**[] arr2){

**int** len1 = arr1.length;

**int** len2 = arr2.length;

**int**[] arr = **new** **int**[len1 + len2];

**for**(**int** i = 0; i < len1; i++){

arr[i] = arr1[i];

}

**for**(**int** i = 0; i < len2; i++){

arr[len1 + i] = arr2[i];

}

**return** arr;

}

**void** minSort(**int**[] arr){

**int** min;

**for**(**int** i = 0; i < arr.length - 2; i++){

min = arr[i];

**for**(**int** j = i + 1; j < arr.length - 1; j++){

**if**(arr[j] < min){

min = arr[j];

arr[j] = arr[i];

}

}

arr[i] = min;

}

}

**public** **static** **void** main(String[] args){

**int**[] arr1 = {1, 4, 5, 8, 17};

**int**[] arr2 = {2, 4, 8, 11, 13, 21, 23, 25};

ArrayMerger am = **new** ArrayMerger();

**int**[] arr = am.merge(arr1, arr2);

System.***out***.println(Arrays.*toString*(arr));

}

}

1. **Big-oh and Little-oh.** Decide whether each of the following is true or false, and in each case, prove your answer.
2. 1 + 4n2 is O(n2)

**True**: Because, for c = 5 and n > 1, 1 + 4n2 <= cn2

1. n2 - 2n is not O(n)

**True**: =

Since the limit is not a finite value, n2 - 2n is not O(n)

1. log(n) is o(n)

**True**: Because for any constant c > 0, we can always find another constant n0 > 1/c for which the following statement is true at all n > n0:

log(n) < cn

Or, using limits and L’ Hopitals theorem

= = 0 which proves the statement

1. n is not o(n)

**True**: = 1 and not 0.

1. **Big-oh Rules**. Suppose you know that f(n) and g(n) are both O(h(n)). Does it follow that the function f(n) +g(n) is also O(h(n))? Prove your answer.

True. The proof is:

Since both f(n) and g(n) are both O(h(n)), it follows that:

**f(n) <= ch(n)** for some constant c > 0, and n >= n0

**g(n) <=c’h(n)** for some constant c’> 0 and n >= n’0.

So, for **k = c + c’** and for n >= max(n0, n’0):

**f(n) + g(n) <= ch(n) + c’h(n) = (c + c’ )h(n) = kh(n)**

This proves that **f(n) +g(n) is O(h(n)).**

1. **Power Set Algorithm:**

**Algorithm**: PowerSet(X)

***Input***: A list X of elements

***Output***: A list P consisting of all subsets of X – elements of P are *Sets*

P ← new list

S ← new Set //S is the empty set

P.add(S) //P is now the set { S }

T ← new Set

**while** (!X.isEmpty() ) **do**

f ← X.removeFirst()

**for each** x **in** P **do**

T ← x U {f} // T is the set containing f & all elements of x

P.add(T)

**return** P

**Implementation of the PowerSet algorithm using Java**

**public** List<Set<Integer>> powerSet(Set<Integer> inputSet){

List<Integer> input = **new** ArrayList<>(inputSet);

//This is P - final result, list containing all subsets

List<Set<Integer>> output = **new** ArrayList<>();

//This is S - emptySet

**final** Set<Integer> emptySet = **new** HashSet<>();

output.add(emptySet);

List<Set<Integer>> temp = **new** ArrayList<>();

Set<Integer> t;

**while**(!input.isEmpty()){

//Get and remove the first element from the input list

Integer f = input.remove(0);

**for**(Set<Integer> x : output){

//Add f to t if it's not already there -- union

t = **new** HashSet<>(x);

t.add(f);

temp.add(t);

}

output.addAll(temp);

temp.clear();

}

**return** output;

}

1. Prove by induction that for all n > 4, Fn > (4/3)n. Then use this result to explain the approximate asymptotic running time of the recursive algorithm for computing the Fibonacci numbers. Is the recursive Fibonacci algorithm fast or slow? Why?

**Solution:**

Let’s attempt to prove that by using total induction. Assume that Fi > (4/3)i is true. If we can prove that Fn > (4/3)n assuming that it’s true for all i > 4, to any i < n.

For i=5: F5 = 5 > (4/3)5 = 4.21

For all n > 5, Fn-1 > (4/3)n-1, and Fn-2 > (4/3)n-2 are true by our assumption.

Fn = Fn-2 + Fn-1 > (4/3)n-2 + (4/3)n-1 ……………………………….(1)

But by general induction, we can prove that (4/3)n-2 + (4/3)n-1 > (4/3)n for all n > 4.

It is true for the base case where n = 5: (4/3)n-2 + (4/3)n-1 = 2.37 + 3.16 = 5.53 > (4/3)n = 4.21. Then assuming that Pn-1 is true we have:

(4/3)n-3 + (4/3)n-2 > (4/3)n-1 ……………………………….(2)

Multiplying both sides of (2) by (4/3) should preserve the inequality in (2), i.e.,

(4/3)(4/3)n-3 + (4/3)(4/3)n-2 > (4/3)(4/3)n-1 …. which leads to:

(4/3)n-2 + (4/3)n-1 > (4/3)n which proves that Pn is also true.

Now, if we replace the left side of (1) by (4/3)n, the inequality statement in (1) should still hold. This gives us:

Fn = Fn-2 + Fn-1 > (4/3)n-2 + (4/3)n-1 > (4/3)n, proving the statement Pn by total induction.

The growth rate of Fibonacci series is much larger than growth rate of of n as n grows bigger.

1. Develop an algorithm that does the following: Given an integer n, the algorithm outputs a randomly generated array in which each of the integers 0, 1, 2, …, n-1 occurs exactly once. Express your algorithm in pseudo-code. Determine (or estimate) the asymptotic running time of your algorithm.

**Algorithm**: RandomArray(n)

***Input***: Size of array given as n

***Output***: A randomly generated array of length n, and all elements from 0 .. n-1 appearing just once.

A ← new array of length n

S ← new Set //S is the empty set

T ← new Set

R ← new Random integer generator

**while** (S.size () < n ) **do**

f ← R.nextInt(n) //Generate next random integer less than n

S ← S U {f} //retain it in the set if it’s not already there

**A**←S.toArray(A) //Copy contents of set to array A

**return** A

To find out the expected number of iterations needed to get the first such array of integers can be computed using probability theory:

The random integer number generator used in the algorithm above generates one integer from the range of 0 to (n-1). There are n different possible outcomes. The probability that the first number generated is unique is 1. We have (n-1) out of n chance of getting the second unique number, and (n-2) out of n chance to get the third unique, and so on. Each of the random integer generation events is independent from others. So, the probability of getting n unique integers generated is given by:

Let Y be a random variable denoting the number of trials. The expected value of Y or E[Y] is then given by:

The running time of the algorithm is hugely influenced by number iterations required to get n random numbers than the constant factors (primitive operation counts within the iteration plus).

So, the running time is asymptotic to , or in other words it’s O(.

1. More big-oh: (Work with someone in your group who is familiar with limits)

a. True or false: 4n is O(2n). Prove your answer.

True: = 2, which is < .

b. True or false: log n is Θ(log3 n). Prove your answer.

True: , which is finite real number > 0

c. True or false: (n/2) log(n/2) is Θ(nlog n). Prove your answer.

True: , which is finite real > 0

1. Below, pseudo-code is given for the recursive factorial algorithm recursive Factorial.

**Algorithm** recursiveFactorial(n)

***Input***: A non-negative integer n

***Output***: n!

**if** (n = 0 || n = 1) **then**

**return** 1

**return** n \* recursiveFactorial(n-1)

Do the following:

1. Use the Guessing Method to determine the worst-case asymptotic running time of this algorithm. Then verify correctness of your formula.

T(0) = T(1) = 2 (comparison plus return)

T(2) = T(1) + 2 = 4 (multiplication plus return)

T(3) = T(2) + 2 = 6

….

T(n) = T(n-1) + 2 = 2n for n > 1

This can be proved by induction, by taking base case of 2. Assuming T(n) is true for all values less than n,

T(n) = T(n-1) + 2 = 2(n-1) + 2 = 2n, which proves the claim

1. Prove the algorithm is correct.

To prove the algorithm is correct, first we check if it’s valid. Does it converge to the base case eventually? The answer is yes. Next, we check if the base case is correct. Is the base case value correct (for 0 or 1)? Yes. Then, is the recursion correct? The answer is yes … n! = n\*(n-1)!. So, the algorithm is correct.

1. Devise an iterative algorithm for computing the Fibonacci numbers and compute its running time. Prove your algorithm is correct.

**Algorithm** iterativeFibonacci(n)

***Input***: A non-negative integer n

***Output***: Fibonnaci of n

**if** (n = 0) **then**

**return 0**

**if** (n = 1) **then**

**return 1**

a 🡨 0

b 🡨 1

temp 🡨 0

**for** i 🡨 2 **to** n **do** n

temp 🡨 b (n-2)

b 🡨 b + a 2(n-2)

a 🡨 temp (n-2)

**return** b

The above algorithm has one for loop which runs for n-2 iterations. In such scenarios, the constant factors (or primitive iterations performed before the loop construct and within the loop) are not important when considering asymptotic running time as the input (n) grows large. So, the running time of the algorithm is O(n).

1. Find the asymptotic running time using the Master Formula:

T(n) = T(n/2) + n; T(1) = 1

In the equation given, a = 1, b = 2, c = 1, k = 1, and d = 1;

We have the scenario:

So, T(n) =

1. You are given a length-n array A consisting of 0s and 1s, arranged in sorted order. Devise an algorithm that counts the total number of 0s and 1s in the array that runs *faster than* Θ(n) time.

**Algorithm** countZerosAndOnes(A, n, B, 2)

***Input***: A – an array of length n containing 0s and 1s sorted in natural order

***Output***: B – an array of length 2, where B[0]=count of zeros, B[1], count of 1s

B 🡨 new Array(2)

**if** (A[0] = 1) **then //**that means we have all 1s in the array A

B[0] 🡨 0

B[1] 🡨 n

**return B**

**if** (A[n-1] = 0) **then //**that means we have all 0s in the array A

B[0] 🡨 n

B[1] 🡨 0

**return B**

mid🡨 floor( //moving mid point

**while**(mid > 0 & mid < n ) **do**

**if(**A[mid-1] != A[mid]**) then //**Got the point separating 0s from 1s

B[0] 🡨mid //all elements at and before index mid-1 are 0s

B[1] 🡨 n – mid //all the rest elements are 1s

**return** b

**if(**A[mid] = 0**) then //**advance to the right

mid 🡨 mid + ceiling(

**if(**A[mid] = 1**) then //**advance to the left

mid 🡨 floor(

B[0] 🡨mid //mid now indicates how many 0s we have in array A

B[1] 🡨 n-mid

**return** b

The Algorithm runs the while loop at most times. Considering the constant factors are not important in evaluating the asymptotic running time of the algorithm as the input array gets large, we have the running time to be O(, which grows slower than n. This indicates, the running time of the algorithm is faster than Θ(n) time.