Module 1: Basic Concepts of Data Structures

System Life Cycle, Algorithms, Performance Analysis, Space Complexity, Time Complexity, Asymptotic Notation, Complexity Calculation of Simple Algorithms.

SYSTEM LIFE CYCLE (SLC)

- Good programmers regard large scale computer programs as systems that contain many complex interacting parts. (Systems: Large Scale Computer Programs.)
- As systems, these programs undergo a development process called System life cycle.(SLC : Development Process of Programs)

  Different Phases of System Life Cycle

  1. Requirements
  2. Analysis
  3. Design
  4. Refinement and coding
  5. Verification

1. Requirement Phase:

- All programming projects begin with a **set of specifications** that defines the purpose of that program.
- Requirements describe the information that the programmers are given (**input**) and the results (**output**) that must be produced.
- Frequently the initial specifications are defined vaguely and we must develop rigorous input and output descriptions that include all cases.

2. Analysis Phase

- In this phase the problem is break down into manageable pieces.
- There are two approaches to analysis:-**bottom up and top down**.
- Bottom up approach is an older, **unstructured** strategy that places an early emphasis on coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many **loosely connected, error ridden segments**.
- Top down approach is a structured approach divide the program into manageable segments.
- This phase generates **diagrams** that are used to design the system.

- Several alternate solutions to the programming problem are developed and compared during this phase

3. Design Phase

- This phase continues the work done in the analysis phase.
- The designer approaches the system from the perspectives of both **data objects** that the program needs and the **operations** performed on them.
- The first perspective leads to the **creation of abstract data types** while the second requires the **specification of algorithms** and a consideration of algorithm design strategies.

  Ex: Designing a scheduling system for university

  Data objects: Students, courses, professors etc

  Operations: insert, remove search etc

  ie. We might add a course to the list of university courses, search for the courses taught by some professor etc.
- Since abstract data types and algorithm specifications are language independent.
- We must specify the information required for each data object and ignore coding details.
  Ex: Student object should include name, phone number, social security number etc.

4. Refinement and Coding Phase

- In this phase we choose representations for data objects and write algorithms for each operation on them.
- Data objects representation can determine the efficiency of the algorithm related to it. So we should write algorithms that are independent of data objects first.
- Frequently we realize that we could have created a much better system. (May be we realize that one of our alternate design is superior than this). If our original design is good, it can absorb changes easily.

5. Verification Phase

- This phase consists of
  - developing correctness proofs for the program
  - Testing the program with a variety of input data.
  - Removing errors.

Correctness of Proofs

- Programs can be proven correct using proofs.(like mathematics theorem)
- Proofs are very time consuming and difficult to develop for large projects.
- Scheduling constraints prevent the development of complete sets of proofs for a larger system.
- However, selecting algorithm that have been proven correct can reduce the number of errors.

Testing

- Testing can be done only after coding.
- Testing requires working code and set of test data.
- Test data should be chosen carefully so that it includes all possible scenarios.
- Good test data should verify that every piece of code runs correctly.
- For example if our program contains a *switch* statement, our test data should be chosen so that we can check each *case* within *switch* statement.

Error Removal

- If done properly, the correctness of proofs and system test will indicate erroneous code.
- Removal of errors depends on the design and code.
- While debugging large undocumented program written in 'spaghetti' code, each corrected error possibly generates several new errors.
- Debugging a well documented program that is divided into autonomous units that interact through parameters is far easier. This especially true if each unit is tested separately and then integrated into system.

ALGORITHMS

Definition: An ***algorithm*** is a finite set of instructions to accomplish a particular task. In addition, all algorithms must satisfy the following criteria:

(1) ***Input***. There are zero or more quantities that are externally supplied.
(2) ***Output***. At least one quantity is produced.
(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

We can describe algorithm in many ways

1. We can use a natural language like English
2. Graphical Representation called flow chart, but they work well only if the algorithm is small and simple.

Translating a Problem into an Algorithm

Example [Selection sort]: Suppose we must devise an algorithm that sorts a collection of n > 1 elements of arbitrary type. A simple solution is given by the following

[Selection Sort: In each pass of the selection sort, the smallest element is selected from the unsorted list and exchanged with the elements at the beginning of the unsorted list]

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

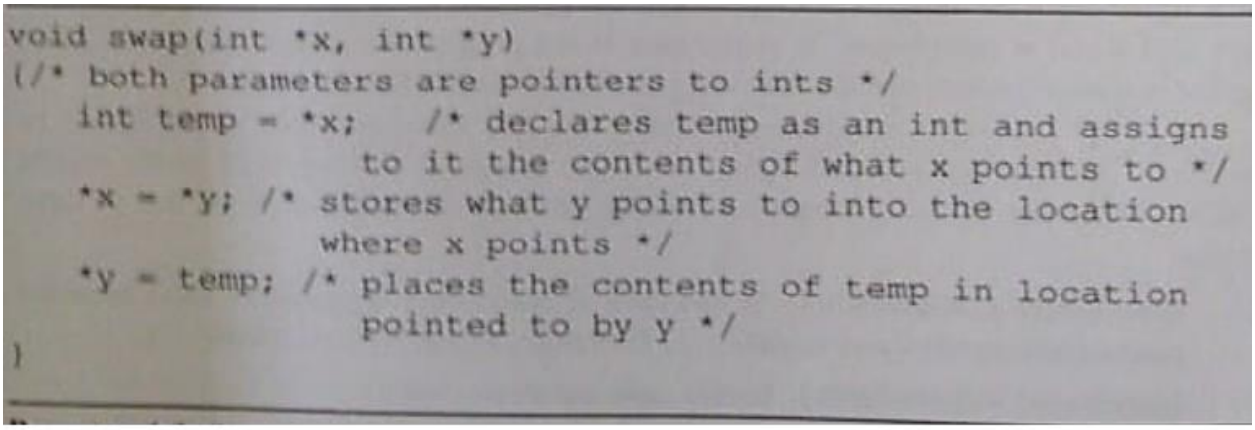
- From those elements that are currently unsorted, find the smallest and place it next in the sorted list
- We assume that the elements are stored in an array 'list', such that the i^{th} integer is stored in the i^{th} Position list[i], 0 <= i <n
- Algorithm 1.1 is our first attempt to deriving a solution

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

1.1 Selection sort algorithm

- We are written this partially in C and partially in English
- To turn the program 1.1 into a real C program, two clearly defined sub tasks are remain: **finding the smallest integer** and **interchanging it with list[i].**
- We can solve this by using a function



```
void swap(int *x, int *y)
(/* both parameters are pointers to ints */
   int temp = *x;   /* declares temp as an int and assigns
                       to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

1.2 Swap Function

- To swap their values one could call swap(&a, &b)
- We can solve the first subtask by assuming that the minimum is the list[i]. Checking list[i] with list[i+1], list[i+2]……,list[n-1]. Whenever we find a smaller number we make it as the minimum. We reach list[n-1] we are finished.

```
#include <stdio.h>
int main()
{
        int a[100], n, i, j, position, swap;
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter %d Numbersn", n);
        for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
        for(i = 0; i < n - 1; i++)
        {
            position=i;
            for(j = i + 1; j < n; j++)
            {
                    if(a[position] > a[j])
                    position=j;
            }
            if(position != i)
            {
                    swap=a[i];
                    a[i]=a[position];
                    a[position]=swap;
            }
        }
        printf("Sorted Array:n");
        for(i = 0; i < n; i++)
        printf("%dn", a[i]);
        return 0;
```

}

- **Correctness Proof**

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1:n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1:q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n-1$ without damaging the correctness of the algorithm. □

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, *list*[0] $\leq$ *list*[1] $\leq \cdots \leq$ *list*[n−1]. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that *list*[i] = *searchnum*. If *searchnum* is not present, we should return −1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = n−1. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare *list*[*middle*] with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle]**. In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle]**. In this case, we return *middle*.

(3) **searchnum > list[middle]**. In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and n − 1. So, we set *left* to *middle* + 1.

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to *list*[*middle*].

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
           return middle;
        else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

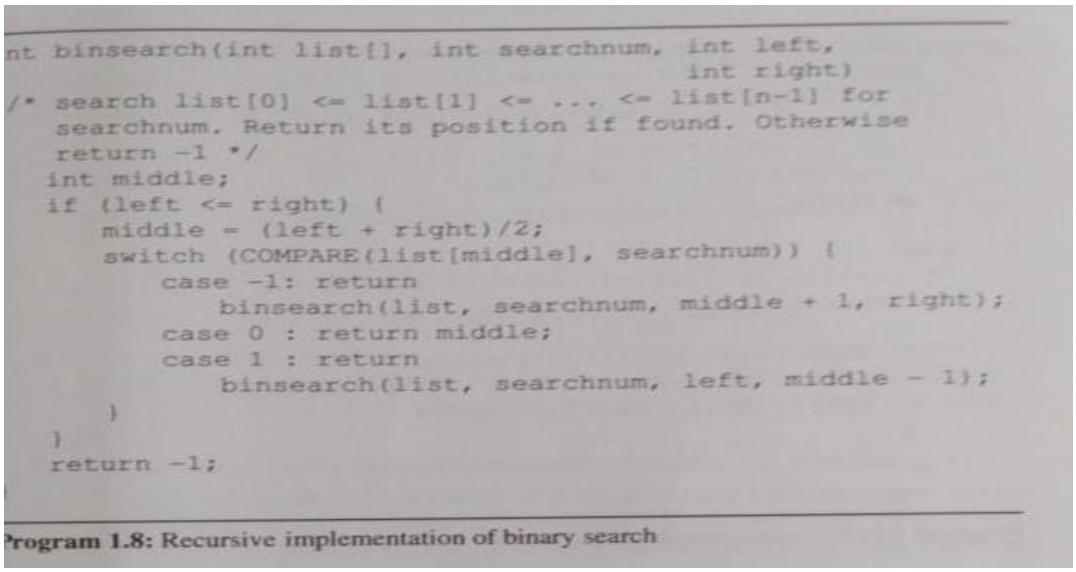
```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Recursive Algorithm

- An algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive.
- Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

- These recursive mechanisms are extremely powerful, but even more importantly; many times they can express an otherwise complex process very clearly.



```
nt binsearch(int list[], int searchnum, int left,
                                        int right)
/* search list[0] <= list[1] <= ... <= list[n-1] for
   searchnum. Return its position if found. Otherwise
   return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
```

Program 1.8: Recursive implementation of binary search

PERFORMANCE ANALYSIS

An algorithm is said to be efficient and fast, if it takes **less time to execute** & **consume less memory space**

Performance is analyzed based on 2 criteria

1. Space Complexity
2. Time Complexity

1. Space Complexity

- Analysis of **space complexity of an algorithm** or program is the **amount of memory it needs to run to completion**.
- The space needed by a program consists of following components.
    - **Fixed space requirements:** Independent on the number and size of the programs input and output. It include
        - Instruction Space (Space needed to store the code)
        - Space for simple variable
        - Space for constants
    - **Variable space requirements:** This component consists of

- Space needed by structured variable whose size depends on the particular instance *I* of the problem being solved
- Space required when a function uses recursion

- Total Space Complexity **S(P)** of a program is

$$\mathbf{S(P) = C + S_p(I)}$$

Here $\mathbf{S_p(I)}$ is Variable space requirements of program P working on an instance *I*.

C is a constant representing the fixed space requirements

- Example :

1.
```
int sum(int A[], int n)
{
    int sum=0, i;
    for(i=0;i<n;i++)
    {
        Sum=sum+A[i];
        return sum;
    }
}
```

Here Space needed for variable n = 1 byte

Sum = 1 byte

i = 1 byte

Array A[i] = n byte

Total Space complexity = [n+3] byte

2.
```
void main()
{
    int x,y,z,sum;
    printf("Enter 3 numbers");
    scanf("%d%d%d",&x,&y,&z);
    sum = x+y+z;
    printf("The sum = %d",sum);
```

}

Here Space needed for variable x = 1 byte

y = 1 byte

z = 1 byte

sum = 1 byte

Total Space complexity = 4 byte

3. sum (a,n)

{

int s=0;

for(i=0;i<n;i++)

for(j=0;j<m;j++)

s=s+a[i][j];

return s;

}

Here Space needed for variable n = 1 byte

m = 1 byte

s = 1 byte

i = 1 byte

j = 1 byte

Array a[i][j] = nm byte

Total Space complexity = nm+5 byte

2. Time Complexity

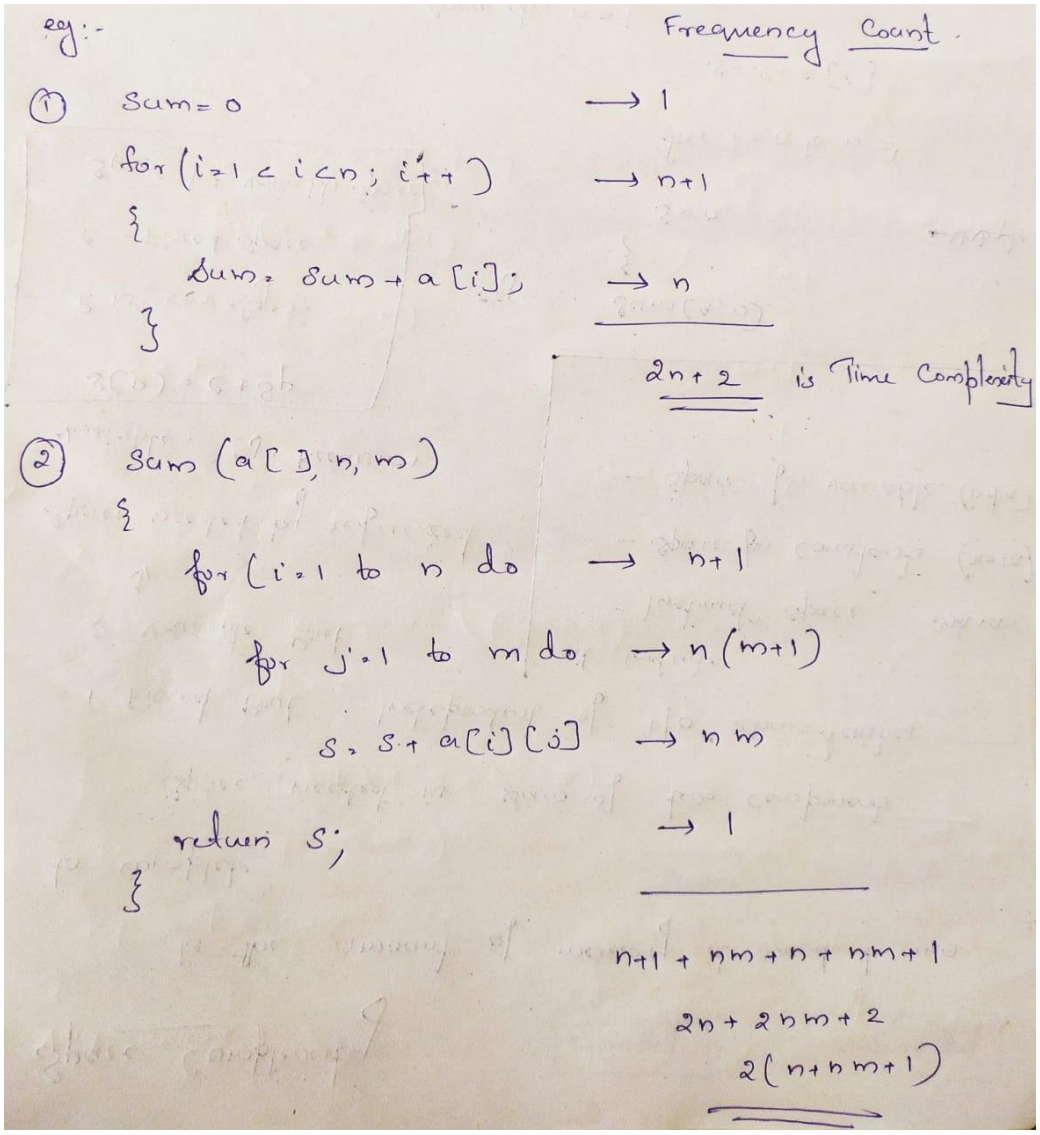
- The **time complexity of an algorithm** or a program is the **amount of time it needs to run to completion**.
- **$T(P)=C+T_P$**

  Here **C** is compile time

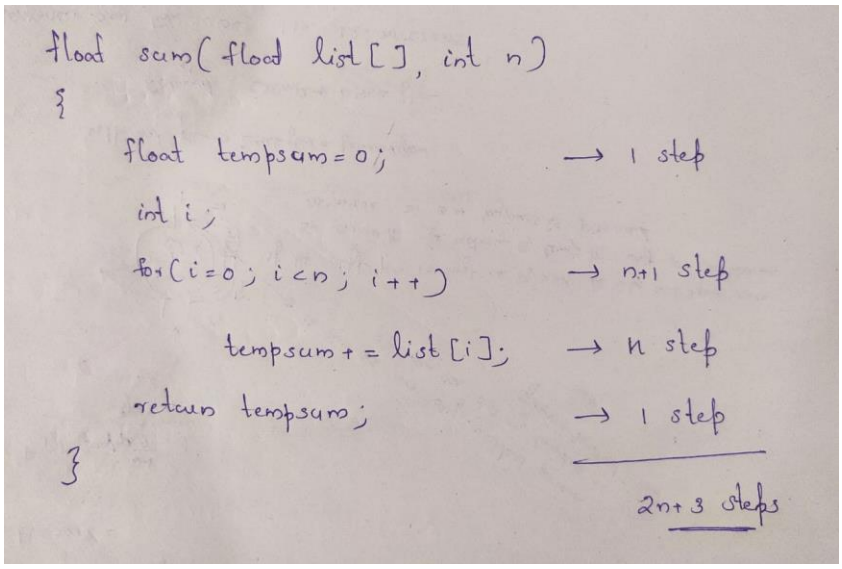
  T_P is Runtime
- For calculating the time complexity, we use a method called **Frequency Count** ie, counting the number of steps
  - Comments – 0 step

- Assignment statement – 1 Step
- Conditional statement – 1 Step
- Loop condition for ‘n’ numbers – n+1 Step
- Body of the loop – n step
- Return statement – 1 Step

- Examples:



3. Iterative function for summing a list of numbers

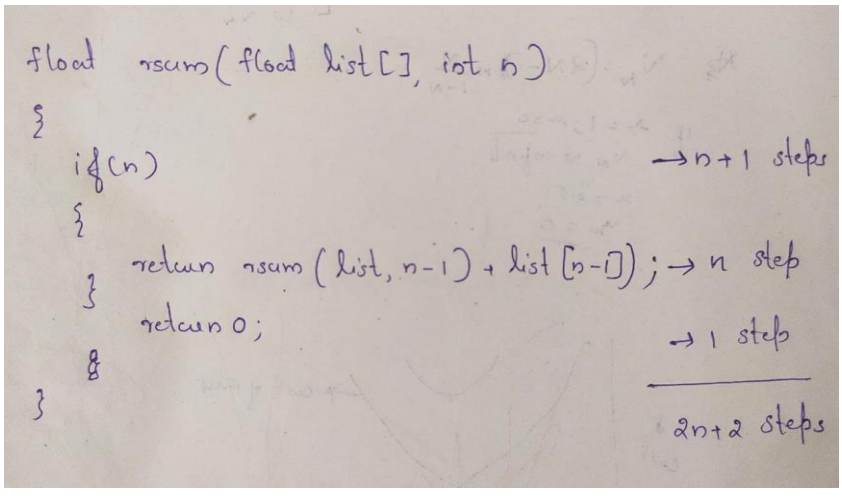


Tabular Method

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

s/e =steps/execution

4. Recursive summing of a list of numbers



Tabular Method

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

- When we analyze an algorithm it depends on the input data, there are three cases :
  a. **Best case:** The best case is the minimum number of steps that can be executed for the given parameters.
  b. **Average case:** The average case is the average number of steps executed on instances with the given parameters.
  c. **Worst case**: In the worst case, is the maximum number of steps that can be executed for the given parameters

ASYMPTOTIC NOTATION

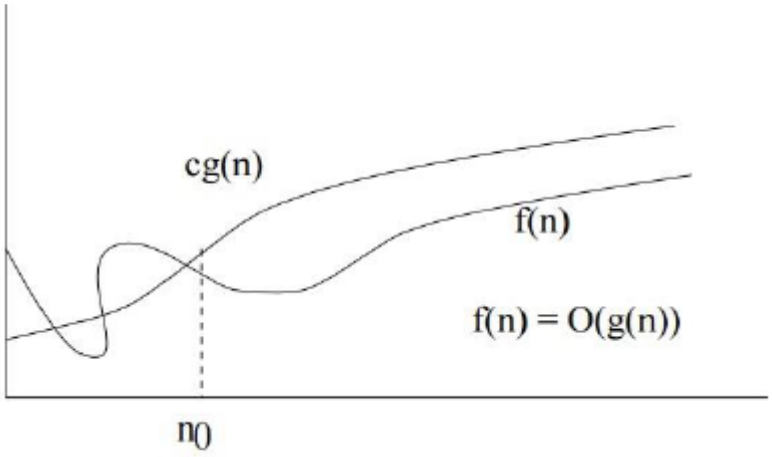
- Complexity of an algorithm is usually a function of n.
- Behavior of this function is usually expressed in terms of one or more standard functions.
- Expressing the complexity function with reference to other known functions is called **asymptotic complexity**.
- Three basic notations are used to express the asymptotic complexity

1. **Big – Oh notation O :** Upper bound of the algorithm
2. **Big – Omega notation Ω :** Lower bound of the algorithm
3. **Big – Theta notation Θ :** Average bound of the algorithm

1. Big – Oh notation O

- Formal method of expressing the upper bound of an algorithm's running time.
- i.e. it is a measure of longest amount of time it could possibly take for an algorithm to complete.
- It is used to represent the **worst case** complexity.

- **f(n) = O(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≤ c g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is big-O of g(n)"**.



- Examples:
  1. Derive the Big – Oh notation for f(n) = 2n + 3

     Ans:

     2n + 3 <= 2n+3n

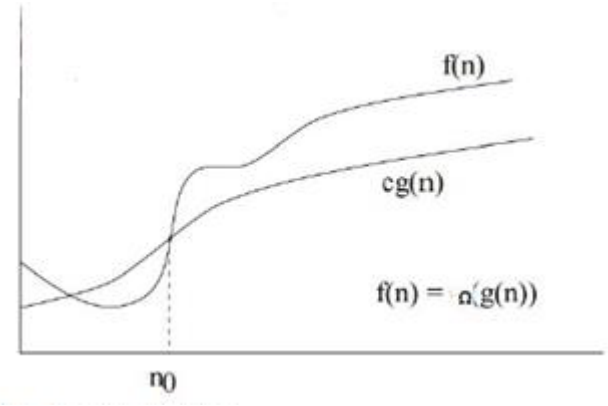
     2n+3 <= 5n for all n>=1

     Here c = 5

     g(n) = n

     so, f(n) = O(n)

2. **Big – Omega notation Ω**
   - **f(n) = Ω(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≥ c g(n)** for all **n ≥ n0.**
   - Then we say that **"f(n) is omega of g(n)"**.



   - Examples:

Derive the Big – Omega notation for f(n) = 2n + 3

Ans:

2n + 3 >= 1n for all n>=1

Here c = 1

g(n) = n

so, f(n) = Ω (n)

3. Big – Theta notation Θ

- **f(n) = Θ(g(n))** if and only if there are three positive constants c1, c2 and n0 such that **c1 g(n) ≤ f(n) ≤ c2 g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is theta of g(n)"**.
- Examples:

  Derive the Big – Theta notation for f(n) = 2n + 3

  Ans:

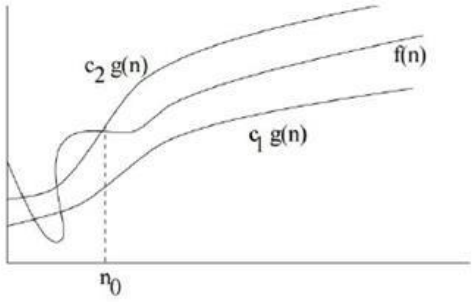
  1n <= 2n + 3 <= 5n for all n>=1

  Here c1 = 1

  C2 = 5

  g1(n) and g2(n) = n

  so, f(n) = **Θ** (n)



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

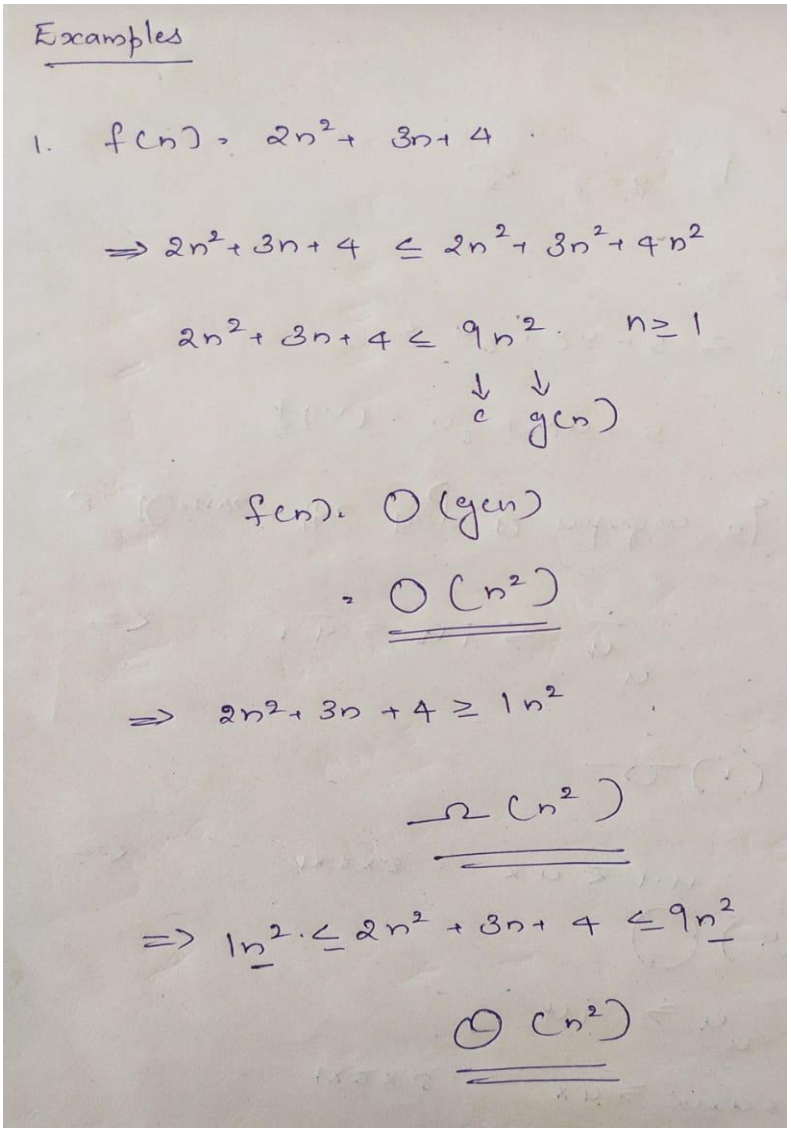
- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

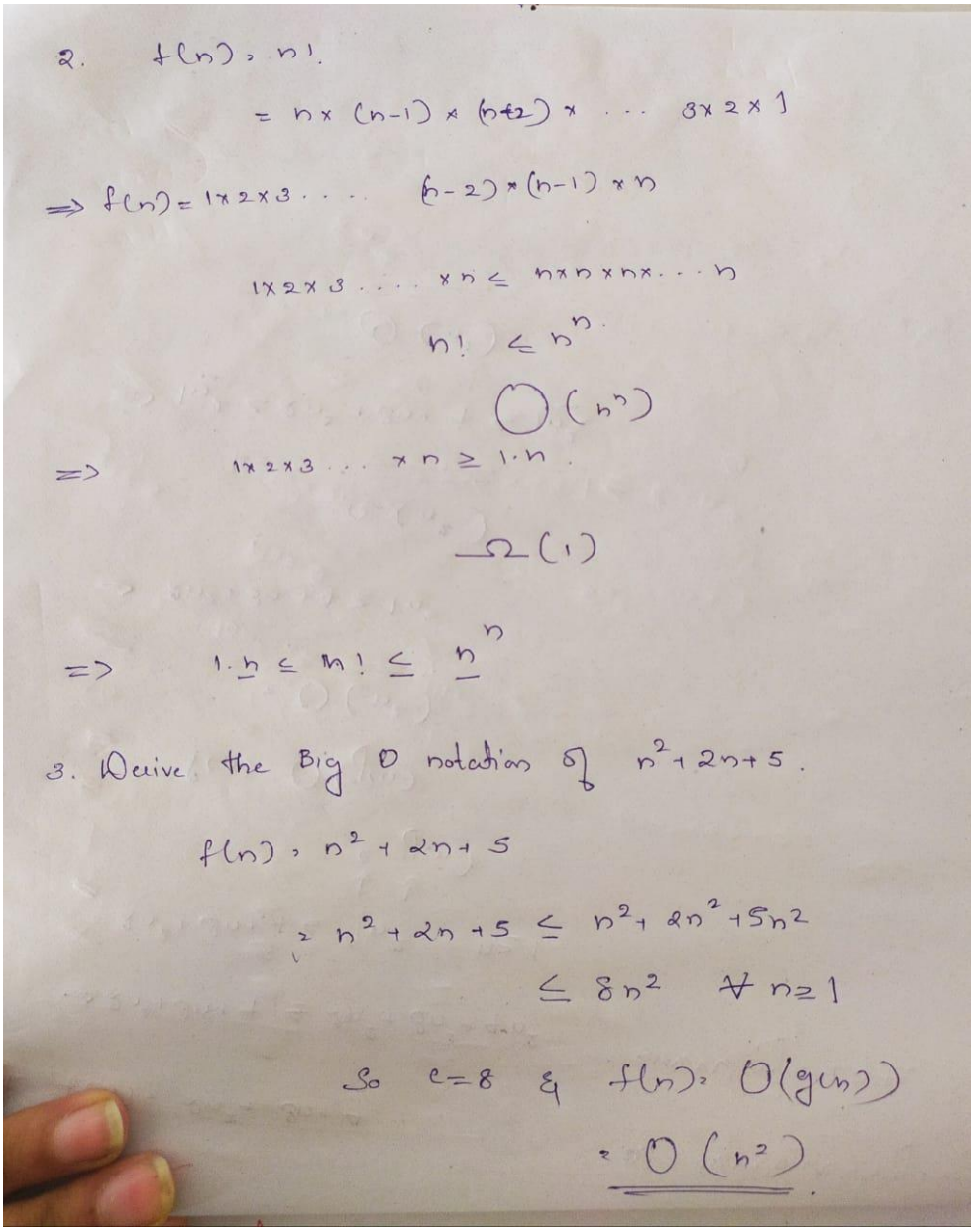
  $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

  Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, c1 = 1, and c2 = 13.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)





TIME COMPLEXITY OF LINEAR SEARCH

- Any algorithm is analyzed based on the unit of computation it performs. For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Best Case	Worst Case	Average Case
1	n	n / 2

TIME COMPLEXITY OF BINARY SEARCH

- In Binary search algorithm, the target key is examined in a sorted sequence and this algorithm starts searching with the middle item of the sorted sequence.
    a. If the middle item is the target value, then the search item is found and it returns True.
    b. If the target **item < middle** item, then search for the target value in the first half of the list.
    c. If the target **item > middle** item, then search for the target value in the second half of the list.
- In binary search as the list is ordered, so we can eliminate half of the values in the list in each iteration.
- Consider an example, suppose we want to search 10 in a sorted array of elements, then we first determine 15 the middle element of the array. As the middle item contains 18, which is greater than the target value 10, so can discard the second half of the list and repeat the process to first half of the array. This process is repeated until the desired target item is located in the list. If the item is found then it returns True, otherwise False.
- In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about ***n/2*** items will be eliminated after first comparison. After second comparison, ***n/4*** items of the list will be eliminated. If this process is repeated for several times, then there will be just one item left in the list. The number of comparisons required to reach to this point is $n/2^i = 1$. If we solve for i, then it gives us $i = \log_2 n$. The maximum number is comparison is logarithmic in nature, hence the time complexity of binary search is ***O(log n).***

Best Case	Worst Case	Average Case
1	O(log n)	O(log n)