Performance Analysis

- **Space Complexity**: The space complexity of a program is the amount of memory it needs to run to completion.
- **Time Complexity**: The time complexity of a program is the amount of computer time it needs to run to completion.

Space Complexity

- A **fixed part** that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables and fixed-size component variables, space for constants, etc.
- A **variable part** that consists of the space needed by component variables whose size is dependent on the particular problem instance I being solved and the recursion stack space.
- The space requirement S(P) of any program P is written as $S(P) = c + Sp(I)$ where c is a constant and I represent Instance Characteristics.

- When analyzing the space complexity of a program, we shall concentrate only on estimating Sp (instance characteristics).
- For a given problem, we shall need to determine which instance characteristics to use to measure the space requirements and is problem specific.

Ex 1:

```
float add(float a, float b, float c) {
   return a+b+c;
}
```

The problem instance is characterized by the specific values of a, b and c. Making assumption that one word is sufficient to store the values of a, b, c and the value returned by add, the space needed by function add is independent of the instance characteristics. Hence,

$Sp(I)=0$

Ex 2:

```
float sum(float  a[], const int n){
   float s=0;
   for(int i=0;i<n;i++) s+=a[i];
   return s;
}
```

Here n, the represents the instance characteristics ie number of elements to be summed. Since n is passed by value, one word must be allocated for it. Also, a is base address of the array, one word is to be allocated again for it. Hence, space needed by fn is independent of n and Sp(n) =0

Ex 3:

```
float rsum(float *a, const int n){
   if (n<=0) return 0;
   return (rsum(a,n-1)+a[n-1]);
}
```

Instances are characterized by n. Recursion stack space includes space for formal parameters, local variables and return address. Each call to rsum requires 12 bytes(space for n, a and return address). Space needed is 12(n+1).

Time Complexity

- The time, T(P), taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. We focus on the run time of a program, denoted by t_p (instance characteristics).

- Program Step – a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.
- The amount of computing represented by one step may be different from that represented by another step.
- The time required to execute each statement that is counted as one step should be independent of the instance characteristics.

Eg: a=2 ;

a= 2*b+3/c;

return(a+b+c*d); - a program step

- We can determine the number of steps a program needs to solve a problem instance by creating a global variable, count, which has an initial value of 0 and then inserting statements that increment count by the number of steps required by each executable statement.

Time Complexity in C++

- General statements in a C++ program

	Step count
Comments	0
Declarative statements	0
Expressions and assignment statements	1
Iteration statements	N
Switch statement	N
If-else statement	N
Function invocation	1 or N
Memory management statements	1 or N
Function statements	0
Jump statements	1 or N

Time Complexity Iterative Example

```
float sum (float *a, const int n)
{
   float s = 0;
   for (int i = 0; i < n; i++)
         s += a[i];
   return;
}
```

Step Count of Iterative Example

```
float sum (float *a, const int n)
{
    float s = 0;
    count++;                    // count is global
    for (int i = 0; i < n; i++)
    {
            count++;// for for
            s += a[i];
            count++;// for assignment
    }
    count++;                    // for last time of for
    count++;                    // for return
    return;
}
```

Step Count of Iterative Example (Simplified)

```
void sum (float *a, const int n)
{
   for (int i = 0; i < n; i++)
         count += 2;
   count +=3;
}
```

If initially count = 0, then the total of steps is 2n + 3.

Time Complexity of Recursive Example

```
float rsum (float *a, const int n)
{
   if (n <= 0) return 0;
   else return (rsum(a, n–1) + a[n-1]);
}
```

Step Count of Recursive Example

```
float rsum (float *a, const int n)
{
    count++;                    // for if conditional
    if (n <= 0) {
            count++;            // for return
            return 0;
    }
    else {
            count++;            // for return
            return (rsum(a, n–1) + a[n-1]);
    }
}
```

Assume $t_{rsum}(0) = 2$

$t_{rsum}(n) = 2 + t_{rsum}(n-1)$

$= 2 + 2 + t_{rsum}(n-2)$

$= 2*2 + t_{rsum}(n-2)$

$= 2n + t_{rsum}(0)$

$= 2n + 2$

Matrix Addition Example

```
line void add (int **a, int **b, int **c, int m, int n)
1    {
2       for (int i = 0; i < m; i++)
3          for (int j = 0; j < n; j++)
4             c[i][j] = a[i][j] + b[i][j];
5    }
```

Step Count of Matrix Addition Example

```
void add (int **a, int **b, int **c, int m, int n)
{
    for (int i = 0; i < m; i++)
    {
      count++;                          // for for i
      for (int j = 0; j < n; j++)
      {
            count++;                    // for for j
            c[i][j] = a[i][j] + b[i][j];
            count++;                    // for assigment
      }
      count++;                          // for last time of for j
    }
    count++;                            // for last time of for i
}
```

Step Count of Matrix Addition Example (Simplified)

```
line void add (int **a, int **b, int **c, int m, int n)
{
   for (int i = 0; i < m; i++)
   {
      for (int j = 0; j < n; j++)
            count += 2;
      count+2;
   }
   count++;
}
```

Step Count = 2mn+2m+1

Time Complexity (Cont.)

- Note that a step count does not necessarily reflect the complexity of the statement.
- Tabular Method:
- Determine the step count for each statement. It is referred as Step per execution (s/e) - the amount by which count changes as a result of the execution of that statement.
- Figure out the number of times that each statement is executed - frequency

Time Complexity Iterative Example

```
float sum (float *a, const int n)
1 {
2  float s = 0;
3  for (int i = 0; i < n; i++)
4          s += a[i];
5  return;
6 }
```

Line	s/e	Frequency	Total steps
1	0	1	0
2	1	1	1
3	1	n+1	n+1
4	1	n	n
5	1	1	1
6	0	1	0
Total no. of steps			2n+3

Step Table of Matrix Addition Example

Line	s/e	Frequency	Total steps
1	0	1	0
2	1	m+1	m+1
3	1	m(n+1)	mn+m
4	1	mn	mn
5	0	1	0
Total number of steps			2mn+2m+1

Fibonacci Numbers

- The Fibonacci sequence of numbers starts as 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

  Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence F_0 then $F_0 = 0$, $F_1 = 1$, and in general

  $$F_n = F_{n-1} + F_{n-2}\ ,\ n \geq 2.$$

C++ Program of Fibonacci Numbers

```
1   void fibonacci (int n)
2    // compute the Fibonacci number F_n
3    {
4           if (n <= 1) cout << n << endl;          // F_0 = 0, and F_1 = 1
5           else {                                  // compute F_n
6              int fn; int fnm2 = 0; int fnm1 = 1;
7              for (int i = 2; i <= n; i++)
8              {
9                 fn = fnm1 + fnm2;
10                fnm2 = fnm1;
11                fnm1 = fn;
12             } // end of for
13             cout << fn << endl;
14          } // end of else
15 } // end of fibonacci
```

Step Count of Fibonacci Program

- Two cases:
  - n = 0 or n = 1
    - Line 4 regarded as two lines: 4(a) and 4(b), total step count in this case is 2
  - n > 1
    - Line 4(a), 6, and 13 are each executed once
    - Line 7 gets executed n times.
    - Lines 8 – 12 get executed n-1 times each.
    - Line 6 has s/e of 2 and the remaining lines have an s/e of 1.
    - Total steps for the case n > 1 is 4n + 1

Asymptotic Notation

- Determining step counts help us to compare the time complexities of two programs that compute the same function and to predict the growth in run time as the instance characteristics change.
- But determining exact step counts could be very difficult. Since the notion of a step count is itself inexact, it may not be worth the effort to compute the exact step counts.
- Definition [Big "oh"]: $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$

Examples of Asymptotic Notation

- $3n + 2 = O(n)$

  $3n + 2 \leq 4n$ for all $n \geq 2$

- $100n + 6 = O(n)$

  $100n + 6 \leq 101n$ for all $n \geq 10$

- $10n^2 + 4n + 2 = O(n^2)$

  $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$

Practical Complexities

- If a program P has complexities O(n) and program Q has complexities $O(n^2)$, then, in general, we can assume program P is faster than Q for a sufficient large n.
- However, caution needs to be used on the assertion of "sufficiently large".

Function Values

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Plot of Function Values