Analysis of Algorithms

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

What is an Algorithm?

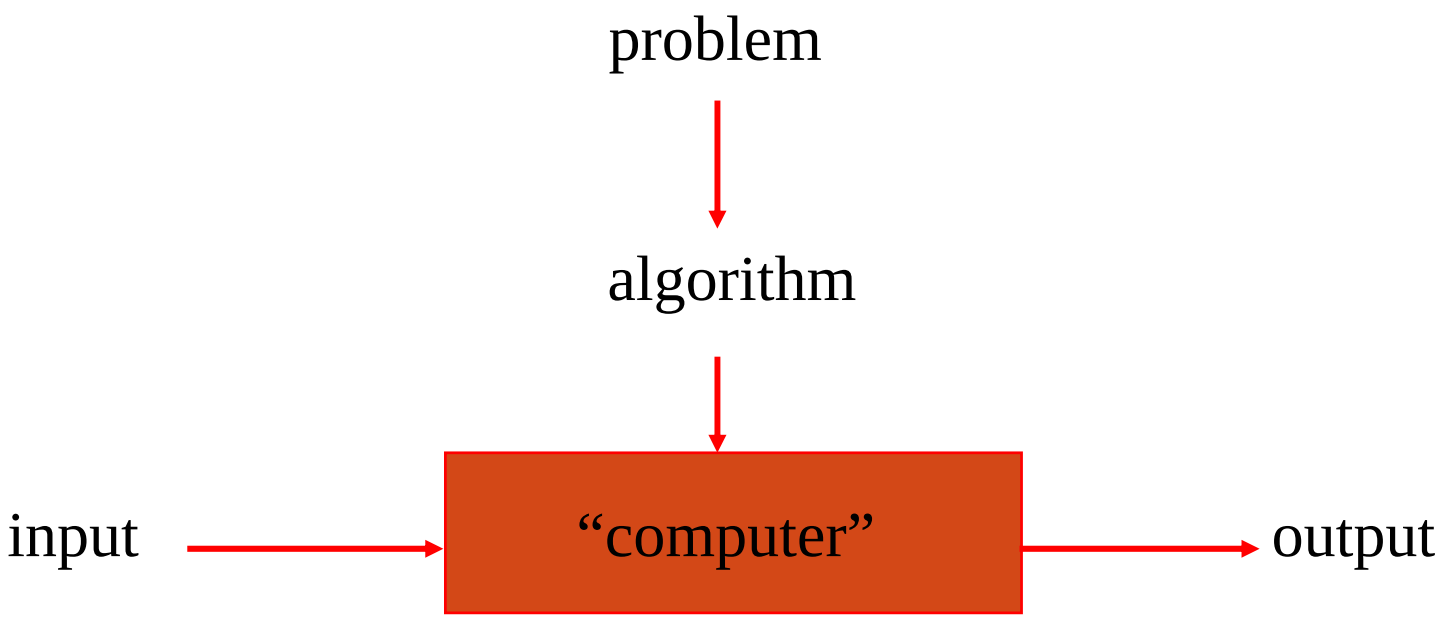
- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
- As a computer professional, it is useful to know a standard set of important algorithms and to be able to design new algorithms as well as to analyze their efficiency.

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost

Notion of algorithm



Algorithmic solution

Problem

- An ***instance of a problem*** consists of all inputs needed to compute a solution to the problem.
- An algorithm is said to be ***correct*** if for every input instance, it halts with the correct output.
- A correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instance, or it might halt with other than the desired answer.

Example: sorting

- Statement of problem:
  - Input:

    A sequence of n numbers

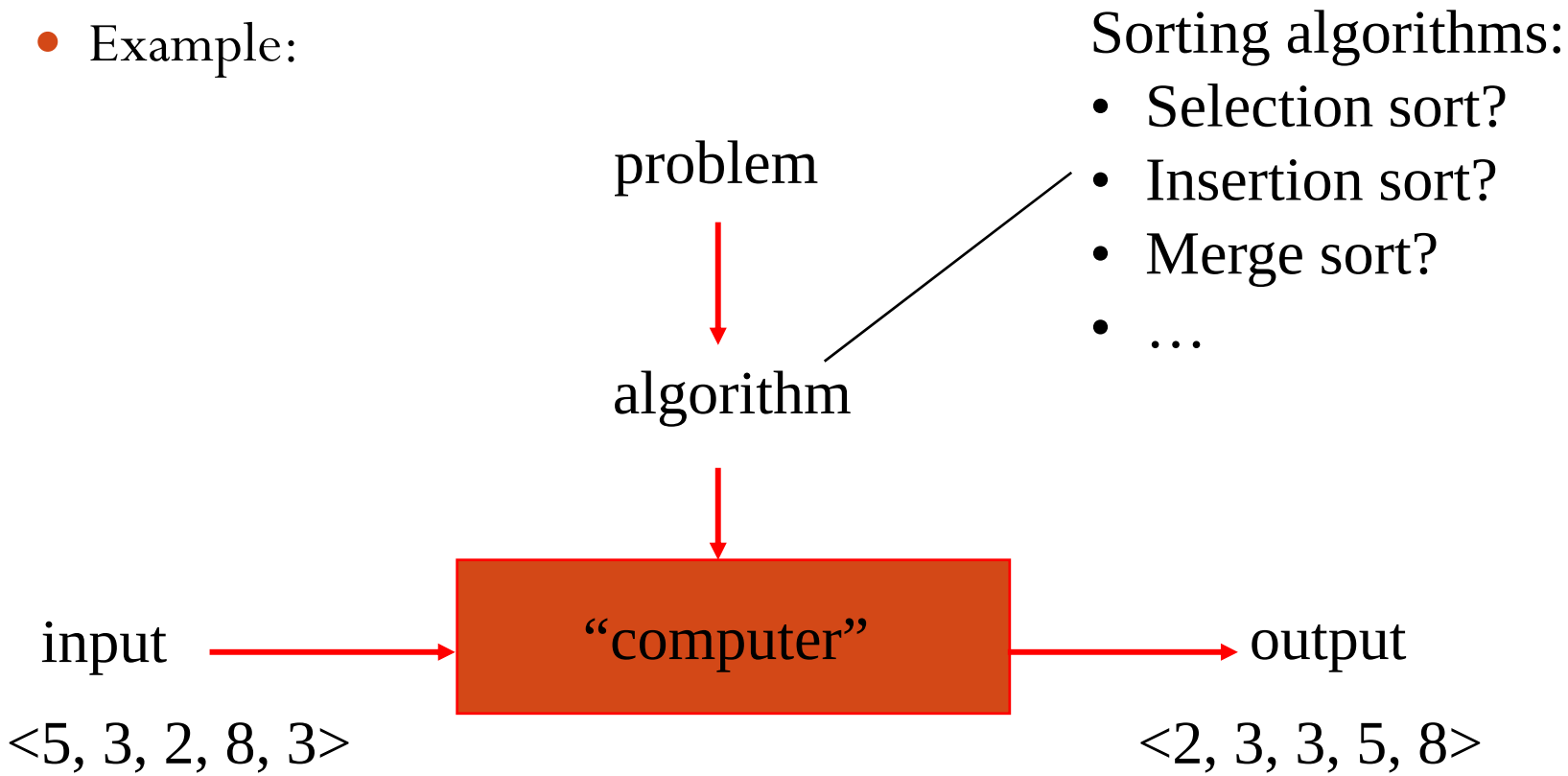
    $$\langle a_1, a_2, \ldots, a_n \rangle$$
  - Output:

    A reordering of the input sequence

    $$\langle b_1, b_2, \ldots, b_n \rangle$$

    so that $b_i \leq b_j$ whenever $i < j$

- Example:


problem
algorithm
Sorting algorithms:
• Selection sort?
• Insertion sort?
• Merge sort?
• …
input
<5, 3, 2, 8, 3>
“computer”
output
<2, 3, 3, 5, 8>

Selection sort

- Input:

  array a[0], …, a[n-1]

- Output:

  array a sorted in non-decreasing order

- Algorithm:

  for i=0 to n-1

  swap a[i] with smallest of a[i],…a[n-1]

Some well-known computational problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Basic issues related to algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness of algorithms
- Efficiency
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound
- Space and time tradeoffs

Analysis of algorithms

- How good is the algorithm?
  - Correctness
  - **Time efficiency**
  - **Space efficiency**
  - Simplicity
  - Generality
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

What is an algorithm?

- **Recipe, process, method, technique, procedure, routine,… with following requirements:**

1. **Finiteness**

   terminates after a finite number of steps

2. **Definiteness**

   rigorously and unambiguously specified

3. **Input**

   valid inputs are clearly specified

4. **Output**

   can be proved to produce the correct output given a valid input

5. **Effectiveness**

   steps are sufficiently simple and basic

Why Analyze Algorithms?

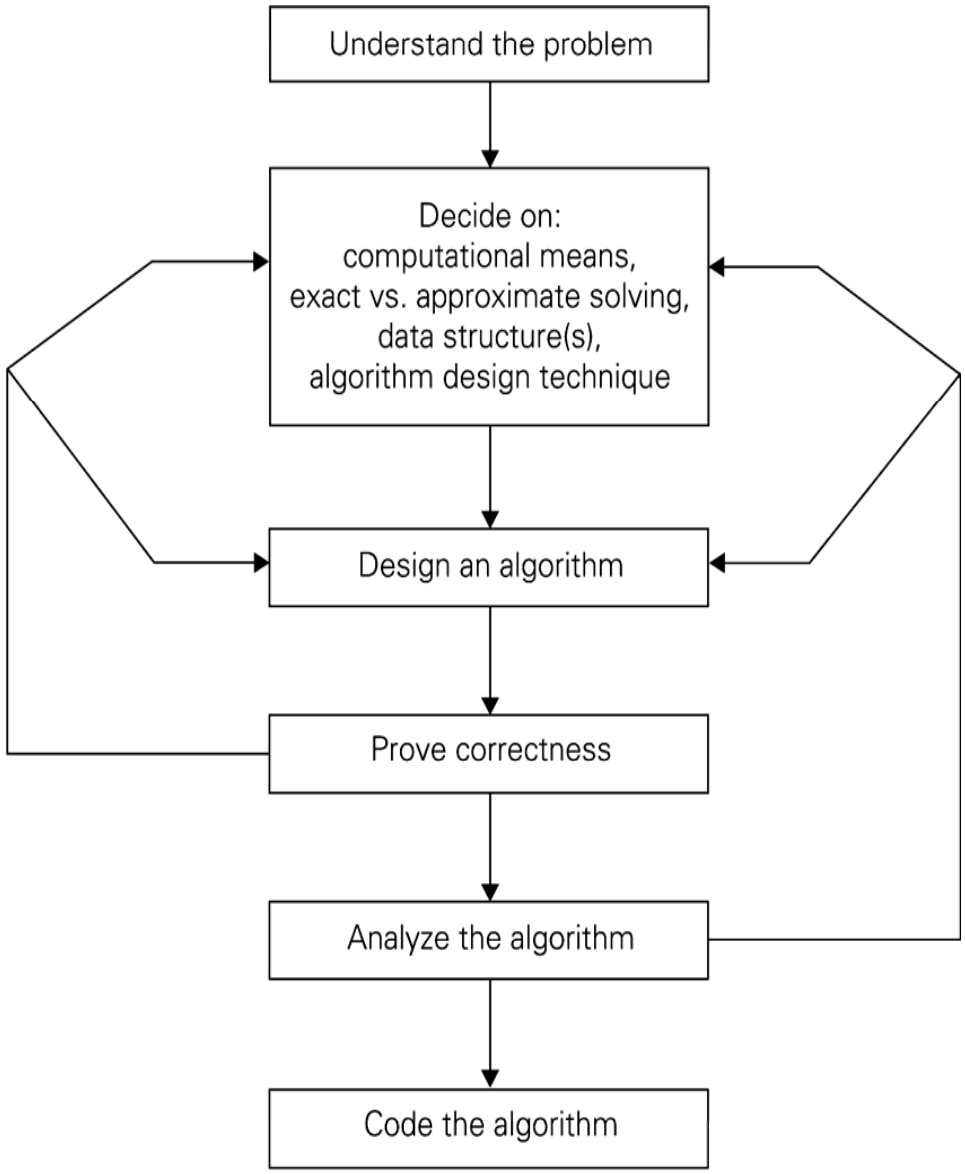
An algorithm can be analyzed in terms of time efficiency or space utilization. We will consider only the former right now. The running time of an algorithm is influenced by several factors:

- Speed of the machine running the program
- Language in which the program was written. For example, programs written in assembly language generally run faster than those written in C or C++, which in turn tend to run faster than those written in Java.
- Efficiency of the compiler that created the program
- The size of the input: processing 1000 records will take more time than processing 10 records.
- Organization of the input: if the item we are searching for is at the top of the list, it will take less time to find it than if it is at the bottom.

Why study algorithms and performance?

- Algorithms help us to understand ***scalability.***
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** *for talking about program behavior.*
- Performance is the ***currency*** *of computing.*
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Algorithm Design and Analysis Process



The goodness of an algorithm

- Time complexity (more important)
- Space complexity
- For a parallel algorithm :
  - time-processor product
- For a VLSI circuit :
  - area-time (AT, AT^2)

Need for analysis

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

Complexity

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal* factors
  - *external* factors

Internal and external factors

- **internal factors**

- The algorithm's efficiency, in terms of:
  - Time required to run
  - Space (memory storage)required to run

- **external factors**

- Speed of the computer on which it is run
- Quality of the compiler
- Size of the input to the algorithm

Complexity measures the *internal* factors (usually more interested in time than space)

Two ways of finding complexity

- **Performance measurement** (Experimental study)

  Through Experiments or Empirical Approach

- **Performance Analysis (Theoretical Analysis)**

  Analytical Method or Theoretical Approach

Experimental study

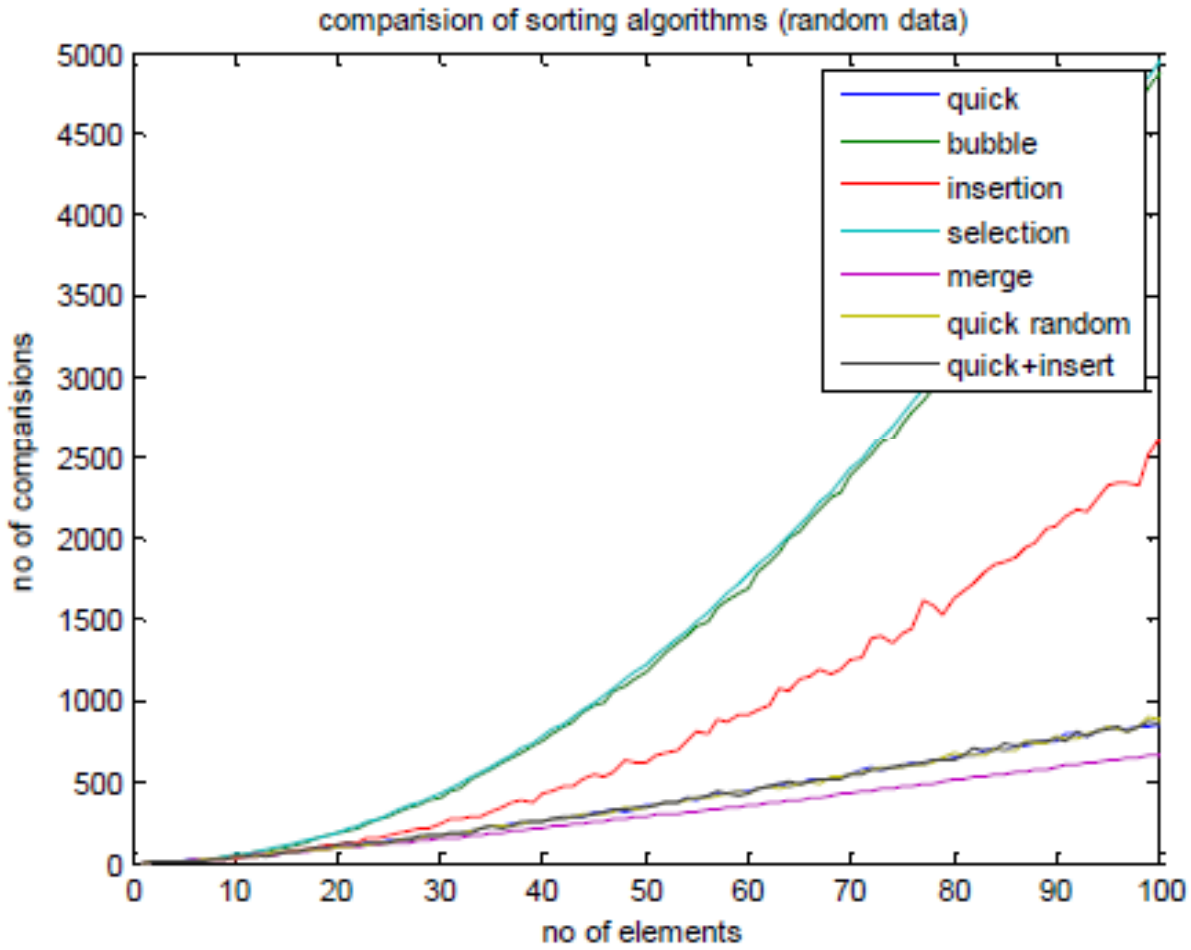
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time

  Use a method like System.currentTimeMillis()
- Plot the results

Java Code – Simple Program

```
import java.io.*;
class for1
{
 public static void main(String args[])
   throws Exception
 {
  int N,Sum;
  N=10000; // N value to be changed.
  Sum=0;
  long
   start=System.currentTimeMillis();
  int i,j;
  for(i=0;i<N;i++)
   for(j=0;j<i;j++)
   Sum++;
   long
    end=System.currentTimeMillis()
    ;
   long time=end-start;
   System.out.println(" The start time
    is : "+start);
   System.out.println(" The end time
    is : "+end);
   System.out.println(" The time
    taken is : "+time);
 }
}
```

Example graph



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space requirements of an algorithm

- **The fixed part of Algorithm**
  - **includes space for**
    - **Instructions**
    - **Simple variables**
    - **Fixed size component variables**
    - **Space for constants**
    - **Etc..**
- **Varaiable Part of Algorithm**

Why Space Complexity

- When memory was expensive we focused on making programs as **space** efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)
- The space complexity is used to estimate the size of the largest problem a program can solve.
- If a program is to run on a multi-user computer system, then it is required to specify the amount of memory to be allocated to the program.

Space Complexity (cont'd) $S(P)=C+S_P(I)$

The space needed by an algorithm is the sum of a fixed part and a variable part

Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:

- - e.g. name of the data collection
- same size for classifying 2GB or 1MB of texts

Variable part: Space needed by variables, whose size is dependent on the size of the problem:

- - e.g. actual text
- - load 2GB of text VS. load 1MB of text

Space Complexity $S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Space Complexity Cont...

- **A variable part of an algorithm is consists of the space needed by component variable whose size depends on the particular problem instance being solved**
- The variable part includes space for
  - Component variables whose size is dependant on the particular problem instance being solved
  - Recursion stack space
  - Etc..

Space Complexity (cont'd)

- S(P) = c + S(instance characteristics)
  - c = constant
- Example:

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

Space? one word for n, one for a [passed by reference!], one for i → constant space!

Time Complexity

Why Time Complexity ?

- Is the algorithm "fast enough" for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose

Why Time Complexity ?

- Often more important than space complexity
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us

- 3-4GHz processors on the market
  - still …
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 Terra HZ computer would take about 1 year to run to completion
- Algorithms running time is an important issue

Time Complexity

The time complexity of an algorithm is the amount of computer time required to run to completion

- The time complexity of a problem is
  - the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm.
- The exact number of steps will depend on exactly what machine or language is being used.
- To avoid that problem, the **Asymptotic notation** is generally used.

Time Complexity $T(P)=C+t_P(I)$

- Compile time (C) independent of instance characteristics.
- Run (execution) time denoted as t_P *(Instance)*
- Many of the factors t_p depends on are not known at the time
- a program is conceived it, is reasonable to attempt only to estimate t_p

$$t_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

- Running time of an algorithm as a function of input size n for large n, where n denotes the instance characteristics.
- Expressed using only the highest-order term in the expression for the exact running time.

Program Steps

- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

Regard as the same unit
machine independent

Methods to compute the step count

- Introduce variable **count** into programs
- **Tabular method**
  - Determine the total number of steps contributed by each statement

    step per execution × frequency
  - add up the contribution of all statements

Iterative summing of a list of numbers

- ***Program 1.12:** Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;              /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;          /* last execution of for */
   return tempsum;
   count++;          /* for return */
}
```

2n + 3 step

- ***Program 1.13:** Simplified version of Program 1.12 (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
       count += 2;
   count += 3;
   return 0;
}
```

2n + 3 step

Recursive summing of a list of numbers

- ***Program 1.14:** Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
        count++;      /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

2n + 2 step

Matrix addition

- ***Program 1.15:** Matrix addition (p.25)

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                        int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
}
```

*Program 1.16: Matrix addition with count statements (p.25)

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int row, int cols )
{
   int i, j;
   for (i = 0; i < rows; i++){
       count++; /* for i for loop */
       for (j = 0; j < cols; j++) {
          count++; /* for j for loop */
          c[i][j] = a[i][j] + b[i][j];
          count++; /* for assignment statement */
       }
       count++;    /* last time of j for loop */
   }
   count++;        /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

*Program 1.17: Simplification of Program 1.16 (p.26)

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
          count += 2;
          count += 2;
    }
    count++;
}
```

$2rows \times cols + 2rows + 1$

Suggestion: Interchange the loops when rows >> cols

Tabular Method: Iterative function to sum a list of numbers

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

Figure 1.2: Step count table for Program 1.10 (p.26)

Recursive Function to sum of a list of numbers

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

Figure 1.3: Step count table for recursive summing function (p.27)

Matrix Addition

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Thanks for Your Attention!

Exercises

Exercise

```
1   Algorithm D(x, n)
2   {
3       i := 1;
4       repeat
5       {
6           x[i] := x[i] + 2; i := i + 2;
7       } until (i > n);
8       i := 1;
9       while (i ≤ ⌊n/2⌋) do
10      {
11          x[i] := x[i] + x[i + 1]; i := i + 1;
12      }
13  }
```

Algorithm 1.23 Example algorithm

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.

   (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).

   (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.

   (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.

Exercise 1

- ***Program 1.18:** Printing out a matrix (p.28)

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i = 0; i < row; i++) {
    for (j = 0; j < cols; j++)
      printf("%d",  matrix[i][j]);
    printf( "\n");
  }
}
```

Exercise 2

- ***Program 1.19:Matrix multiplication function**(p.28)

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int
c[ ][MAX_SIZE])
{
  int i, j, k;
  for (i = 0; i < MAX_SIZE; i++)
    for (j = 0; j< MAX_SIZE;  j++) {
       c[i][j] = 0;
    for (k = 0; k < MAX_SIZE; k++)
         c[i][j]  +=  a[i][k] * b[k][j];
    }
}
```

Exercise 3

- ***Program 1.20:Matrix product function**(p.29)

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
                                                    int rowsa, int colsb, int
colsa)
{
  int i, j, k;
  for (i = 0; i < rowsa; i++)
    for (j = 0; j< colsb;  j++) {
       c[i][j] = 0;
    for (k = 0; k< colsa; k++)
         c[i][j]  +=  a[i][k] * b[k][j];
    }
}
```

Exercise 4

- ***Program 1.21:Matrix transposition function** (p.29)

```
void transpose(int a[ ][MAX_SIZE])
{
  int i, j, temp;
  for (i = 0; i < MAX_SIZE-1; i++)
    for (j = i+1; j < MAX_SIZE;  j++)
      SWAP (a[i][j], a[j][i], temp);
}
```

Next class : Asymptotic Notations ...

- $f(n) = \mathbf{O}(g(n))$, There exist $c > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq f(n) \leq cg(n)}$ for each $n \geq n_0$

- $f(n) = \mathbf{\Omega}(g(n))$, There exist $c > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq cg(n) \leq f(n)}$ for each $n \geq n_0$

- $f(n) = \mathbf{\Theta}(g(n))$, There exist $c_1, c_2 > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$ for each $n \geq n_0$