**UNIT-I**

**Introduction : Algorithm, pseudo code for expressing algorithms.**

**Definition:** An *algorithm* is a sequence of unambiguous instructions for solving a problem. It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

**Characteristics of an algorithm:**

Every algorithm must be satisfied the following characteristics.

Input : Zero / more quantities are externally supplied.

Output : At least one quantity is produced.

Definiteness : Each instruction is clear and unambiguous.

Finiteness : If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.

Efficiency : Every instruction must be very basic and runs in short time with effective results better than human computations.

**Pseudo code for Expressing Algorithms:**

1. An algorithm is a procedure. It has two parts; the first part is head and the second part is body.

2. The Head section consists of keyword Algorithm and Name of the algorithm with parameter list.

E.g. Algorithm name1(p1, p2,…,p3)

The head section also has the following:

//Problem Description:

//Input:

//Output:

3. In the body of an algorithm various programming constructs like if, for, while and some statements like assignments are used.

4. The compound statements may be enclosed with { and } brackets. if, for, while can be open and closed by {, } respectively. Proper indention is must for block.

5. Comments are written using // at the beginning.

6. The identifier should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.

7. The left arrow “:=” used as assignment operator. E.g. v:=10

8. Boolean operators (TRUE, FALSE), Logical operators (AND, OR, NOT) and Relational operators (<,<=, >, >=,=, ≠, <>) are also used.

9. Input and Output can be done using read and write.

10. Array[], if then else condition, branch and loop can be also used in algorithm.

**Example:**

The greatest common divisor(GCD) of two nonnegative integers *m* and *n* (not-both-zero,m<=n), denoted gcd*(m, n)*, is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero.

*Euclid’s algorithm* is based on applying repeatedly the equality gcd*(m, n)* = gcd*(n, m* mod *n),* where *m* mod *n* is the remainder of the division of *m* by *n*, until *m* mod *n* is equal to 0. Since gcd*(m,*0*)* = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*.

gcd*(*60*,* 24*)* can be computed as follows:gcd*(*60*,* 24*)* = gcd*(*24*,* 12*)* = gcd*(*12*,* 0*)* = 12*.*

**Euclid’s algorithm for computing gcd*(m, n)* in simple steps**

Step 1 If *n* = 0, return the value of *m* as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3 Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

Euclid’s algorithm for computing gcd*(m, n)* expressed in pseudocode

*ALGORITHM Euclid\_gcd(m, n)*

*{*

*//Computes gcd(m, n) by Euclid’s algorithm*

*//Input: Two nonnegative, not-both-zero integers m and n*

*//Output: Greatest common divisor of m and n*

*while n ≠ 0 do*

*{*

*r := m mod n;*

*m:=n;*

*n:=r;*

*}*

*return m;*

*}*

**FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING**

(i) Understanding the Problem

* This is the first step in designing of algorithm.
* Read the problem’s description carefully to understand the problem statement completely.
* Ask questions for clarifying the doubts about the problem.
* Identify the problem types and use existing algorithm to find solution.
* Input (*instance*) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device

1. In *random-access machine* (*RAM*), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called *sequential algorithms*.
2. In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called *parallel algorithms*.
3. Choice of computational devices like Processor and memory is mainly based on space and time efficiency

(b) Choosing between Exact and Approximate Problem Solving

* + 1. The next principal decision is to choose between solving the problem exactly or solving it approximately.
    2. An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
    3. If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

1. An *algorithm design technique* (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
2. Algorithms*+ Data Structures = Programs*
3. Though Algorithms and Data Structures are independent, but they are combined to develop program. Hence the choice of proper data structure is required before designing the algorithm.
4. Implementation of algorithm is possible only with the help of Algorithms and Data Structures
5. Algorithmic strategy / technique / paradigm is a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(iii) **Methods of Specifying an Algorithm**

There are three ways to specify an algorithm. They are:

a. Natural language

b. Pseudocode

c. Flowchart

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. **Natural Language**

It is very simple and easy to specify an algorithm using natural language. But many times, specification of algorithm by using natural language is not clear and thereby we get brief specification.

b**. Pseudocode**

Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

c. **Flowchart**

In the earlier days of computing, the dominant method for specifying algorithms was a *flowchart*, this representation technique has proved to be inconvenient. *Flowchart* is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps.

(iv) **Proving an Algorithm’s Correctness**

Once an algorithm has been specified then its *correctness* must be proved. An algorithm must yields a required result for every legitimate input in a finite amount of time.

Example: Addition of a and b

Start

Input the value of a;

Input the value of b;

c: = a + b;

Display the value of c;

Stop

(v) **Analyzing an Algorithm**

For an algorithm the most important is *efficiency*. In fact, there are two kinds of algorithm efficiency. They are:

*Time efficiency*, indicating how fast the algorithm runs, and

*Space efficiency*, indicating how much extra memory it uses.

The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.

So factors to analyze an algorithm are:

1. Time efficiency of an algorithm
2. Space efficiency of an algorithm
3. Simplicity of an algorithm
4. Generality of an algorithm

(vi) **Coding an Algorithm**

The coding / implementation of an algorithm is done by a suitable programming language

like C, C++, JAVA.

1. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduced by inefficient implementation.
2. Standard tricks like computing a loop’s invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
3. Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
4. It is very essential to write an optimized code (efficient code) to reduce the burden of
5. compiler.

**Performance Analysis.**

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

1. Analysis Framework.
2. Asymptotic Notations and its properties.
3. Mathematical analysis for Recursive algorithms.
4. Mathematical analysis for Non-recursive algorithms.
5. **Analysis Framework**: There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

*Time efficiency*, indicating how fast the algorithm runs, and

*Space efficiency*, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

* + 1. Measuring an Input’s Size
    2. Units for Measuring Running Time
    3. Orders of Growth
    4. Worst-Case, Best-Case, and Average-Case Efficiencies

1. Measuring an Input’s Size: An algorithm’s efficiency is defined as a function of some parameter *n* indicating the algorithm’s input size. In most cases, selecting such a parameter is quite straightforward.

For example, it will be the size of the list for problems of sorting, searching. For the problem of evaluating a polynomial *p(x)* = *anxn* + *. . .* + *a*0 of degree *n*, the size of the parameter will be the polynomial’s degree or the number of its coefficients, which is larger by 1 than its degree.

In computing the product of two *n* × *n* matrices, the choice of a parameter indicating an input size does matter.

Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.

In measuring input size for algorithms solving problems such as checking primality of a positive integer *n*. the input is just one number.

The input size by the number *b* of bits in the *n*’s binary representation is b=(log2 n)+1.

(ii) Units for Measuring Running Time : Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

1. Dependence on the speed of a computer.
2. Dependence on the quality of a program implementing the algorithm.
3. The compiler used in generating the machine code.
4. The difficulty of clocking the actual running time of the program.

So, we need metric to measure an *algorithm*’s efficiency that does not depend on these extraneous factors. One possible approach is to *count the number of times each of the algorithm’s operations* *is executed.* This approach is excessively difficult.

The most important operation (+, -, \*, /) of the algorithm, called the *basic operation*. Computing the number of times the basic operation is executed is easy. The total running time is

determined by basic operations count.

(iii**) Orders of Growth**

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.

For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid’s algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only. For large values of *n*, it is the function’s order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms.

Table 1.1 Growth of function order



(iv) **Worst-Case, Best-Case, and Average-Case Efficiencies** *Consider Sequential Search* algorithm some search key *K*

*ALGORITHM SequentialSearch(A[0..n - 1], X)*

*{*

*//Searches for a given value in a given array by sequential search*

*//Input: An array A[0..n - 1] and a search key X*

*//Output: The index of the first element in A that matches K or -1 if there are no*

*// matching elements*

*i ←0;*

*while i < n and A[i] ≠ X do*

*i ←i + 1;*

*if i < n return i*

*else return -1;*

*}*

Clearly, the running time of this algorithm can be quite different for the same list size *n*. In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

***Worst-case efficiency***

The *worst-case efficiency* of an algorithm is its efficiency for the worst case input of size *n*. The algorithm runs the longest among all possible inputs of that size. For the input of size *n,* the running time is *Cworst(n)* = *n*.

***Best case efficiency***

The *best-case efficiency* of an algorithm is its efficiency for the best case input of size *n*. The algorithm runs the fastest among all possible inputs of that size n. In sequential search, If we search a first element in list of size *n. (i.e.* first element equal toa search key), then the running time is *Cbest(n)* = 1

***Average case efficiency***

The Average case efficiency lies between best case and worst case. To analyze the algorithm’s average case efficiency, we must make some assumptions about possible inputs of size *n*.

**Time complexity-Space Complexity**

* Two criteria are used to judge algorithms: (i) time complexity (ii) space complexity.
* Space Complexity of an algorithm is the amount of memory it needs to run to completion.
* Time Complexity of an algorithm is the amount of CPU time it needs to run to completion.

**Space Complexity:**

Memory space S(P) needed by a program P, consists of two components:

* + A fixed part: needed for instruction space (byte code), simple variable space, constants space etc. 🡪 c
  + A variable part: dependent on a particular instance of input and output data. 🡪 Sp(instance)

**S(P) = c + Sp(instance)**

**Example 1:**

Algorithm abc (a, b, c)

{

1. return a+b+b\*c+(a+b-c)/(a+b)+4.0;

}

For every instance 3 computer words required to store variables: a, b, and c. Therefore Sp()= 3. S(P) = 3.

**Example 2:**

Algorithm Sum(a[], n)

{

1. s:= 0.0;
2. for i = 1 to n do
3. s := s + a[i];
4. return s;

}

Every instance needs to store array a[] & n.

* 1. Space needed to store n = 1 word.
  2. Space needed to store a[ ] = n floating point words (or at least n words)
  3. Space needed to store i and s = 2 words

Sp(n) = (n + 3). Hence S(P) = (n + 3).

**Time Complexity:**

* How to measure T(P)?
  + Measure experimentally, using a “stop watch”

🡪 T(P) obtained in secs, msecs.

* + Count program steps 🡪 T(P) obtained as a step count.
* Fixed part is usually ignored; only the variable part tp() is measured.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Statements** | **S/E** | **Freq.** | **Total** |
| 1 | Algorithm Sum(a[],n) | 0 | - | 0 |
| 2 | { | 0 | - | 0 |
| 3 | S = 0.0; | 1 | 1 | 1 |
| 4 | for i=1 to n do | 1 | n+1 | n+1 |
| 5 | s = s+a[i]; | 1 | n | n |
| 6 | return s; | 1 | 1 | 1 |
|  |  |  |  |  |
| 7 | } | 0 | - | 0 |
| Total Count | | | | 2n+3 |

What is a program step?

* + a+b+b\*c+(a+b)/(a-b) 🡪 one step;
  + comments 🡪 zero steps;
  + while (<expr>) do 🡪 step count equal to the number of times <expr> is executed.
  + for i=<expr> to <expr1> do 🡪 step count equal to number of times <expr1> is checked.

**Asymptotic notations**

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program. The efficiency analysis framework concentrates on the order of growth of an algorithm’s basic operation count as the principal indicator of the algorithm’s efficiency. To compare and rank such orders of growth, computer scientists use five notations, they

are:

O - Big oh notation

Ω - Big omega notation

Θ - Big theta notation

o- Little oh notation

ω-Little omega notation

*Asymptotically Non-Negative*: A function *g*(*n*) is *asymptotically nonnegative*, if *g*(*n*)>=0 for all *n*>=*n*0 where *n*0 in N={0,1,2,3,…}

**Asymptotic Upper Bound: *O(Big-oh)***

**Definition:** Let *f*(*n*) and *g*(*n*) be asymptotically non-negative functions. We say  
 *f* (*n*) is in *O* ( *g* ( *n* )) if there is a real positive constant *c* and a positive Integer *n*0 such that for every *n* >= *n*0 , 0 <=*f* (*n*)<= *c* \* *g* (*n* ).

(Or)

*O*(*g*(*n*))= { *f*(*n*) *|* there exist a positive constant *c* and a positive integer *n*0such that

0 <=*f*( *n*) <= *c* \* *g* (*n* ) for all *n* >= *n*0 }

The Figure 1.1 shows the growth function of f(n) and g(n) for case of asymptotic upper bound

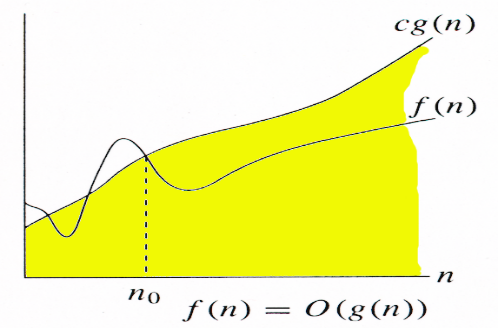


Figure 1.1 f(n)=O(g(n) growth function

**Example 1**: Verify 5n+2 = O(n).

Solution:

From the definition of Big Oh, there must exist c>0 and integer *n*0 >0 such that

0 <= 5*n*+2<=*c\*n* for all *n*>= *n*0.

Dividing both sides of the inequality by n>0 we get:

0 <= 5+2/*n* <= *c*.

Cleary 2/n <= 2, since 2/*n*>0 becomes smaller when n increases.

There are many choices here for c and *n*0.

If we choose *n*0 =1 then *c* >= 5+2/1= 7.

If we choose *c*=6, then 0 <= 5+2/*n*<=6*.* So *n*0>= 2.

In either case (we only need one!) we have a *c*>o and *n*0 >0 such that 0 <= 5*n*+2<=*cn* for all *n>= n*0 .

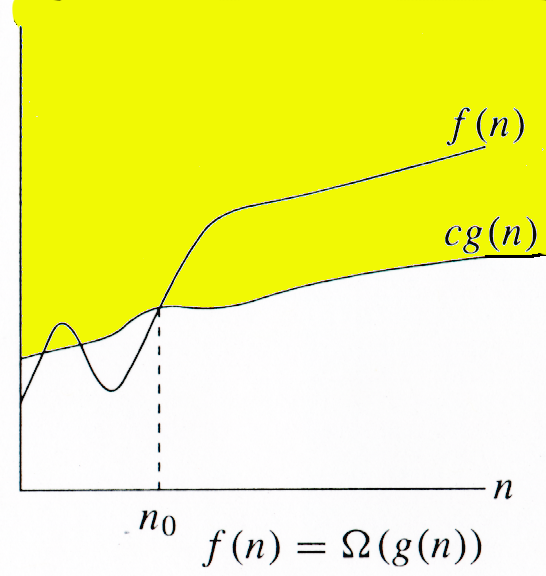
So the definition is satisfied and 5*n*+2 = O(n)

**Asymptotic Lower Bound: *Ω(Big-Omega)***

Definition:   
 Let *f*(*n*) and *g*(*n*) be asymptotically non-negative functions. We say  
 *f* (*n*) is ***Ω***( *g* ( *n* )) if there is a positive real constant *c* and a positive integer *n*0 such that for every *n* >= *n*0 0 <= *c* \* *g* (*n* ) <= *f* ( *n*).

(Or)

***Ω***( *g* ( *n* )) = { *f* (*n*)| there exist positive constant *c* and a positive integer *n*0 such that 0 <= *c* \* *g* (*n* ) <= *f* ( *n*) for all *n* >= *n*0 }



From the definition of Omega, there must exist c>0 and integer *n*0>0 such that 0 <= c\**n* <= 5*n*-20 for all *n*>= *n*0

Dividing the inequality by *n*>0 we get: 0 <= c <= 5-20/*n* for all *n*>= *n*0.

20/n <= 20, and 20/n becomes smaller as *n* grows.

There are many choices here for c and *n*0.

Since *c* > 0, 5 – 20/n >0 and *n*0 >4

**For example**, if we choose c=4, then 5 – 20/n <= 4 and *n*0 >= 20

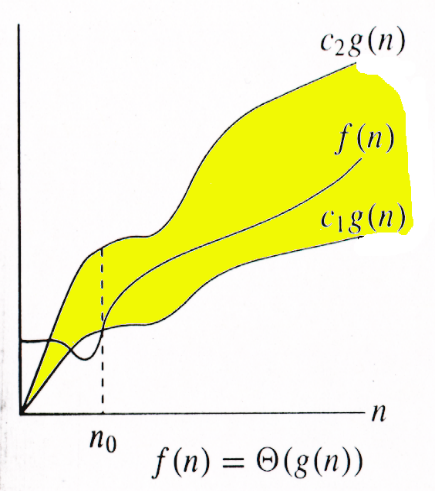
In this case we have a c>o and *n*0>0 such that 0 <= c\**n* <= 5n-20 for all *n* >=*n*0. So the definition is satisfied and 5*n*-20 in Ω (*n*)

**Asymptotic Tightly Bound: θ*(Theta)***

**Definition**: Let *f* (*n*) and *g*(*n*) be asymptotically non-negative functions. We say *f* (*n*) is θ( *g* ( *n* )) if there are positive constants *c, d* and a positive integer *n*0such that for every *n* >= *n*0   
 0 <= *c* \* *g* (*n* ) <= *f* ( *n*) <= *d* \* *g* ( *n* ).

(Or )

θ (*g*(*n*))={*f*(*n*)|there exist positive constants *c, d* and a positive integer *n*0 such that 0 <= *c* \* *g* (*n* ) <= *f* ( *n*) <= *d* \* *g* ( *n* ). for all *n* >= *n*0 }



**Example**: Prove that

Proof:

It is enough to prove that





****

****

**Asymptotic *o(Little-oh)***

**Definition:** Let *f* (*n*) and *g*(*n*) be asymptotically non-negative functions. We say *f* ( *n* ) is *o* ( *g* ( *n*)) if for every positive real constant *c* there exists a positive integer *n*0such that for all *n>=n*0   
0 <= *f*(*n*)< *c* \* *g* (*n* ).

(Or)

*o*(*g*(*n*))={*f*(*n*): for any positive constant *c* >0, there exists a positive integer *n*0 *> 0* such that 0 <= *f*( *n*) < *c* \* *g* (*n* ) for all *n* >= *n*0 }

**Calculating the running time of programs:**

Let us now look into how big-O bounds can be computed for some common algorithms.

**Example** :

|  |  |
| --- | --- |
| 2n2 + 5n – 6 = O (2n) | 2n2 + 5n – 6   (2n) |
| 2n2 + 5n – 6 = O (n3) | 2n2 + 5n – 6   (n3) |
| 2n2 + 5n – 6 = O (n2) | 2n2 + 5n – 6 =  (n2) |
| 2n2 + 5n – 6  O (n) | 2n2 + 5n – 6   (n) |
| 2n2 + 5n – 6   (2n) | 2n2 + 5n – 6 = o (2n) |
| 2n2 + 5n – 6   (n3) | 2n2 + 5n – 6 = o (n3) |
| 2n2 + 5n – 6 =  (n2) | 2n2 + 5n – 6  o (n2) |
| 2n2 + 5n – 6 =  (n) | 2n2 + 5n – 6  o (n) |

**Example:**

If the first program takes 100n2 milliseconds and while the second takes 5n3 milliseconds, then might not 5n3 program better than 100n2 program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, 5n3 program be better than the 100n2 program.

5 n3/100 n2 = n/20

for inputs n < 20, the program with running time 5n3 will be faster than those the one with running time 100 n2. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was O(n3)

However, as ‘n’ gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the O(n3) program will take significantly more time than the O(n2) program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function’s such as O(n) or O(n log n) are always better.

**Example:**

**Analysis of simple for loop**

Now let’s consider a simple for loop:

*for (i = 1; i<=n; i++)*

*v[i] = v[i] + 1;*

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n). The actual number of instructions might be 50n, while the running time might be 17n microseconds. It might even be 17n+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it’s a linear function which is proportional to n, the correct notation is O(n) and the code is said to have *linear* running time.

**Example:**

**Analysis for nested for loop**

Now let’s look at a more complicated example, a nested for loop:

*for (i = 1; i<=n; i++)*

*for (j = 1; j<=n; j++)*

*a[i,j] = b[i,j] \* x;*

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes n  n = n2 times. The assignment statement in the inner loop takes constant time, so the running time of the code is O(n2) steps. This piece of code is said to have *quadratic* running time.

**Example:**

**Analysis of matrix multiply**

Lets start with an easy case. Multiplying two n  n matrices. The code to compute the matrix product C = A \* B is given below.

*for (i = 1; i<=n; i++)*

*for (j = 1; j<=n; j++)*

C[i, j] = 0;

*for (k = 1; k<=n; k++)*

*C[i, j] = C[i, j] + A[i, k] \* B[k, j];*

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes n\*n\*n = n3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes O(n3) time.

**Example :**

**Analysis of bubble sort**

The main body of the code for bubble sort looks something like this:

*for (i = n-1; i<1; i--)*

*for (j = 1; j<=i; j++)*

*if (a[j] > a[j+1])*

*swap a[j] and a[j+1];*

This looks like the double. The innermost statement, the if, takes O(1) time. It doesn’t necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = n-1 times. The second time it runs n-2 times, etc. The total number of times the inner if statement executes is therefore:

(n-1) + (n-2) + ... + 3 + 2 + 1

This is the sum of an arithmetic series.

The value of the sum is n(n-1)/2. So the running time of bubble sort is O(n(n-1)/2), which is O((n2-n)/2). Using the rules for big-O given earlier, this bound simplifies to O((n2)/2) by ignoring a smaller term, and to O(n2), by ignoring a constant factor. Thus, bubble sort is an O(n2) algorithm.

**Example :**

**Analysis of binary search**

Binary search is a little harder to analyze because it doesn’t have a for loop. But it’s still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

n, n/2, n/4, ..., 8, 4, 2, 1

It’s not obvious how long this sequence is, but if we take logs, it is: log2 n, log2 n - 1, log2 n - 2, ..., 3, 2, 1, 0

Since the second sequence decrements by 1 each time down to 0, its length must be

log2 n + 1. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is log2n + 1. So binary search is an O(log2 n) algorithm. Since the base of the log doesn’t matter in an asymptotic bound, we can write that binary search is O(log n).

**General rules for the analysis of programs**

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is ‘n’ the input size.

1. The running time of each assignment read and write statement can usually be taken to be O(1). (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).
2. The running time of a sequence of statements is determined by the sum rule.

I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.

1. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally O(1) the time for an if–then–else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.
2. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is O(1)). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.

**Probabilities Analysis**

**Probabilities Analysis:**

this analysis uses probability

**Example:**

A sample space S will for us be some collection on elementary events.

For instance, results of coin flips. Then S={HH, TH, HT, TT}.

An event E is any subset of S.

For example, E= {TH, HT} be the event of S

A probability distribution P{} on S is a mapping from events on S to the real numbers satisfying for any events A and B. A’ be the complement of A

(a) P{A} >= 0 (b) P{S} = 1 (c) P{A∪ B} = P{A} + P{B} if A ∩ B = ∅

Result 1 : P{S∪ ∅} = P{S} + P{∅} = 1 + P{∅}. So P{∅} = 0.

Result 2 : P{S}= P{A∪ A’} = P{A} + P{A’}. So P{A’}=1- P{A}.

**Conditional Probability and Independence**

The conditional probability of an event A given an event B is defined to be: P{A|B} = P{A∩B}/P{B}.

• Two events are independent if Pr{A∩B} = Pr{A}Pr{B}

• Given a collection A1, A2,… Ak of events we say they are pairwise independent if Pr{Ai ∩ Aj } = Pr{Ai }Pr{Aj } for any i and j.

• They are mutually independent if for an subset Ai\_1, A2,… Ai\_m of then Pr{Ai\_1 ∩… Ai\_m} = Pr{Ai\_1}\* \*Pr{Ai\_m}

A discrete random variable X is a function from a finite sample space S to the real numbers.

• Given such a function X we can define the probability density function for X as: f(x) = Pr{X = x} where the little x is a real number.

The expected value of a random variable X is defined to be:

• The variance of X, Var[X] is defined to be: E[(X- E(X))2]= E[X2] -(E[X])2 • The standard deviation of X, σX, is defined to be the (Var[X])1/2.

**Indicator Random Variables**

• In order to analyze the hiring problem we need a convenient way to convert between probabilities and expectations.

• We will use indicator random variables to help us do this.

• Given a sample space S and an event A. Then the indicator random variable I{A} associated with event A is define as:

**Example:**

Suppose our sample space S={H,T} with P{H}=P{T}=1/2.

We can define an indicator random variable XH associated with the coin coming up heads:

XH=

The expected number of heads in one coin flip is then

E[XH]=P(H)\*I(H)+P(T)\*I(T)

= ½\*1+1/2\*0 =1/2.

Lemma 1: Given a sample space S and an event A in S, let XA=I{A}. Then E[XA]=P{A}.

Proof: E[XA] = E[I{A}]

= 1\*P{A}+ 0\*P{A}

=P{A}.

Indicator random variables are more useful if we are dealing with more than one coin flip.

Let Xi be the indicator that indicates whether the result of the ith coin flip was a head.

Consider the random variable: X=

Then the expected number of head in n tosses is

E[X]=E[]===n/2

**The Hiring Problem**

We will now begin our investigation of randomized algorithms with a toy problem:

• You want to hire an office assistant from an employment agency.

• You want to interview candidates and determine if they are better than the current assistant and if so replace the current assistant.

• You are going to eventually interview every candidate from a pool of n candidates.

• You want to always have the best person for this job, so you will replace an assistant with a better one as soon as you are done the interview.

• However, there is a cost to fire and then hire someone.

• You want to know the expected price of following this strategy until all n candidates have been interviewed.

**Algorithm Hire-Assistant(n)**

{

best := dummy candidate;

for i := 1 to n do

{

interview of candidate i ;

if (candidate i is better than best) then

{

best := i;

hire candidate i;

}

}

}

• Interviewing has a low cost ci .

• Hiring has a high cost ch.

• Let n be the number of candidates to be interviewed and let m be the number of people hired.

• The total cost then goes as O(n\*ci +m\*ch)

• The number of candidates is fixed so the part of the algorithm we want to focus on is the m\*ch term.

• This term governs the cost of hiring.

**Worst-case Analysis**

• In the worst case, everyone we interview turns out to be better than the person we currently have.

• In this case, the hiring cost for the algorithm will be O(n\*ch).

• This bad situation presumably doesn’t typically happen so it is interesting to ask what happens in the average case.

**Probabilistic analysis**

• Probabilistic analysis is the use of probability to analyze problems.

• One important issue is what is the distribution of inputs to the problem.

• For instance , we could assume all orderings of candidates are equally likely.

• That is, we consider all functions rank: [0..n] --> [0..n] where rank[i] is supposed to be the ith candidate that we interview. So <rank(1),rank(2),…,rank(n)>should be a permutation of <1,2,3,…,n>.

• There are n! many such permutations and we want each to be equally likely.

• If this is the case, the ranks form a uniform random permutation.

**Randomized algorithms**

• In order to use probabilistic analysis, we need to know something about the distribution of the inputs.

• Unfortunately, often little is known about this distribution.

• We can nevertheless use probability and analysis as a tool for algorithm design by having the algorithm we run do some kind of randomization of the inputs.

• This could be done with a random number generator. i.e.,

• We could assume we have primitive function Random(a,b) which returns an integer between integers a and b inclusive with equally likelihood.

• Algorithms which make use of such a generator are called randomized algorithms.

• In our hiring example we could try to use such a generator to create a random permutation of the input and then run the hiring algorithm on that.

**Analysis of the Hiring Problem**

• Let Xi be the indicator random variable which is 1 if candidate i is hired and 0 otherwise.

• Let

• By our lemma E[Xi ] = Pr{candidate i is hired}

• Candidate i will be hired if i is better than each of candidates 1 through i-1.

• As each candidate arrives in random order, any one of the first candidate i is equally likely to be the best candidate so far. So E[Xi ] =1/i.

E[X]=E[]===ln(n)+O(1)

Assume that the candidates are presented in random order, then algorithm Hire-Assistant has a hiring cost of O(ch\*ln n)

**Amortized Analysis**

**Amortized Analysis**

**What is Amortized Analysis ?**

* In amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed.
* No involvement of probability
* Average performance on a sequence of operations, even some operation is expensive.
* Guarantee average performance of each operation among the sequence in worst case.
* Methods of Amortized Analysis

**Aggregate Method**: we determine an upper bound *T(n)* on the total sequence of *n* operations. The cost of each will then be *T(n)/n*.

**Accounting Method**: we overcharge some operations early and use them to as prepaid charge later.

**Potential Method**: we maintain credit as potential energy associated with the structure as a whole.

1. **Aggregate Method**

* Show that for all n, a sequence of n operations take worst-case time T(n) in total
* In the worst case, the average cost, or amortized cost , per operation is T(n)/n.
* The amortized cost applies to each operation, even when there are several types of operations in the sequence.

**Aggregate Analysis: Stack Example**

|  |  |  |  |
| --- | --- | --- | --- |
| 3 ops: |  |  |  |
| Push(S,x) | Pop(S) | Multi-pop(S,k) |
| Worst-case cost: | O(1) | O(1) | O(min(|S|,k)  = O(n) |

* Sequence of n push, pop, Multipop operations
  + Worst-case cost of Multipop is O(n)
  + Have n operations
  + Therefore, worst-case cost of sequence is O(n2)
* Observations
  + Each object can be popped only once per time that it’s pushed
  + Have <= n pushes => <= n pops, including those in Multipop
  + Therefore total cost = O(n)
  + Average over n operations => O(1) per operation on average
* Notice that no probability involved

1. **Accounting Method**

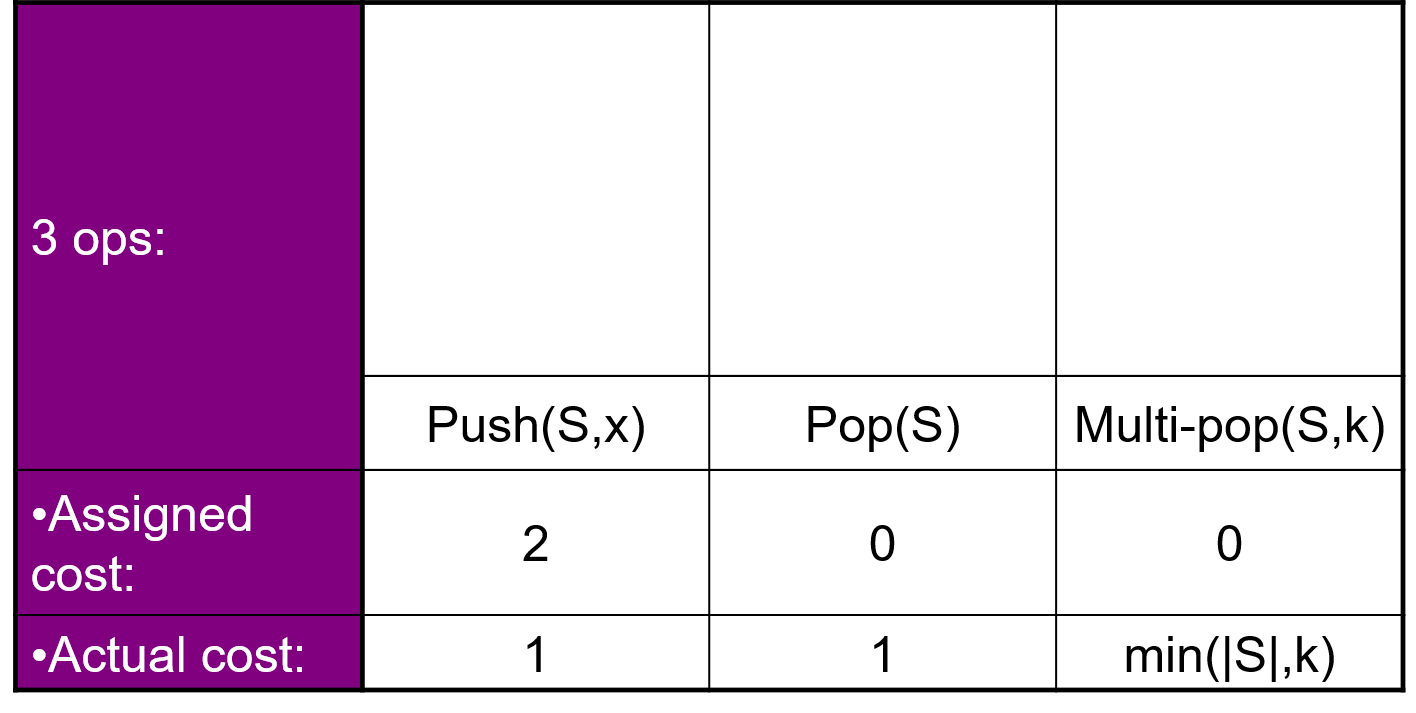
Charge i th operation a fictitious amortized cost ĉi, where $1 pays for 1 unit of work (i.e., time).

* + Assign different charges (amortized cost ) to different operations
    - * Some are charged more than actual cost
      * Some are charged less
* This fee is consumed to perform the operation.
* Any amount not immediately consumed is stored in the bank for use by subsequent operations.
* The bank balance (the credit) must not go negative!

*We must ensure that*

*for all n.* **

* Thus, the total amortized costs provide an upper bound on the total true costs.



* When pushing an object, pay $2
  + $1 pays for the push
  + $1 is prepayment for it being popped by either pop or Multipop
  + Since each object has $1, which is credit, the credit can never go negative
  + Therefore, total amortized cost = O(n), is an upper bound on total actual cost

**Accounting Method: Binary Counter**

* k-bit Binary Counter: A[0..k−1]



INCREMENT(*A*)

1. *i* ← 0

2. while *i < length*[*A*] and *A*[*i*] = 1

3. do *A*[*i*] ← 0 ⊳ *reset a bit*

4. *i* ← *i +* 1

5. if *i* < *length*[*A*]

6. then *A*[*i*]← 1 ⊳ *set a bit*

Consider a sequence of *n* increments. The worst-case time to execute one increment is Q(*k*). Therefore, the worst-case time for *n* increments is *n* · Q(*k*) = Q(*n⋅ k*).

WRONG! In fact, the worst-case cost for *n* increments is only Q(*n*) ≪ Q(*n⋅ k*).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ctr | A[4] | A[3] | A[2] | A[1] | A[0] | *Cost* |
| 0 | 0 | 0 | 0 | 0 | 0 | *0* |
| 1 | 0 | 0 | 0 | 0 | 1 | *1* |
| 2 | 0 | 0 | 0 | 1 | 0 | *3* |
| 3 | 0 | 0 | 0 | 1 | 1 | *4* |
| 4 | 0 | 0 | 1 | 0 | 0 | *7* |
| 5 | 0 | 0 | 1 | 0 | 1 | *8* |
| 6 | 0 | 0 | 1 | 1 | 0 | *10* |
| 7 | 0 | 0 | 1 | 1 | 1 | *11* |
| 8 | 0 | 1 | 0 | 0 | 0 | *15* |
| 9 | 0 | 1 | 0 | 0 | 1 | *16* |
| 10 | 0 | 1 | 0 | 1 | 0 | *18* |
| 11 | 0 | 1 | 0 | 1 | 1 | *19* |

Total cost of *n* operations

A[0] flipped every op *n*

A[1] flipped every 2 ops *n*/2

A[2] flipped every 4 ops *n*/22

A[3] flipped every 8 ops *n*/23

… … … … …

A[*i*] flipped every 2*i* ops *n*/2*i*

Cost of n increments



Thus, the average cost of each increment operation is Q(*n*)/*n* = Q(1).

3. **Potential Method**

IDEA: View the bank account as the potential energy (as in physics) of the dynamic set.

FRAMEWORK:

* Start with an initial data structure *D*0.
* Operation *i* transforms *Di*–1 to *Di*.
* The cost of operation *i* is *ci*.
* Define a *potential function* F : {*Di*} → R,

such that F(*D*0 ) = 0 and F(*Di* ) ³ 0 for all *i*.

* The *amortized cost* *ĉi* with respect to F is defined to be *ĉi* = *ci* + F(*Di*) – F(*Di*–1).
* Like the accounting method, but think of the credit as *potential* stored with the *entire data structure*.
  + Accounting method stores credit with specific objects while potential method stores potential in the data structure as a whole.
  + Can release potential to pay for future operations
* Most flexible of the amortized analysis methods ).
* *ĉi* = *ci* + F(*Di*) – F(*Di*–1)
* If DF*i* > 0, then *ĉi* > *ci*. Operation *i* stores work in the data structure for later use.
* If DF*i* < 0, then *ĉi* < *ci*. The data structure delivers up stored work to help pay for operation *i*.

The total amortized cost of *n* operations is



Summing both sides telescopically



 since F(*Dn*) ³ 0 and F(*D*0 ) = 0.

Stack Example

Define: φ(Di) = #items in stack Thus, φ(D0)=0.

Plug in for operations:

Push: ĉi = ci + φ(Di) - φ(Di-1)

= 1 + j - (j-1)

= 2

Pop: ĉi = ci + φ(Di) - φ(Di-1)

= 1 + (j-1) - j

= 0

Multi-pop: ĉi = ci + φ(Di) - φ(Di-1)

= k’ + (j-k’) - j k’=min(|S|,k)

= 0

**Potential Method: Binary Counter**

Define the potential of the counter after the *i*th operation by F(*Di*) = *bi*, the number of 1’s in the counter after the *i*th operation.

Note:

* F(*D*0 ) = 0,
* F(*Di*) ³ 0 for all *i*.

Example

0 0 0 1 0 1 0

0 0 0 1$1 0 1$1 0

Assume *i*th INCREMENT resets *ti* bits (in line 3).

Actual cost *ci* = (*ti* + 1)

Number of 1’s after *i*th operation: *bi* = *bi*–1 – *ti* + 1

The amortized cost of the i th INCREMENT is

*ĉi* = *ci* + F(*Di*) – F(*Di*–1)

= (*ti* + 1) + (1 − *ti*)

= 2

Therefore, *n* INCREMENTs cost Q(*n*) in the worst case