* **Algorithm**
* Algorithm is a set of **clearly** specified steps to follow to solve a problem.
* Algorithm must have a **starting** point to start and an **end** pint to come to an end.
* Algorithm must be **correct** to solve a problem.
* A computer program is an algorithm expressed in a computer programming language.
* **Algorithm Analysis**
* Once the correctness of an algorithm is proven, the next issue is it’s **time** and **space** complexity.
* It discusses the **performance** issue of an algorithm in terms of **time** and **space**.
* One of the reasons for using computer is to process a **large amount** of data.
* If so, we need to make sure:
  + It terminates within a **reasonable amount of time**
  + It can finish the job using the **limited** amount of available memory.
* We discuss the time complexity of an algorithm.
* **Execution Time of a Program**
* The execution time of a program ***almost always*** depends on the amount of data it must process.
* So, the execution time of a program is a function of its input.
* The value of this function depends on many factors: compiler, speed of computer, the quality of the program.
* **Plot of different functions**
* We can plot the values of different functions on a graph to show how they grow when size of input increases.
* The following slide shows the plot of these four functions:
  + Linear F(N)
  + Logarithmic F(N log N)
  + Quadratic F(N2)
  + Cubic F(N3)
* **Order of efficiency**
* Figures 6.1 and 6.2 (previous slides) show that **Linear**, **N log N**, **Quadratic** and **Cubic** represent running time in order of decreasing performance.
* It means Linear is the most efficient and Cubic is the least efficient of these four functions.
* **An example**
* Suppose we want to download a file from the Internet and:

There is an initial 2-sec delay (to set up a connection). Data transfer rate is 1.6 KB/sec.

File size is N kilobytes.

The time to download is T(N) = 2 + N /1.6

* This is a linear function.

Downloading an 80KB file takes approximately 52 sec, a file of twice as large (160KB) takes about 102 sec, or roughly functions twice as long.

* This property, in which time essentially is directly proportional to the amount of input, is the nature of a linear algorithm, which is the most efficient algorithm.
* **Dominant Term of a Function**
* In a function the term with largest power is called the dominant term of that function.
* In a linear function like: T(N) = N /1.6 + 2, the dominant term is a constant c time N. The power of N is one.
* In a logarithmic function: N logN , c = log N.
* In a Quadratic function: constant c times N2.
* In a Cubic function: constant c time N3.
* **Big-Oh Notation**
* Big-Oh notation will be used to capture the most dominant term of a function as an **estimated** **execution** time of the function.
* Example: if the execution time of an algorithm is a polynomial like: 3N2+5N+25

we say the execution of this algorithm is O(N2).

* This notation (Big-Oh) can be used to establish a relative ordering among the functions.
* **Common Functions**
* The following are the functions that commonly describe algorithms running times in order of increasing growth rate:

**Function Name**

c Constant

log N Logarithmic

log2 N Log-squared

N Linear

N log N N log N

N2 Quadratic

N3 Cubic

2N Exponential

* Examples
* MINIMUM ELEMENT IN AN ARRAY

Given an array of n items, find the smallest item. It is O(N), Linear. Examine every item.

* CLOSEST POINTS IN THE PLANE

Given n points in a plane (that is, an x-y coordinate system), find the pair of points that are closest together. It is O(N2), Quadratic.

N(N-1)/2.

This problem can be solved in O(N logN) and even in O(N).

* COLINEAR POINTS IN THE PLANE

Given n points in a plane (that is, an x-y coordinate system), determine if any three form a straight line. It is O(N3), Cubic.

N(N-1)(N-2)/6.

* **The Maximum Contiguous Subsequence Sum Problem**
* Given (possibly negative) integers A1, A2, ..., An,

find (and identify the sequence corresponding to) the maximum value of SUM(Ai,…, Aj).

* The maximum contiguous subsequence sum is zero if all the integers are negative.
* As an example, if the input is {-2, 11, -4, 13, -5, 2}, the answer is 20, which represents the contiguous subsequence encompassing items 2 through 4.
* As a second example, for the input {1, -3, 4, -2, -1, 6}, the

answer is 7, the last four items.

* **The first Program With O(N3)**

1 // Cubic maximum contiguous subsequence sum algorithm.

2 // seqStart and seqEnd represent the actual best sequence.

3 template <class Comparable>

4 Comparable maxSubsequenceSum( const vector<Comparable> & a, int & seqStart, int & seqEnd )

6 {

7 int n = a.size( );

8 Comparable maxSum = 0;

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

11 for ( int j = i; j < n; j++ )

12 {

13 Comparable thisSum = 0; .

14 for( int k = i; k <= j; k++ ) . It is O(N3)

15 thisSum += a[ k ];

17 if( thisSum> maxSum )

18 {

19 maxSum = thisSum;

20 seqStart = i;

21 seqEnd = j;

22 }

23 }

25 return maxSum;

26 }

* **Another Program With O(N2)**

1 // Cubic maximum contiguous subsequence sum algorithm.

2 // seqStart and seqEnd represent the actual best sequence.

3 template <class Comparable>

4 Comparable maxSubsequenceSum( const vector<Comparable> & a,

S int & seqStart, int & seqEnd )

6 {

7 int n = a.size( );

8 Comparable maxSum = 0;

9

1. for (int i = 0; i < n; i++)
2. {
3. Comparable thisSum = 0;.

13 for ( int j = i; j < n; j++ )

14 {

15 thisSum += a[ j ];

16 This is O(N2)

17 if( thisSum> maxSum )

18 {

19 maxSum = thisSum;

20 seqStart = i;

21 seqEnd = j;

22 }

23 }

24 }

25 return maxSum;

26 }

* **The Third Program With O(N)**

1 // Cubic maximum contiguous subsequence sum algorithm.

2 // seqStart and seqEnd represent the actual best sequence.

3 template <class Comparable>

4 Comparable maxSubsequenceSum( const vector<Comparable> & a,

S int & seqStart, int & seqEnd )

6 {

7 int n = a.size( );

8 Comparable thisSum = 0;

9 Comparable maxSum = 0;

1. for( int i = 0, j = 0; j < n; j++ ) It is O(N)
2. {

12 thisSum += a[ j ];

15 if( thisSum> maxSum )

16 { This O(N)

17 maxSum = thisSum;

18 seqStart = i;

19 seqEnd = j;

20 }

21 else if (thisSum < 0)

22 {

23 i = j + 1;

24 thisSum = 0;

25 }

26 }

28 return maxSum;

26 }

* **REPEATED DOUBLING**

Starting from X = 1, how many times should X be doubled before it is at least as large as N?

Suppose that we start with $1 and double it every year. How long would it take to save a million dollars?

In this case, after 1 year we would have $2; after 2 year, $4; after 3 year, $8, and so on.

In general, after K years we would have 2K dollars, so we want to find smallest K satisfying 2K >= N. So K=┌logN┐.

After 20 yr, we would have more than a million dollars. The repeated doubling principle holds that, starting from 1, we can repeatedly double only ┌log N┐ times until we reach N.

* **REPEATED HALVING**

Starting from x = n, if n is repeatedly halved, how many iterations must be applied to make N smaller or equal to 1? The answer is ┌log N┐. If the division rounds down then the answer is └log N┘

* **Sequential Search**
* If the list is not sorted, we have to use sequential search to look for an item which is O(N).
* If the list is sorted, we can use binary search which is O(log N).
* **Binary Search**
* In binary search, we compare the target with middle element. If equal, we found it. If not we divide the list into two halves, lower half and upper half. And repeat the process on one of theses halves until we find the target or conclude that it is not there.
* Binary search is O( log N). But the list must be sorted.
* Examples

1. sum = 0;

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

sum++;

This is O(n)

2. sum = 0;

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

for (int j = 0; j <n; j++)

sum++;

This is O(n2)

* *Continued…*

3. sum = 0;

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

for (int j = 0; j <n \* n; j++)

sum++;

This is O(n3)

4. sum = 0;

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

for (int j = 0; j <i; j++)

sum++;

What is the execution time of this?



int f(int n)

{

if(n == 0)

return 0;

else

return 2\*f(n-1) + n\*n;

}



int bad(int n)

{

if (n == 0)

return 0;

else

return bad(n/3 + 1) + n - 1;

}