Lecture 9

Complexity Classes i.e., Constant, Linear, Quadratic; Analysis of Iterative Algorithms, Empirical Measurements of Performance; and Time & Space Tradeoffs in Algorithms.

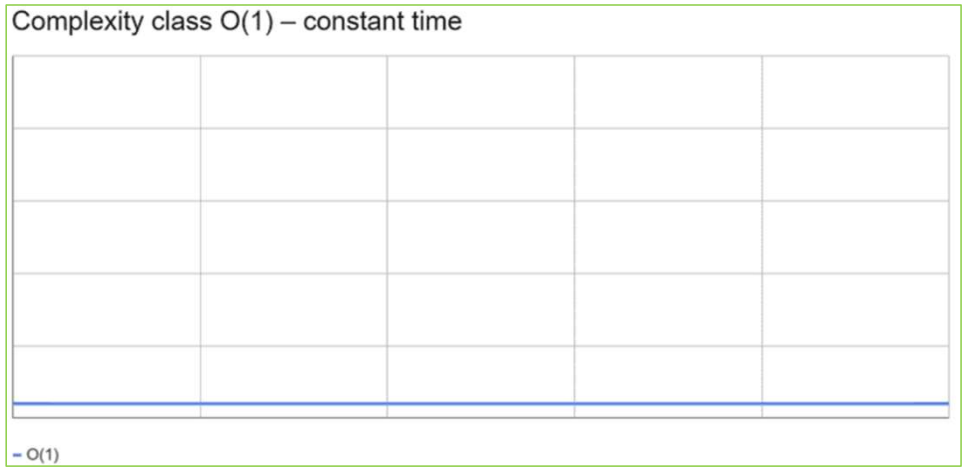
Complexity classes

- Algorithms can be divided into so-called complexity classes.
- A complexity class is identified by the Landau symbol *O* ("big O").

Complexity Class	F(N)	F(2N)	F(2N) / F(N) -- ratio
Constant	1	1	1
Linear	N	2N	2
Quadratic	N^2	$4N^2$	4
Cubic	N^3	$8N^3$	8
Logarithm	log(N)	1 + log(N)	1 + 1/log(N)
N-Log-N (or linearithmic)	N*log(N)	2N*log(N) + 2N	2 + 2/log(N)
Exponential	2^N	2^{2N}	2^N

Constant Time: O(1)

- Pronounced: "Order 1", "O of 1", "big O of 1"
- The runtime is constant, i.e., independent of the number of input elements *n*.
- In the following graph, the horizontal axis represents the number of input elements *n* (or more generally: the size of the input problem), and the vertical axis represents the time required.
- Since complexity classes can only be used to *classify* algorithms, but not to calculate their *exact running time*, the axes are not labeled.

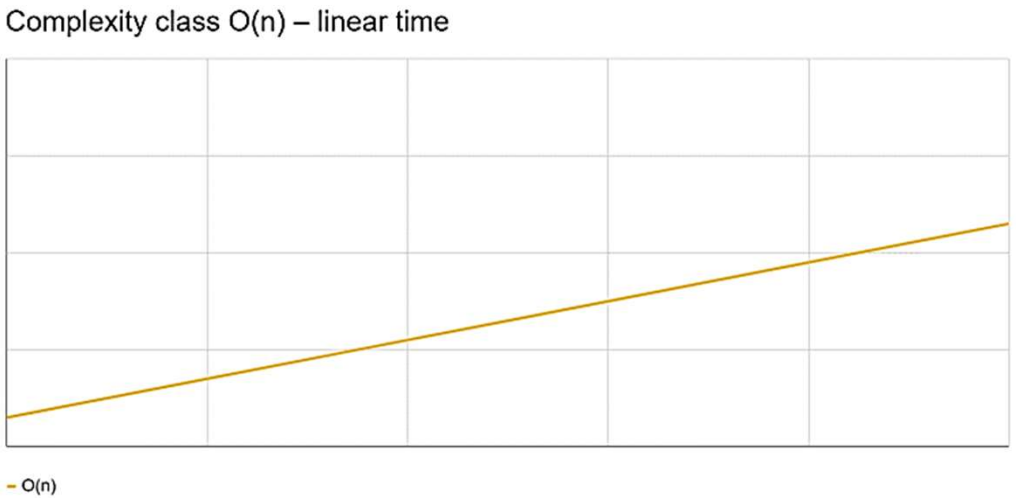


Two Examples

- Accessing a specific element of an array of size n: No matter how large the array is, accessing it via array[index] always takes the same time (Always not true).
- Inserting an element at the beginning of a linked list

Linear Time – O(n)

- Pronounced: "Order n", "O of n", "big O of n"
- The time grows linearly with the number of input elements *n*: If *n* doubles, then the time approximately doubles, too.
- "Approximately" because the effort may also include components with lower complexity classes. These become insignificant if *n* is sufficiently large, so they are omitted in the notation.

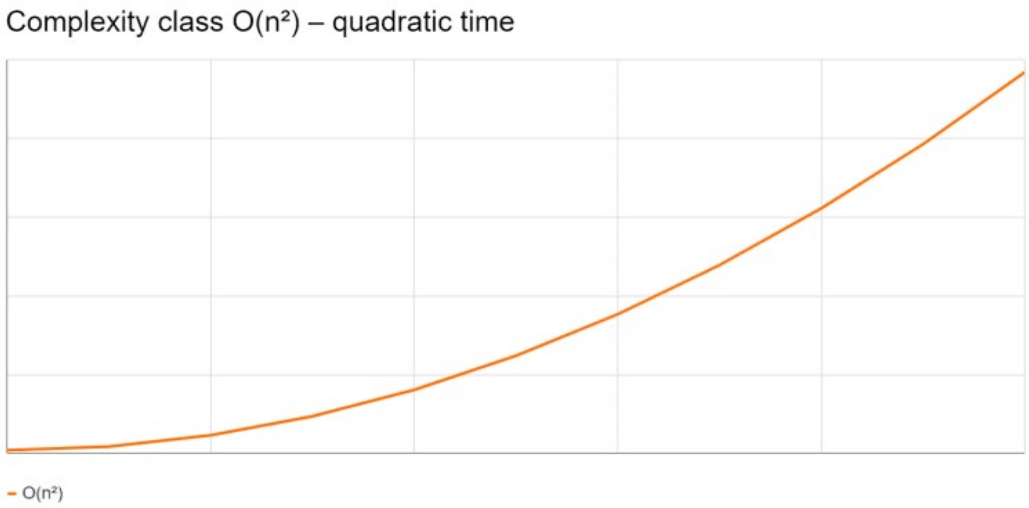


Examples

- Finding a specific element in an array: All elements of the array have to be examined – if there are twice as many elements, it takes twice as long.
- Summing up all elements of an array: Again, all elements must be looked at once – if the array is twice as large, it takes twice as long.

Quadratic Time – $O(n^2)$

- Pronounced: "Order n squared", "O of n squared", "big O of n squared"
- The time grows linearly to the square of the number of input elements: If the number of input elements *n* doubles, then the time roughly quadruples. (And if the number of elements increases tenfold, the effort increases by a factor of one hundred!)

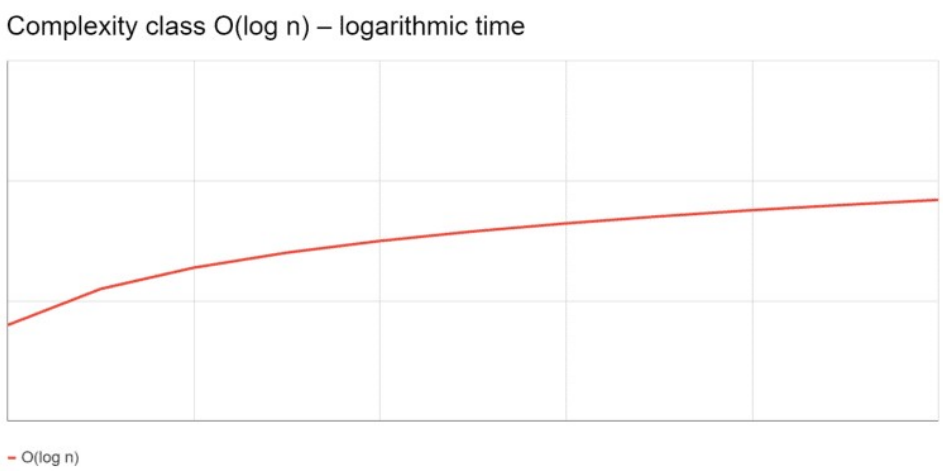


Examples of quadratic time are simple sorting algorithms like

- Insertion Sort,
- Selection Sort, and
- Bubble Sort.

Logarithmic Time – O(log n)

- Pronounced: "Order log n", "O of log n", "big O of log n"
- The effort increases approximately by a constant amount when the number of input elements doubles.
- For example, if the time increases by one second when the number of input elements increases from 1,000 to 2,000, it only increases by another second when the effort increases to 4,000. And again, by one more second when the effort grows to 8,000.



An example of logarithmic growth is the binary search for a specific element in a sorted array of size n.

Complexity Classes or Types

- The following are the important complexity classes:
  - **P Class**
  - **NP Class**
  - **CoNP Class**
  - **NP hard**
  - **NP complete**

Types of Complexity

P CLASS

- The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.
- This class contains many natural problems like:
  - **Calculating the greatest common divisor.**
  - **Finding a maximum matching.**
  - **Decision versions of linear programming.**

NP CLASS

- The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- This class contains many problems that one would like to be able to solve effectively:
  - **Boolean Satisfiability Problem (SAT).**
  - **Hamiltonian Path Problem.**
  - **Graph coloring.**

Types of Complexity

CO-NP CLASS

- Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.
- Some example problems for C0-NP are:
  - **To check prime number.**
  - **Integer Factorization.**

NP-HARD

- An NP-hard problem is at least as hard as the hardest problem in NP, and it is the class of the problems such that every problem in NP reduces to NP-hard.
- Some of the examples of problems in Np-hard are:
  - **Halting problem.**
  - **Qualified Boolean formulas.**
  - **No Hamiltonian cycle.**

Types of Complexity

NP-COMPLETE CLASS

- A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.
- Some example problems include:
  - **Decision version of 0/1 Knapsack.**
  - **Hamiltonian Cycle.**
  - **Satisfiability.**
  - **Vertex cover.**

```
Complexity   Algorithm: O(n log n)                                   Greedy Method
O(n)         FOR i=1 to n
                  Calculate Profit/Weight
O(n log n)        Sort Objects in decreasing order of Profit/Weight ratio
O(n)              FOR i=1 to n
                        IF m>0 and wi≤m THEN
                              m=m-wi
                              p=p+pi
                        else  break;
                        if m>0
                                    p=p+pi*(m/wi)
                        end if
```

For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item, then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item.

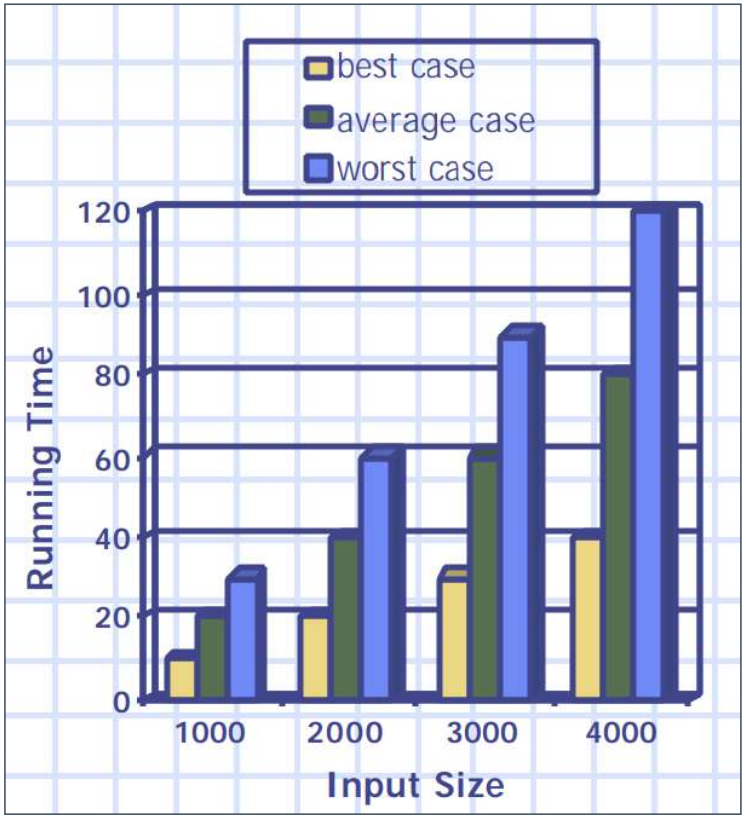
Analysis of Algorithms

- Two essential approaches to measuring algorithm efficiency:
- Empirical analysis:
  - Program the algorithm and measure its running time on example instances
- Theoretical analysis
  - Employ mathematical techniques to derive a *function* which relates the **running time to the** ***size of instance***

- In this cousre our focus will be on Threoretical Analysis.

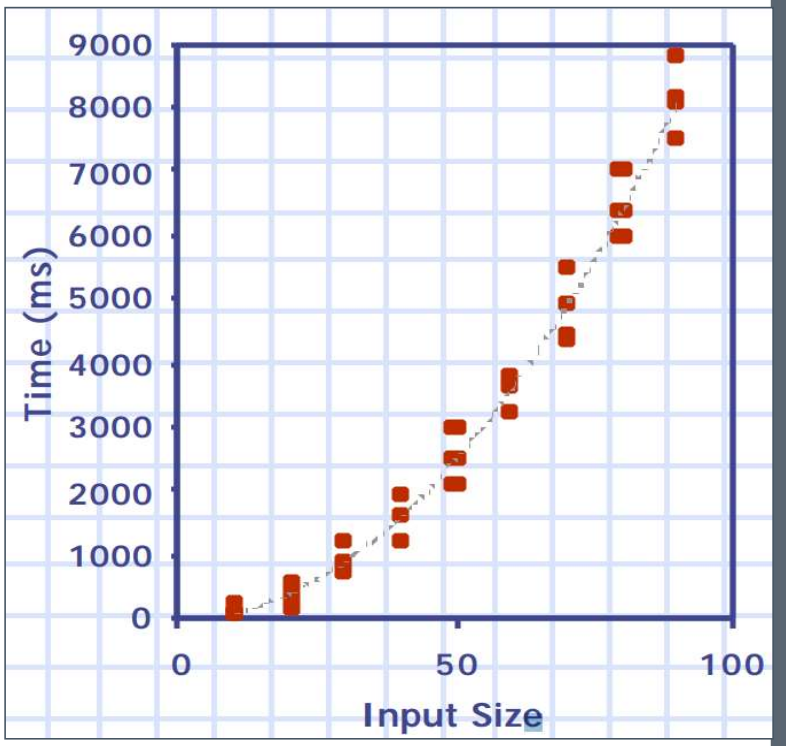
Empirical Analysis

- Most algorithms transform input objects into output objects
- The running time of an algorithm typically grows with the input size
- Average case time is often difficult to determine
- We focus on the worst-case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



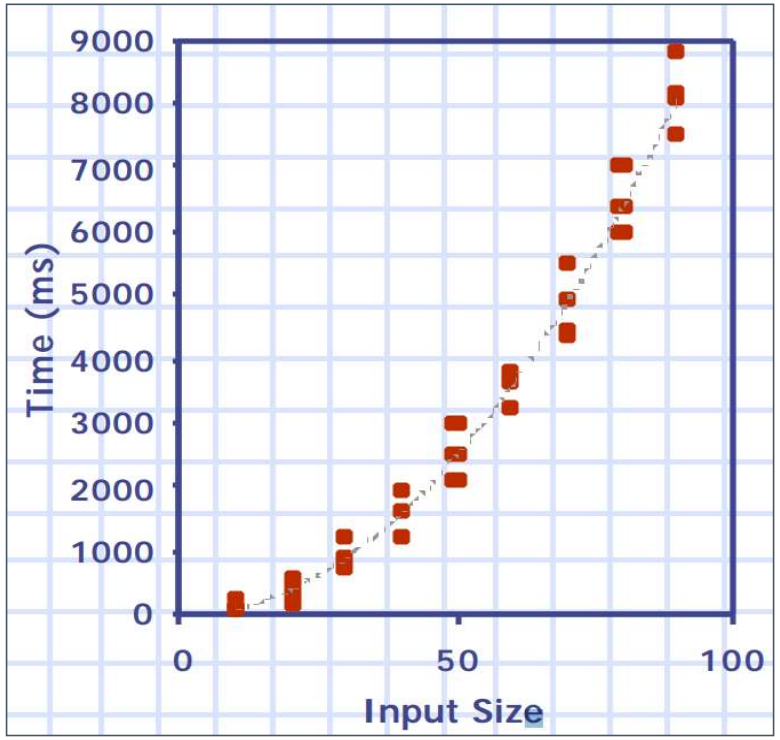
Empirical Analysis

- Write a program implementing the algorithm
- Run the program with inputs of varying size and compositions
- Use timing routines to get an accurate measure of the actual running time
  e.g. *System.currentTimeMillis()*
- Plot the results



Limitations of Empirical Analysis

- Implementation dependent
  - Execution time differ for different implementations of same program
- Platform dependent
  - Execution time differ on different architectures
- Data dependent
  - Execution time is sensitive to amount and type of data minipulated.
- Language dependent
  - Execution time differ for same code, coded in different languages



∴ absolute measure for an algorithm is not appropriate

Theorerical Analysis

- Data independent
  - Takes into account all possible inputs
- Platform independent
- Language independent
- Implementatiton independent
  - not dependent on skill of programmer
  - can save time of programming an inefficient solution
- Characterizes running time as a function of input size, n. Easy to extrapolate without risk

Why Analysis of Algorithms?

- For real-time problems, we would like to prove that an algorithm terminates in a given time.
- Algorithmics may indicate which is the best and fastest solution to a problem without having to code up and test different solutions
- Many problems are in a complexity class for which no practical algorithms are known
  - better to know this before wasting a lot of time trying to develop a ”perfect” solution: **verification**

Complexity of an Algorithm

- The complexity of an algorithm is the amount of work the algorithm performs to complete its task. It is the **level of difficulty** in solving mathematically posed problems as measured by:
  - Time (time complexity)
  - No. of steps or arithmetic operations (computational complexity)
  - Memory space required (space complexity)
- Complexity is a function T(n) which yields the time (or space) required to execute the algorithm of a problem of size 'n'.

Analysis Example

Algorithm:	Number of times executed		
1. n = read input from user	1		
2. sum = 0	1		
3. i = 0	1		
4. while i < n	n		
5. number = read input from user	n	or	$\sum_{i=0}^{n-1} 1$
6. sum = sum + number	n	or	$\sum_{i=0}^{n-1} 1$
7. i = i + 1	n	or	$\sum_{i=0}^{n-1} 1$
8. mean = sum / n	1		

The computing time for this algorithm in terms on input size n is:

$T(n) = 1 + 1 + 1 + n + n + n + n + 1$

$T(n) = 4n + 4$

Counting Primitive Operations

```
Algorithm ArrayMax(A,n)
        {An array A storing N integers
          and find the largest element in A.}

        currentMax  =   A[0]          2 steps + 1 to initialize i
        for (i=1;i<=n-1;i++)          2 step each time (compare i to n, inc i)
                if (currentMax < A[i])    2 steps
                currentMax  =   A[i]      2 steps
        return currentMax             1 step
```

n-1 times — braces the loop body (the `if` and the assignment)

How often done??

Between 4(n-1) and 6(n-1) in the loop

It depends on the order the numbers appear in in A[]

Run Time Analysis

› *Pseudo code to find product of two numbers*

```
int method()
{
   int a,b,c;   ---------  c1
   a=2;         ---------  c2
   b=3;         --------- c2
   c= a*b;      ---------  c3
    printf("Product of a and b is %d", c);
   return 0;    ----------- c4
}
```

Run Time Complexity will be =

c1+c2+c2 + c3 + c4

= c1 + 2c2 + c3 + c4 (as all constant)

= C

Run Time Analysis (Cont !!!)

```
int method()
{
  int a,b,large;   ---------  c1
  a=2;         ---------  c2
  b=3;         ---------  c2
  if(a>b)      --------  c3
    large=a;  -------- c2
 else
    large = b;  ------- c2
  printf("Large number is %d", large);
Return 0;    ---------- c4
}
```

Run Time Complexity will be =

c1+c2+c2 + c3 +c2 + c2 + c4

= c1 + 4c2 + c3 +c4 (as all constants)

= C

Note:- =You can say there should be 3c2 instead of 4c2 (Coefficient have no impact)

Run Time Analysis (Cont !!!)

```
int method()
{
  int I, codd, ceven, a[5]={5,4,3,2,1}; ---------  c1
ceven=0;codd=0;   ----------- c2
for(i=0;i<=4;i++)    ------ c3
if(a[i]%2==0)     --------  c4
     ceven++;-------- c5
 else
    codd++; ------- c5
   printf("Total evens %d and Total Odd %d", ceven, codd);
Return 0;    ---------- c6
}
```

Run Time Complexity will be = c1+c2

+ n* (c3 +c4 + c5)+ c6

= c1 + c2 + n*c +c6

= n*c + c

= n

Run Time Analysis (Cont !!!)

```
int method()
{
  int I, a[5]; ---------  c1
N for(i=0;i<=4;i++)    ------ c2
    Scanf("%d", &a[i]);
  for(i=0;i<=4;i++)
N     A[i]=a[i]+5;  ----------- c3
  for(i=0;i<=4;i++)
N       print("%d", a[i]);
  return 0;    ---------- c4
}
```

Run Time Complexity will be = c1 + n * c2 + n*(c1+c3) + n * c2 + c4

= c1 + 2n*c2 + n * c

= c1 + n * c + n * c

= 2n*c +c1

= 2n

=n

Run Time Analysis (Cont !!!)

```
int method()
{
  int r,c, a[][]= { {1,2}, {1,3}}; ---------  c1
  for(c=0;c<=1;c++)    ------ c2
   for(r=0;r<=1;r++)
        a[r][c]=a[r][c]+5;---------- c3+c4+c5
   for(c=0;c<=1;c++)
   for(r=0;r<=1;r++)
        printf("%d", a[r][c]);
  return 0;    ---------- c6
}
```

$N * m = n^2$

$N * m = n^2$

Note:-

- ***C3 refer to array scripts use***
- ***C4 refer to arithmetic op +***
- ***C5 refer to assignment***

Run Time Complexity will be = c1 + n * m * (c2+c3+c4+c5) + n * m * c4 + c6

= c1 +c6 + n*m*c + n*m*c +

= 2*n*m*c + c

= n*m*c

= n*m = n^2

Run Time Analysis (Cont !!!)

```
int method()
{
  int r,c,,s, a[][]= { {1,2}, {1,3}}; ---------  c1
S=0;                  ------------- c2
for(c=0;c<=1;c++)    ------ c3
{    for(r=0;r<=1;r++)
      a[r][c]=a[r][c]+5;   ---- c4+c5 +c2
s=s+a[r][0];   ------------- c4+c5 +c2
}
Printf("Sum of element of 1st row %d", s);
return 0;    ---------- c6
}
```

N times
$N * m = n^2$

Note:-

- ***C4 refer to array scripts use***
- ***C5 refer to arithmetic op +***
- ***C2 refer to assignment***

Run Time Complexity will be = c1 + n * m * (c2+c3+c4+c5) + n (c2+c3+c4+c5)+c6

$= c1+c6 + n*m*c + n*c$

$= n*m + n + c$

$= n^2 + n$

How do we analyze algorithms (Cont !!!)

(3). Express running time as a function of the input size n (i.e., f(n)).

- *To compare two algorithms with running times f(n) and g(n), we need a **rough measure** of **how fast a function grows**.*
- *Such an analysis is independent of machine time, programming style, etc.*

Example-1. (Run time Cost Analysis)

- *Associate a "cost" with each statement.*
- *Find the "total cost" by finding the total number of times each statement is executed.*

Algorithm 1		***Algorithm 2***	
Cost		***Cost***	
arr[0] = 0;	c_1	*for(i=0; i<N; i++)*	c_2
arr[1] = 0;	c_1	*arr[i] = 0;*	c_1
arr[2] = 0;	c_1		
…	*…*		
arr[N-1] = 0;	c_1		
---		---	
$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 =$	$(c_2 + c_1) \times N + c_2$

Example-2 . (Run time Cost Analysis)

- ***Algorithm 3*** *Cost*

```
sum = 0;                          c1
for(i=0; i<N; i++)                c2
  for(j=0; j<N; j++)              c2
      sum += arr[i][j];           c3
                                  ----------
```

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Time and Space Tradeoff

- Definition
  - A way of solving a problem in computer science with
    - Less time by using more space or memory
    - Less space with a long time
  - Is known as a Time-Space tradeoff.
- Example

More time, less space	Less time, More space
1. int a,b; 2. printf("enter value of a\n"); 3. scanf("%d",&a); 4. printf("enter value of b\n"); 5. scanf("%d",&b); 6. b=a+b; 7. printf("output is:%d",b);	1. int a,b,c; 2. printf("enter values of a,b and c \n"); 3. scanf("%d%d%d",&a,&b,&c); 4. printf("output is: %d",c=a+b);

Thank You!!!

Have a good day