Youtube

Time Complexity

①

```
#include <stdio.h>
int main()
{
    int n1, n2, result
    n1 = 10;
    n2 = 20;
    result = n1 + n2
    result 0;
}
```

এখানে Operation : 3 টি assign
1 টি addition
Total 4 টি

But complexity O(1)
কারণ input যাই থেকে operation একবক থাকে, তাই O(1) হবে।
(O(4) হবে না)

②

```
int main()
{ int n, result;
  scanf("%d", &n);
  result = n*(n+1)/2;
  printf("result = %d\n", result);
  return 0;
}
```

Operation : ৪টি
৩টি assign
৩টি গাণিতিক
Total = 4টি

n এর যেকোনো মানের জন্য operation একবক তাই এর complexity O(1)

③

(3)

```
#include <stdio.h>
int main()
{ int i, n, result;
  scanf("%d", &n);
  result = 0;
  for(i = 1; i<=n ; i++){
    result = result + i ;
  }
  printf("result = %d\n", result);
  return 0;
}
```

$n=3$ হলে

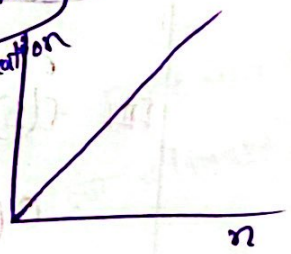
Operation: $n = n+1$, $n = n+2$, $n = n+3$ — প্রত্যেকবার ১ বার assign ও ১ বার add হচ্ছে

$n=1$ হলে, Operation 2
$n=2$ " " 4
$n=3$ " " 6
$n=10$ " " 20

তাই Complexity $= O(2n)$
$= 2 \times O(n)$
$= O(n)$

(2 একটি Constant তাই ignore করলাম)

$\therefore$ Complexity $O(n)$ বা linear complexity কারণ: operation এখানে n, operation এর সাথে linearly বাড়ে)



(4)

```
int main()
{ int, i, n, j, count;
  scanf("%d", &n);
  count = 0;
  for (i=0; i<n; i++){
    for (j=0; j<n; j++){
      count = count+1
    }
  }
  printf("n=%d , count=%d \n", n, count);
  return 0;
}
```

যখন 2টি loop, n এর উপর নির্ভর করছে তখন, ⇒

n	count (operation)
1	1
2	4
3	9
10	100

Complexity $O(n^2)$

- ২টি loop যদি n তের উপর নির্ভর করে, তাহলে $O(n^2)$
- যদি nested loop n " " নির্ভর না করে nested loop যদি ~~i=0~~ (j=0; j<3; j++) হতো তাহলে আমার complexity আসতো $O(3n)$ তখন হতো constant ignore করলে $O(n)$ আসতো।

- ~~O(1), 0~~ → complexity. ভালো (Time কম লাগে) O(1) (Time complexity কম)

$O(1) > O(\log n) > O(\sqrt{n}) > O(n) > O(n^2) > O(n^3) > O(n^n)$

- Time বেশি লাগে $O(n^n)$, তাই এটি খারাপ complexity. অর্থাৎ Time complexity বেশি।

- O(1), O(log n), O(n), O(n log n), O(n^2), O(2^n), O(n!) → (খারাপ) complexity (বেশি Time লাগে)

```
int main()
{ int i,j, n, count;
scanf("%d", &n);
count=0;
for(i=0;i<n;i++){
  for(j=0; j<n ; j++){
    count = count+1;
}
}
  for(i=0; i<n ; i++){
    count = count+1;
}
printf("count = %d\n", count);
return 0; }
```

2টি আলাদা n এর উপর নির্ভর করে তাই complexity $O(n^2)$

এখানে একটি loop O(n)

Total = $O(n^2) + O(n) = O(n^2+n)$ (কমটি লিখতে হয়) $= O\,n^2$

- $O(n^4 + n^3 + n^2 + n)$ হলে Time Complexity = $O(n^4)$ হবে।
- $O(n! + n^3 + n^2 + n)$ হলে Time Complexity = $O(n!)$ হবে।
  (cz এখানে n^3 এর থেকে $n!$ বেশি দ্রুত বাড়ে)
- $O(4n^4 + 2n^3 + 3n^2 + n)$ হলে Time Complexity = $O(n^4)$ হবে

~~Space Complexity~~

(*) linear search is an algorithm. and algorithm is a strategy. complexity : Big O(n). (*) easy to implement.

Insersion :

(*) Insertion sort is a sorting algorithm. problem is: sort. ~~best~~ worst case $O(n^2)$, $O(n)$.
→ যখন list sort থাকে almost।

(*) Asending ~~order~~ order বা disending sort করতে চাইলে $O(n^2)$

(*) outer loop starts from 2nd element.. times (k+1) if starts from k.

(*) Binary search :

complexity $O(\log_2 n) = \log_2^{15} = 4$.

for linear search : $O(n) = 15$

~~Q.111~~

in Binary search

$$mid = \frac{0+13}{2} = 6.5 \approx 6$$ (ez integer show করবে)

* Write a program to sort 1st half of an array in ascending order and 2nd half of the array in descending order.

Insertion sort code

```
#include <stdio.h>
void insertionsort (int A[], int length) {
for (int j=1 ; j<length; j++) {
 int key = A[j];
 int i = j-1;
 while (i>=0 && A[i] > key) {
    A[i+1] = A[i];
      i--;
   }
  A[i+1] =key;
  }
}
void descendingorder (int A[], int length) {
    for (int j=1 ; j< length; j++) {
      int key = A[j];
      int i = j- 1;
    }
    A[i+1] = key;
  }
}
int main() {
 int A[] = {2, 4,5, 9,71};
 int length = sizeof (A) /sizeof A [0] );
 printf("original Array:) ;
```

```
for (int i = 0; i< length; i++){
    printf("%d", A[i]);
}
 printf("\n");
insertionsort(A, length);
printf("sorted array in ascending:");
for (int i =0; i< length; i++){
    printf("%d", A[i]);
   }
 printf("\n");
descendingorder(A, length);
printf("sorted array (Descending):");
   for(int i=0; i<length; i++) {
     printf("%d", A[i]);
   }
    printf("\n");
   return 0;
}
```

Space Complexity

```
#include<stdio.h>
int main()
{ int n;
  scanf("%d", &n);
 if (n%2==0){
  printf("%d is even number.\n",n);
  } else{
      printf("%d is odd number\n",n);
  }
   return 0;
}
```

Time complexity
O(1)

Space complexity
O(1)
এই প্রোগ্রাম যত বার রান হোক না কেন আমরা ~~শুধু~~ একটি variable use করেছি।

```
#include<stdio.h>
int main()
{ int i, n, even[101];
for (i=0; i<101; i++){
 even[i]=0;
}
scanf("%d", &n);
if (even[n]){
 printf("%d\n");
} else{
    printf("%d", n);
}
 return 0;
}
```

O(n)

এই n এর উপর নির্ভর ?

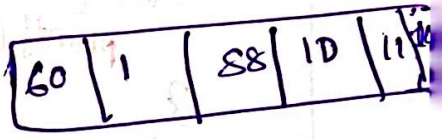
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    s[i][j] = a[i][j] + b[i][j];
  }
}
```

Space complexity :-

$O(n^2)$

Linear Search

```
int linear_search (int A[], int n, int x)
{
    int i ;
    for(i=0; i<n; i++){
        if (A[i] == x){
            return i;
        }
    }
    i = -1 ;
    return i;
}
```



- worst complexity = $O(n)$
- কারণ i variable টাকে use করে কাজ করছি তাহলে space complexity O(1).

Linear search এ n যতটা তারপরই খোঁজা লাগবে।

Binary Search

শর্ত: ⊛ এই array টি ছোট থেকে বড় অথবা বড় থেকে ছোট সাজানো থাকতে হবে।

⊛ এটি middle point হিসেবে mid করে ⇒ $\frac{high+low}{2}$.

-1	4	6	7	10	19	22	23	30	35	39	46	49	50	52	55	61	67	70	71
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- $\log_2 n = \square$ হলে Binary search যদি n সংখ্যক column এর array তে তাহলে Operation হবে $\square$ বার।

Example : • $\log_2 16 = 4$. বা, $2^4 = 16$

Here, Operation ~~Here~~ 4 বার, $n = 16$. মানে 16 টি ঘরের একটি array থাকবে।

• $\log_2 32 = 5$ বা $2^5 = 32$.

32 ঘরের একটি array থাকলে Operation হবে 5 বার। কিন্তু linear search এ operation হতো 32 বার।

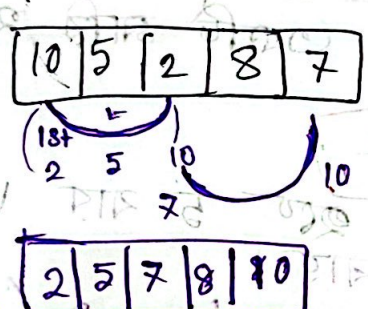
তাই Binary Search এর time complexity $O(\log_2 n)$.

best O(1)
Avg O(log n)
worst O(log n)

```
int binary_search (int A[], int n, int x)
{
    int left, right, mid;
    left = 0;
    right = n-1;
    while (left <= right){
        mid = (left + right) / 2;
        if (A[mid] == x){
            return mid;
        }
        if (A[mid] < x) { left = mid+1; }
```

```
else {
    right = mid - 1 ;
    }
}
return -1 ;
}
```

সিলেকশন সর্ট (Selection Sort)



$n + n-1 + n-2 + \cdots + 1$

$\Rightarrow$ বারার অ্যাভারেজ $= n \times \frac{(n-1)}{2n} \Rightarrow \frac{1}{2}(n^2 - n)$

Complexity $= O(n^2)$

(সবথেকে ছোটটিকে ক্রমান্বয়ে ১ম জায়গা থেকে select করে বড় থেকে ছোট থেকে বড় সাজানো হয়)

Implementation code:

```
void selection_sort (int A[] , int n)   → কয়টি সংখ্যা আছে
{
    int i , j, index_min, temp ;
    for(i =0; i < n-1 ; i++){
        index_min = i;
        for(j = i+1; j < n; j++){
            if (A[j] < A[index_min]) {
                index_min = j ;
            }
        }
```

```
if (index_min != i) {
    temp = A[i];
    A[i] = A[index_min];
    A[index_min] = temp;
    }
  }
}
```

বাবল সর্ট (Bubble sort)

10	5	2	8	7

এর নিয়ম ২য় ইন্ডেক্স মধ্যে ছোট বড় তুলনা করে করে সাজাতে হবে।

5 10 2 8 7

5 2 10 8 7

5 2 8 10 7

5 2 8 7 | 10 সবচেয়ে বড় সংখ্যা সবার শেষে তাই এটি আর করা যায় না।

2 5 8 7 | 10

2 5 7 | 8 | 10 সবচেয়ে বড়।

17 বড়।

15 বড়।

2

Complexity $O(n^2)$
কারণ ২টি loop চলে।

Space complexity $O(1)$
constant কারণ প্রতি array assign করে বার বার সেটি দিয়েই কাজ করছি।

Code implementation :

```
void bubble_sort (int A [ ], int n)
{
    int i, j, temp ;
    for(i=0 ; i<n ; i++) {
        for(j=0; j<n-i-1 ; j++) {
            if (A[j] > A [j+1]) {
                temp =A[j] ;
                A[j] = A[j+1] ;
                A[j+1] =temp;
            }
        }
    }
}
```

ইনসার্শন সর্ট (Insertion sort)

1	2	3	5	4

ছোট থেকে বড় :-

⇓ অলরেডি আছে।

শুধুই এই দুটি Change হবে

তাই এটির কাজ কম এটি Time Complexity হবে Best case.

∴ Best case Time Complexity O(n)

2	4	3	1	5

এখানে এক এক করে সবগুলো Change হবে তাই এর Time complexity হবে $O(n^2)$.

sorted ↑ unsorted

0	1	2	3	4
43	31	26	29	12

n=5.

temp = 31 → 1

0	1	2	3	4
31	43	26	29	12

Sorting ← → unsorting

temp=26

0	1	2	3	4
26	31	43	29	12

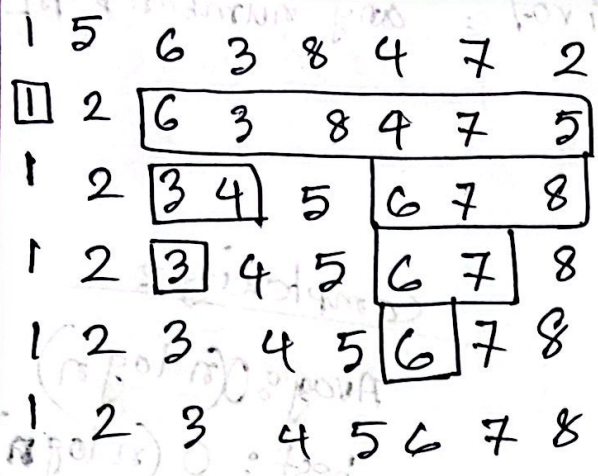
temp=29

26	29	31	43	12

temp=12

12	26	29	31	43

Quick sort




```
void quick_sort(int A[], int low, int high)
{
    if (low >= high){
        return;
    }
    int p;
    p = partition (A, low, high);
    quick_sort(A, low, p-1);
    quick_sort(A, p+1, high);
}
```

(Quick sort, Algorithm)

Let, pivot = any number from A[].

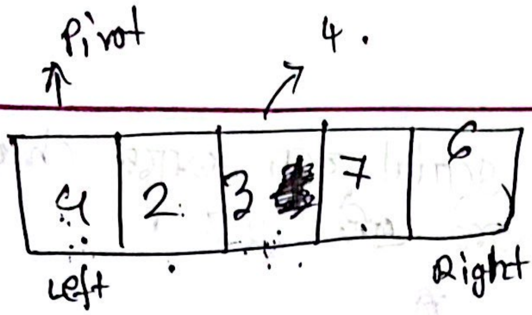
```
{
lower value = i
higher value = j
while (i<j)
{ while A[i] <= pivot
  { i++;
  }
while A[j] > pivot
  { j--;
  }
swap হবে যদি i<j হয়।

যদি i<j
  {
    swap (a[i], a[j])
  }
}
যদি i >= j হয় তাহলে
  {
    swap (pivot and a[j])
    অথবা A[l]
  }
return j;
}
```

Complexity:

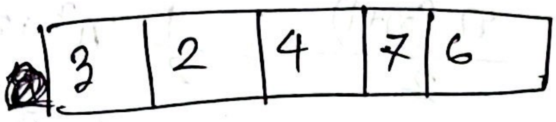
Avg: O(n log n)

best: O(n log n)

worst: O(n²)



find the exact position for first pivot.

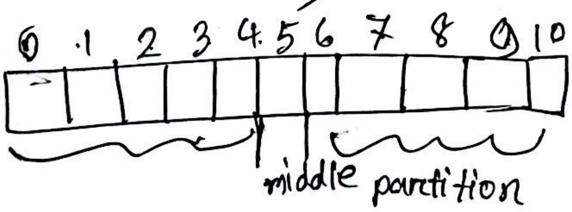


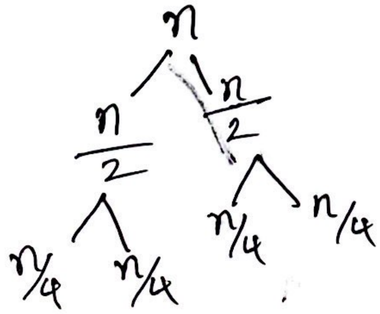
=α=

Algorithm : Quicksort (A, l, h)

```
{ If (l<h)
  {
    j = partition (A, l, h);
    Quicksort (A,l,j-1);
    Quicksort (A,j+1, h);
  }
}
```

Best case:





$\frac{n}{2^k} = 1.$

$\Rightarrow n = 2^k$

$\Rightarrow \log n = \log 2^k$

$\Rightarrow \log n = k \log 2$

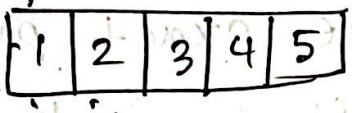
$\Rightarrow k = \log n$

$O(n \log n)$

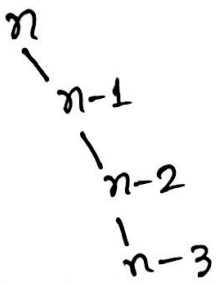
Computer Binary log, কাজে লাগে তাই log কাটা যা

~~$\Rightarrow \frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \log n = \log 2^k \Rightarrow \log n = k \log_2$~~

worst case:



এক্ষেত্রে serial array তাহে change হয় না তাই worst হয়।



$$\Rightarrow \frac{n(n+1)}{2} = \frac{1}{2}n^2 + n$$
$$= O(n^2)$$

∴ worst case $O(n^2)$
Average " $O(n \log n)$
Best case $O(n \log n)$

— ০ —