DATA STRUCTURES & ALGORITHMS (CS1004)

function oriented → supports lambda oriented functions

procedure oriented → doesn't support hoops concept

COURSE CONTENT :
- review of elementary data structures
- analysis of recurrence relations

Linear Search → Best: 1st element.
worst: last element.

Binary Search →
- if elements sorted, we can do.
- check if size of reqd. element is greater than / lesser than the middle element & search accordingly.

BOOK → M.A. Weiss, Data Structures and Algorithms.

EVALUATION :

CS1004
- 35 marks for pass, then on relative.
- 25 marks for continuous evaluation. (10 marks assignment, 15 marks for surprise test)
- 25 marks for mid-sem.
- 50 marks for end-sem.

CS1005
- min 35 marks, then on relative.
- 25 marks continuous evaluation. (surprise test or viva)
- 25 marks for mid-sem.
- 50 marks for end-sem.

ELEMENTARY DATA STRUCTURES

- Data Structure → way of organizing & storing data in memory / external storage devices.
  → helps facilitate insertion, retrieval and manipulation of data.

Volume, Variety and Velocity (VVV) of data

DATA STRUCTURES → Linear
→ Non-Linear

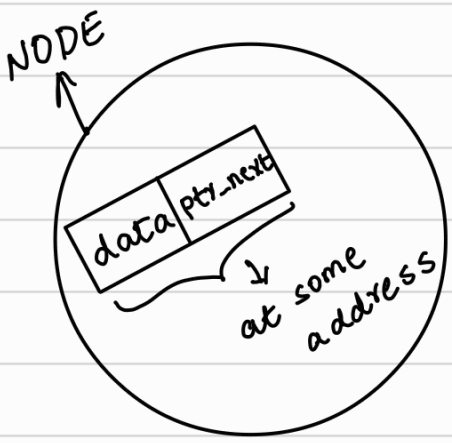
Linear : • Allows data elements to be arranged in a sequential / linear fashion.
• each element → attached to previous and next element.
• E.g. Array, Stack, Queue, Linked List.

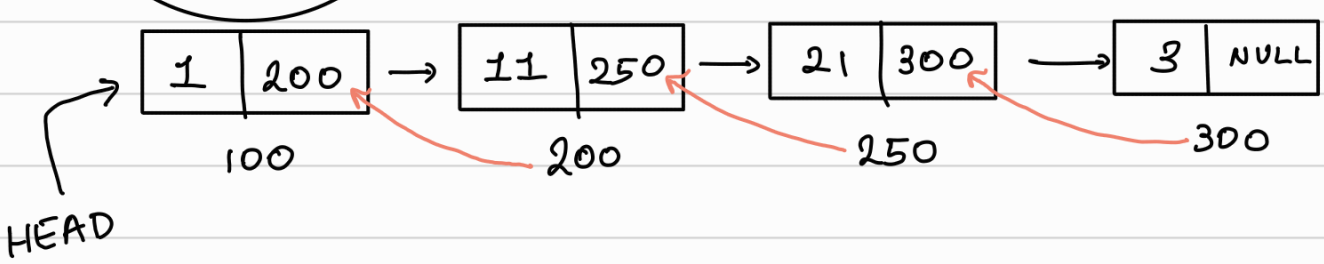
Non-Linear : • Data isn't arranged sequentially
• can't traverse thru all elements in single run.
• E.g. Graphs and Trees.

ARRAY → • Static memory allocation → can't add more elements
• stores only one data_type
• deletion of data / insertion of data is cumbersome.

So we use * Linked List *

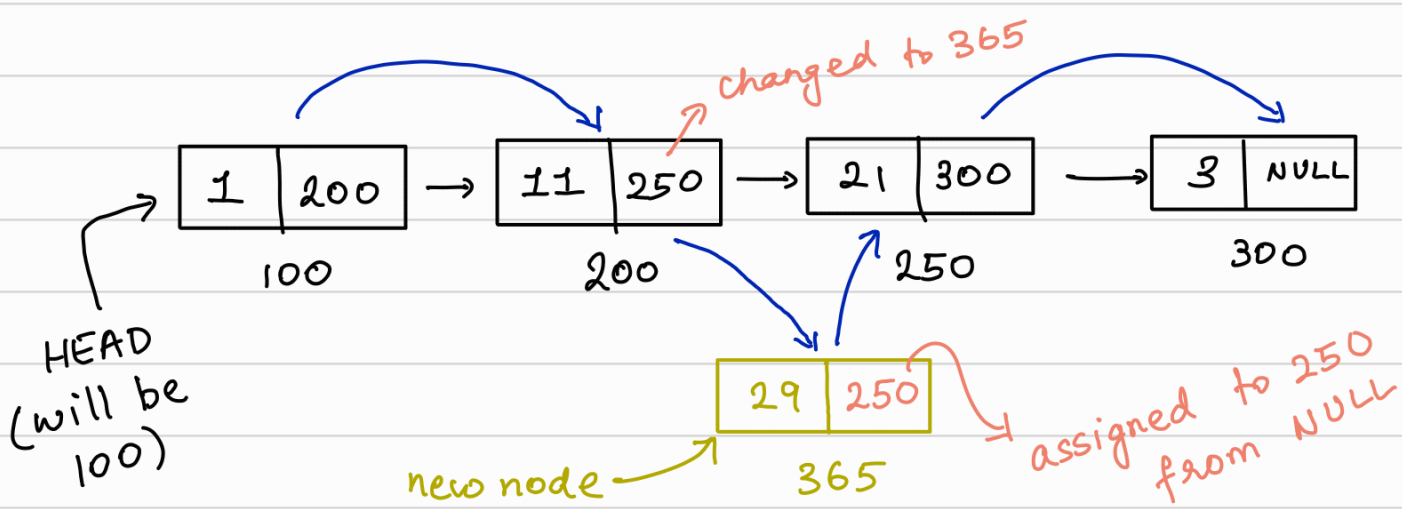
LINKED LIST → • Linear Data Structures in which the elements (nodes) are connected together via pointers (or) references.
• They allow Dynamic Memory Allocation, and data aren't stored in a contiguous manner. This makes insertion & deletion of elements more flexible.





GRAPHS → collection of edges and vertices forming a loop.

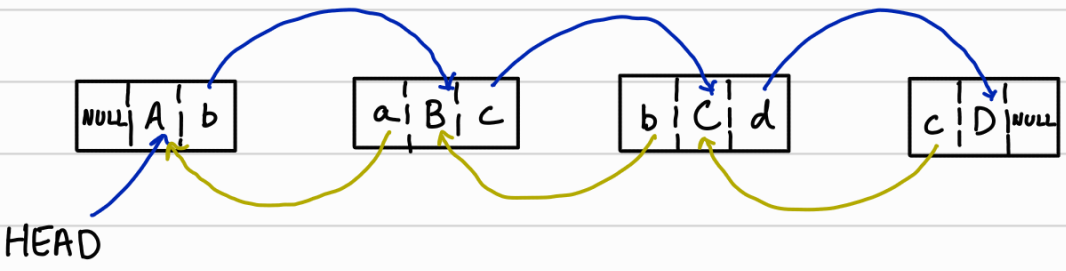
SINGLE LINKED LIST : • singly linked list if elements are not stored in contiguous memory locations, & each node is ONLY connected to its next element, using pointers.



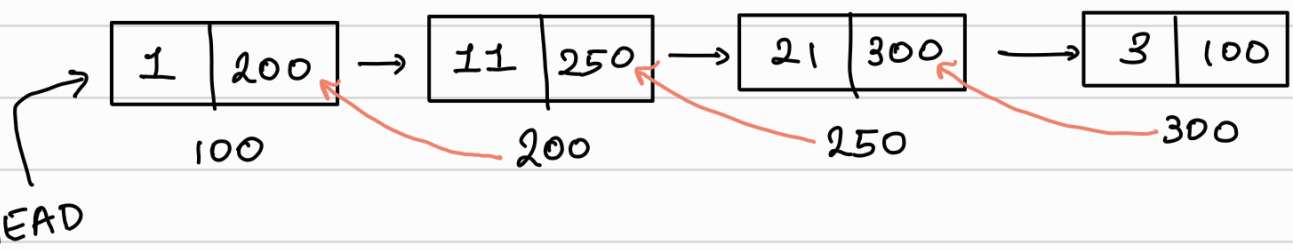
➤ To add a new node / data at ANY RANDOM POINT.

DRAWBACK of SINGLE LINKED LIST : Move only in forward direction.

DOUBLE LINKED LIST : • traverse forward and backward



CIRCULAR LINKED LIST : Last node is connected to first



STACK : follows Last In First Out (LIFO)
(or)
First In Last Out (FILO)

when stack is empty → top = -1

to insert element → push (check if stack is full/not)

to delete element → pop

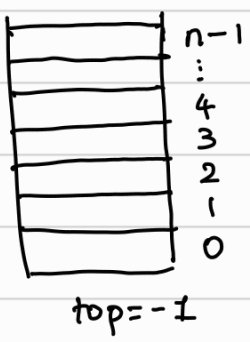
PUSH

stack is full if : top = size-1

if top != size-1, then :

top++;

a[top] = variable.



if top = some K, then some var is stored at K. so, top++ (goes above) and then push.

then printf("%d", a[top]);

POP

stack is empty if : top = -1.

if top != -1, then remove the element

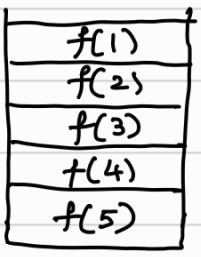
(can use linked list → yet to be taught)

(or) printf("%d", a[top]);

top--;

Recursion → implemented using stack.

find 5!



f(1) = 1.

2 × f(1) = 2 × 1 = 2 = f(2)

3 × f(2) = 3 × 2 = 6 = f(3)

4 × f(3) = 4 × 6 = 24 = f(4)

5 × f(4) = 24 × 5 = 120 = f(5)

∴ f(5) = 120

defining a node :

```
Struct node
{
   int data;
   Struct node * next;
};
Struct node * head = NULL;
```

Create node

```
Struct node * newnode, *temp;
newnode = (struct node*) malloc (sizeof (node));
printf ("enter value");
scanf ("%d", &newnode → data);
newnode → next = NULL;
if (head == NULL)
{
   head = newnode;
   temp = head;
}
else
{
   temp → next = newnode;
   temp = temp → next;
}
```

print data

```
void display()
{
   Struct node *temp = head;
   if (head == NULL) { printf ("empty");}
   else
   {
      while ( temp != NULL)
      {
         printf ("%d\n", temp → data);
         temp = temp → next;
      }
   }
}
```

WACP : to insert a new node at the beginning of a linked list.

```
void begin()
{
   struct node *newnode;
   newnode = (struct node *)malloc(sizeof(node));
   scanf("%d", &newnode → data);
   newnode → next = head;
   head = newnode;
```

```
}
```

WACP: insert at end.

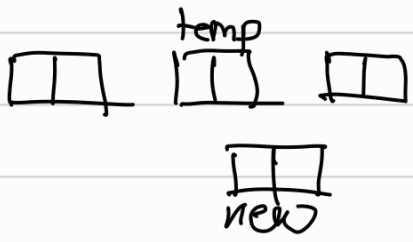
```
Void end ()
 {
  struct node * newnode = (struct node *) malloc
                                  (sizeof(
                                    struct
                                      node));

  scanf("%d", &newnode→data);
  newnode→next = NULL;
  while (temp→next != NULL)
   {
      temp = temp→NULL;
   }
   temp→next = newnode;
 }
```

WACP: to insert node at a random position.

① count nodes:

```
Void add()
{
    int count = 0;
    struct node * temp = head;
     while (temp != NULL)
     {
        count++;
        temp= temp→next;
     }
     if (n > count)
     {
        printf ("N.A");
     }
     else
     {
        struct node * newnode = (struct node*)
                                  malloc(sizeof(struct
                                      node));
        scanf("%d", &newnode → data);
```

```
newnode → next = NULL;
for(int i=1 ; i<n ; i++)
{
   temp = temp→next;
}
newnode → next = temp → next
```



This only works if position is not at the ends.

deletion :

- assume list is created
- Struct node
```
{
   int data;
   Struct node *next;
};
Struct node *head = NULL;
```

ⓐ delete 1st :

```
if [head→next != NULL] ← { Struct node * temp = head;
                           head = temp→next;
                           free(temp);
                           temp = NULL;

if [head→next == NULL] ← { free(head);
                           head = NULL;

if [head = NULL] ← { enjoy
```

ⓑ delete at end:

```
Struct node * temp = head;
Struct node * prevnode = NULL;
while (temp→next != NULL)
{
   prevnode = temp;
   temp = temp→next;
}
free(temp);
temp = NULL;
prevnode→next = NULL;
```

© delete i'th index;

```
Struct node *temp=head, *prevnode = NULL;
int j=1;
while (j<i)
{
    prevnode = temp;
    temp = temp→next;
    j++;
}
prevnode → next = temp→next;
free (temp);
temp = NULL;
```

Assuming i< count

if i=3

1 → p=1 j=1
t=2

2 → p=2 j=2
t=3

We have the node to be deleted.

else → printf ("poda loosu");
return 0;

count → got by:

```
int count =0 ; temp = head
while ( temp!=NULL)
{
    count++;
    temp = temp→next;
}
```

display elements in reverse order:
(assume list is ready)

```
Struct node *temp = head, *prevnode=NULL,
temp2 = head;
while (temp2 !=NULL)
{
    temp2 = temp2→next;
    temp→next = prevnode;
    head = temp;
    prevnode = temp;
    temp = temp2;
}
    head=prevnode;
```

take

1	100	2	200	3	NULL
50		100		200	

head =50.

1st: temp2 = 100;
temp→next = NULL;
prevnode = temp = 50
temp = temp2 =100.

1	NULL	2	200	3	NULL
50		100		200	

2nd: temp2 = 200;
temp->next = 50;
prevnode = 100;
temp = 200;

1	NULL	2	50	3	NULL
	50		100		200

3rd temp2 = NULL;
temp->next = 100;
prevnode = 200;
temp = NULL;

1	NULL	2	50	3	100
	50		100		200

IMPLEMENTATION of STACK using Arrays

PUSH

```
int x;
if (top == size-1)
 {
   printf("stack is full \n");
 }
else
 {
   printf ("enter value: ");
   scanf ("%d", &x);
    top++;
   a[top] = x;
}
```

size → size of Stack.
top → where at stack the last value lies.

if n(elements) = 0, then top = -1 (initially)

POP

top--; if top == -1 → underflow

PEAK

if (top == -1) → empty

```
printf("%d", a[top]);
```

DISPLAY

```
temp = top;
while (temp != -1)
 {
   printf ("%d", a[temp--]);
 }
```

IMPLEMENTATION of STACK using Linked List

```
struct node
 {
    int data;
    struct node * next;
 };
 struct node * top = NULL;
```

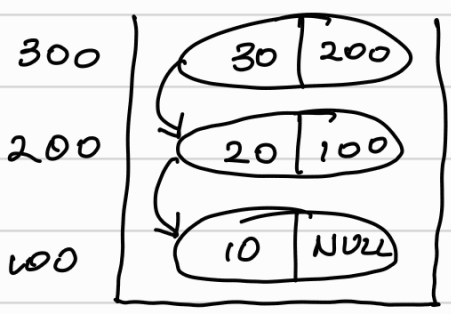
> push (or) pop in the linked list → at first node.
> as push / pop ⟶ O(1)

PUSH :

```
struct node *newnode = (struct node *) malloc
                                (sizeof (struct node));
newnode → next = top;
scanf ("%d", & newnode → data);
top = newnode;
```

POP :

```
if (top == NULL) ⟶ empty.
else
{
   struct node *temp = NULL;
    temp = top;
    top = top → next
    temp → next = NULL;
    free (temp);
    temp = NULL;
}
```



DOUBLY LINKED LIST

* addr. of prevnode, nextnode + the data itself.

```
struct node
 {
    struct node *prev, *next;
    int data;
 }
 struct node * head = NULL, *tail = NULL;
```

create node

```
struct node *newnode = (struct node*) malloc (sizeof(
                                    struct node));
scanf ("%d", &newnode->data);
newnode->next= NULL;
newnode-> prev = NULL;
if(head ==NULL)
 {
    head = tail = newnode;
 }
else
 {
    tail-> next = newnode;
    newnode->prev= tail;
    tail = tail->next;
 }
```

display elements

```
temp = head;
while (temp !=NULL)
{
    printf("%d", temp->data);
    temp = temp->next;
}
```

→ forward dir.

```
temp = head;
    while (temp->next !=NULL)
    {
       temp= temp-> next;
    }
    while( temp !=NULL)
    {
       printf("%d", temp->prev);
       temp = temp->prev;
    }
```

→ backward dir (dumb)

```
temp = tail;
  while (temp !=NULL)
  {
     printf("%d", temp-> data);
     temp = temp -> prev;
  }
```

→ backward dir (genius)

Insert newnode

(a) At beginning :

```
struct node *newnode = (struct node*) malloc (sizeof(
                                    struct node));
scanf ("%d", &newnode->data);
newnode->next = NULL;
newnode->prev = NULL;
if (head == NULL)
 {
    head = tail = newnode;
 }
else
 {
    head->prev = newnode;
    newnode->next = head;
     head = newnode;
 }
```

(b) At end:

```
struct node *newnode = (struct node*) malloc (sizeof(
                                    struct node));
scanf ("%d", &newnode->data);
newnode->next = NULL;
newnode->prev = NULL;
 tail->next = newnode;
 newnode->prev = tail;
  tail = newnode;
```

(c) at i'th index:

```
int i, count = 0;
scanf("%d", &i);
temp = head;
while (temp != NULL)
{
    count++;
    temp = temp->next;
}
```

```
if (i > count)
 {
    printf ("gay");
 }
 else
 {
     temp = head;
    for (int j = 1; j < i ; j++)
     {
        temp = temp->next;
     }
     struct node *newnode = (struct node*) malloc (sizeof(
                                    struct node));
      scanf ("%d", &newnode->data);
      newnode->next = NULL;
      newnode->prev = NULL;
     newnode->prev = temp;
     newnode->next = temp->next;
       temp->next   = newnode;
       temp->next->prev = newnode;
 }
```

delete nodes

(a) at beginning :

```
temp = head;
head = temp->next;
temp->next = NULL;
temp->prev = NULL;
free(temp);
```

(b) At end :

```
temp = tail;
tail = temp->prev;
temp->next = NULL;
temp->prev = NULL;
free(temp);
```

P.T.O

© i'th index :

```
int i, count = 0;
scanf("%d", &i);
temp = head;
while (temp != NULL)
{
    count++;
    temp = temp->next;
}
if (i > count)
{
    printf("gay");
}
else
{
    temp = head ; temp2 = NULL;
    for (int j=1 ; j<i ; j++)
    {
      temp2 = temp;
      temp = temp->next;
    }
    temp2->next = temp->next;
    temp->next->prev = temp2;
    temp->next = NULL;
    temp->prev = NULL;
    free(temp);
}
```

temp2 not required!

temp->next->prev = temp->prev

temp->prev->next = temp->next

we stored temp->prev as temp2

REVERSE doubly linked list

(SOLVE as H.W.) tail = head;

head = tail. prev.

foo easi.

CIRCULAR LINKED LIST

for single L.L.

```
Struct node
{
   int data;
   Struct node *next;
}
Struct node *head, *tail;
   head = tail = NULL;
```

Create node :

```
struct node *newnode = (struct node*) malloc
                        (sizeof(struct node));
scanf("%d", &newnode->data);
newnode->next = NULL;
if (head == NULL)
 {
  head = tail = newnode;
  tail->next = head;
 }
 else
 {
   tail->next = newnode;
    tail = newnode;
   tail->next = head;
 }
```

insert newnode at beginning :

```
newnode = (struct node*)malloc(sizeof(struct node));
newnode->next = head;
head = newnode;
scanf("%d", &newnode->data);
tail->next = head;
```

after i'th index → same as single linked list (the difference is only in the first & last node)

```
temp = head;
do
 {
   i++;
   temp = temp->next;
 } while (temp != head);
```

Delete nodes

(a) Beginning :

```
tail->next = head->next;
head->next = NULL;
free(head)
head = tail->next;
```

① check if head = NULL
② check if head->next = head
③ else

(b) end:

```
temp = head; temp2;
while (temp->next != head)
{
    temp2 = temp;
    temp = temp->next;
}
temp2->next = temp->next;
temp->next = NULL;
free(temp);
temp = NULL;
```

check if
① head = NULL;
② head->next = head;

QUEUE

* using Linked List:

enqueue -

```
int x;
newnode = (struct node *)
          malloc (sizeof(struct node));
scanf("%d", &x)
newnode->data = x;
newnode->next = NULL;
if (front == NULL)
{
 front = rear = newnode;
}
else
{
 rear->next = newnode;
 rear = newnode;
}
```

dequeue -

```
if (front == NULL)
 {
    printf("empty\n");
 }
else if (front->next == NULL)
 {
    front = rear = NULL;
 }
else
 {
      temp = front;
      front = front->next;
      temp->next = NULL;
```

```
        free(temp);
    }
```

display -

```
    temp = front;
    while (temp != NULL)
    {
        printf("%d", temp->data);
        temp = temp->next;
    }
```

peak -

```
    printf("%d", front->data);
```

EFFICIENCY of ALGORITHMS

* determined wrt time (time complexity) and storage (space complexity)
* measured with the help of asymptotic notations.
* Algorithms - diff performance with diff inputs.

POSTERIOR ANALYSIS - evaluates actual performance of the implemented algorithm based on real-world data (or) emperical measurements
- usually not done as its not universal (diff. computers may give diff. measurements)

PRIOR ANALYSIS - Predicts performance of an algorithm before its execution, often using theoretical models & asymptotic analysis
- we use this.

P.T.O.

CLASSIFICATION of ALGORITHMS :

* Iterative : computational procedure that repeats a set of instructions until a condition is met.
* Recursive : calls a function repeatedly.

TIME & SPACE COMPLEXITY

* Time : measure of time of execution as a fn. of sizeof(input)
* Space : space taken by the algorithm wrt its input size. Includes auxiliary space & space used by input.

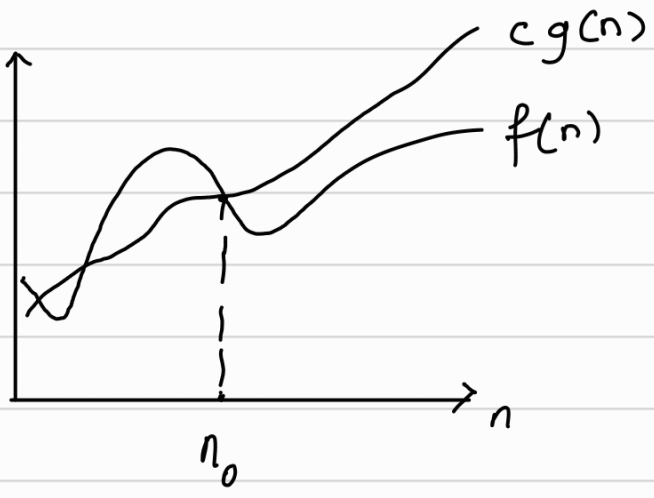
ASYMPTOTIC NOTATION

* mathematical notation - used to describe limiting behaviour of a fn. as the input $\to \infty$.
* Big O, big Omega, big Theta.

$g(n) = O(f(n)) \longrightarrow$ if $g(n) \leq O(f(n))$;

Big O → upper bound / worst case scenario.

$$O(g(n)) = \{f(n) : \exists\, c, n_0 \geq 0,\ n_0 \geq 1 \text{ s.t. } \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$



set of all fns. whose rate of growth is same as or lower than that of g(n).

$f(n) = O(g(n))$.

Big Omega Notation (Ω) :

* $\Omega(g(n))$ represents a lower bound on the growth rate of a function.
* $\Omega(g(n)) = \{f(n) : \exists\, c, n_0 \geq 0,\ n_0 \geq 1,\ 0 \leq cg(n) \leq f(n)\}$

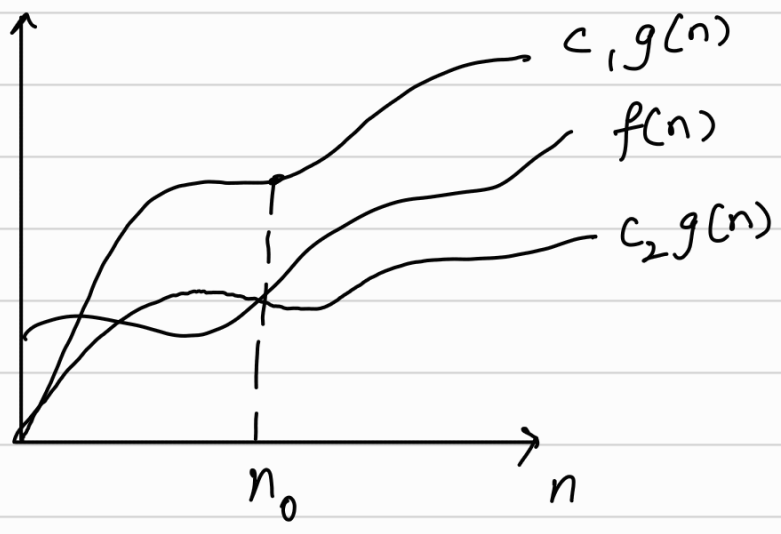
set of all functions whose rate is same as or higher than that of g(n).

then, $f(n) = \Omega(g(n))$

Big Theta (Θ) :

- $\Theta(f(n))$ → upper & lower bounds of the time complexity.
- provides a tight bound.

$$c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall\ n \geq n_0,\ n_0 \geq 1 \quad \exists\ c_1, c_2 > 0$$



$$\Theta(g(n)) = \{f(n) : \exists\ c_1, c_2, n_0 > 0,\ n_0 \geq 1 \text{ s.t. } \forall n \geq n_0,\ 0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n)\}$$

(work on this later :)

```
A(n)
{
  i=1,s=1;
  while(s<=n)
  {
    i++;
    s+=i;
    printf("save me\n");
  }
}
```

$i=1,\ s=1$
① $i=2,\ s=3$ ↳2
② $i=3,\ s=6$ ↳3
③ $i=4,\ s=10$ ↳4
④ $i=5,\ s=15$ ↳5

$S_n = 1, 3, 6, 10, 15$

```
A(n)
{
  i=1;
  for(i; i*i<=n; i++)
  {
    printf("save me\n");
  }
}
```

$i^2 <= n$
$\Rightarrow i <= \sqrt{n}$

So, $O(\sqrt{n})$

```
A(n)
{
  int i,j,K,n;
  for(i=1;i<=n;i++)
  {
    for(j=1; j<=i ;j++)
    {
      for(K=1; K<=100; K++)
      {
        printf("save me\n");
      }
    }
  }
}
```

i	1	2	3	...	n
j	1	2	3	...	n
K	100	2x100	3x100	...	nx100

$100(1+2+\cdots+n)$

$= 100 \times \frac{n(n+1)}{2}$

$= 50(n^2+n)$

$n^2+n \approx n^2$

$O(n^2)$

```
A(n)
{
  int i,j,K,n;
  for(i=1;i<=n;i++)
  {
    for(j=1;j<=i*i;j++)
    {
      for(K=1; K<=n/2; K++)
      {
        printf("saveme\n");
      }
    }
  }
}
```

i	1	2 ...	K ...	n
j	1	4	K^2 ...	n^2
K	n/2	n/2	n/2	n/2

$\frac{n}{2} + 4 \times \frac{n}{2} + 9 \times \frac{n}{2} + \cdots + n^2 \times \frac{n}{2}$

$\frac{n}{2}(1+4+9+\cdots+n^2)$

$\frac{n}{2}\left(\frac{n(n+1)(2n+1)}{6}\right)$

$O(n^4)$

```
A(n)
{
  for(i=1; i<=n; i*=2)
  {
    printf("saveme\n");
  }
}
```

1	2	4	8	16 ...	2^{n-1}

Let loop run x times

$2^{x-1} <= n$

$\Rightarrow x-1 <= \log_2 n$

$\Rightarrow x <= \log_2 n - 1$

$O(\log_2 n)$

```
A(n)
{
  int i,j,K;
  for(i=n/2; i<=n; i++)
   {
     for(j=1; j<=n/2 ;j++)
      {
        for(K=1; K<=n ; K*=2)
         {
           printf("save me \n");
         }
      }
   }
}
```

$i \rightarrow n/2$

$j \rightarrow n/2$

$K \rightarrow \log_2 n$

$n/2 \times n/2 \times \log_2 n$

$O(n^2 \log n)$

```
A(n)
{
  int i,j,K;
  for(i=n/2; i<=n ; i++)
   {
     for(j=1 ;j<=n ; j*=2)
     {
       for (K=1 ; K<=n; K*=2)
       {
         printf("save me \n");
       }
     }
   }
}
```

$n/2 \cdot \log_2 n \cdot \log_2 n$

$O(n \log_2^2 n)$

```
A(n)
{
   for(i=1; i<=n; i++)
   {
     for(j=0 ; j<=n; j=j+i)
      {
        printf("save me \n");
      }
   }
}
```

$i =$	1	2	3	4	$\cdots$	n
$j =$	0	0	0	0		0
	1	2	3	4		K
	2	4	6	8		2K
	3	6	9	12		⋮
	⋮	⋮	⋮	⋮		
	n	n	n	n		n
n	$\frac{n}{2}$	$\frac{n}{3}$	$\frac{n}{4}$	$\cdots$	$\frac{n}{n}$	

$$n\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\cdots+\frac{1}{n}\right)$$

$$\frac{1(x^n-1)}{x-1} = 1+x+x^2+\cdots$$

$$\int \frac{dx}{x-1} = \int (1+x+x^2+\cdots)\,dx$$

$$\Rightarrow \ln(x-1) = 1+\frac{x}{2}+\frac{x^3}{3}+\cdots$$

order $\longrightarrow O(\log n)$.

So, net $\equiv O(n \log n)$

```
A(n)
{
  i=1; s=1;
  while (s<=n)
   {
     i++;
     s+=i;
     printf("save me\n");
   }
}
```

i : 1 2 3 4 5 6 .... K

S : 1 3 6 10 15 21 ....

1 2 3 4 5 6

1 3 6 10 15 21

n'th value of s

$S_n = 1+2+3\cdots+n$

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

$S_K = \frac{K(K+1)}{2}$

$S_K <= n$

$\Rightarrow \frac{K(K+1)}{2} <= n$

$\Rightarrow \frac{K^2}{2} <= n$

$\Rightarrow \boxed{K <= \sqrt{n}}$

∴, complexity $\rightarrow O(\sqrt{n})$

```
A(n)
{
  while (n>1)
   {
     printf("save me\n");
     n=n/2;
   }
}
```

$n = K, K/2, K/4 \cdots K/2^{n-1}$

$\frac{K}{2^{n-1}} < 1$

$\Rightarrow K < 2^{n-1}$

$\Rightarrow \log_2 K < n$

∴, $\boxed{\text{complexity} \rightarrow O(\log_2 n)}$

RECURSIVE ALGORITHMS:

- Back-substitution mechanisms
- recursive trees.
- master's theorem ⟶ advanced master's theorem. (newer one)

```
A(n)
{
  if (n>1)
   {
    return A(n-1);
   }
}
```

(if n>1)

[if n=1 ↓ runs only ONCE.]

$t(n) = 1 + t(n-1)$ - (1)

$t(n-1) = 1 + t(n-2)$ - (2)

$t(n-2) = 1 + t(n-3)$ - (3)

(2) in (1)

$\Rightarrow t(n) = 1 + 1 + t(n-2) = 2 + t(n-2)$ - (4)

(3) in (4)

$\Rightarrow t(n) = 3 + t(n-3)$ - (5)

$\Rightarrow$ for some K,

$\Rightarrow t(n) = K + t(n-K)$ - (6)

We know $t(1) = 1$

$\Rightarrow n - K = 1$

$\Rightarrow \boxed{K = n-1}$

$\therefore t(n) = (n-1) + t(1)$

$= (n-1) + 1$

$= n$

$\therefore \boxed{t(n) = n}$ ⟶ algorithm runs for 'n' times.

$\therefore O(n)$

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

$T(n-1) = (n-1) + T(n-2)$

$T(n-2) = (n-2) + T(n-3)$

⋮

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

⋮

$T(1) = 1 + T(0)$

$T(1) = 1$.

$n - K = 1$

$\Rightarrow \boxed{K = n-1}$

$$(n-0)+(n-1)+(n-2)+\ldots\ldots+(n-(n-2))+(n-(n-1))$$

↓

n terms.

$$n^2-(1+2+\ldots\ldots+n-1)$$

$$= n^2 - \frac{(n-1)(n)}{2}$$

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

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

complexity → $O(n^2)$

SORTING TECHNIQUES

- Arrange elements in specific order.
- → Bubble Sort → Merge sort
  → Selection Sort → Heap sort
  → Insertion Sort → Counting sort
  → Quick Sort → Radix sort

BUBBLE SORT:

Take → 5 9 2 15 6 11 → apply bubble sort to sort by ascending order

① 5<9 → TRUE, no need to swap.
② 9<2 → FALSE → swap (9,2) (5 2 9 15 6 11)
③ 9<15 → TRUE, no swap
④ 15<6 → FALSE → swap (15,6) (5 2 9 6 15 11)
⑤ 15<11 → FALSE → Swap (15,11) (5 2 9 6 11 15)

↓

go back to 1st index

⑥ 5<2 → FALSE → swap (5,2) (2 5 9 6 11 15)
⑦ 5<9 → TRUE
⑧ 9<6 → FALSE → swap (9,6) (2 5 6 9 11 15)
⑨ 9<11 → TRUE
⑩ 11<15 → TRUE

↓

go back to 1st index.

REPEAT for (n-1) times

Same for descending, except: check if current > next is true...

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

i =	0	1	3	...	n-2
j =	n-1	n-2	n-3		n-1-(n-2) = 1

$$\overbrace{n-1}^{} + \overbrace{n-2}^{} + \overbrace{n-3}^{} + \dots n-k$$

$$= nK - \frac{K(K+1)}{2}$$

$$= nK - \frac{(n-1)n}{2}$$

$$= n(n-1) - \frac{n(n-1)}{2}$$

$$= \frac{n(n-1)}{2}$$

SELECTION SORT :

⊙ selects max | min of unsorted part and pushes it to the last/first index. This is done until all the elements are sorted.

Example: 5 9 2 15 6 11

① min = 2, swap 5 and 2
[2] 9 5 15 6 11

② min = 5, swap 9 and 5
[2 5] 9 15 6 11

③ min = 6, swap 9 and 6
[2 5 6] 15 9 11

④ min = 9, swap 15 and 9
[2 5 6 9] 15 11

⑤ min = 11, swap 15 and 11
[2 5 6 9 11 15]

P.T.O

```
void selectionsort(int a[], n)
{
  int i, j, min;
  for(i=0; i<n-1; i++)
  {
    min = i;
    for(j=i+1; j<n; j++)
    {
      if (a[j] < a[min])
      {
        min = j;
      }
    }
    if(i!=min)
    {
      temp = a[i];
      a[i] = a[min];
      a[min] = temp;
    }
  }
}
```

$i = 0 \quad 1 \quad \cdots \quad K \quad \cdots\cdots \quad (n-2)$

$n-1 \quad n-2$

$n-(0+1) \quad n-(0+2) \cdots n-(0+K) \cdots n-(0+n-2)$

$= 2$

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

$= (n-2)n - \frac{(n-2)(n-1)}{2}$

$\downarrow$

$O(n^2)$

INSERTION SORT :

[5] 1 8 2 16

① pick 1: Check $5>1$ (or) $1<5 \rightarrow$ sort.

[1 5] 8 2 16

② pick 8: Check $5<8$ (or) $8<5 \rightarrow$ sort.

[1 5 8] 2 16

③ pick 2: Check $8>2$ (or) $8<2 \rightarrow$ sort.

Check $5>2$ (or) $2>5 \rightarrow$ sort.

Check $1>2$ (or) $2>1 \rightarrow$ sort.

[1 2 5 8] 16 $\longrightarrow$ Sorted...

P.T.O

```
void insertionsort(int a[],n)
{
  int i,j,temp;
  for(i=1;i<n;i++)
  {
    temp=a[i];
    j=i-1;
    while(j>=0 && a[j]>temp)
    {
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=temp;
```

$\omega(n)$ → best case

QUICK SORT

$\omega(n) = 1$
$O(n) = n^2$
avg → $n \log n$

56, 28, 95, 18, 78, 33, 45, 57, 20

↓ after pass 1

33 28 20 18 45 56 78 57 95

↓ perform Quick sort for RHS and LHS

LHS :

33 28 20 18 45

↓ i=0 ↓ j=4

pivot = 33.

a[i] = a[0] = pivot = 33. ⇒ 33 <= pivot i++.

33 28 20 18 45

↑ i=1 ↓ j=4

28 <= pivot i++.

33 28 20 18 45

↑ i=2

20 <= pivot i++.

i=3 j=4

33 28 20 18 45 18 <= pivot i++;

```
void swap (int *a, int *b)
{
    int t=*a;
    *a=*b;
    *b=t;
}
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i= low-1;
    for(int j=low; j<=high-1; j++)
    {
      if (arr[j]<pivot)
      {
         i++;
         swap (&arr[i], &arr[j]);
      }
    }
    swap (&arr[i], &arr[j]);
  }
  swap (&arr[i+1], &arr[high]);
  return (i+1);
}
void quicksort (int arr[], int low, int high)
{
  if (low<high)
  {
    int p= partition (arr, low, high);
    quicksort(arr, low, p-1);
    quicksort (arr, p+1, high);
  }
int main()
{
  int n;
  scanf("%d", &n);
  int a[n];
  for(int i=0; i<n; i++)
  {
     scanf("%d", &a[i]);
  }
 quicksort (arr, 0, n-1);
}
```

33 28 20 18 45

↓

i=4, j=4

45 <= pivot → false.

a[4] = 45 ? pivot → j--

33 28 20 18 45

↓ ↓

j=3 i=4

a[3] = 18 ? pivot → false.

i < j → false. so, swap pivot and a[j]

| 18 28 20 33 45 | → pass 1 completed for LHS.

RHS :

78 57 95

↓ ↓

i=0 j=2

pivot = 78.

78 <= pivot → true. i=1

78 57 95

i=1 j=2

57 <= pivot → true i=2

78 57 95

i=2, j=2

95 <= pivot → false.

95 > pivot → true j=1

78 57 95

j=1 i=2

57 > pivot → false.

i < j → false.

So, swap pivot and 57.

| 57 78 95 | → pass 1 completed for RHS...

like this we do until its sorted - code -> write later.

P.T.O.

MERGE SORT

learn on your own ....

HEAP SORT

binary tree → each node : 2 children at max (0,1,2)

arrange in ↑↑ order → maxHeap.
arrange in ↓↓ order → minHeap.

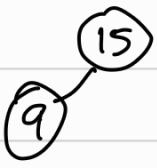
maxHeap :

9 15 16 1 5 30

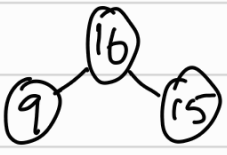
parent >= children

(9) → parent.

Check: 15>9 → swap; (9) → child, (15) → parent



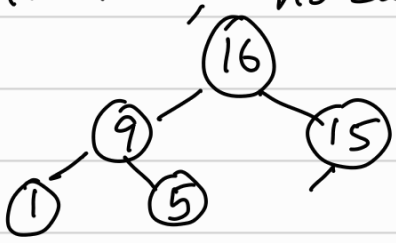
Check: 16>15 → swap ; (16) → parent, (15) → child



(9) → parent.

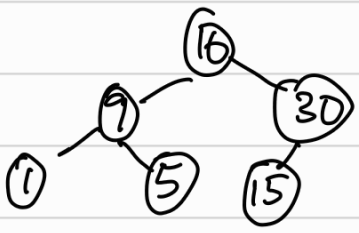
Check: 971 , no swap

Check: 975 , no swap.

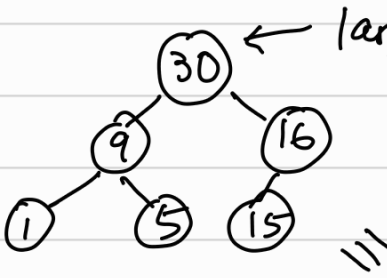


(15) → parent.

Check: 30>15 → swap.



30>16 : swap.



← largest.

30 9 16 1 5 15

↓

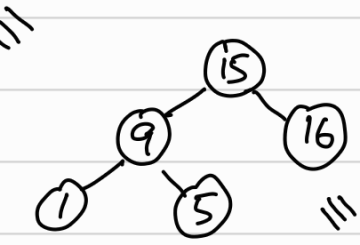
15 9 16 1 5 (30)

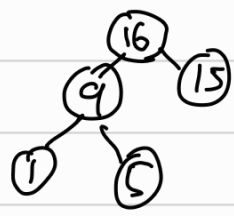
↓

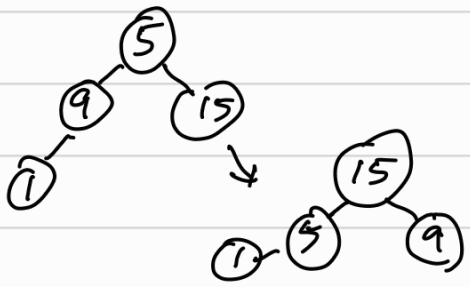
16 9 15 1 5

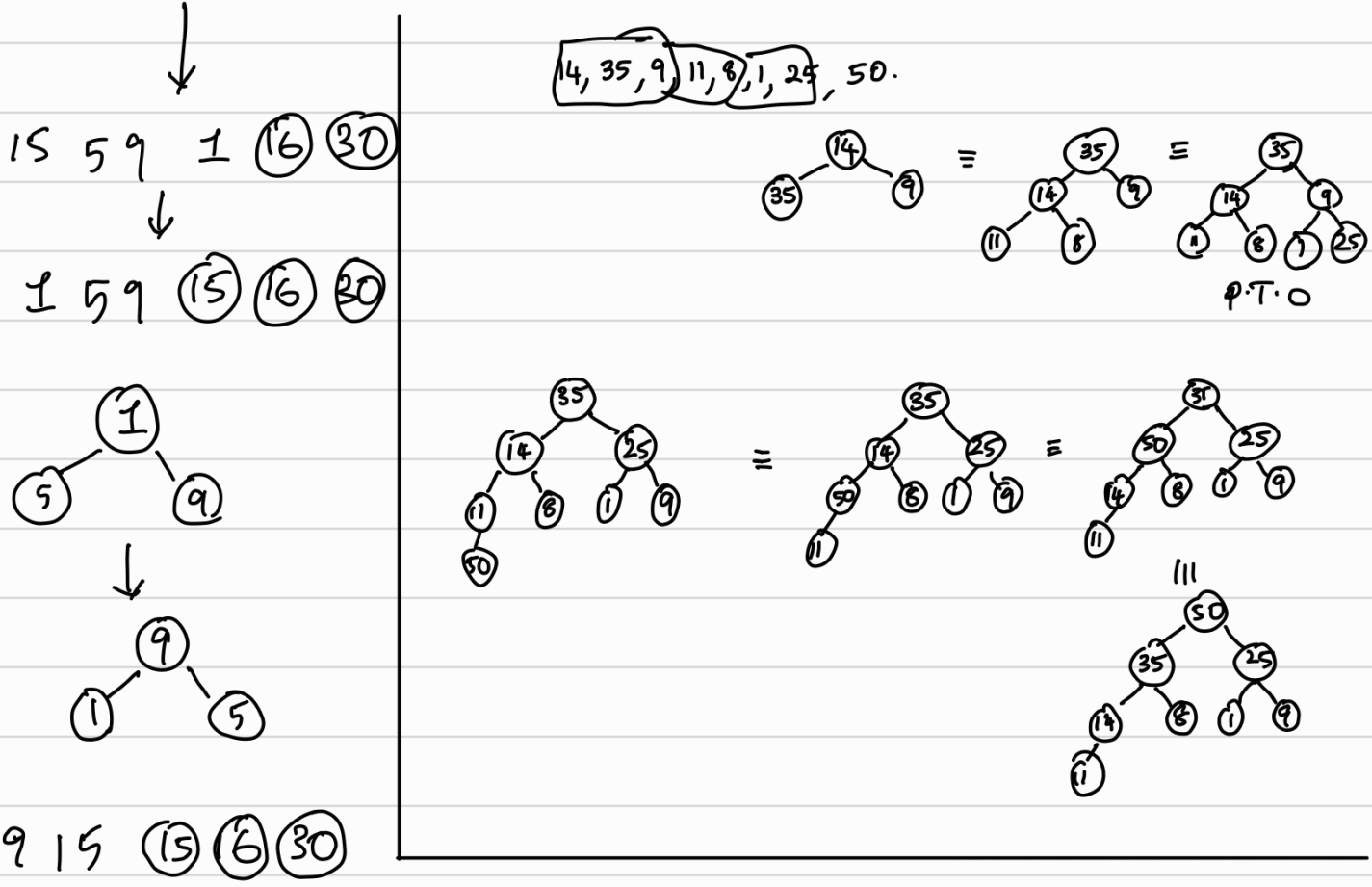
↓

5 9 15 1 (16) (30) →









do maxHeap for: 16 1 5 8 13 12 → done.

for quickSort →

$T(1) = 1$

$T(n) = 2 \times T(n/2) + n$

$T(n/2) = 2 \times T(n/4) + n/2$

$T(n/4) = 2 \times T(n/8) + n/4$

$\therefore, \quad T(n) = 4 \times T(n/4) + 2n$

$= 4 \times \left(2 \times T(n/8) + n/4\right) + 2n$

$= 8 \times T(n/8) + 3n$

$\vdots$

$= 2^K \times T(n/2^K) + nK$

$= 2^K \times T(1) + n \log n$

$= n + n \log n$

$\Omega(n \log n)$

$\frac{n}{2^K} \geq 1$

$n \geq 2^K$

$\Rightarrow \log_2 n \geq K$

$O(n^2)$ → if we need ascending, but given descending.

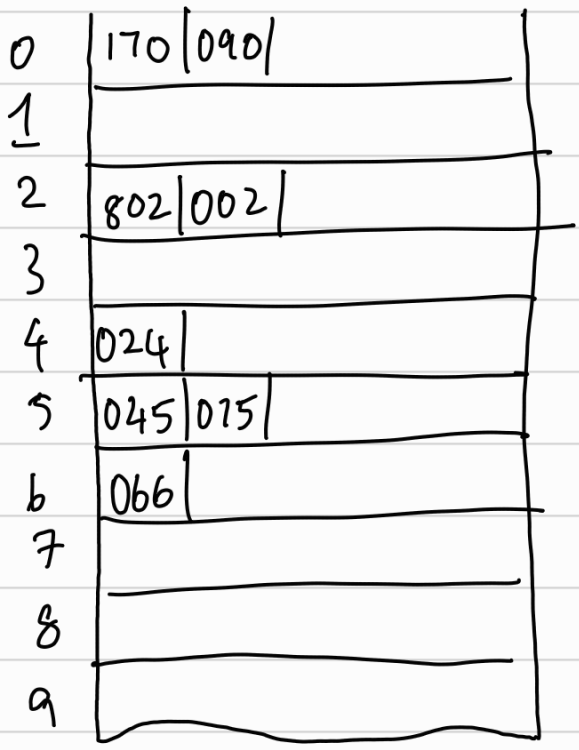
RADIX SORT

- no comparisons
- distribute elements into buckets based on individual digits
- processes digits - from LSD to MSD (or) vice versa.

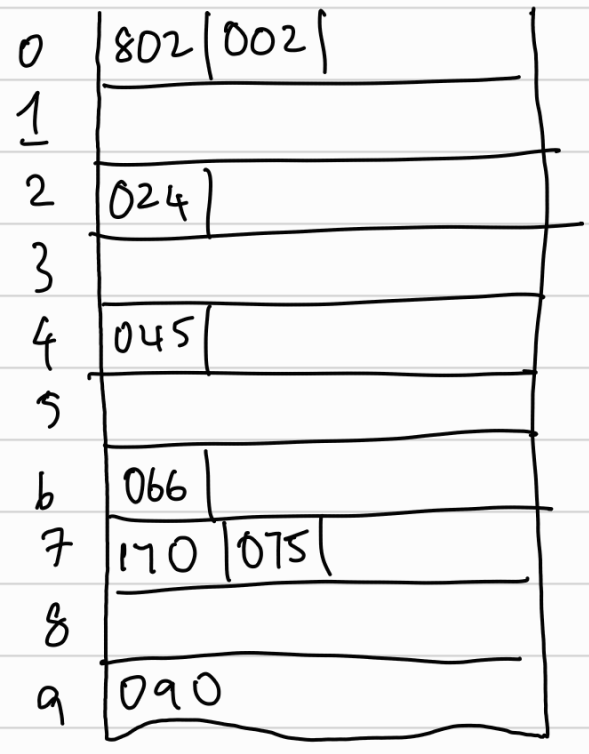
170, 45, 75, 90, 802, 24, 2, 66

↓

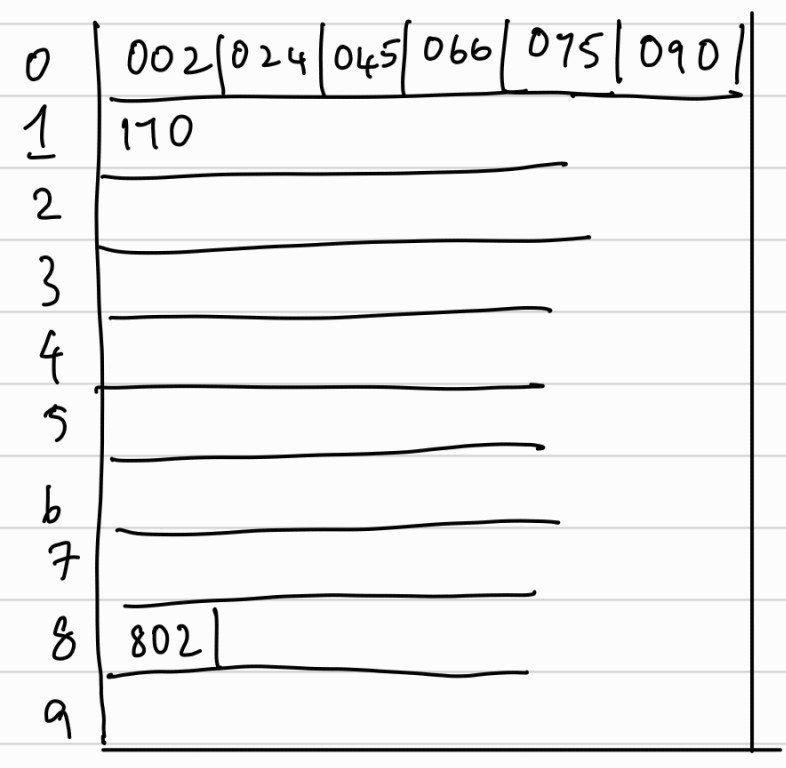
170, 045, 075, 090, 802, 024, 002, 066



170, 090, 802, 002, 024, 045, 075, 066



802, 002, 024, 045, 066, 170, 075, 090



002, 024, 045, 066, 075, 090, 170, 802

COUNTING SORT

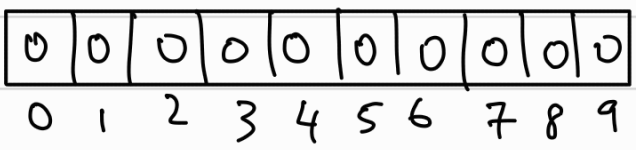
- non comparitive - counts the n(occurrences) of each element in input.
  - works well if Δ of range b/w elements is not that great.

2 9 7 4 1 8 4

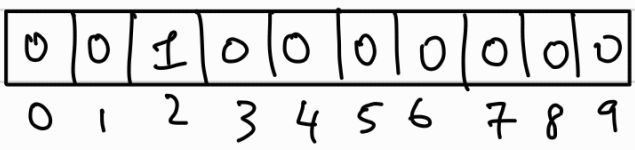
↓

largest element.

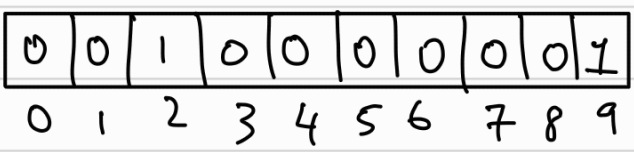
Create arr of (9+1) = 10.



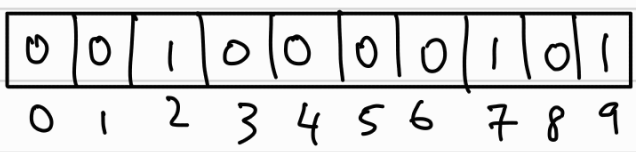
① 2 → go to 1st ↑↑ value by 1:



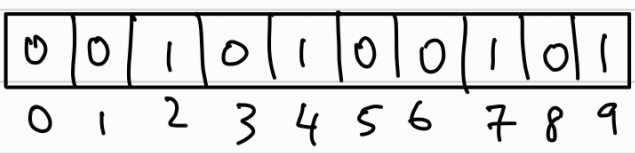
② 9 →



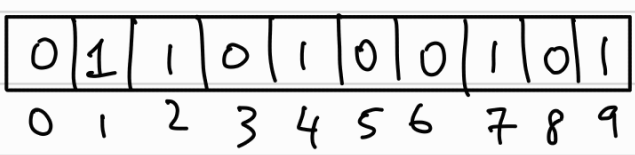
③ 7 →



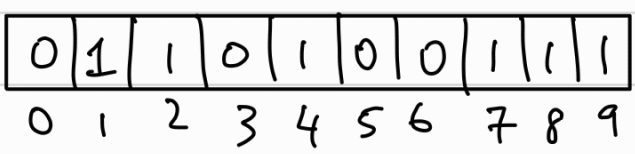
④ 4 →



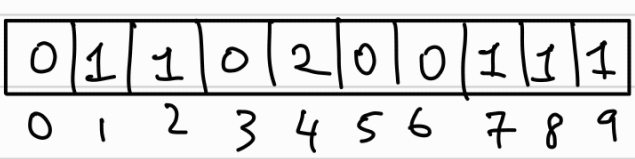
⑤ 1 →



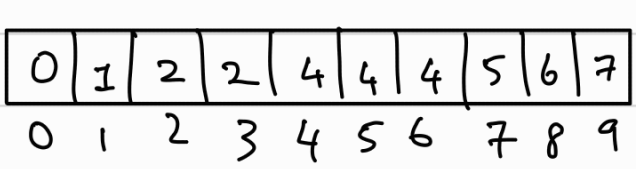
⑥ 8 →



⑦ 4 →



add cumulatively →



Start from last 2 9 7 4 1 8 (4)

(4) Check value at 4th index → 4
subt. 1 → 4 − 1 = 3.

add 4 at 3rd index.

_ _ _ 4 _ _ _

(8) → at 8th → 6 → 6 − 1 = 5
add 8 at 5th index

_ _ _ 4 _ 8 _

(1) → at 1 → 2 → 2 − 1 = 1
add 2 at 1.

_ 2 _ 4 _ 8 _

(4) → at 4 → 4 → 4 − 1 = 3
add 4 at 3

_ 2 4 4 _ 8 _

(7) → at 7, 5 → 5 − 1 = 4.
add 7 at 4

_ 2 4 4 7 8 _

(9) → at (9), 7 → 7 − 1 = 6
add 9 at 6

_ 2 4 4 7 8 9

(1) → at 1, 1 → 1 − 1 = 0
add 1 at 1

1 2 4 4 7 8 9 //

P.T.O

INPLACE SORTING ALGORITHMS

- doesn't require extra space.
- sorts within its own memory.
- Bubble, Insertion, Selection, Heap, Quick → inplace

STABLE SORTING ALGORITHMS

- relative order of equal elements in the sorted output follow their relative order in the unsorted input.
  → merge, insertion, bubble, selection, counting, radix.

TIME COMPLEXITY of Algorithms

- Bubble → best: $O(n)$
  worst: $O(n^2)$
  avg: $O(n^2)$
- Selection → best: $O(n^2)$
  worst: $O(n^2)$
  avg: $O(n^2)$
- Insertion → best: $O(n)$
  worst: $O(n^2)$
  avg: $O(n^2)$
- Merge & Heap → best: $O(n \log n)$
  worst: $O(n \log n)$
  avg: $O(n \log n)$
- quick → best: $O(n \log n)$
  worst: $O(n^2)$
  avg: $O(n \log n)$
- counting → best: $O(n+K)$; $K \rightarrow$ range of input.
  worst: $O(n+K)$ of non-negative
  avg: $O(n+K)$ key values.
- Radix → best: $O(n \cdot K)$
  worst: $O(n \cdot K)$; $K \rightarrow$ n(digits) in max no.
  avg: $O(n \cdot K)$

P.T.O

MASTER'S THEOREM

$$T(n) = a\,T(n/b) + \theta(n^K (\log n)^P)$$

$a >= 1$, $b > 1$, $K >= 0$, p is Real Number.

1. if $a > b^K$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^K$,
   - (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} (\log n)^{p+1})$
   - (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log(\log n))$
   - (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^K$,
   - (a) if $p >= 0$, then $T(n) = \theta(n^K (\log n)^P)$.
   - (b) if $p < 0$, then $T(n) = \theta(n^K)$.

$T(n) = 2T(n/2) + n$; $a = 2$, $b = 2$, $K = 1$, $p = 0$

$b^K = 2^1 = 2$.

$\Rightarrow \boxed{a = b^K}$

$p = 0 \Rightarrow p > -1 \Rightarrow T(n) = \theta(n^{\log_b a} (\log n)^{p+1})$

$= \theta(n^{\log_2 2} (\log n)^1)$

$= \theta(n \log n)$

$\Rightarrow \boxed{T(n) = \theta(n \log n)}$

$T(n) = 2T(n/2) + n\log n$

(1) (1)

$aT(n/b) + \theta(n^K (\log n)^P)$

$a = 2$, $b = 2$, $K = 1$, $p = 1$.

$a = b^K$ and $p > -1$.

$\Rightarrow T(n) = \theta(n^{\log_b a} (\log n)^{p+1})$

$= \theta(n^{\log_2 2} (\log n)^2)$

$= \theta(n (\log n)^2)$

$\therefore \boxed{T(n) = \theta(n (\log n)^2)}$

$T(n) = 2T(n/2) + \theta(n/\log n)$

$a = 2$, $b = 2$, $K = 1$, $p = -1$.

$a = b^k$ and $p = -1$.

$$\theta(n^{\log_b a} \cdot \log(\log n))$$
$$= \theta(n^{\log_2 2} \cdot \log(\log n))$$
$$= \theta(n \log(\log n))$$

$\therefore$ $\boxed{T(n) = \theta(n \log(\log n))}$

$T(n) = 3T(n/2) + n^2$

$a = 3,\ b = 2,\ k = 2,\ p = 0$

$a < b^k$ $(3 < 4)$ and $p >= 0$

$\Rightarrow T(n) = \theta(n^k (\log n)^p)$

$= \theta(n^2)$

$\Rightarrow \boxed{T(n) = \theta(n^2)}$

$T(n) = 2T(n/2) + n^{0.51}$

$a = 2,\ b = 2,\ k = 0.51$ $p = 0$

$a > b^k$ and $p = 0$

$\Rightarrow T(n) = \theta(n^{\log_b a})$

$= \theta(n^{\log_2 2})$

$= \theta(n)$

$\Rightarrow \boxed{T(n) = \theta(n)}$

$T(n) = 0.5T(n/2) + 1/n$; $\boxed{\text{let } T(1) = 1}$

$a = 0.5,\ b = 2,\ k = -1,\ p = 0 \rightarrow$ NO MASTER'S THEOREM ($a >= 1$, here its 0.5)

$T(n) = 3T(n/2) + (\log n)^2$

$a = 3,\ b = 2,\ k = 0,\ p = 2$

$a > b^k$ $(3 > 2^0)$

$\theta(n^{\log_b a}) = \theta(n^{\log_2 3})$

P.T.O.

$T(n) = 2T(n/2) + n(\log n)^2$

$a=2,\ b=2,\ K=1,\ p=2$

$a = b^K$, $p > -1$: $T(n) = \theta(n^{\log_b a} \cdot (\log n)^{2+1})$

$= \theta(n^{\log_2 2} \cdot (\log n)^3)$

$= \theta(n(\log n)^3)$

$T(n) = 16T(n/4) + n$

$a=16,\ b=4,\ K=1,\ p=0$

$16 > 4 \quad (a > b^K) \Rightarrow T(n) = \theta(n^{\log_b a})$

$= \theta(n^{\log_4 16})$

$= \theta(n^2)$

$T(n) = 2^n T(n/2) + n^n \rightarrow$ CAN'T APPLY MASTER'S THEOREM!
(use back substitution)

LITTLE OH :

if $f(n) <= cg(n)$
then $f(n) = O(g(n))$

$f(n) < c \cdot g(n) \quad ; c > 0$

$f(n) = n$ and $g(n) = n^2$

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n)) \rightarrow$ Little Oh.

$\lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} \frac{1}{n} = 0$ $\therefore$ $\boxed{f(n) = o(g(n))}$

find 'c'. s.t $f(n) < c \cdot g(n)$ ALWAYS!

$f(n) = \log n, \quad g(n) = n.$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{n} = 0$

$\boxed{f(n) = o(g(n))}$

$\boxed{\log n = o(n)}$

LITTLE OMEGA :

↳ Lower bound, not tight

↳ if $f(n) = \omega(g(n))$, then: $\boxed{|f(n)| > c|g(n)|}$

↳ $f(n)$ grows much faster than $g(n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \boxed{f(n) \to \omega(g(n))}$$

$f(n) = n$; $g(n) = \log n$

$$\frac{f(n)}{g(n)} = \frac{n}{\log n} \equiv \frac{1}{1/n} = n.$$

$n \to \infty \Rightarrow \boxed{\frac{f(n)}{g(n)} = \infty}$

$\therefore$, $f(n) = \omega(g(n))$

$f(n) = 3^n$, $g(n) = 2^n$

$$\frac{f(n)}{g(n)} = \frac{3^n}{2^n} = \left(\frac{3}{2}\right)^n \quad \text{as } n \to \infty, \boxed{\left(\frac{3}{2}\right)^n = \infty}$$

$\therefore$, $\boxed{f(n) = \omega(g(n))}$

SPACE COMPLEXITY :

↳ measure of amount of storage space an algorithm / program uses to solve a problem as a fn. of size of the input.

↳ expressed in "Big Oh" notation.

printf ("%d", a+b); (or) s = a+b → printf ("%d", s);

$SP = \underbrace{C}_{\text{independent}} + \underbrace{S_p}_{\text{dependent}}$

$O(2) \equiv O(1)$

```
int n, a[n];
sum=0
for(int i=0; i<n; i++)
{
  sum+=a[i];
}
```

$SP = C + S_p$

$SP = 3 + n \rightarrow$ {n integers}

$= O(n+3)$

$= O(n)$ //

another defn → SP ≡ extra memory storage used.

for Sorting:

- bubble : O(1)
- selection : O(1)
- Insertion : O(1)
- merge : O(n)
- quick : O(n)
- heap : O(1)
- counting : O(K)
- radix : O(n+K)

SEARCHING ALGORITHMS :-

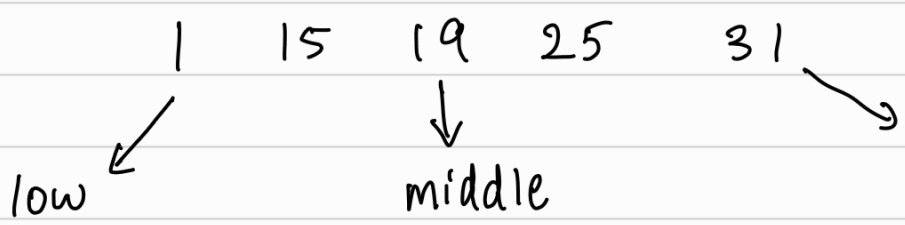
Linear Search :

→ sequential search. checks until match is found.

```
int linearSearch (int a[], int target, int n)
 {
    for (int i=0; i<n; i++)
      {
         if (a[i] == target)
           {
             return i; // found, return index
           }
      }
     return -1;    // not found error 404 :<
 }
```

Binary Search

↳ for sorted arrays
↳ divides search in half : narrows down possible locations for the target value.



low = 0 ; high = 4 middle = $\left(\frac{low + high}{2}\right) = \frac{0+4}{2} = 2$

Compare a[m] == target (19 ≠ 1) → false.

since false, we narrow search to LHS of m as target < middle

low = 0 , high = middle - 1 = 2 - 1 = 1

middle = $\frac{(0+1)}{2} = 1/2 = 0$

compare a[middle] == target → TRUE!

so, found. THIS IS BINARY SEARCH.

(if a[middle] < target, m = m + 1 and high doesn't change).

```
int binarySearch (int a[], int low, int high, int target)
{
    while (low<=high)
    {
      int mid = (low+high)/2 ;
      if (a[mid]==target)
      {
        return mid;
      }
      else if (arr[mid] < target )
       {
          low = mid+1;
       }
      else
       {
          high= mid-1;
       }
    }
   return -1;
}
```

for binary search: $T(n) = T(n/2) + 1$

(1)

$T(n) = aT(n/b) + \theta(n^k (\log n)^p)$

$a = 1, \ k = 0, \ p = 0$

$a = b^k$

$p > -1$ $\quad T(n) = n^{\log_b a} (\log n)^{p+1}$

$T(n) = \log n$