19/9

Module: IV

Sorting & ~~Selection~~ Searching

- Sorting
  - → Selection
  - → Insertion
  - → Quick
  - → Heap
  - → Merge
  - → Radix
- Searching
  - → Linear ✓
  - → Binary ✓
  - → Hashing

Bubble Sort [complexity : $O(n^2)$]

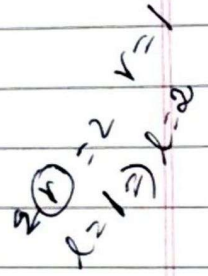
```
for(i=0; i<n-1; i++) { // for passes
    for(j=0; j<n-i-1; j++){
        if (a[j] > a[j+1] ){
            temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp
}}}
```

Selection Sort [complexity: $O(n^2)$]

eg:

	elements	0	1	2	3	4	5	6	7	8
		→ passes								
0	10	(10)	-1	-1	-1	-1	-1	-1	-1	-1
1	6	6	(6)	2	2	2	2	2	2	2
2	12	12	12	(12)	6	6	6	6	6	6
3	8	8	8	8	[8]	8	8	8	8	8
4	42	42	42	42	42	(42)	10	10	10	10
5	-1	[-1]	10	10	10	10	(42)	11	11	11
6	22	22	22	22	22	22	22	(22)	12	12
7	18	18	18	18	18	18	18	18	[18]	18
8	2	2	[2]	[6]	12	12	12	[12]	22	(22)
9	11	11	11	11	11	11	[11]	42	42	42

```
selection_sort (int a[], int n){
 // sort elements in array 'a' in ascending order
 for (i=0; i<n ; i++)
     min = a[i];
     min_index = i;
     for (j= i+1; j≤ n-1 ; j++)
          if  a[j] < min
               min = a[j]
               min_index = j
     }}// end of for
     swap ( a[i] and, a[min_index]);
}
```



Insertion Sort O(n²)

	0	1	2	3	4	5	6	7	8	9
1	90	(6)	12	10	22	89	46	32	11	-1
2	6	90	(12)							
3	6	12	90	(10)						
4	6	10	12	90	(22)					
5	6	10	12	22	90	(89)				
6	6	10	12	22	89	90	(46)			
7	6	10	12	22	46	89	90	(32)		
8	6	10	12	22	32	46	89	90	(11)	
9	6	10	11	12	22	32	46	89	90	(-1)

```
insertion_sort (int a[], int n){
 for i=1 to n   do{  // no. of passes
      temp= a[i]
      j = i-1
      while (j≥0 && a[j] >temp){
```

```
        a[j+1] = a[j]
        j--
    } // end while
    a[j+1] = temp
```

29/9

Quick Sort O(nlogn) / Partition Exchange Sort

- idea of divide & conquer
- by a series of exchanges, it does sorting hence known as Partition Exchange sort
- it was the 1st sorting algorithm that have order nlogn : Quick sort
  Invented by Hoare in 1962

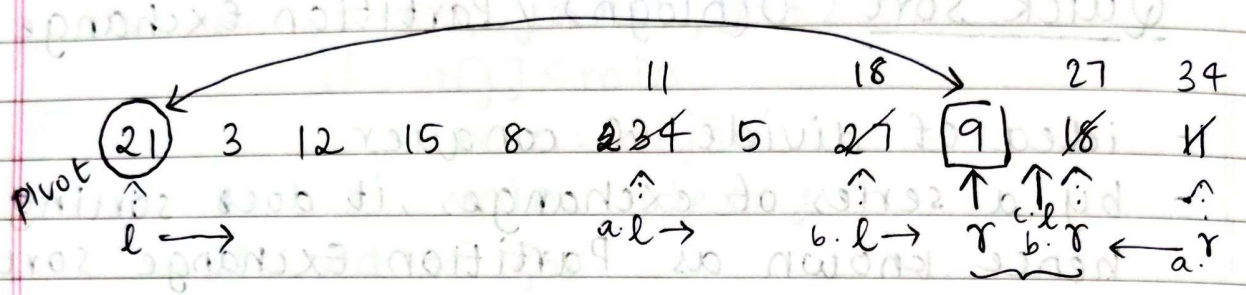
```
quick_sort (int a[], int start, int end){
  if (start < end){
     pivot = a[start]
     l = start, r = end
     while (l<r){
        while (a[l] ≤ pivot) & (l ≤ end)
              l++
        while (a[r] > pivot) & (r ≥ start)
              r--
        if l<r
              swap (a[l] & a[r])
     } end while
     swap (a[r], a[start])
     Quicksort (a, start, r-1)
     Quicksort (a, r+1, end)
  } // end of if
  return ()
} // end of quicksort.
```

Eg:

0	1	2	3	4	5	6	7	8	9	10
21	3	12	15	8	34	5	27	9	18	11

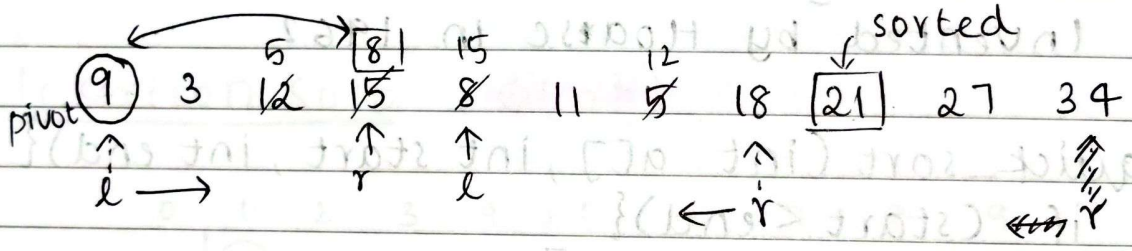
pivot

Iteration : 1

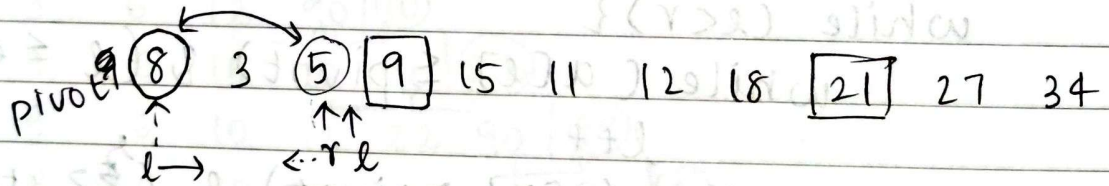


r < l no swap

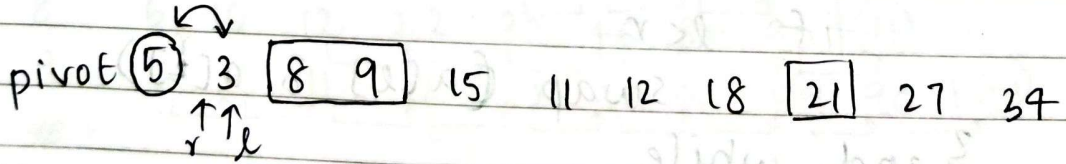
Iteration : 2



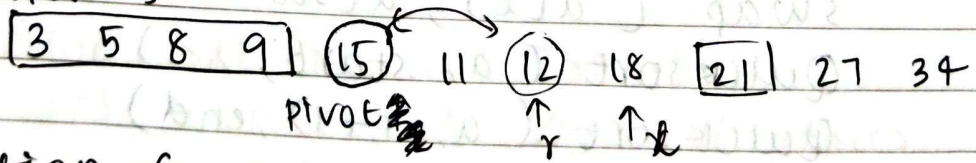
Iteration : 3



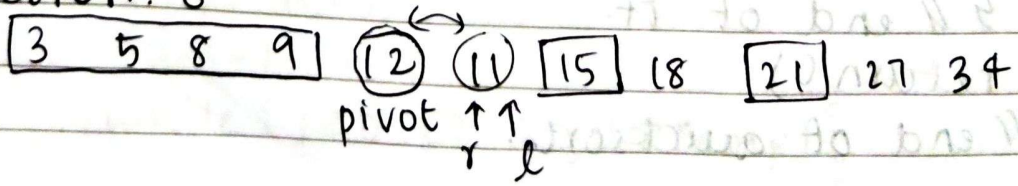
Iteration : 4



Iteration : 5



Iteration : 6



Iteration 7

[3 5 8 9] 11 [12 15] 18 [21] 27 34

[11 12] [18 21] (27) 34

r ↑ pivot ↑ l

hence,

3 5 8 9 11 12 15 18 21 27 34

Q-1 3 1 4 1 5 9 2 6 5 3

→ pivot (3) 1 4 (1) 5 9 2 6 5 3

r ↑ l ↑

pivot (1) (1) 2 [3] 5 9 4 6 5 3

r ↑ l ↑

[1 1 2 3] (5) 9 4 6 5 9

pivot r ↑ l ↑

[1 1 2 3] (4) 3 [5] 6 5 9

pivot r ↑ l ↑

[1 1 2 3 3 4 5] (6) 5 9

pivot r ↑ l ↑

[1 1 2 3 3 4 5] 5 [6] 9

sorted sorted

Hence, 1 1 2 3 3 4 5 5 6 9

Q-2. 48 29 8 59 72 88 42 65 95 19 82

→ (48) 29 8 59 72 88 42 65 95 19 82

pivot r ↑ l ↑

(42) 29 8 (19) [48] 88 72 65 95 59 82

pivot r ↑ l ↑

(19) 29 8 [42 48] 88 72 65 95 59 82

pivot r ↑ l ↑

8 | 19 | 29 | 42 48 | 88 72 65 95 59 82

pivot

sorted sorted

8 19 29 42 48 | 88 | 72 65 95 59 82

pivot

8 19 29 42 48 | 59 | 72 65 82 | 88 | 95

pivot r l

ie, 59 sorted

8 19 29 42 48 59 | 72 | 65 82 | 88 | 95

pivot r l

8 19 29 42 48 59 | 65 | 72 | 82 | 88 | 95

sorted sorted sorted

Hence,

8 19 29 42 48 59 65 72 82 88 95.

Q.3 12 65 34 9 56 43 10

→ 12 65 34 9 56 43 10

pivot r l

9 10 | 12 | 34 56 43 65

pivot r l

9 10 12 | 34 | 56 43 65

pivot r l

ie, 34 is sorted (at crrt position)

9 10 12 34 | 56 | 43 65

pivot r l

9 10 12 34 | 43 | 56 | 65

sorted sorted

hence

9 10 12 34 43 56 65

25/9

Complexity of Quick sort

$T(n) = n + 2T(n/2)$ —— ①

$T(n/2) = n/2 + 2T(n/4)$ —— ②

sub ② in ① ⇒

$T(n) = n + 2\left[\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right]$

$= n + n + 4T\left[\frac{n}{4}\right] = 2n + 4T\left[\frac{n}{4}\right]$

generalise,

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

recursion stops when $T(1)$

$2^k = n$

$k = \log_k(n)$

⇒ $T(n) = (\log n)n + n(T(1))$

$= n + n\log n$

$T(n) = O(n \log n)$

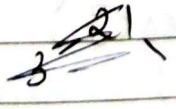
Heap Sort : $O(n \log n)$

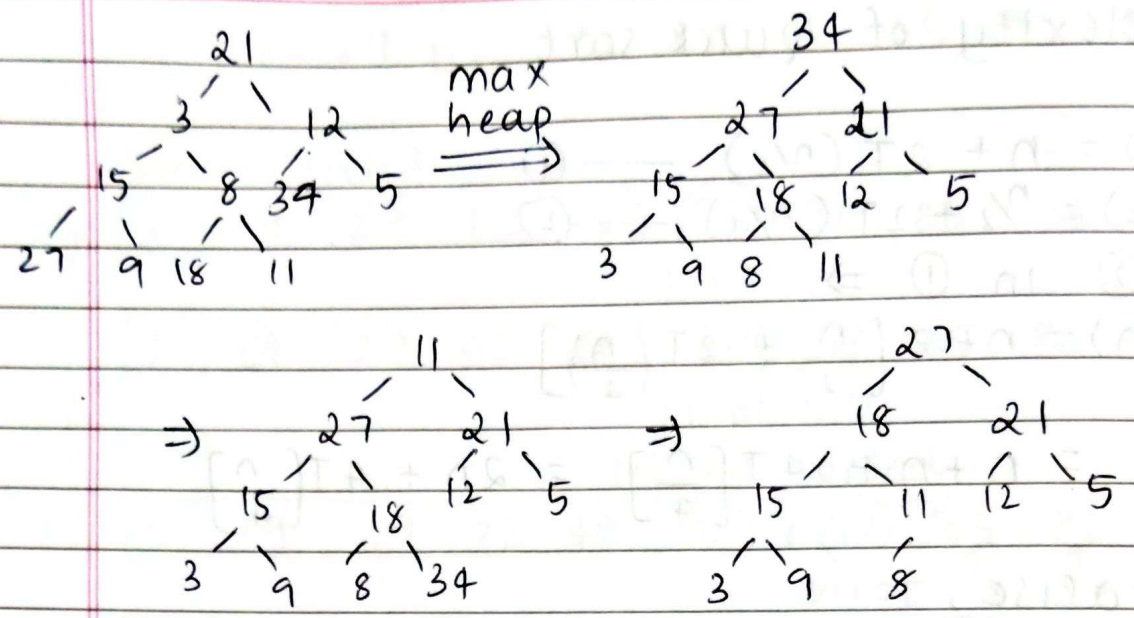
heap

Max (asc) — Min

(Every parent greater than children)

Eg: 21 3 12 15 8 34 5 27 9 18 11





⇒

Algorithm for heap sort : O(nlogn)

```
void heapSort(int a[], int n){
  for(int i= n/2 -1 ; i>=0 ; i--)
        heapify (a, i, n)
  for (int i = n-1 ; i>=1 ; i--){
        swap(a[0], a[i])
        heapify (a, 0, i-1)
}
heapify (a, i, n){
   largest = i
   l = 2i+1 ;  r = 2i+2 ;
   if (l<=n-1) && (a[l]>a[largest])
        largest = l
   if (r<=n-1) && (a[r]> a[largest])
        largest = r
```

```
if (largest != i)
    swap(a[i], a[largest])
    heapify(a, largest, n)
}
```

26/9

Merge Sort : O(nlogn)

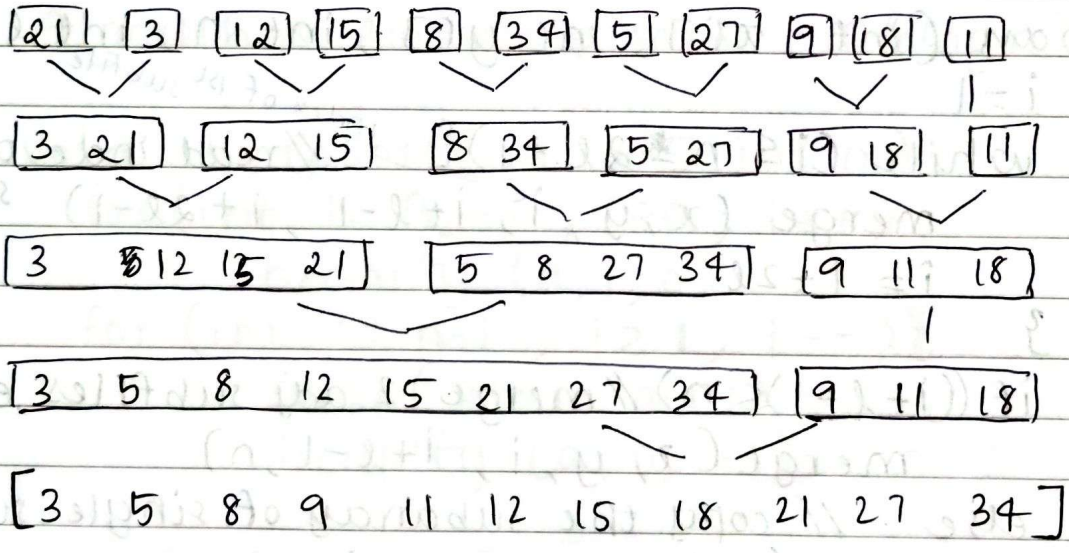
```
Algorithm [array index from 1 ... n]
mergesort (int a[], int n){
    l = 1 // l: size of subarray being merged.
    while (l<n){
        mpass(x, y, n, l)        // i/p o/p size size of subarray
        l = 2l
        mpass (y, x, n, l)
        l = 2l
}}

mpass (int x[], int y[], int n, int l){
    i = 1
    while (i <= n - 2l + 1){     // index of 1st subfile
        merge (x, y, i, i+l-1, i+2l-1)   // read inde of 1st subfile
        i = i + 2l
    }
    if ((i+l-1) < n) // merge 2 adj subfiles, equal siz
        merge(x, y, i, i+l-1, n)
    else  // copy the subarray of single subfile
        for (t=i; t<=n; t++)
            y[t] = x[t]
}

merge(int x[], int y[], int l, m, u){
    i=0; j=m+1, k=l
```

```
while (i≤m) && (j≤u)
    if  x[i] < x[j]
        y[k] = x[i]
        i++
    else
        y[k] = x[j]
        j++
    k++
if (i>m)
  for(t=j ; t≤n; t++)
      g[k] = x[t]
      k++
else
  for (t=i ; t≤m; t++)
      j[k] = x[t]
      k++
```

Eg:



Here, index: i, i+l-1, i+l, i+2l-1 (merging 2 subarrays of same size)

8/10

Hashing : Searching Techniaue.

- Search time is constant , O(1).

Eg: If array: 18 16 77 89 24 90 15

It is stored in hash table using hashing funcn

Hash function

$h(k) = k \bmod 10$

$h(18) = 18 \% 10 = 8$

$h(16) = 16 \% 10 = 6$

$h(77) = 77 \% 10 = 7$

90	0
	1
	2
	3
24	4
15	5
16	6
77	7
18	8
89	9

→ Hash key : index into the hash table

→ Hash table : array

→ Bucket : single location in the array

Properties of a good Hash Function

1. Function must be easy enough to
2. Minimise collision as far as possible
3. Distribute keys as uniformly as possible.

Resolve Collision

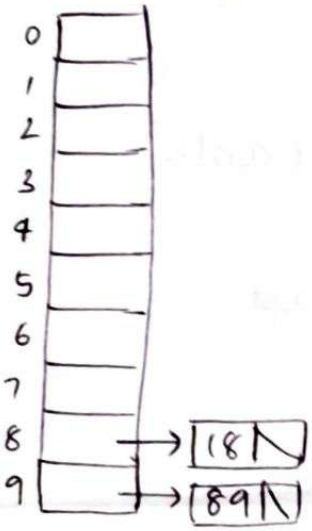
Open Hashing (chaining)

Closed hashing
↳ linear probing
↳ quadratic ..
↳ double hatching

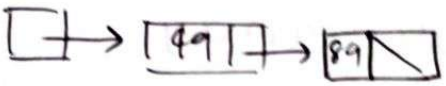
Open Hashing : LL (any no. of elements can be stored array size is not relevant)

⇒ chaining

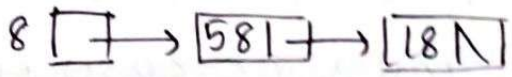
1. 89 18 49 58 9 (Bucket stores address)



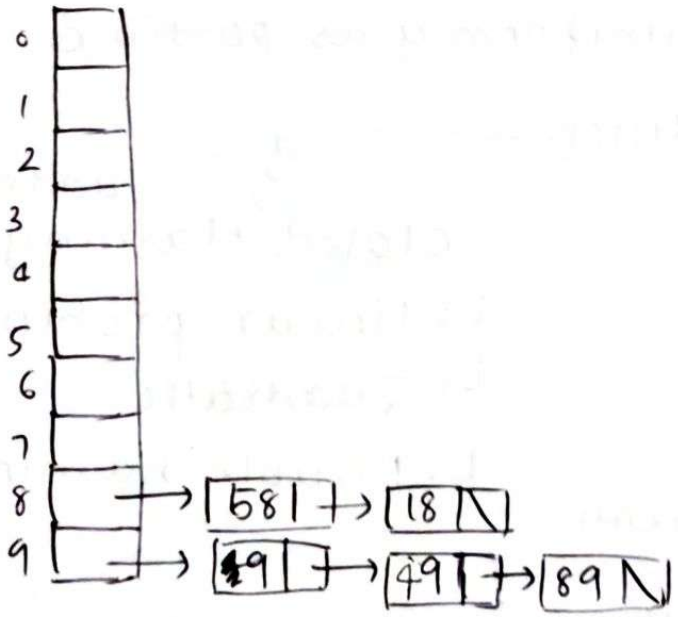
· when 49 comes
Insert at front



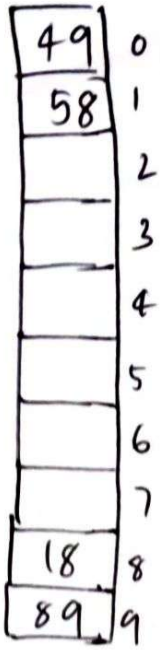
· when 58 comes



ie,



Linear Probing



$h(89) = 89 \% 10 = 9$

$h(18) = 18 \% 10 = 8$

$h(49) = 49 \% 10 = 9$ (collision) ✗

∴ next empty location

$(9+1) \% 10 = 0$

$h(58) = 58 \% 10$ ✗

Probe seq

$(8+1) \% 10 = 9$ ✗

$(8+2) \% 10 = 0$ ✗

$(8+3) \% 10 = 1$ ✓

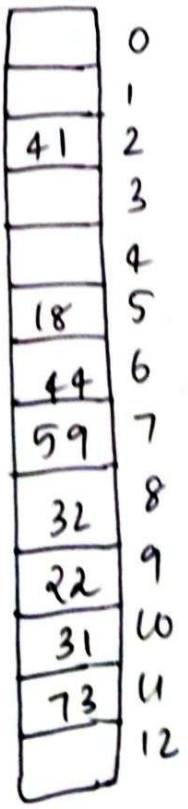
Probe sequence

~~h(k)~~ $(h(k)+1) \bmod 10$ if $h(k) \cdot k \bmod 10$

$i = 1, 2, 3 \ldots \Rightarrow$ offset of probe

(search next empty)

2) 18 41 22 44 59 32 31 73 mod 13



Hash function: $h(k) = k \bmod 13$

1. $h(18) = 18 \% 13 = 5$
2. $h(41) = 41 \% 13 = 2$
3. $h(22) = 22 \% 13 = 9$
4. $h(44) = 44 \% 13 = 5$ } $\Rightarrow (5+1) \% 13 = 6$
5. $h(59) = 59 \% 13 = 7$
6. $h(32) = 32 \% 13 = 6$ } $(6+1) \% 13 = 7$
   $(7+1) \% 13 = 8$
7. $h(31) = 31 \% 13 = 5$ } $(5+1) \% 13 = 6$
   $(6+1) \% 13 = 7$
   $(7+1) \% 13 = 8$
   $(8+1) \% 13 = 9$

(9+1) % 13 = 10

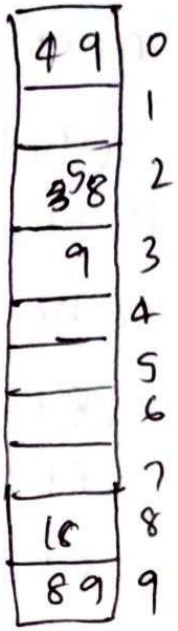
8. h(73) = 73 % 13 = 11

Disadvantage : crete clustu element

Quadratic Probing

Probe sequence :

$(h(k) + i^2) \bmod n$



Disadvantage : not all the location are probed.

Double Hashing

$(h_1(k) + i h_2(k)) \% n \quad i = 1, 2, 3 \ldots$

eg :- $h_1(k) = k \bmod 13$

$h_2(k) = 8 - (k \bmod 8)$

Rehashing

: done when hash table is full.

- Load ~~hash~~ factor

eg:- If load factor = 0.5 is it means 50% when hashing is 50% done then a new hash table is created and the rehashing done using new hash function (including all elements).

- If old hash table size = 10
  new hash table size = 20

Radix Sort

- uses a subfunction called : counting function
- non comparison based sorting algm.
- sorting digit by digit (1st lsb ....)

Eg: 170 45 75 90 802 24 2 66

{max = 802
{d = 3 (max digit)

I st : 170 90 802 2 24 45 75 66

II nd : 802 2 24 45 66 170 75 90

III rd : 2 24 45 66 75 90 170 802

- Stable sorting algorithm :
  eg: counting sort