Sorting Techniques

bitCode technologies

learn.innovate.share

Sorting

- Sorting is the operation of arranging the records/elements/keys in some sequential order according to an ordering criterion.
- Ordering criterion
- - Ascending order
- - Descending order
- Sorting data is preliminary step to searching.

Types of sorting

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Radix sort
- Quick sort

Algorithm for Bubble sorting

- two records are compared & interchanged immediately upon discovering that they are out of order
- This method will cause records with small keys to move or "bubble up"
- After the first pass, the record with the largest key will be in the nth position

Bubble sorting

Unsorted		Pass number(i)					Sorted
j	K_j	1	2	3	4	5	6
1	42	23	23	11	11	11	11
2	23	42	11	23	23	23	23
3	74	11	42	42	42	36	36
4	11	65	58	58	36	42	42
5	65	58	65	36	58	58	58
6	58	74	36	65	65	65	65
7	94	36	74	74	74	74	74
8	36	94	87	87	87	87	87
9	99	87	94	94	94	94	94
10	87	99	99	99	99	99	99

Contd.

- On each successive pass, the records with the next largest key will be placed in position n - 1, n - 2, . . ., 2, respectively, thereby resulting in a sorted table.

Contd.

```
/* Bubble sort for integers */

#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

 /*State the advantage of using macro instead of function */

 void bubble( int a[], int n )    {

        int i, j,passes, comps;
        passes = n-1;
        comps = n-1;

        for(i=0;i<passes;i++) {
           for(j=0;j<(comps-i);j++) {

                if( a[j]>a[j+1] )
                   SWAP(a[j],a[j+1]);
           }
        }
    }
```

Rules for Selection sorting

- Select the highest one and put it where it belongs.
- Select the second highest and place it in order.
- Do this for all the keys to be sorted.

+ Contd.

Initial Order	Pass I	Pass II	Pass III	IV
50	50	40	20	
70	30	30	30	
20	20	20	40	
40	40	50	50	
30	70	70	70	

Algorithm for Insertion sorting

- This makes use of the fact the keys are partly ordered in the entire set of keys.
- One key from the unordered array is selected and compared with the keys which are ordered.
- The correct location of this key would be when a key greater than this particular key is found

Contd..

- The keys from that key onwards in the ordered array are shifted one position to the right.
- The hole developed is then filled in by the key from the unordered array.
- This is repeated for all keys in the unordered array

Original	34	8	64	51	32	21	Positions Moved
After p = 2	8	34	64	51	32	21	1
After p = 3	8	34	64	51	32	21	0
After p = 4	8	34	51	64	32	21	1
After p = 5	8	32	34	51	64	21	3
After p = 6	8	21	32	34	51	64	4

Merge Sort

- Merging is a process of combining two or more sorted arrays/files into third sorted array/file
- In merge sort the first elements in both tables are compared & the smallest key is then stored in a third table.
- This process is repeated till end of the both arrays
- In this way two sorted tables are merged to form third sorted table

- Table 1 : 11 23 42
- Table 2 : 9 25

- We obtain the following trace:
- Table 1 11 23 42
- Table 2 25
- New Table 9

- Table I 23 42
- Table 2 25
- New Table 9 11

Quick Sort terminology

- Pivot :The key whose exact location is to be found in the sorted array.
- Keys to the left of pivot are smaller than the pivot and keys to the right of pivot are greater than the pivot.
- This is known as partitioning.

Quick Sort(A, lb,up)

If (lb >= ub) then return

If lb < ub then pivot_loc = partition (A, lb,ub)

Quick_sort(A, lb, pivot_loc)

Quick_sort(A, pivot_loc+1, ub)

Partition algorithm

- Let down = lower bound of array and up = upper bound of array, pivot = A[lb]
- Repeatedly increase the pointer down by one position while A[down] < pivot
- Repeatedly decrease pointer up by one position while A[up] >= pivot
- If up > down then interchange A[down] with A[up]
- If down < up goto 2
- Else interchange pivot, A[up]
- Pivot_loc = up
- Return pivot_loc

Quick Sort

Line No.	Key[1]	Key[2]	Key[3]	Key[4]	Key[5]	Key[6]	Key[7]	Key[8]	Key[9]	key[10]
										←
1	15	20	5	8	95	12	80	17	9	55
		→						←		
2	9	20	5	8	95	12	80	17	()	55
3	9	( )	5	8	95	12	80	17	20	55
			→							
4	9	12	5	8	95	( )	80	17	20	55
					←					
5	9	12	5	8	( )	95	80	17	20	55
6	9	12	5	8	15	95	80	17	20	55

bitCode technologies

Quick Sort

- Steps Involved

1. Remove the first data item, 15 as the pivot, mark it's position, scan the array from right to left, comparing the data item value 15. When you find the first smaller value remove it from it's current position and put it in position key[i] (Shown in line 2).

2. Scan line 2 from left to right beginning with position key[2], comparing value 15. when you find the first value greater than 15, store it in position marked by parenthesis (shown in line no 4).

3. Begin the right to left scan of line 3 with position key[8] looking for a value smaller than 15. When found store it in position marked by parenthesis.

4. Begin scanning line 4 from left to right at position key[3]. Find a value greater than 15, remove it, mark it's position, store it inside parenthesis in line 4. (shown in line 5).

5. Now when you attempt to scan line 5 from right to left beginning at position key[5], you are immediately at a parenthesized position determined by the previous left to right scan. This is the position to put the pivot data item 15. At this stage 15 is in correct position relative to the final sorted array.

Radix Sort

- In radix sort method the given set of unsorted numbers are compared on the basis of columns (digit place like units tens hundreds etc).
- Each comparison through the entire set is termed as Pass.
- The number of passes required to sort the given set of numbers depends on the number of digits of the largest number
- After each pass the number are placed in the respective pockets.

- 42, 23, 74, 11, 65, 57, 94, 36, 99, 87, 70, 81, 61

After the first pass on the unit digit position of each number we have:

		61								
		81			94			87		
	70	11	42	23	74	65	36	57		99
poc	0	1	2	3	4	5	6	7	8	9

- combining the contents of the pockets so that the contents of the "0" pocket are on the bottom and the contents of the "9" pocket are on the top, we obtain:

- 70, 11, 81, 61, 42, 23, 74, 94, 65, 36, 57, 87, 99

							61	70	81	94
		11	23	36	42	57	65	74	87	99
Poc	0	1	2	3	4	5	6	7	8	9

Comparison of Sorting methods

Algorithm	Average	Worst Case	
SELECTION	$n^2/4$	$n^2/4$	
BUBBLE	$n^2/4$	$n^2/2$	
MERGE	$0(n\log_2 n)$	$0(n\log_2 n)$	
QUICK	$0(n\log_2 n)$	$n^2/2$	
RADIX	0(m+n)	0(m+n)	

Complexity of Algorithm

- In general the *complexity* of an algorithm is

the amount of time and space (memory use) required to execute it.

- Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time,

- however it is possible to measure the length of the computation in other ways, say by the number of operations performed.

- the following loop performs the statement x := x + 1 exactly *n* times,

```
for i := 1 to n do
    x := x + 1
```

- The following double loop performs it *n*2 times:

```
for i := 1 to n do
for j := 1 to n do
x := x + 1
```

- The following one performs it $1 + 2 + 3 + \cdots + n = n(n + 1)/2$ times:

```
for i := 1 to n do
    for j := 1 to i do
        x := x + 1
```

- Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the *size* of the input.
- Since the execution time may be different for inputs of the same size, we define the following kinds of times:
- Best Case Time
- Worst Case Time
- Average Case Time

- 1. *Best-case time*: minimum time needed to execute the algorithm among all inputs of a given size *n*.
- 2. *Worst-case time*: maximum time needed to execute the algorithm among all inputs of a given size *n*.
- 3. *Average-case time*: average time needed to execute the algorithm among all inputs of a given size *n*.

Example

- For instance, assume that we have a list of n objects one of which is

  colored red and the others are colored blue, and we want to find the one that is colored red by examining the objects one by one. We measure time by the number of objects examined. In this problem the minimum time needed to find the red object would be 1 (in the lucky event that the first object examined turned out to be the red one). The maximum time would be n (if the red object turns out to be the last one).

- The average time is the average of all possible times: 1, 2, 3, . . . , n, which is $(1+2+3+\cdots+n)/n = (n+1)/2$.

- So in this example the best-case time is 1, the worst-case time is n and the average-case time is $(n + 1)/2$.

Complexity for Bubble Sort

- Since bubble sort is just a double loop its inner loop is executed

$(n - 1) + (n - 2) + \cdots + 1 = n(n - 1)/2$ times,

so it requires $n\ (n - 1)/2$ comparisons and possible swap operations.

- Hence its execution time is O(n2).