Sorting

(Section 7)

By:

Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Introduction and types

The general idea behind implementation of sorting techniques is that, the resulting output would be useful for easier searching. Searching in unsorted or unorganized items is quite inconvenient. The number of items also plays important role in sorting and searching.

Normally, a set of data that contains 'n' number of elements
e.g. 10, 8, 3, 32, 4, 5, 45, 645, 7,657, 567, 324, 433
is known as a **file** of size 'n'. Every item in the file is known as a **record**.

We know what a sorted data is. But, a general representation of sorted file is that

If d[] is the file, then
d[i] < d[j]
where, i = 0 → j-1
j = i + 1
The explanation of 'i' and 'j' depends on whether we are defining an ascending or a descending sort.

Types

There are two types of sorts depending on the place where the file is kept. If the file is kept in main memory and sorted, then it is known as '**internal**' sort and if the file also resides on auxiliary storage like magnetic disks, then it is known as '**external**' sort.

Stable sort

Let's say, we have duplicate search keys as shown below;

Name	Address
AAB	*BBC*
CDI	LKK
AAB	*BBA*
KKA	KSO
IEH	IEU

Look at record number 0 and 2. If we are sorting the records for name then if the sorting technique sorts the data as;

Name	Address
AAB	*BBC*
AAB	*BBA*
CDI	LKK
IEH	IEU
KKA	KSO

The sort is known as stable sort. In normal considerations, the data is sorted like;

AAB BBA
AAB BBC

But, the stable sort keeps track of original pattern unless specified.

Efficiency

Whenever efficiency of a given algorithm is addressed, we point to two important complexities.

1. Time complexity
2. Space complexity

Time complexity defines how much time is taken by a given algorithm to compute the result and space complexity defines how much memory space is needed by the algorithm. They are inversely proportional to each other.

Let's have look at an example to swap values of two variables.

Algorithm – 1

```
temp = b;
b = a;
a = temp;
```

Algorithm – 2

```
a = a + b;
b = a – b;
a = a – b;
```

In algo-1, 3 variables are used and three assignment operations finish the swap. In algo-2, there are only two variables but 3 arithmetic operations and assignment operations. If compared, algo-1 uses one more variable and algo-2 uses 3 more arithmetic operations.

For time complexity, algo-1 runs fast as it does only three assignment operations where as algo-2 are slower. It uses 3 arithmetic operations plus three assignment operations. You can see clearly that, for high time complexity, there is low space complexity and for low time complexity there is high space complexity. Generally, this property holds for all algorithms. If both of the time and space complexities increase for certain data set for an algorithm, then it is a bad algorithm.

The exact amount of time required to execute an algorithm will depend on the implementation (language) of the algorithm and on the actual machine. Hence, normally only the order of magnitude for the time required is expected from the analysis.

The time requirements will normally depends on the amount of input. Therefore, the time required is usually expressed as function of the size of the input (source) like, T(n) and the space as S(n).

Complexity of algorithms

The 'complexity' of an algorithm is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data.

Example: Linear search

- ITEM to be searched in the array DATA.

Worst case: C(n) = n (n – comparisons)
Average case: It is equally likely to occur ITEM at any position in the array DATA. Accordingly, the number of comparisons can be any of the nos. 1, 2, 3, 4, 5, ... , n, and each no. occurs with probability of p = 1/n.

Then,

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

$$= \frac{1}{n}(1+2+3+....+n)$$

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

i.e. $C(n) \approx \frac{n}{2}$

C(n) = k. n

Where, K is a constant.

Rate of growth: Big O notation

From above, clearly the complexity of algorithm increases as 'n' increases. It is usually the rate of growth of C(n) that we can examine. To talk about growth rate of functions we use what is known as 'big oh' notation. The searching time of linear search algorithm is said to be of order 'n' i.e. O(n), when it is proportion to n.

The sorting time is $O(n^2)$ means that it is proportional to n^2. That is $O(n^2) = k.n^2$ where 'k' is a positive integer.

➔ Suppose M is an algorithm suppose 'n' is the size of the input data. Clearly, the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) that we want to examine. This is usually done by comparing f(n) with some standard function such as;

$Log_2 n$, n, $n \log_2 n$, n^2, n^3, 2^n

The rates of growth for these standard functions are indicated in this table.

n\g(n)	logn	n	n.logn	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

The constants involved in the equations are not considered for the complexity comparison. For high degree of 'n', the constants are negligible.

Look at: Figure 6.1.3, page 333 in *Data Structures using C and C++* by Langsam, Augenstein, Tenenbaum

Negative example

Let's see whether $x^4 \in O(3x^3 + 5x^2 - 9)$ or not.

Let's say, $C(3x^3 + 5x^2 - 9) \geq x^4$ is always true. Easiest way is with limits (yes Calculus is good to know):

$$\lim_{x\to\infty} \frac{x^4}{C(3x^3+5x^2-9)} = \lim_{x\to\infty} \frac{x}{C(3+5/x\ -9/x^3)}$$

$$= \lim_{x\to\infty} \frac{x}{C(3+0-0)} = \frac{1}{3C} \cdot \lim_{x\to\infty} x = \infty$$

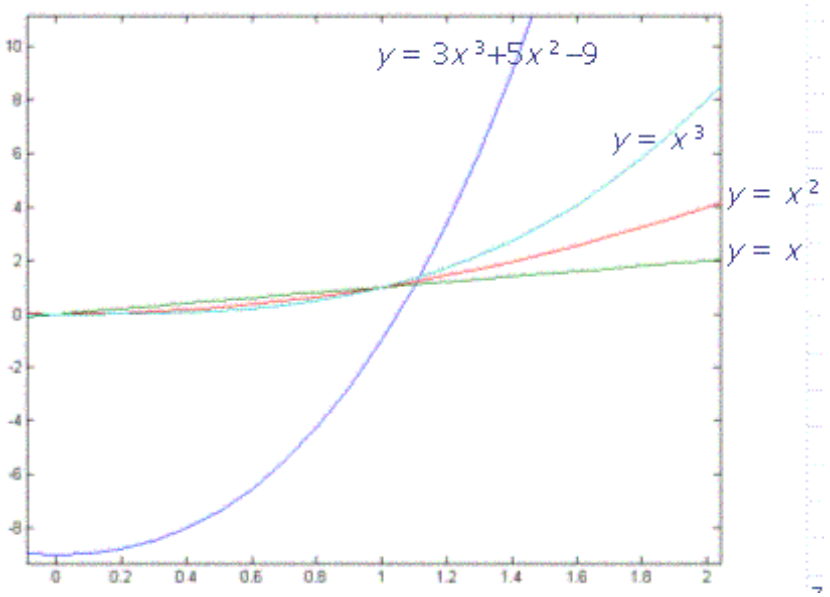
It shows that we were wrong at our assumption.

Positive example

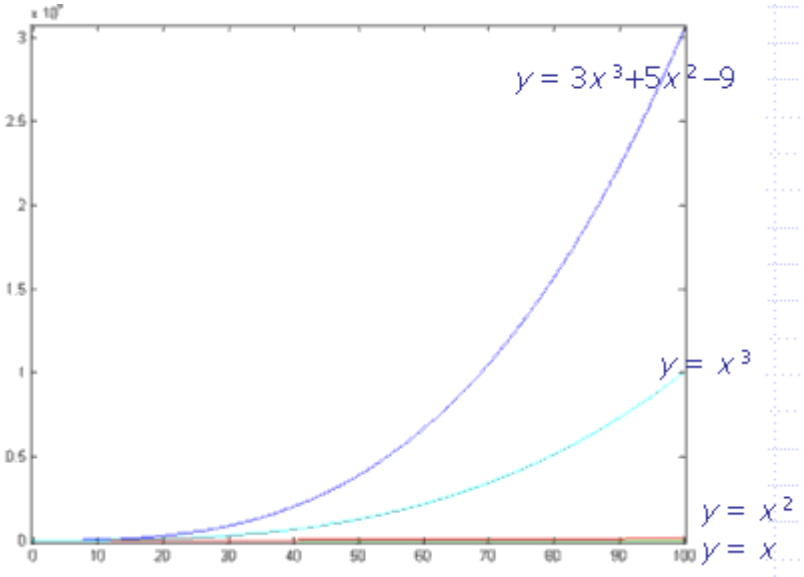
$3x^3 + 5x^2 - 9 = O(x^3)$. Compute:

$$\lim_{x\to\infty} \frac{x^3}{3x^3+5x^2-9} = \lim_{x\to\infty} \frac{1}{3+5/x\ -9/x^3} = \frac{1}{3}$$

yes, it holds.



Order of [0, 2]



order of [0, 100]

Space analysis

- easier than time analysis
- same techniques are used
- done for the space to store data, does not include the space to store algorithm itself.
- the space function also is usually in order notation.

Exchange sorts

Bubble sort

- Simplest algorithm.

- Start from the beginning.
- Swap if A[j] > A[j+1]. Compare a given element with each and every item that it follows.
- Stop swapping if A[j] > A[j+1] do not hold for any element.

Consider the list:

25, 57, 48, 37, 12, 92, 86, 33

Passes	1	2	3	4	5	6	7	8
1	25	57	48	37	12	92	86	33
2	25	48	37	12	57	86	33	92
3	25	37	12	48	57	33	86	92
4	25	12	37	48	33	57	86	92
5	12	25	37	33	48	57	86	92
6	12	25	33	37	48	57	86	92

In the worst case of bubble sort, the number of comparisons can go up to n^2 [(n-1) * (n-1) comparisons give order of n^2].

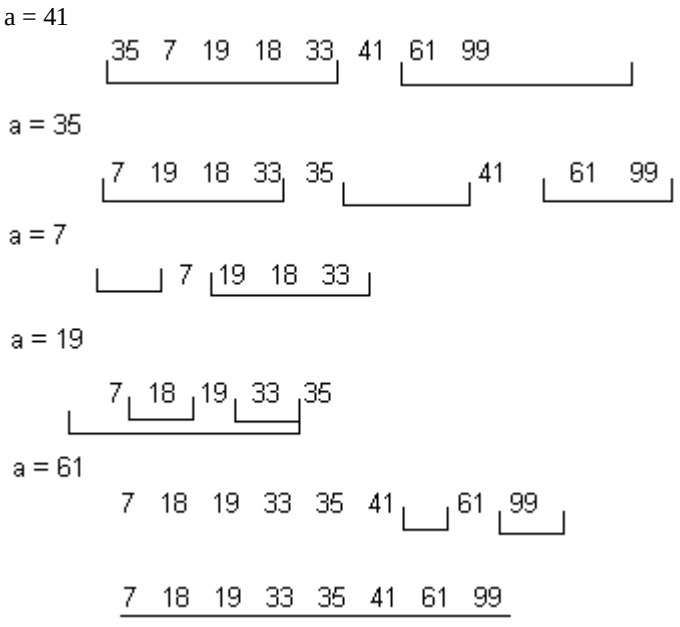
Quick sort

One kind of divide and conquer algorithm.

General idea is;

- select a key
- now start selecting the remaining elements. If the element is greater than the key, put on the right hand side, otherwise put on left hand side.
- sort all of the elements on right and left hand side in the same manner.

41, 35, 61, 7, 19, 99, 18, 33



Partition Illustration (algorithmic)

25, 57, 48, 37, 12, 92, 86, 33

Key	25	57	48	37	12	92	86	33
a = 25	Down→							up

move 'down' until the greater value is found

	25	57	48	37	12	92	86	33
		down						←up

move 'up' until a < k[j] i.e. 'up'

	25	57	48	37	12	92	86	33
		down			up			

if down < up, or hasn't crossed, exchange

	25	12	48	37	57	92	86	33
		Down→			up			
	25	12	48	37	57	92	86	33
			down		←←up			
	25	12	48	37	57	92	86	33
		up	down					

if up and down crossed, exchange the values of k[up] ←> a

	(12)	25	(48	37	57	92	86	33)
		pivot point						

Here, the left pan has only one element, so sorted. Let's do for right pan.

a = 48	12	25	48	37	57	92	86	33
			Down→→					up
	12	25	48	37	57	92	86	33
					down			←up

k[up] is less than 'a' already, so stop and exchange 'down' & 'up'

	12	25	48	37	33	92	86	57
					Down→			up
	12	25	48	37	33	92	86	57
						down		←←up
	12	25	48	37	33	92	86	57
					up	down		

crossed, so exchange 'a' and k[up]

	12	25	(33	37)	48	(92	86	57)
					pivot			

a = 33	12	25	33	37
			Down→	up
			33	37
				←up

crossed, so exchange 'a' and k[up]

			( ) 33	(37)

both pans are already sorted

a = 92	12	25	33	37	48	92	86	57
						Down		up
						→→		
	12	25	33	37	48	92	86	57
								up down

exchange 'a' and k[up]

a = 57	12	25	33	37	48	(57	86)	92 ( )
						Down	Up←	
						→		
	12	25	33	37	48	57	86	92
						up	down	

exchange 'a' and k[up]

	12	25	33	37	48	( ) 57	(86)	92

NULL and one element in left and right pan. Therefore sorted.

General Procedure: Qsort(k, LB, UB)

```
1. flag ← true
2. [perform sort]
      if LB <= UB then              (down < up)
            i ← LB
            j ← UB + 1
            key ← k[LB]

            repeat while(flag)
                  i ← i + 1
                  repeat while k[i] < key
                        i ← i + 1

                  repeat while k[ j] > key
                        j ← j – 1
                  if(i < j) then
                        k[i] <-> k[ j]
                  else
                        flag ← false
            k[LB] <-> k[ j]
            call Qsort(k, LB, j – 1)
            call Qsort(k, j + 1, UB)
3. Finish. Return
```

Quick sort is one of the efficient sorting algorithms. The sorting time is in the order of $(n \cdot \log_n)$.

Selection and tree sorts

In selection sort, a key is selected and then put on the proper place.

Straight selection sort

General algorithm:

1. Repeat through step -5 a total of (n-1) times.
2. Record the portion of the array already sorted.
3. Repeat step 4 for the elements in the unsorted portion of the list.
4. Record location of smallest element in unsorted portion list.
5. Exchange first element in unsorted list with smallest element.

Tracing

	Unsorted	Pass numbers (i) – sorted								
i	**k[j]**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	42	**11**	11	11	11	11	11	11	11	11
2	23	23	**23**	23	23	23	23	23	23	23
3	74	74	36	**36**	36	36	36	36	36	36
4	11	42	42	42	**42**	42	42	42	42	42
5	65	65	65	65	65	**58**	58	58	58	58
6	58	58	58	58	58	65	**65**	65	65	65
7	94	94	94	94	94	94	94	**74**	74	74
8	36	36	74	74	74	74	74	94	**87**	87
9	99	99	99	99	99	99	99	99	99	**94**
10	87	87	87	87	87	87	87	87	94	99

Straight selection sort has time complexity of $O(n^2)$ as all the elements are compared with the current key.

Binary tree sort

General idea is to create a binary search tree and access the elements either in LVR and RVL for ascending and descending order.

But, in case of inbalanced tree (right skewed and left skewed), the search time goes approximately n^2. Therefore, to minimize the search time, AVL trees are maintained. This will increase performance up to $(n \cdot \log_n)$. Still, BST requires some time to search and retrieve the data. After deletion of elements, there are some burden to maintain the BST property. It mean, the tree is accessed 2 times. To minimize the time for retrieval, heap is created. In heap sort, the heap creation takes time, but the retrieval takes no time.

Heap sort

Definition:

A heap is defined to be a binary tree with a key in each node, such that;

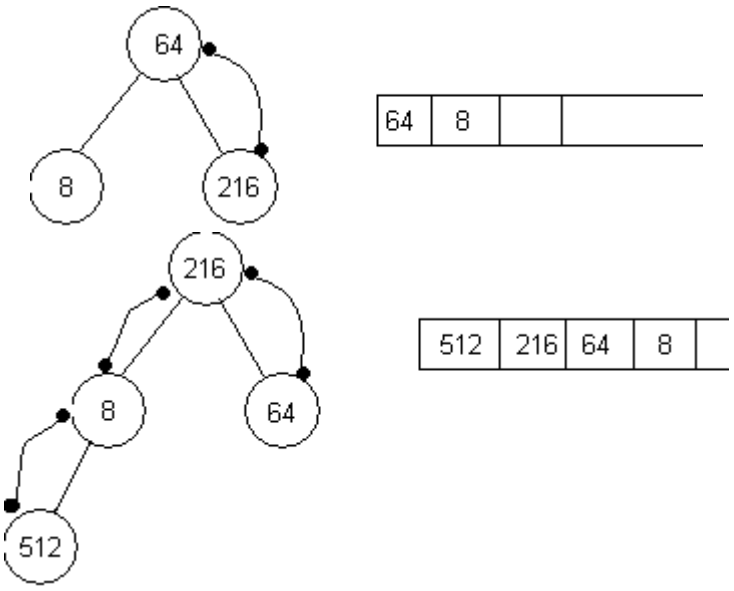
1. All the leaves of the tree are on two adjacent levels.
2. Additions on the lowest level occur to the left and all levels, except possibly the lowest, are filled.
3. The key in the root is at least as large as the keys in its children (if any), and the left & right sub-trees (if they exist) are again heaps (in case of max. heap).

Two types

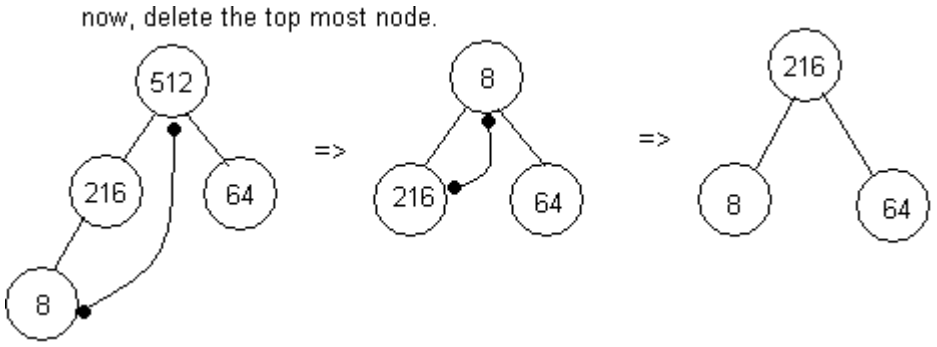
Max heap: descending heap. The largest element is kept at the top (as root node).
Min heap: ascending heap. The smallest element is kept at the top (as root node).

Heap construction

Inserting new record into existing heap such that new heap is formed after performing the insertion (adjustments might be needed).



For deletion, the root node is deleted and nodes are adjusted again so that the resulting tree again holds the heap property.



The deleted elements are stored at the last of the array.

Classwork: create heap of given data:

42, 23, 74, 11, 65, 58, 94, 36, 99, 87

procedure: create_heap(K, n)

```
1. Repeat thru step 7 for Q = 2, 3, 4, . . . , N
2. I ← Q,   key ← K[Q].
3. J ← I/2
4. Repeat thru step 6 while I > 1 & key > K[J].
5. K[I] ← K[J]
6. I ← J, J ← I / 2
            if, J < 1, then J ← 1
7. K[I] ← key
      return
```

procedure: heap(K, n)

```
1. call create_heap(K, n)
2. repeat thru step 10 for Q = N, N-1, . . . , 2
3. K[1] ←→ K[Q]
4. I ← 1, key ← K[1], J ← 2
5. If J+1 < Q
            then, if K[J+1] > K[J]
                  then J ← J + 1
6. Repeat thru step 10
            while J<= Q-1  and  K[J] > key
7. K[I] ← K[J]
```

```
8. I ← J, J ← 2 * I
9. If J+1 < Q
            then, if K[J + 1] > K[J]
                  then J ← J + 1
            else if J > N then
                  J ← N
10.   K[1] ← key
11.   return
```

Insertion sorts

Here, the elements are inserted into the resulting array so that it will always forms sorted array.

Simple insertion

The algorithm is directly applied to the file.

Number of passes = n-1.

General procedure:

```
1. Set A[0] = -∞ [initialize sentinel element]
2. Repeat steps 3 to 5 for k = 2, 3, ……, N
3. Set temp = A[k] & ptr = k-1.
4. Repeat while, temp < A[ptr]
      i. set A[ptr + 1] = A[ptr]          i.e. move element forward
      ii. set ptr = ptr – 1.
5. Set A[ptr+1] = temp
            - insert element in proper place
            [End of the step-2 loop].
Return
```

Tracing

Data: 77, 33, 44, 11, 88, 22, 66, 55

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K= 1 (original data)	-∞	77	33	44	11	88	22	66	55
k = 2	-∞	33	77						
3	-∞	33	44	77					
4	-∞								
5	-∞								
6	-∞								
7	-∞								

When k = 2, A[k] = 33, temp = 33, ptr = 1, A[ptr] = 77
now, while temp < A[ptr]

A[ptr + 1] = A[2] = 77
ptr = 0

When k = 3, A[k] = 44, temp = 44, ptr = 2, A[ptr] = 77
while temp < A[ptr]

A[ptr + 1] = A[2] = 44

ptr = 1, temp > A[ptr],
end.

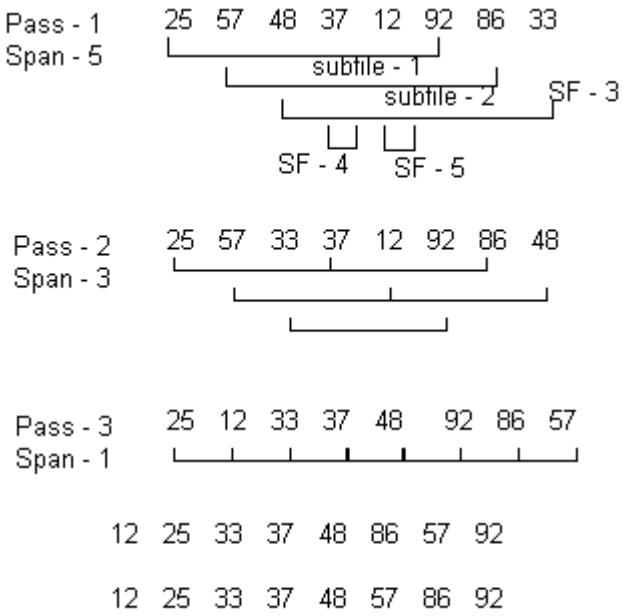
Shell sort

- due to Shell, D. L (1959)
- sometimes called **diminishing increment sort (advanced insertion)**.
- the idea behind shell sort is:

> "For small and almost sorted files, insertion sort is very efficient. Thus for large files to be sorted, sub-files are sorted as insertion sort in an increment & again sub-files are divided with diminishing increment. Finally increment will be one."

Tracing with data

Original file: 25, 57, 48, 37, 12, 92, 86, 33



No. of span = no. of sub-files.

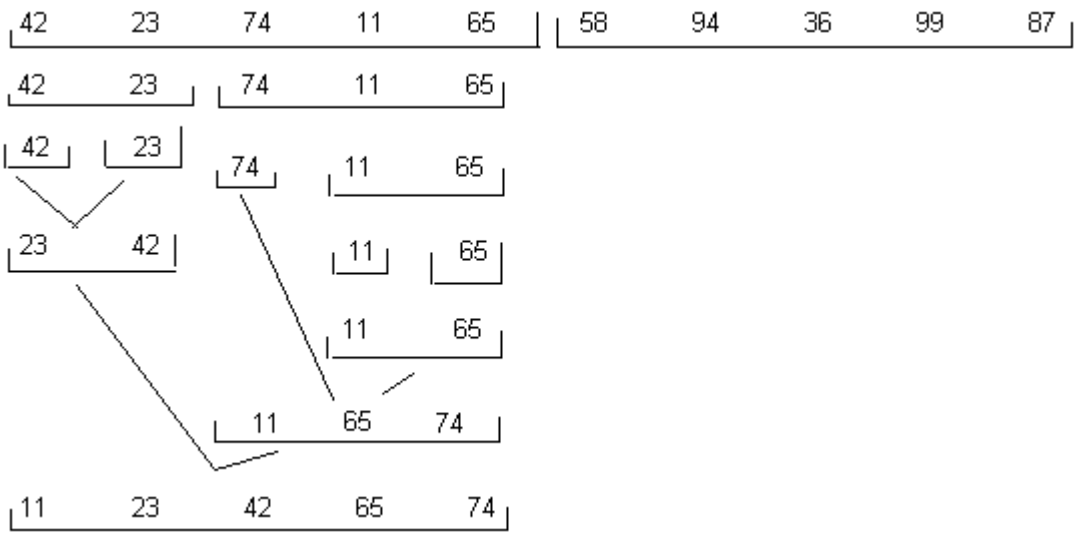
Merge and radix sorts

Merge sort

The file is divided and sorted. The sorted portions are again merged to get final result. Also a divide and conquer algorithm.

```
void sort(list){
      if (list has length greater than 1){
            partition the list into lowList, highList;
            sort(lowList);
            sort(highList);
            Combine(lowList, highList);
      }
}
```

Sample: 42, 23, 74, 11, 65, 58, 94, 36, 99, 87



Merging

- arranging two ordered sub-tables into a table.

Consider:

K: 11 23 42 9 25

Let, first – pointer = 11
second - pointer = 25

A general algorithm (procedure) mSort(K, start, finish)

```
1.    size ← finish – start + 1
2.    if size <= 1, return;
3.    middle ← start + (size / 2) + 1
4.    call mSort(k, start, middle)
5.    call mSort(k, middle + 1, finish)
call simple merge (k, start, middle + 1, finish)
```

Procedure: simple-merge(k, first, second, last)

```
1. [initialize]
      i ← first
      j ← second
      L ← 0

2:    [compare & output the smallest]
      repeat while i<second and j<= last
            if k[i] <= k[j] then
```

```
                    L ← L + 1
                    temp[L] ← k[i]
                    i ← i + 1
              else
                    L ← L + 1
                    temp[L] ← k[j]
                    j ← j + 1
3:    [copy remaining elements unprocessed in output area]
      if i >= second then,
              repeat while j <= last
                    L ← L + 1
                    temp[L] ← k[ j]
                    j ← j + 1
      else,
              repeat while i < second
                    L ← L + 1
                    temp[L] ← k[i]
                    i ← i + 1
4.    [copy elements from temporary area to original area]
      repeat for i = 1, 2, …., L
              k[first – 1 + i] = temp [i]
```

Radix sort (bucket sort)

General idea is that; find the radix for each and every digit, and place in corresponding place.

Demonstration:

Input 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

(number of steps = maximum number of digits.)

Buckets after first-step (put according to Least Significant digit)

0	1	512	343	64	125	216	27	8	729
0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***

Buckets after second-step (put according to second-least significant digit)

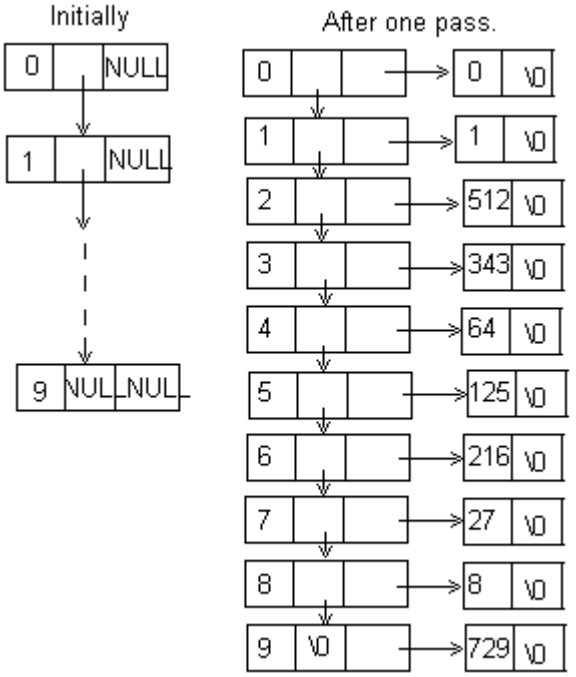
8 1 0		729 27 125		343		64			
0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***

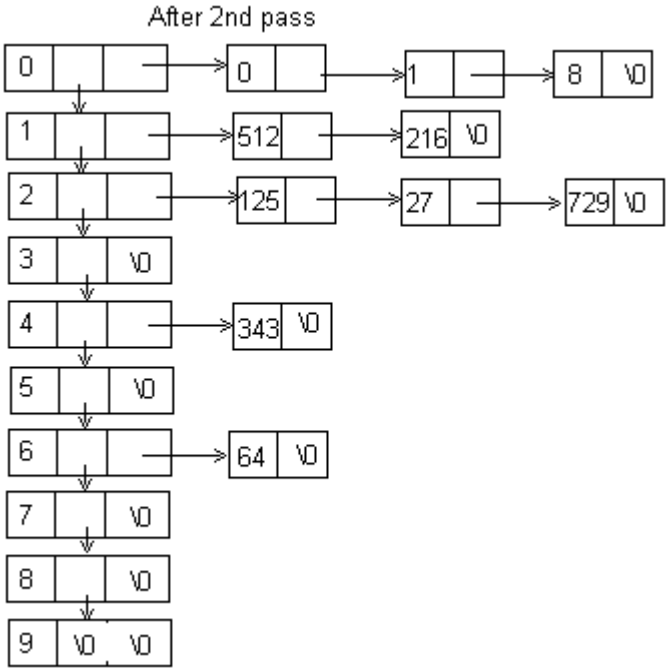
Buckets after third step (put according to most significant digit)

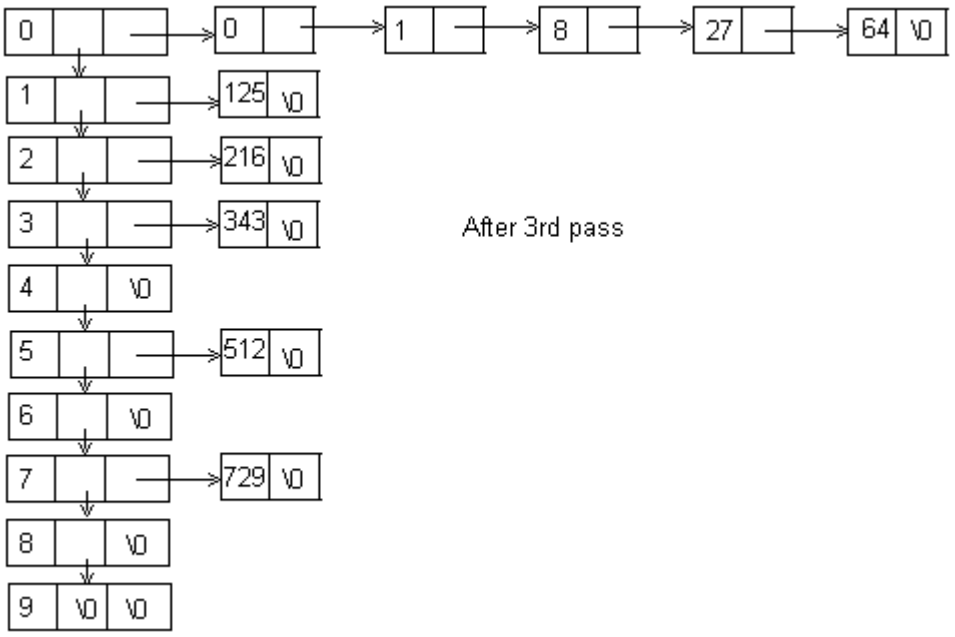
64 27 8 1 0	125	216	343		512		729		
0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***

Now, to read the sorted data, just start reading elements upwards from array[0] to array[9].

Multi-linked list implementation







Comparisons of sorting algorithms

While comparing the sorting algorithms, we always look for the time complexity. Due to high availability and economy of the memory chip, most f the time, only time comp lexity is addressed.

Before comparing the sorting algorithm, it's necessary to calculate the order of the time complexity of a given algorithm. We have already seen how time complexity of a given algorithm is calculated?

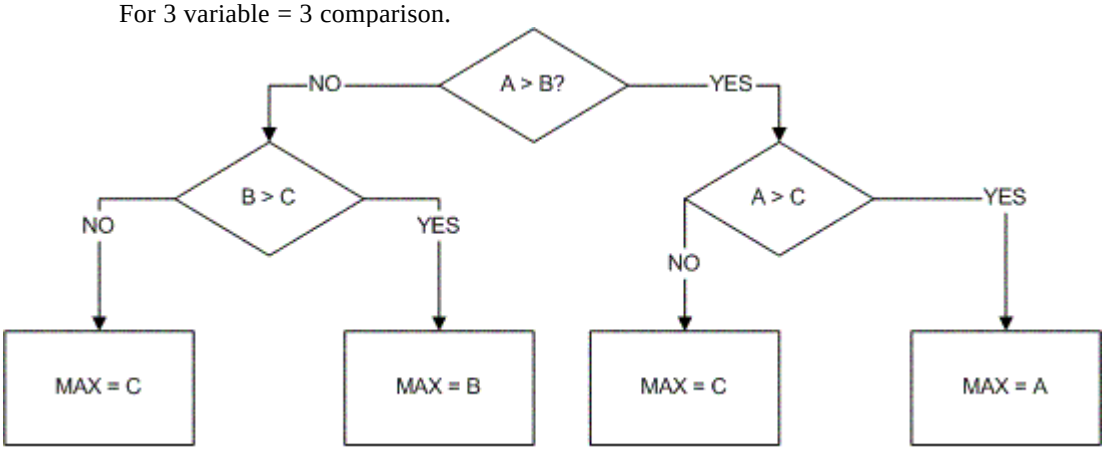
Have a look at **Complexity of algorithms, page 2.**

Algorithm Analysis (case study)

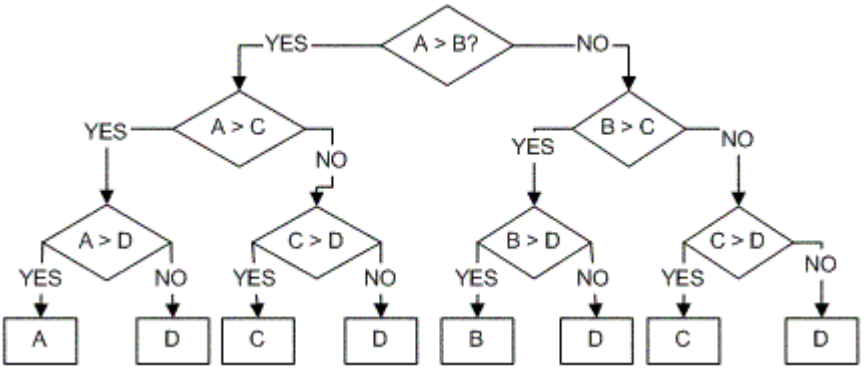
Algorithm to find maximum number.

Algorithm – 1

For 1 variable = no comparison.
For 2 variable = 1 comparison.
For 3 variable = 3 comparison.



For 4 variable = 7 comparison

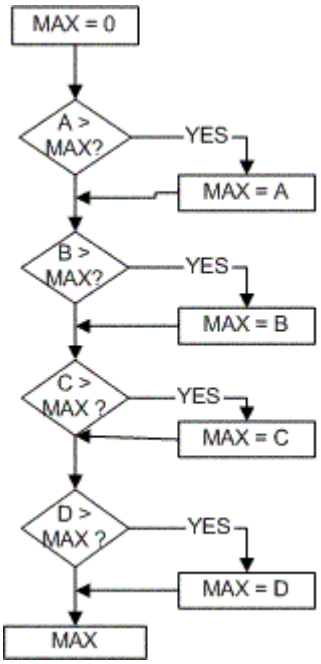


Analysis

No. of variables	Comparison	
1	0	$2^0 - 1$
2	1	$2^1 - 1$
3	3	$2^2 - 1$
4	7	$2^3 - 1$
.	.	
.	.	
n	$2^{n-1} - 1$	

For large n, runtime efficiency is proportionally equal to 2^{n-1}.
Or, run time efficiency ~ 2^n.

Algorithm – 2



Analysis

Number of Variables	Comparison
1	0
2	1
3	2
.	.
n	n-1

For n sufficiently large, runtime efficiency is directly proportional to n.

Here,

Algorithm – 1 has time efficiency in the order of 2^n $O(2^n)$
Algorithm – 2 has time efficiency in the order of n $O(n)$

Further, let's again look at **time complexity of binary search**.

1. In binary search, each comparison in the binary search reduces the number of possible candidates by a factor of 2. Thus the maximum number of comparisons is $\log_2 n$. **How?**

   Let's say, we require $f(n)$ comparisons to locate a given item.

   Then, the number of nodes in the tree can be related with the comparisons as;

   $$2^{f(n)} > n$$
   or
   $$f(n) = [\log_2 n] + 1$$

   Thus, searching time is order of $O(\log_2 n)$.

   Example:

   Suppose a file contains 15 items, then we need;

   $$f(n) = [\log_2 n] + 1$$
   $$f(n) = \log_2 15 + 1$$

   or, simply
   $$2^4 > 15$$

   Therefore, $f(n) = 4 + 1 = 5$. i.e. in worst case, there will be 5 comparisons at max.

Example:

Suppose a file contains 1,000,000 items.

And, we know	$2^{10} = 1024$
Hence,	$2^{20} > 1000^2$
	$= 1,000,000$
i.e.	$f(n) = 20$

Here, we saw that, using binary search algorithm, one requires only about 20 comparisons to find the location of an item in a data array with 1,000,000 elements. But one should remember that this algorithm requires two conditions.

i. The list must be sorted.
ii. One must have direct access to the middle element in any sublist.

2. You can use inary search in conjunction with the indexed sequential search organization method.

3. Not: binary search on indices can be used only if the index table is sorted as an array. Particularly useless in situations where there are many insertions or deletions, so that an array structure is inappropriate.

Sorting time summary

The appropriate number of comparisons and the complexity of the various sorting algorithms are summarized in the following table.

Algorithm	Worse case	Average case
Bubble sort	$\frac{n*(n-1)}{2} = O(n^2)$	$\frac{n*(n-1)}{2} = O(n^2)$
Quick sort	$\frac{n*(n+3)}{2} = O(n^2)$	1.4 n logn = O(n.logn)
Insertion sort	$\frac{n*(n-1)}{2} = O(n^2)$	$\frac{n*(n-1)}{4} = O(n^2)$
Selection sort	$\frac{n*(n-1)}{2} = O(n^2)$	$\frac{n*(n-1)}{2} = O(n^2)$
Merge sort	n.logn = O(n.logn)	n.logn = O(n.logn)
Heap sort	n.logn = O(n.logn)	n.logn = O(n.logn)

Selecting a sort

Algorithm	Comments
Bubble sort	Good for small n (n < 10)
Quick sort	Excellent for virtual memory environment
Insertion sort	Good for almost sorted data
Selection sort	Good for partially sorted data and small 'n'
Merge sort	Good for external file sorting
Heap sort	As efficient as quick sort in the average case and far superior to quick sort in the worst case