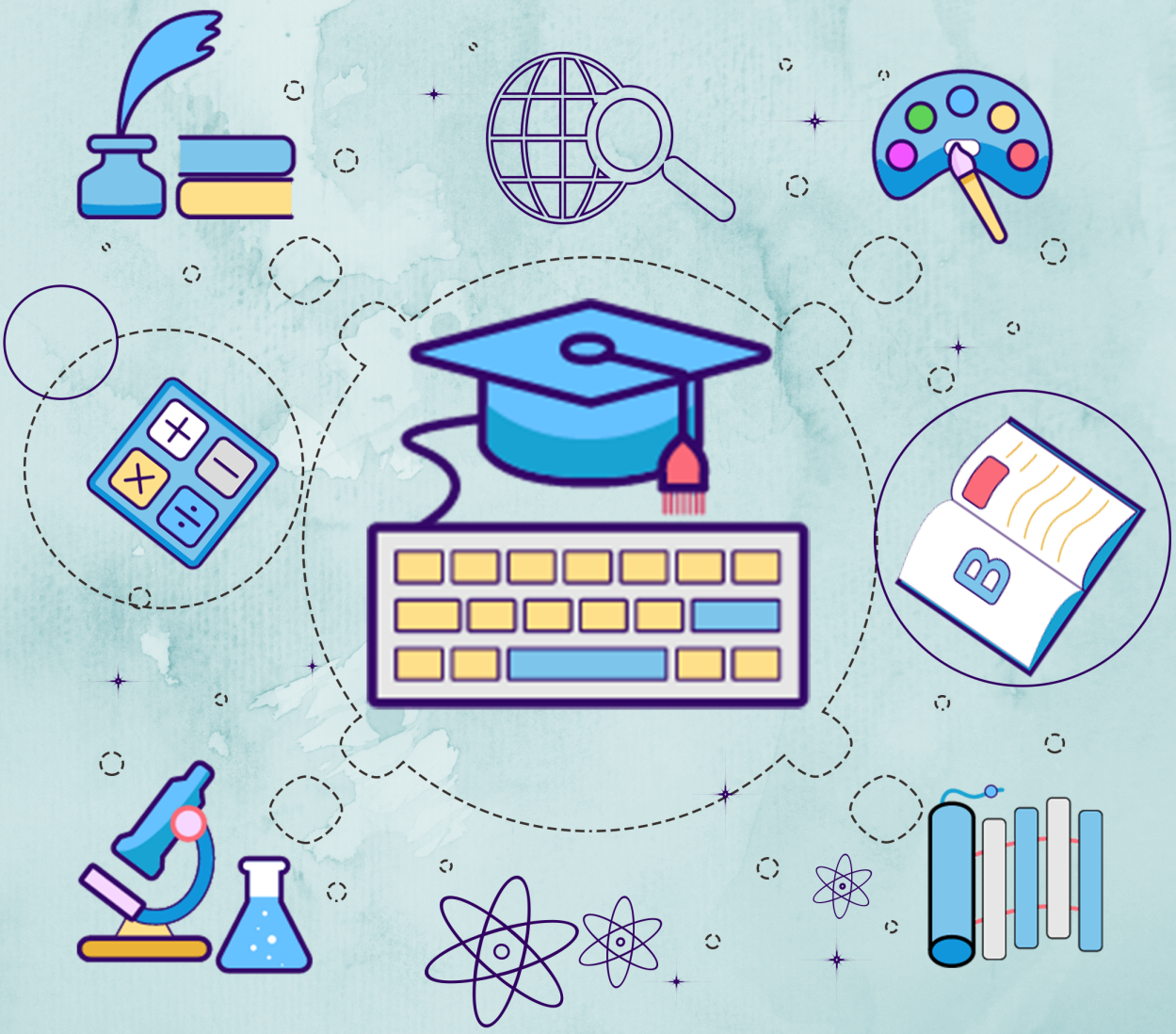
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

DATA STRUCTURES

CST 201

Module 5

Related Link :

- KTU S3 STUDY MATERIALS
- KTU S3 NOTES
- KTU S3 SYLLABUS
- KTU S3 TEXTBOOK PDF
- KTU S3 PREVIOUS YEAR SOLVED QUESTION PAPER

MODULE 5

Data Structure

Sorting method can be classified into two.

1)Internal Sorting

2)External Sorting

In internal sorting the data to be sorted is placed in main memory. In external sorting the data is placed in external memory such as hard disk, floppy disk etc. The internal sorting can be classified into

1)n^2 sorting

2)n log n sorting

n^2 sorting - it can be classified into

1)Bubble sort

2) Selection sort

3) Insertion sort

nlogn sorting - It can be classified into

1) Merge sort

2)Quick sort

3) Heap sort

Bubble sort

Bubblesort(a[],n)

Input- an array a[size], n is the no. of element currently present in array

Output- Sorted array

DS- Array

Algorithm

1. Start
2. i=0
3. While i<n-1
    1. j=0
        2. While j<n-1-i
            1. If a[j]>a[j+1]
                1.temp=a[j]
                2. a[j]=a[j+1]
                3. a[j+1]=temp
            2.end if
            3. j=j+1

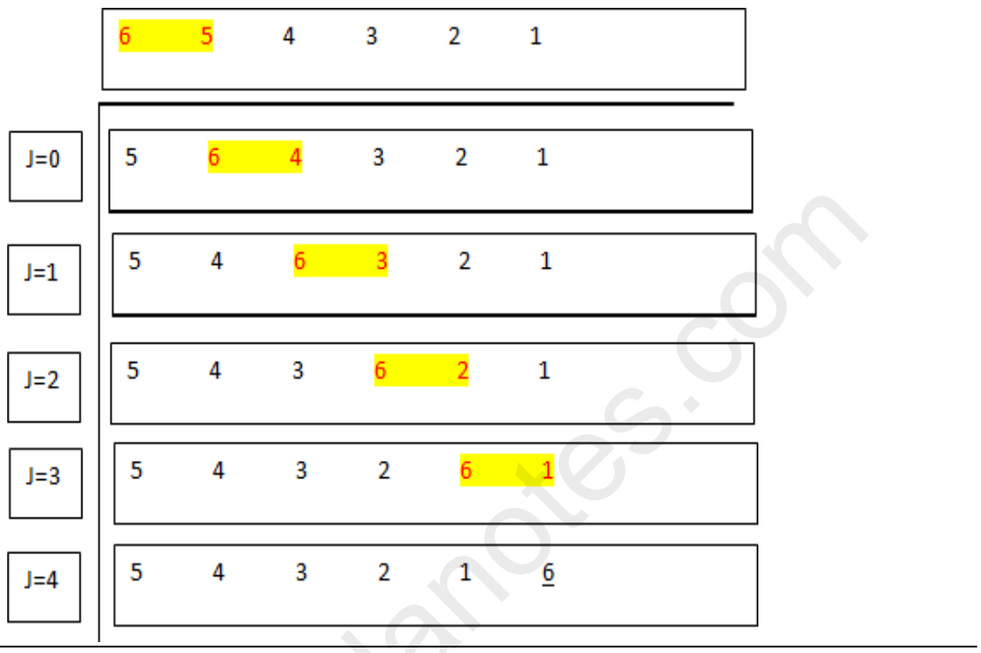
3.end while

4.i=i+1

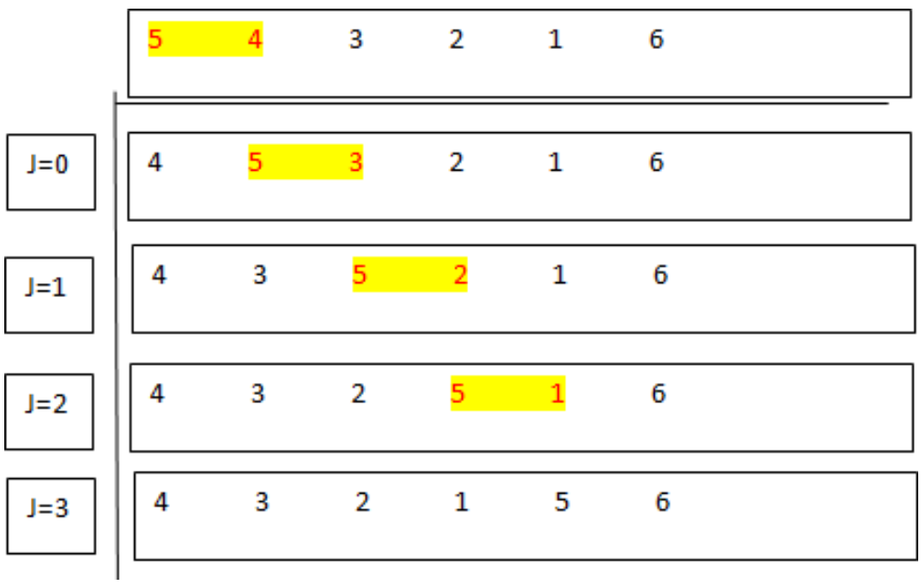
4. end while

Here first element is compared with second element. If first one is greater than second element then swap each other. Then second element is compared with third element . If second element is greater than third element then perform swapping. This process is continued until the comparison of (n-1)th element with nth element. These process continues (n-1) times. Consider the example-

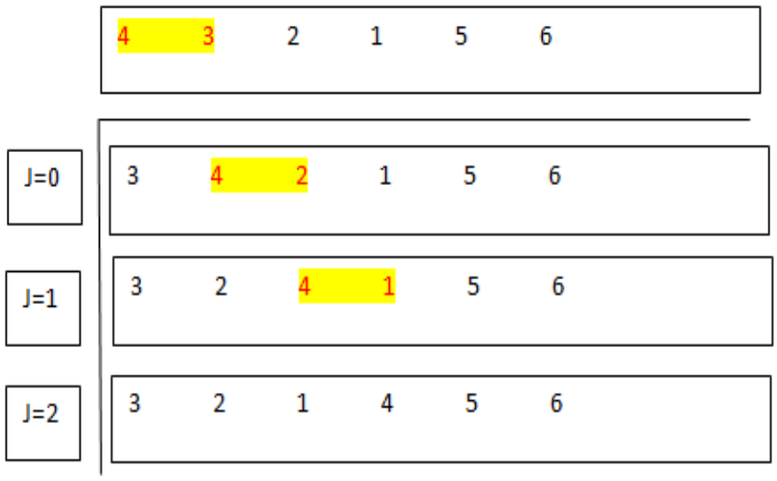
i=0



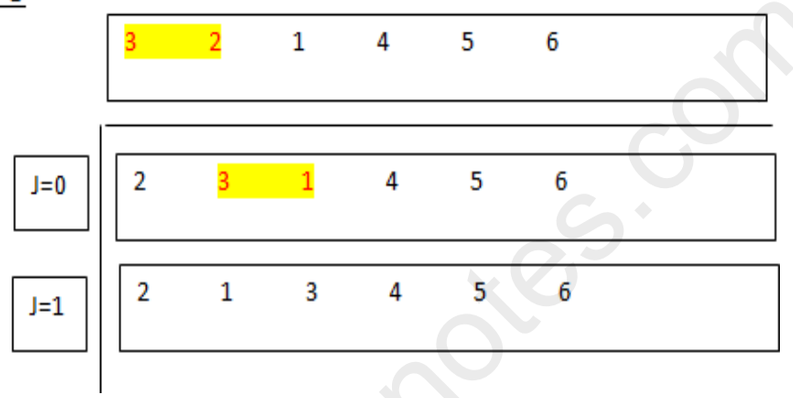
i=1



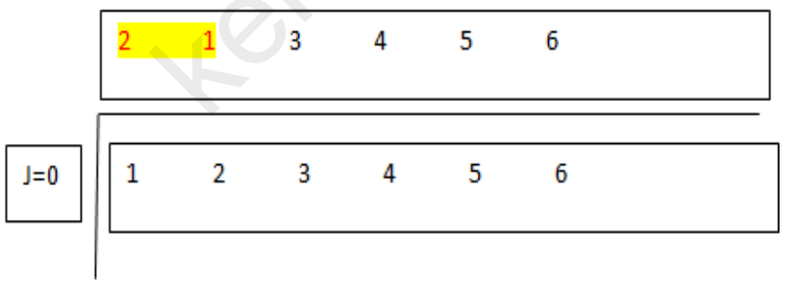
i=2



i=3



i=4



Thus i have values from 0 to n-1 and j have values from 0 to n-1-i.

Analysis

Here during the first iteration n-1 comparisons are required. During the second iteration n-2 comparisons are required etc..during last iteration, 1 comparison is required. Therefore total comparisons

= (n-1)+(n-2)+(n-3)+.....+1

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

=O(n^2)

SELECTION SORT

Selectionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

1: Start

2. i=0

3. while i<n-1 do

    1. j=i+1
    2. small=i
    3. while j<n do
        1. if a[small]>a[j]
            1. small=j
        2. end if
        3. j=j+1
        4. end while
        5. if i!=small
            1. temp=a[i]
            2. a[i]=a[small]
            3. a[small]=temp
        6. end if
        7. i=i+1

7. end while

8. stop

Here the first element in the array is selected as the smallest element. Then check for any element smaller than this from the second position to the last. If found any, then they are interchanged. Similarly next we check for the smallest element from third position to last. And the process continues upto (n-1)th position.

Analysis

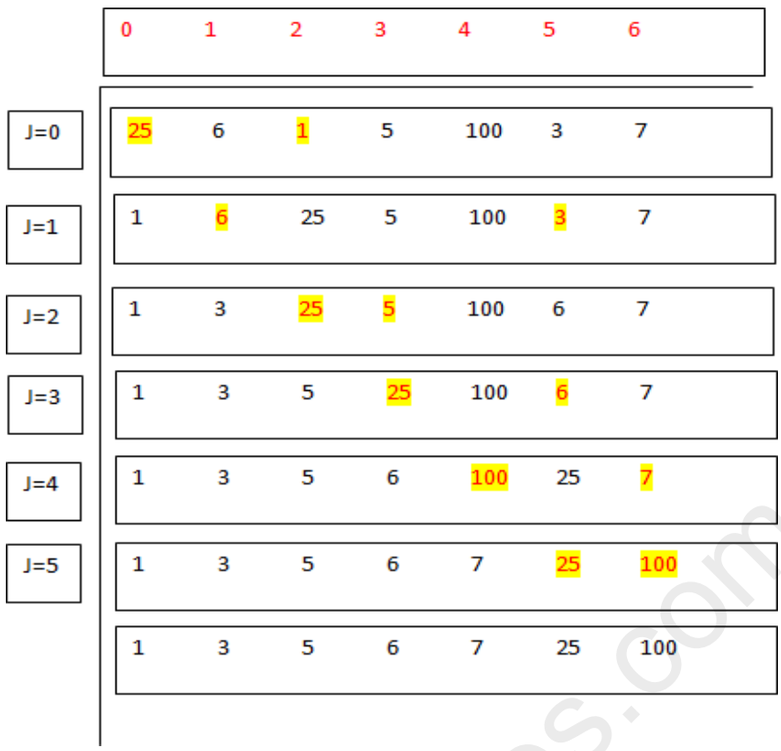
Here first iteration when i=0 takes n-1 comparisons, during second iteration takes n-2 comparison and so on.

Total no. of comparisons=(n-1)+(n-2)+(n-3)+.....+2+1

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

= O(n^2 **)**

Example



INSERTION SORT

insertionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

1. Start
2. i=1
3. While i<n
    1. j=i
    2. While a[j]<a[j-1] and j>0
        1. t=a[j]
        2. a[j]=a[j-1]
        3. a[j-1]=t
        4. j=j-1
    3. end while
    4. i=i+1
4. end while
5. stop

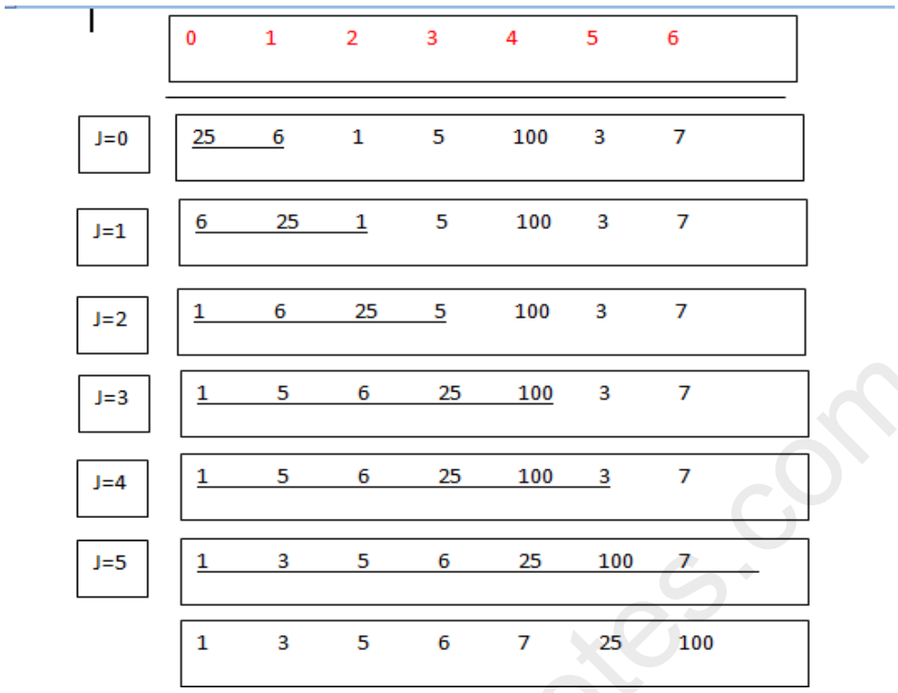
Here first iteration takes 1 comparison, 2nd iteration takes 2 comparison, and last iteration takes (n-1) comparison

Total comparison=1+2+...+n-1

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

$$= O(n^2)$$

Example



MERGE SORT

mergesort(start,end)

Input: An unsorted array a[], n is the no.of elements, start indicates lower bound of the array, end indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

1.start

2. if(start!= end)

    1. mid=(start+end)/2
    2.mergesort(start,mid)
    3.mergesort(mid+1,end)
    4. merge(start,mid,end)

3. end if

4.stop

Algorithm for merge

Merge(start,mid,end)

1. i=start
2. j=mid+1
3. k=start
4. while i<=mid and j<= end do
    1. if a[i]<=a[j]
        1.temp[k]=a[i]
        2. i=i+1
        3. k=k+1
    2. Else
        1.Temp[k] =a[j]
        2. J=j+1
        3. k=k+1
    3. end if

1.end while

2. while i<=mid do
    1.temp[k]=a[i]
    2. i=i+1
    3. k=k+1

7. end while

8. While j<=end
    1.Temp[k]=a[j]
    2. j=j+1
    3. k=k+1

9. end while

10. k=start

11. while k<=end
    1.a[k]=temp[k]
    2. k=k+1

12.end while

13.stop

Merge sort follows the strategy divide and conquer method. Here the given base array is divided into two sub lists. These 2 sub lists is again divided into 4 sub lists. The process is continued until subsists contain single element. Then repeatedly merge these two sub lists to a single sub list. So that a sorted array is created from sorted sub lists. The process continues until a sub list contains all the elements that are sorted**.**

Analysis of merge sort

Suppose the merge sort follows the recurrence relation.

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

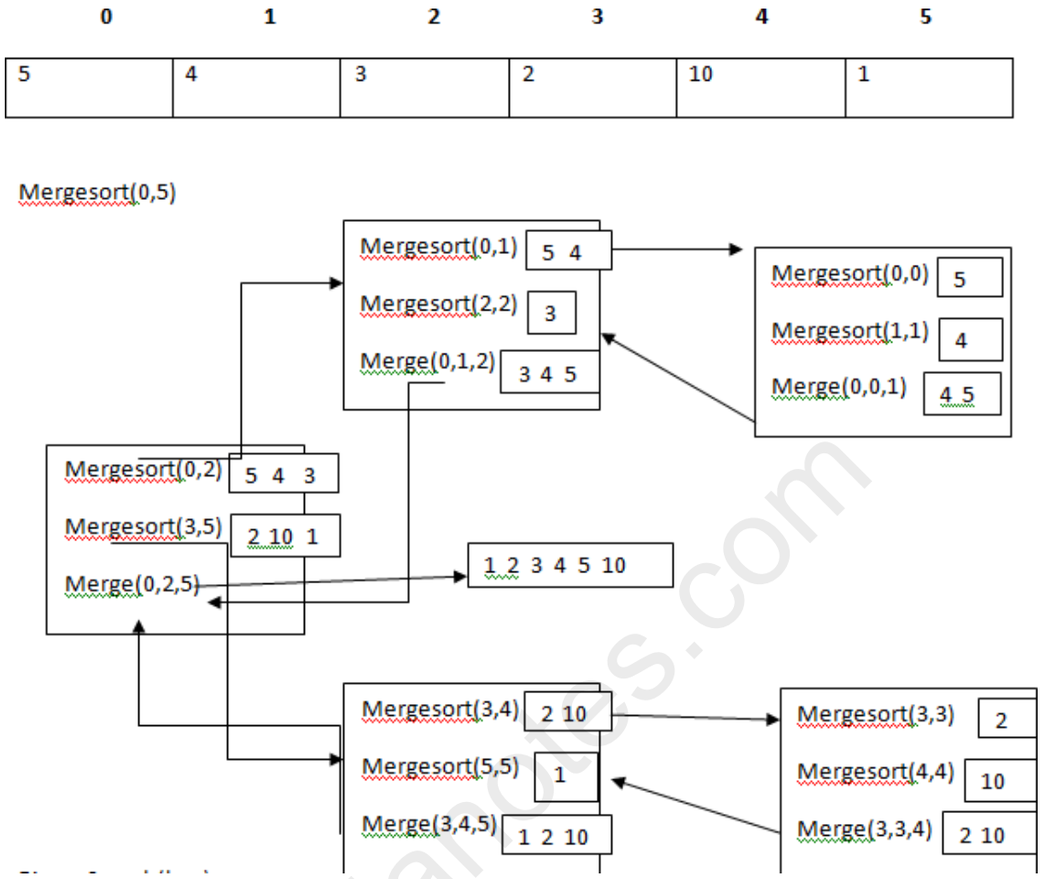
Comparing with standard recurrence equation $T(n)=aT(n/b)+f(n)$

a=2, b=2, f(n)=n

then $n^{\log_b a}$ =n

therefore time complexity is $\Theta(n \log n)$

Consider the following example



QUICK SORT

Quicksort(lb,ub)

Input: An unsorted array a[], n is the no.of elements, lb indicates lower bound of the array, ub indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

1.start

2.if lb<ub

1.loc=partition(lb,ub)

2.quicksort(lb,loc-1)

3.quicksort(loc+1,ub)

3. end if

4. stop

Algorithm for partition

Partition(lb,ub)

1. pivot=a[lb]
2. up=ub
3. down=lb
4. while down< up
    1. while pivot>=a[down] and down<=up 1.down=down+1
    2. end while
    3. while a[up]>pivot
        1. up=up-1
    4. end while
    5. if down< = up
        1. swap(a[down], a[up])
    6. end if
5. end while
6. swap(a[lb],a[up])
7. return up
8. stop

Quick sort algorithm basically takes the following steps

1. Choose a pivot element(the pivot element may be in any position). Normally first element is chosen as pivot

2. Perform partition function in such a way that all the elements which are lesser than pivot goes to the left part of the array and all the elements greater than pivot go to the right part of the array. The partition function also places pivot in exact position.

3. Recursively perform quick sort algorithm in these two sub arrays

Analysis

The recurrence relation ofquick sort in worst case is

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

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

Comparing with the standard recurrence equation

$T(n)=aT(n/b)+f(n)$

Substitute the values of a and b in $n^{\log_b a}$ and compare it with f(n).

In this case a=2, b=2 then $n^{\log_b a}=n$

Whereas f(n)= n^2. Therefore time complexity of quick sort is $\Theta(n^2)$

SEQUENTIAL SEARCH/ Linear Search Sequential_search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

DS: Array

1. Start
2. i=0
3. flag=0
4. While i<n and flag=0
    1. If a[i]=key
        1. Flag=1
    2. Index=i
    2.end if
    3. i=i+1
5. end while
6. if flag=1
    1. print “the key is found at location index”
7. else
    1.print “key is not found”
8. end if
9. stop

Analysis

In this algorithm the key is searched from first to last position in linear manner. In the case of a successful search, it search elements up to the position in the array where it finds the key. Suppose it finds the key at first position, the algorithm behaves like best case, If the key is at the last position, then algorithm behaves like worst case. Thus the worst case time complexity is equal to the no. of comparison at worst case ie., equal to O(n). The time complexity in best case is O(1).

The average case time complexity =(no. of comparisons required when the key is in the first position + no. of comparisons required when the key is in second position+…+ no. of comparison when key is in nth position)/n

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

Binary Search

Binary Search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

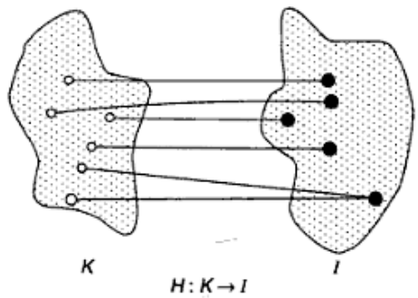
DS: Array

1. Start
2. Start=0,end=n-1
3. Middle=(start+end)/2
4. While key!=a[middle] and start<end
    1. If key>a[middle]
        1. Start=middle+1
        2.else
            1. end= middle-1
    3. end if
    4. middle=(start+end)/2
5. end while
6. if key=a[middle]
    1. print “the key is found at the position”
7. else
    1. print “the key is not found”
8. end if
9. stop

HASHING

We have seen about different search techniques (linear search, binary search) where search time is basically dependent on the no of elements and no. of comparisons performed.

Hashing is a technique which gives constant search time. In hashing the key value is stored in the hash table depending on the hash function. The hash function maps the key into corresponding index of the array(hash table). The main idea behind any hashing technique is to find one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in figure below where K denotes a set of key values, I denotes a range of indices, and H denotes the mapping function from K to I.



All key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the hash function. There are two principal criteria in deciding hash function H:K->I as follows.

1) The function H should be very easy and quick to compute
2) It should be easy to implement

As an example let us consider a hash table of size 10 whose indices are 0,1,2,...9.Suppose a set of key values are 10,19,35,43,62,59,31,49,77,33.Let us assume the hash function as stated below

1) Add the two digits in the key
2) Take the digit at the unit place of the result as index , ignore the digits at tenth place if any

Using this hash function, the mapping from key values to indices and to hash tables are shown below.

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

H: K→I

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

HASH FUNCTIONS

There are various methods to define hash function

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number n of keys in K. The hash function H is then defined by

H(k)=k(MOD h) if indices start from 0

Or

H(k)=k(MOD h)+1 if indices start from 1

Where k€K, a key value. The operator MOD defines the modulo arithmetic operator operation, which is equal to dividing k by h. For example if k=31 and **h=13 then,**

H(31)=31 MOD 13=5 (OR)

H(31)=31(MOD 13)+1=6

h is generally chosen to be a prime number and equal to the sizeof hash table

MID SQUARE METHOD

Another hash function which has been widely used in many applications is the mid square method. The hash function H is defined by H(k)=x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. example-

k	**:**	**1234**	**2345**	**3456**
k^2	**:**	**1522756**	**5499025**	**11943936**
H(k)	**:**	**525**	**492**	**933**

For a three digit index requirement, after finding the square of key values,the digits at 2nd, 4th and 6th position are chosen as their hash values.

FOLDING METHOD

Another fair method for a hash function is folding method. In this method, the key k is partitioned into a number of parts k 1 , k2..kn where each part has equal no.of digits as the required address(index) width. Then these parts are added together in the hash function.

H(k)=k1+k2+...+kn. Where the last carry, if any is ignored. There are mainly two variations of this method.

1)fold shifting method

2)fold boundary method

Fold Shifting Method

In this method, after the partition even parts like k2, k4 are reversed before addition.

Fold boundary method

In this method, after the partition the boundary parts are reversed before addition

Example

-Assume size of each part is 2 then, the hash function is computed as follows

Key values k :	**1522756**	**5499025**	**11943936**
Chopping :	**01 52 27 56**	**05 49 90 25**	**11 94 39 36**
Pure folding :	**01+52+27+56=136**	**05+49+90+25=169**	**11+94+39+36=180**
Fold Shifting:	**10+52+72+56=190**	**50+49+09+25=133**	**11+94+93+36=234**
Fold Boundary :	**10+52+27+65=154**	**50+49+90+52=241**	**11+94+39+63=207**

DIGIT ANALYSIS METHOD

This method is particularly useful in the case of static files where the key values of all the records are known in advance. The basic idea of this hash function is to form hash address by extracting and/or shifting the extracted digits of the key. For any given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and rearrangement is finalized after analysis of hash functions under different criteria.

Example: given a key value 6732541, it can be transformed to the hash address 427 by extracting the digits from even position. And then reversing this combination.ie 724 is the hash address.

Collision resolution and overflow handling techniques

There are several methods to resolve collision. Two important methods are listed below:

1) **Closed hashing(linear probing)**
2) **Open hashing (chaining)**

CLOSED HASHING

Suppose there is a hash table of size h and the key value is mapped to location i, with a hash function. The closed hashing can then be stated as follows.

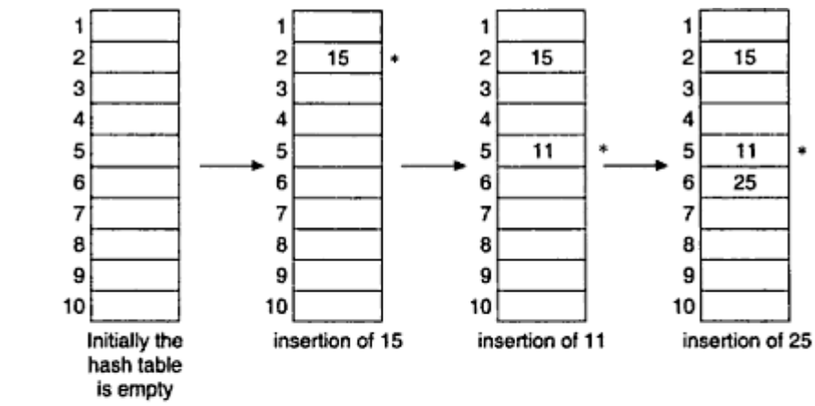
Start with the hash address where the collision has occurred,let it be i. Then carry out a sequential search in the order:- i, i+1,i+2..h,1,2...,i-1

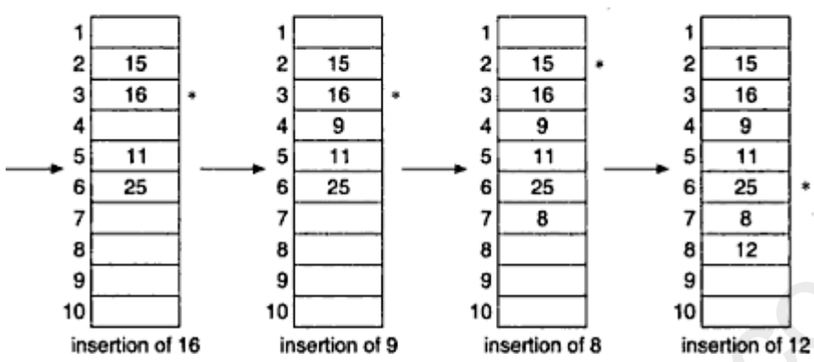
The search will continue until any one of the following occurs

- The key value is found
- An unoccupied location is found
- The searches reaches the location where search had started

The first case corresponds to successful search , and the other two case corresponds to unsuccessful search.Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is alternatively termed as linear probing.

Example- Assume there is a hash table of size 10 and hash function uses the division method of remainder modulo 7, namely H(k)=k(MOD 7)+1.The construction of hash table for the key values 15,11,25,16,9,8,12,8 is illustrated below.





Drawback of closed hashing and its remedies

The major drawback of closed hashing is that as half of the hash table is filled, there is a tendency towards clustering.That is key values are clustered in large groups.and as a result sequential search becomes slower and slower. This kind of clustering is known as primary clustering.

The following are some solutions to avoid this situation

1)Random probing

2)Double hashing

3)Quadratic probing

Random Probing

Instead of using linear probing that generates sequential locations in order, a random location is generated using random probing.

An example of pseudo random number generator that generates such a random sequence is given below:

I=(i+m)MOD h+1

Where m and h are prime numbers. For example if m=5, and h=11 and initially=2 then random probing generates the sequence

8,3,9,4,10,5,11,6,1,7,2

Here all numbers are generated between 1 and 11 in a random order. Primary clustering problem is solved. Where as there is an issue of clustering when two keys are hashed into the same location and then they make use of the same sequence locations generated by the random probing, which is called as secondary clustering

Double Hashing

An alternative approach to solve the problem of secondary clustering is to make use of second hash function in addition to the first one. Suppose H1(k) is initially used hash function and H2(k) is the second one. These two functions are defined as

$$H1(k)=(k \quad MOD \quad h)+1$$

$$H2(k)=(k \text{ MOD } (h-4))+1$$

Let h=11, and k=50 for an instance, then

H1(50)=7 and H2(50)=2.

Now let k=28, then H1(28)=7 and H2(28)=5

Thus for the two key values hashing to the same location, rehashing generates two different locations alleviating the problem of secondary clustering.

Quadratic Probing

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations i+1, i+2..etc are probed.But in quadratic probing next locations to be probed are $i+1^2$,$i+2^2$,$i+3^2$ ..etc . This method substantially reduces primary clustering, but it doesn't probe all the locations in the table.

Open Hashing

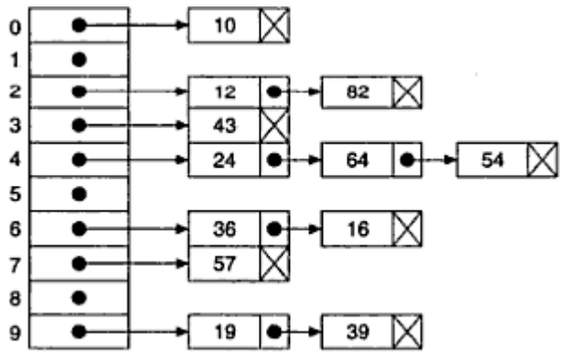
Closed hashing method for collision resolution deals with arrays as hash tables and thus random positions can be quickly refer red. Two main disadvantages of closed hashing are

1) It is very difficult to handle the problem of overflow in a satisfactory manner
2) The key values are haphazardly intermixed and, on the average majority of the key values are from their hash locations increasing the number of probes which degrades the overall performance

To resolve these problems another hashing method called open hashing or separate chaining is used.

The chaining method uses hash table as an array of pointers. Each pointer points a linked list. That is here the hash table is an array of list of headers.

Illustrated below is an example with hash table of size10.



For searching a key in hash table requires the following steps

1)Key is applied to hash function

2) Hash function returns the starting address of a particular linked list(where key may be present)

3)Then key is searched in that linked list

Performance Comparison Expected

Algorithm Name	Best Case	Average Case	Worst Case
Quick Sort	**O(n log n)**	**O(n log n)**	**$O(n^2)$**
Merge Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Heap Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Bubble Sort	**O(n)**	**$O(n^2)$**	**$O(n^2)$**
Selection Sort	**$O(n^2)$**	**$O(n^2)$**	**$O(n^2)$**
Insertion Sort	**O(n)**	**$O(n^2)$**	**$O(n^2)$**
Binary Search	**O(1)**	**O(log n)**	**O(log n)**
Linear Search	**O(1)**	**O(n)**	**O(n)**