DATA STRUCTURES AND ALGORITHMS(CSE2003)

PROJECT TITLE: SORTING ALGORITHMS

MENTOR: NAGARAJU M

MEMBERS:

TANAY NAGARSHETH(17BCE2230)

ARYAN AGARWAL(17BCE0600)

ROHAN GUPTA(17BCE0717)

NALIN PANDOHI(17BCE0966)

❖ INTRODUCTION:

- Algorithms have a vital and key role in solving the computational problems, informally an algorithm is a well-defined computational procedure that takes input and produces output. Algorithm is a tool or a sequence of steps to solve the computational problems . The existence of algorithms goes way back as they were in existence even before the existence of computers.

- There are various methodologies and techniques based on which various different kinds of algorithms are designed. Out of all these problem solving algorithms, let us talk about the sorting algorithms. In case of sorting, it is required to arrange a sequence of numbers into a given order, generally non-decreasing.

- In computer science, an algorithm that puts elements of a list into a certain order is known as sorting algorithm. The orders that are mainly used are numerical order and lexicographical order. For making use of other algorithms (like search and merge algorithms) sorted lists are required to work correctly; it is also often useful for conforming to well-established patterns or rules of data and for producing such output which is easy to read and understand. There are two conditions enlisted that output must satisfy. These conditions are:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order)

2. The output is a permutation or reordering of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004).

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

Sorting Algorithm used in computer science:

Computational complexity (worst, average and best behaviour) of element comparisons in terms of the size of the list. For typical sorting algorithms good behaviour is O(n logn) and bad behaviour is O(n2).

Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only O(1) memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.

Recursion: Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).

Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).

Aims of the Algorithms:

- The algorithm had several aims:
- Speed.
- Good memory utilization. The number of elements that can be sorted should closely approach the physical limits of the machine.
- In order for the algorithm to be truly general purpose the only operator that will be assumed is binary comparison. This rule out methods such as radix sort .
- To obtain good memory utilization when sorting small elements linked lists are avoided. Thus, the lists of elements referred to below are implemented using arrays, without any storage overhead for pointers.

Summary of Popular sorting algorithms:

1. *Bubble Sort:*

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, then it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

This algorithm's average and worst case performance is O(n2), so it is rarely used to sort large, unordered, data sets. This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. Bubble sort can be used to sort a small number of items (where its inefficiency is not a high penalty). Bubble sort may also be efficiently used on a list that is already sorted except for a very small number of elements. For example, if only one element is not in order, bubble sort will take only 2n time. If two elements are not in order, bubble sort will take only at most 3n time. Bubble sort average case and worst case are both $O(n^2)$.

Pros:

- Simplicity and ease of implementation.
- Auxiliary Space used is O (1).

Cons:

- Very inefficient. General complexity is O (n2). Best case complexity is O(n).

CODE:

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in
       place for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */ void
printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above
functions int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11,
    90}; int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array:
    \n"); printArray(arr, n);
    return 0;
}
```

OUTPUT:

```
Sorted array:

11 12 22 25 34 64 90
```

2. *Insertion sort:*

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list.

In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort is a variant of insertion sort that is more efficient for larger lists.

Pros:

- Auxiliary space used is O (1).

Cons:

- General Complexity is O (n2). Best Case is O (n) when the list is already sorted.

CODE:

```
/* Insertion sort ascending order */

#include <stdio.h>

int main()
{
  int n, array[1000], c, d, t;

  printf("Enter number of
  elements\n"); scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++) {
    scanf("%d", &array[c]);
  }

  for (c = 1 ; c <= n - 1; c++)
    { d = c;

    while ( d > 0 && array[d-1] > array[d])
      { t = array[d];
      array[d] = array[d-1];
      array[d-1] = t;

      d--;
    }
  }

  printf("Sorted list in ascending order:\n");

  for (c = 0; c <= n - 1; c++)
    { printf("%d\n", array[c]);
  }

  return 0;
}
```

OUTPUT:

```
Enter number of elements

Enter 7 integers

Sorted list in ascending

order: 4 8 11 15 19 22 35
```

3. *Heap sort:*

Heap sort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree.

Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root.

Using the heap, finding the next largest element takes O (log n) time, instead of O (n) for a linear scan as in simple selection sort. This allows Heap sort to run in O (n log n) time, and this is also the worst case complexity.

Pros:

- Time complexity of the algorithm is O (n log n).
- Auxiliary Space required for the algorithm is O (1).
- In-space and non-recursive makes it a good choice for large data sets.

Cons:

- Works slow than other such DIVIDE-AND-CONQUER sorts that also have the same O (n log n) time complexity due to cache behaviour and other factors.
- Unable to work when dealing with linked lists due to non convertibility of linked lists to heap structure.

2. Quick Sort:

Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivots are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place.

The lesser and greater sub-lists are then recursively sorted. Efficient implementations of quick sort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O (log n) space usage, quick sort is one of the most popular sorting algorithms and is available in many standard programming libraries.

The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower O (n^2) performance, if at each step the median is chosen as the pivot then the algorithm works in O (n log n).

Finding the median however, is an O (n) operation on unsorted lists and therefore exacts its own penalty with sorting.

Pros:

- One advantage of parallel quick sort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.
- All comparisons are being done with a single pivot value, which can be stored in a register.
- The list is being traversed sequentially, which produces very good locality of reference and cache behaviour for arrays.

Cons:

- Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of O (n2) which can prove very fatal for large data sets.

CODE:

```
#include <stdio.h>

void quick_sort(int[],int,int);
int partition(int[],int,int);

int main()
{
   int a[50],n,i;
   printf("How many
   elements?"); scanf("%d",&n);
   printf("\nEnter array elements:");

   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

   quick_sort(a,0,n-1);
   printf("\nArray after sorting:");

   for(i=0;i<n;i++)
      printf("%d ",a[i]);

   return 0;
}

void quick_sort(int a[],int l,int u)
{
   int j;
   if(l<u)
   {
      j=partition(a,l,u);
      quick_sort(a,l,j-1);
      quick_sort(a,j+1,u);
   }
}

int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;

   do
   {
      do
         i++;

      while(a[i]<v&&i<=u);


      do j-
         -;
      while(v<a[j]);

      if(i<j)
```

```
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);

    a[l]=a[j];
    a[j]=v;

    return(j);
}
```
OUTPUT:

```
How many elements: 7

Enter array elements:

Array after sorting:4 8 11 15 19 22 35
```

5. *Merge Sort:*

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second.

It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O (n log n).

Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages Perl, Python (as timsort), and Java (also uses timsort as of JDK7), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.

Pros:

- Marginally faster than the heap sort for larger sets.
- Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Cons:

- At least twice the memory requirements of the other sorts because it is recursive. This is the BIGGEST cause for concern as its space complexity **is very high. It requires about a Θ (n) auxiliary space for its working.**
- Function overhead calls (2n-1) are much more than those for quick sort (n). This causes it to take more time marginally to sort the input data.

CODE:

```
     #include<stdio.h>

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
    int a[30],n,i;
    printf("Enter no of
    elements:"); scanf("%d",&n);
    printf("Enter array elements:");

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    mergesort(a,0,n-1);

    printf("\nSorted array is
    :"); for(i=0;i<n;i++)
        printf("%d ",a[i]);

    return 0;
}

void mergesort(int a[],int i,int j)
{
    int mid;

    if(i<j)
    {
        mid=(i+j)/2;
        mergesort(a,i,mid);        //left recursion
        mergesort(a,mid+1,j);    //right recursion
        merge(a,i,mid,mid+1,j);    //merging of two sorted sub-arrays
    }
}

void merge(int a[],int i1,int j1,int i2,int j2)
{
    int temp[50]; //array used for merging
    int i,j,k;
    i=i1;    //beginning of the first list
    j=i2;    //beginning of the second list
    k=0;

    while(i<=j1 && j<=j2)    //while elements in both lists
    {
        if(a[i]<a[j])
            temp[k++]=a[i++];
        else
            temp[k++]=a[j++];
    }

    while(i<=j1)    //copy remaining elements of the first list
       temp[k++]=a[i++];

    while(j<=j2)    //copy remaining elements of the second list
       temp[k++]=a[j++];

    //Transfer elements from temp[] back to a[]
    for(i=i1,j=0;i<=j2;i++,j++)
       a[i]=temp[j];
}
```

OUTPUT:

```
Enter no of elements:7

Enter array elements:

Sorted array is :4 8 11 15 19 22 35
```

6. *Counting Sort:*

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in O (|S| + n) time and O(|S|) memory where n is the length of the input. It works by creating an integer array of size |S| and using the ith bin to count the occurrences of the ith member of S in the input.

Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order.

This sorting algorithm cannot often be used because S needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behaviour as n increases. It also can be modified to provide stable behaviour.

Pros:

- The algorithm has a time complexity of O (n+m), where n is the number of data while m is the range of the data, which implies that the most efficient use will be when m<<n. In that case the time complexity will turn out to be linear.
- This sort works optimally in the case when the data is uniformly distributed.

Cons:

- If the range m>>n, the complexity will not be linear in n and thus this sort does not remain useful anymore. This is because chances of introduction of gaps, that is counters for those elements which do not exist in the list, will cause a higher space complexity.

CODE:

```
  #include <stdio.h>

/* Counting sort function  */
void counting_sort(int A[], int k, int n)
{
    int i, j;
    int B[15], C[100];
    for (i = 0; i <= k; i++)
        C[i] = 0;
     for (j = 1; j <= n; j++)
         C[A[j]] = C[A[j]] + 1;
     for (i = 1; i <= k; i++)
         C[i] = C[i] + C[i-1];
     for (j = n; j >= 1; j--)
     {
```

```c
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
    printf("The Sorted array is : ");
    for (i = 1; i <= n; i++)
        printf("%d ", B[i]);
}
/*  End of counting_sort()  */

/*  The main() begins  */
int main()
{
    int n, k = 0, A[15], i;
    printf("Enter the number of input : ");
    scanf("%d", &n);
    printf("\nEnter the elements to be sorted :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &A[i]);
        if (A[i] > k) {
            k = A[i];
        }
    }
    counting_sort(A, k, n);
    printf("\n");
    return 0;
}
```

OUTPUT:

```
Enter the number of input : 7

Enter the elements to be sorted :

The Sorted array is : 4 8 11 15 19 22 35
```

Divide and Conquer sorting algorithms:

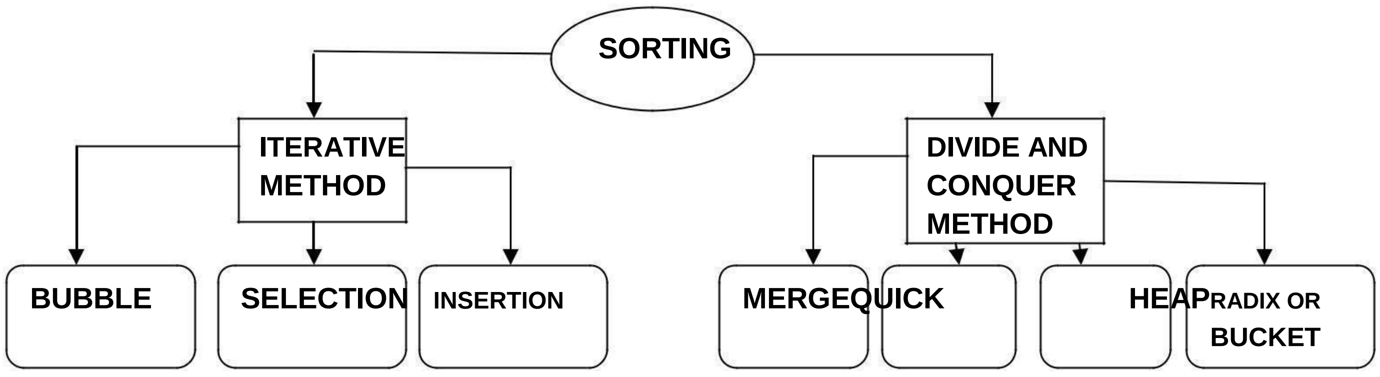
1. Introduction:

Researches in various architectural and database management systems have given rise to various sorting algorithms among which Divide and Conquer techniques holds the most promise in terms of both Space complexity as well as Time complexity .

Sorting can be done in iterative as well as recursive way. The iterative approach makes repeated passes over some portions of the list of elements being sorted which in the worst case may lead to scanning almost the entire list of n elements. Therefore, the worst case time complexity is slightly high, usually O(n2). However the memory used (Space complexity) does not require additional overheads in terms of Stack management.

The recursive methods, on the other hand, are mainly based on Inductive steps which can easily be formulated in terms of Divide and Conquer approach. These methods, compared to the iterative sorting algorithms, though elegant in terms of reduced number of steps needed to sort the list of n elements, may put additional burden in terms overhead required to maintain and manage the Stack. The Divide and Conquer approach has been implemented in many conventional sorting algorithms like Merge Sort, Quick Sort, Heap Sort, Radix Sort, etc. They usually follow a time complexity of O(n log2 n).

Apart from these, many researchers have studied and formulated new algorithms, based on this approach, which has applications in diverse fields.



Comparison Sort	Time Complexity					
	BEST	*WORST*	*AVG.*	*SPACE*	*STABLE*	*COMMENTS*
Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	YES	For each pair of indices, swap the elements if they are out of order.
Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	YES	At each pass check if the array is already sorted. Best case – array already sorted.
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	YES	Swap happens only once in a single pass.
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	YES	Very small constant factor even if the complexity O(n^2). Best case – array already sorted. Worst case – sorted in reverse order.
Quick sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	YES	Best case – when pivot divides in 2 equal halves. Worst case – array already sorted
Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	YES	Pivot chosen randomly
Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	YES	Best to sort Linked list. Best for very large no. of elements which can't fit in memory.
Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	NO	
Non-Comparison Sort						
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k^2)	YES	K= range of numbers in the list.
Radix Sort	O(n.k/s)	O(2^s.n.k/s)	O(n.k/s)	O(n)	NO	
Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	YES	

CONCLUSION :

Sorting has many real time applications. It helps tatistical record management so that a quick search can be performed for accessing any particular data from the record. The various conventional algorithms allow this quick access but fails in case of large lists. The usual time complexities of such algorithms is O(n). Hence, in case of large array, the numbers of steps for sorting is very large and the complexity thus increases. The Divide and Conquer sorting approaches solve this problem as they follow an iterative as well as recursive path while handling the data.

They usually sort the data with a complexity of O(n log2n) which is very less compared to the traditional iterative methods. In this brief, we have discussed the conventional as well as newly proposed sorting algorithms based on Divide and Conquer.

From researches it is found that the RNS decoding algorithm is highly efficient among other algorithms. It requires less space in RAM and has a time complexity of O (n). The merge join parallel sorting algorithm is a new concept but it can work up to 128 processors. Yet it requires fewer steps and has a good load balancing factor. The Divide and Conquer sorting algorithms has a wide range of applications in DBMS, in various record keepings like telephone directories, dictionaries, bank directories, etc.