Data Structures and Algorithms (CS-2001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

4 Credit

Lecture Note

Chapter Contents

Sr #	Major and Detailed Coverage Area	Hrs
7	**Sorting**	4
	Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort	

Introduction

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:

- **Telephone Directory –** The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary –** The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Important

- **Increasing order –** A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.
- **Decreasing Order –** A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.
- **Non-increasing order –** A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.
- **Non-decreasing Order –** A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Introduction cont...

In-place Sorting and Not-in-place

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting. However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Sorting

Sr#	Traditional Sorting	Sorting based on Divide & Conquer	Sorting based on Trees
1	Bubble Sort	Merge sort	Heap Sort
2	Insertion sort	Quick sort	
3	Selection sort		
4	Radix Sort		

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity is of $\mathbf{O(n^2)}$ where n are no. of items.

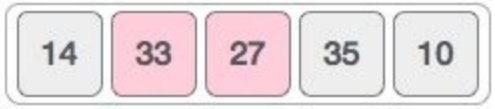
How bubble sort

Take an unsorted array

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, 33 is greater than 14, so it is already in sorted locations. Next, it compares 27 with 33.

We find that 27 is smaller than 33 and these two values must be swapped.



Post to swapping, the new array should looks like

Bubble Sort cont...

Next it compares 35 with 33 and both are already in sorted positions.

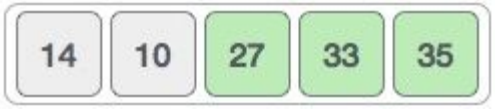
Then we move to next two values, 35 and 10.

We know than 10 is smaller 35. Hence they are not sorted.

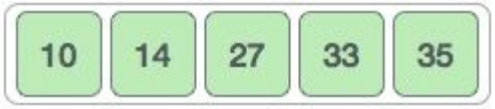
We swap these values. We find that we reach at the end of the array. After one iteration the array should look like this

To be precise, we are now showing that how array should look like after each iteration. After second iteration, it should look like this

Notice that after each iteration, at least one value moves at the end



When there's no swap required, bubble sorts learns that array is completely sorted.



Bubble Sort cont...

Algorith

Suppose LA is an array with n integer elements and swap function is to exchange the values for the array elements. Below is the Bubble Sort algorithm to arrange the elements in ascending order

1. Start
2. Set i=0
3. Set j = 0
4. Repeat steps 5 till 9 while i < n
5. Set j = 0
6. Repeat steps 7 till 8 while j < n – i - 1
7. If (LA[j] > LA[j+1]) then swap (LA[j], LA[j+1])
8. Set j = j + 1
9. Set i = i + 1
10. Stop

C Code

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j]; a[j] = a[j+1]; a[j+1] = temp;
    }
  }
}
```

Bubble Sort cont...

Time

n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

(n-1)+(n-2)+(n-3)+.....+3+2+1
Sum = n * (n-1)/2
i.e $O(n^2)$

Hence the time complexity of Bubble Sort is

- **$O(n^2)$ in worst case**
- **$\theta(n^2)$ in average case**
- **$\Omega(n)$ in best case**

Space

Space complexity is **O(1)** since only single additional memory space is required for temp variable

Animatio

https://cathyatseneca.github.io/DSAnim/web/bubble.html

Insertion Sort

Insertion sort is **in-place** comparison based sorting algorithm. A sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be inserted in the sorted sub-list, has to find its appropriate place and insert it there. Hence the name insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list **(in the same array)**. This algorithm is not suitable for large data sets as its average and worst case complexity are of **$O(n^2)$** where n are no. of items.

How Insertion sort

Take an unsorted array for the example

Insertion sort compares the first two elements

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort cont...

Insertion sort moves ahead and compares 33 with 27

And finds that 33 is not in correct position

It swaps 33 with 27. Also it checks with all the elements of sorted sublist. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping.

By now 14 and 27 in the sorted sub list. Next it compares 33 with 10.

These values are not in sorted order

So swap them

Insertion Sort cont...

But swapping makes 27 and 10 unsorted

So we swap them too

Again we find 14 and 10 in unsorted order

And we swap them. By the end of third iteration we have a sorted sub list of 4 items.

This process goes until all the unsorted values are covered in sorted sub list.

Insertion Sort cont...

Algorith

Suppose LA is an array with n integer elements and below is the Insertion Sort algorithm to arrange the elements in ascending order

1. Start
2. Set i=1, j = 0, key = 0
4. Repeat steps 5 till 10 while i < n
5. Set key = LA[i]
6. Set j = i - 1
7. While j >= 0 AND key < LA[j] repeat 8 and 9
8. LA[j + 1] = LA [j]
9. Set j = j – 1
10. LA[j + 1] = key
11. Stop

C Code

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
  key = a[i];
  j = i-1;
  while(j>=0 && key < a[j])
  {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

Insertion Sort cont…

Time

Time complexity of Insertion Sort is

- **$O(n^2)$ in worst case**
- **$\Omega(n)$ in best case**
- **$\theta(n^2)$ in average case**

Space

Space complexity is **$O(1)$**

Animatio

https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort/insertioncardsort.swf

Selection Sort

Selection sort is **a in-place** comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexity are of $\mathbf{O(n^2)}$ where n are no. of items.

How Selection sort

Take an unsorted array for the example

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort cont...

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of sorted list

For the second position, where 33 is residing, we start scanning the rest of the list in linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in the sorted manner.

Selection Sort cont...

The same process is applied on the rest of the items in the array and pictorial depiction of entire sorting process is

Selection Sort cont...

C Code Snippet

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, min, temp;
for(i=0; i<6; i++)
{
 min= i;
 for(j=i+1; i<6; j++)
 {
    if (a[j]  < a[min])
    {
       min = j;
    }
 }
 temp = a[i];
 a[i] = a[min];
 a[min] = temp;
}
```

Pseudo code

```
procedure selection sort
  list  : array of items
  n     : size of list

  for i = 1 to n - 1
    min = i /* set current element as minimum*/
    for j = i+1 to n   /* check the element to be minimum */
      if list[j] < list[min] then
        min = j;
      end if
    end for
    /* swap the minimum element with the current element*/
    if indexMin != i  then
      swap list[min] and list[i]
    end if
  end for
end procedure
```

Selection Sort cont...

Time

Time complexity of Selection Sort is

- **$O(n^2)$ in worst case**
- **$\Omega(n)$ in best case**
- **$\theta(n^2)$ in average case**

Space Complexity

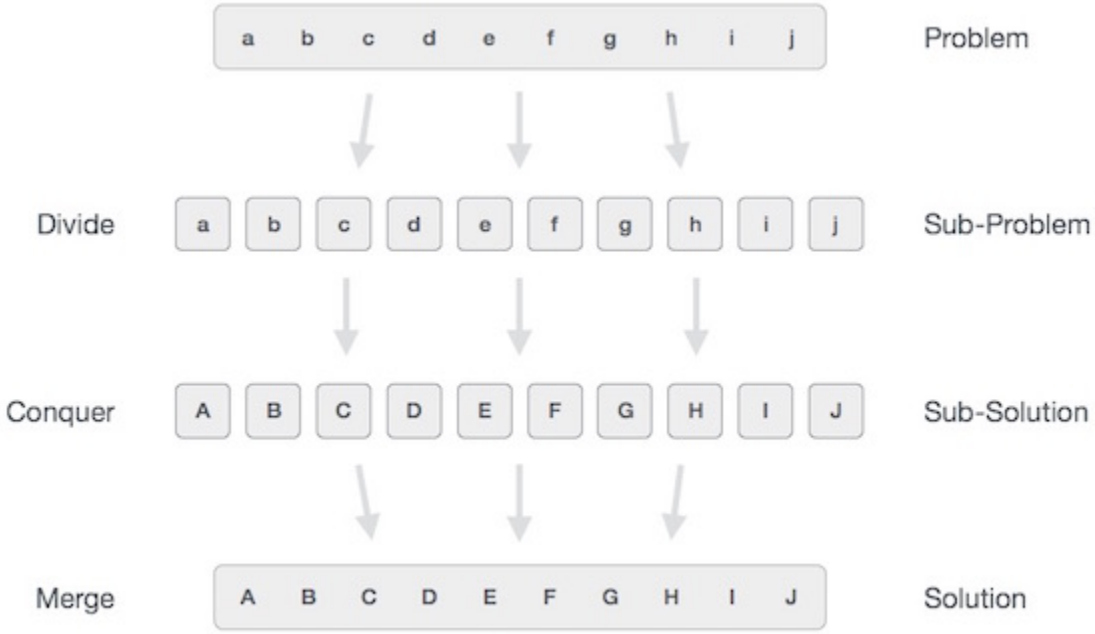
Space complexity is **$O(1)$**

Animatio

http://www.algostructure.com/sorting/selectionsort.php

Divide & Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more dividation is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.



Divide & Conquer cont...

Broadly, we can understand divide-and-conquer approach as three step process.

- **Divide/Break:** This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.
- **Conquer/Solve:** This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.
- **Merge/Combine:** When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

The following computer algorithms are based on divide-and-conquer programming approach

- Merge Sort
- Quick Sort

This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Quick Sort

Quick sort is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say **pivot** based on which the partition is made and another array holds values greater than pivot value. The quick sort partitions an array and then calls itself recursively twice to sort the resulting two sub-array.

How Quick Sort works?

Quick Sort Algorithm

Step 1 – Start
Step 2 – Make the right-most index value pivot
Step 3 – partition the array using pivot value
Step 4 – quicksort left partition recursively
Step 5 – quicksort right partition recursively
Step 6 – Stop

Quick Sort Pseudo code

```
procedure quickSort(left, right)
   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if
end procedure
```

Quick Sort cont...

Quick sort Pivot Algorithm

Step 1 – Choose the highest index value has pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – left points to the low index

Step 4 – right points to the high

Step 5 – while value at left is less than pivot move right

Step 6 – while value at right is greater than pivot move left

Step 7 – if both step 5 and step 6 does not match swap left and right

Step 8 – if left ≥ right, we stop. The point where they met is new pivot. swap the pivot value with the split point.

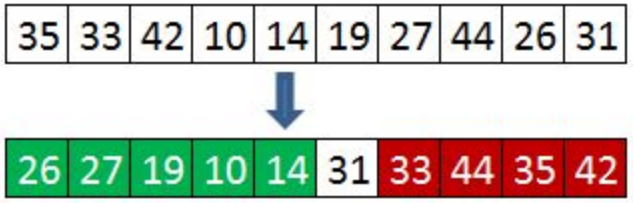
Quick sort Pivot Pseudo code

```
function partitionFunc(left, right, pivot)
  leftPointer = left -1
  rightPointer = right

  while true do
    while A[++leftPointer] < pivot do
      //do-nothing
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
      //do-nothing
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if

  end while

  swap leftPointer,right
  return leftPointer
end function
```

Quick Sort

Quick sort is a **divide and conquer** algorithm. Its divided large list in mainly three parts:

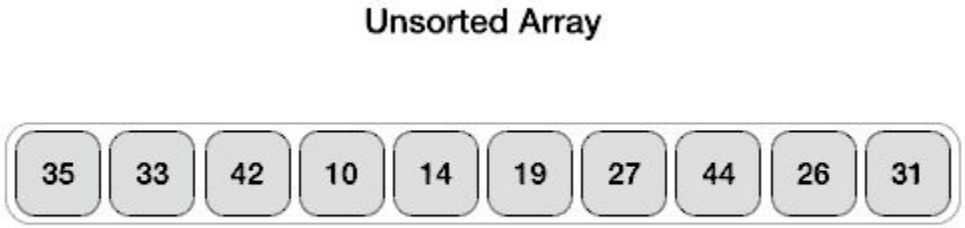
1. Elements less than (i.e. LT) pivot element.
2. Pivot element.
3. Elements greater than (i.e. GT) pivot element.



Post to the Partition, Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Elements in Green represents LT pivot element
Elements in Red represents GT pivot element
Element with no color represents pivot element

Partition in Quick



Quick Sort Code Snippet & Complexity Analysis


```
int a[6] = {5, 1, 6, 2, 4, 3};
int partition(int left, int right, int pivot)
{
  int leftPointer = left;
  int rightPointer = right-1;
  while(true)
  {
    while(a[leftPointer++] < pivot)
    {
      //do nothing
    }
    while(rightPointer > 0 && a[rightPointer--] > pivot)
    {
      //do nothing
    }
    if(leftPointer >= rightPointer)
    {
      break;
    }
    else
    {
      swap(leftPointer, rightPointer);
    }
  }
  swap(leftPointer, right);
  return leftPointer;
}
```

```
//continuation of program
void quickSort(int left, int right)
{
  if(right-left <= 0)
  {
    return;
  }
  else
  {
    int pivot = a[right];
    int partitionPoint = partition(left, right, pivot);
    quickSort(left, partitionPoint-1);
    quickSort(partitionPoint+1, right);
  }
}
void swap(int num1, int num2)
{
  int temp = a[num1];
  a[num1] = a[num2];
  a[num2] = temp;
}
int main()
{
  quickSort(0,5);
  return 0;
}
```

Time

Time complexity of Selection Sort is

- **$O(n^2)$ in worst case**
- **$\Omega(n \log n)$ in best case**
- **$\theta(n \log n)$ in average case**

Space Complexity

Space complexity is **O(log n)**

Merge Sort

Merge sort is a sorting technique based on **divide and conquer** technique. With worst-case time complexity being **O(n log n)**, it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Quick sort

Take an unsorted array

Merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. Here the array of 8 items is divided into two arrays of size 4.

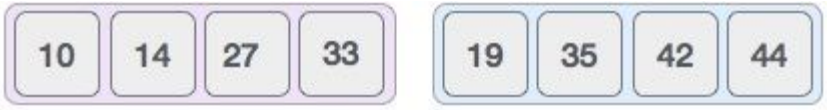
This does not change the sequence of appearance of items in the original. Now these two arrays are divided into halves.

Merge Sort cont...

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly same manner they were broken down. Please note the color codes given to these lists. We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.

In next iteration of combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in sorted order.



After final merging, the list should look like this

Merge Sort cont...

Algorithm

Step 1 - Start

Step 2 – if it is only one element in the list it is already sorted, return.

Step 3 – divide the list recursively into two halves until it can no more be divided.

Step 4 – merge the smaller lists into new list in sorted order.

Step 5 - Stop

Time Complexity

- **O(n log n) in worst case**
- **Ω(n log n) in best case**
- **θ(n log n) in average case**

Space Complexity

Space complexity is **O(n)**

Pseudo code

```
procedure mergesort( var a as array )
  n = length of a
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure
```

```
procedure merge( var a as array, var b as array )
  var c as array

  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c
end procedure
```

Merge Sort C Code


```
#define max 10

int a[10] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44 };
int b[10];

void merging(int low, int mid, int high)
{
  int l1, l2, i;

 for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
 {
    if(a[l1] <= a[l2])
      b[i] = a[l1++];
    else
      b[i] = a[l2++];
  }

  while(l1 <= mid)
    b[i++] = a[l1++];

  while(l2 <= high)
    b[i++] = a[l2++];

  for(i = low; i <= high; i++)
    a[i] = b[i];
}
```

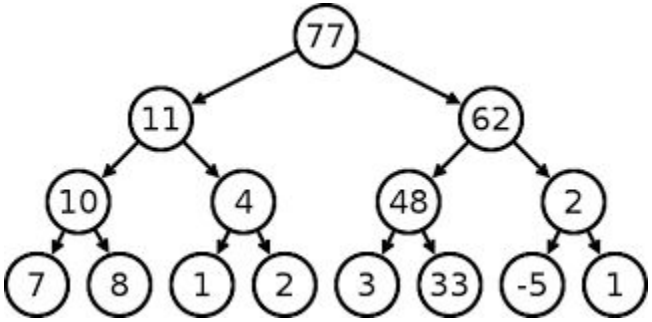
```
void sort(int low, int high)
{
  int mid;

  if(low < high)
  {
    mid = (low + high) / 2;
    sort(low, mid);
    sort(mid+1, high);
    merging(low, mid, high);
  }
  else
  {
    return;
  }
}
```

```
int main()
{
  int i;

  printf("List before sorting\n");

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

  sort(0, max-1);

  printf("\nList after sorting\n");

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

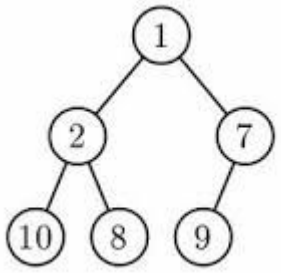
Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. **Complete Binary Tree:** A binary tree T with n levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side. A **Binary Heap:** It is a *Complete Binary Tree* where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as **max heap** and the latter is called **min heap**.

Max Heap



Min Heap



Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

How Heap Sort Works?

- In this algorithm we first build a heap using the given elements
- If we want to sort the elements in **ascending** order, we create a **Min Heap**
- If we want to sort the elements in **descending** order, we create a **Max Heap**
- Once the heap is created, we delete the root node from the heap and put the last node in the root position and repeat the process till we have covered all the elements

Pointes to Remember for ascending order

1. Build Heap
2. Transform the heap into Min Heap
3. Delete the root node
4. Put the last node of the heap in root position
5. Repeat from step 2 till all nodes are covered

Pointes to Remember for descending order

1. Build Heap
2. Transform the heap into Max Heap
3. Delete the root node
4. Put the last node of the heap in root position
5. Repeat from step 2 till all nodes are covered

Heap Sort Example

Heap Sort Code Snippet & Complexity Analysis

Time Complexity

Time complexity of Selection Sort is

- **O(n log n) in worst case**
- **Ω(n log n) in best case**
- **θ(n log n) in average case**

Space Complexity

Space complexity is **O(1)**

C Code

Max Heap

Radix Sort

Radix Sort is a clever and intuitive little sorting algorithm. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Example:

Original, unsorted list: 170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: 17**0**, 9**0**, 80**2**, **2**, 2**4**, 4**5**, 7**5**, 6**6**

Sorting by next digit (10s place) gives: 8**0**2, 2, **2**4, **4**5, **6**6, 1**7**0, **7**5, **9**0

Sorting by most significant digit (100s place) gives: 2, 24, 45, 66, 75, 90, **1**70, **8**02

Time Complexity

Time complexity of Radix Sort is

- **O(w*n) in worst case**
- **Ω(w*n) in best case**
- **θ(w*n) in average case**

Note: **w** is the word size and n is the number of elements

Space Complexity

Space complexity is **O(w+n)**

Radix Sort C Code

Class Work

Sr #	Algorithm	Time Complexity			Space Complexity
		Best	Average	Worst	
1	Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
2	Insertion Sort				
3	Selection Sort				
4	Quick Sort				
5	Merge Sort				
6	Heap Sort				
7	Radix Sort				

Assignments

1. Let A[1], A[2], . . . , A[n] be an array containing n very large positive integers. Describe an efficient algorithm to find the minimum positive difference between any two integers in the array. What is the complexity of your algorithm? Explain.

2. Design an efficient algorithm to sort 5 distinct keys.

3. Let A = A[1], . . . , A[n] be an array of n distinct positive integers. An inversion is a pair of indices i and j such that i < j but A[i] > A [j]. For example in the array [30000, 80000, 20000, 40000, 10000], the pair i = 1 and j = 3 is an inversion because A [1] = 30000 is greater than A[3] = 20000. On the other hand, the pair i = 1 and j = 2 is not an inversion because A [1] = 30000 is smaller than A [2] = 80000. In this array there are 7 inversions and 3 non-inversions. Describe an efficient algorithm that counts the number of inversions in any array. What is the running time of your algorithm?


THANK
YOU!

Home Work

Experiment with the C implementations of the various sorting algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Radix Sort

Supplementary Reading

- http://www.geeksforgeeks.org/sorting-algorithms/
- https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
- http://www.studytonight.com/data-structures/introduction-to-sorting
- http://nptel.ac.in/courses/Webcourse-contents/IIT-%20Guwahati/data_str_algo/Module_5/binder1.pdf
- http://nptel.ac.in/courses/106105164/

FAQ

- **Bubble Sort :** Exchange two adjacent elements if they are out of order. Repeat until array is sorted.
- **Insertion sort :** Scan successive elements for an out-of-order item, then insert the item in the proper place.
- **Selection sort :** Find the smallest element in the array, and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted.
- **Quick sort :** Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively.
- **Merge sort :** Divide the list of elements in two parts, sort the two parts individually and then merge it.
- **Heap sort :** The first maximum element is searched and place it at the end and the process is repeated for remaining elements.
- **Radix sort :** Sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.