Algorithms in C

Agenda

- Algorithms
- Recursion
- Searching Algorithms
- Sorting Algorithms

Algorithms

What is an Algorithm?

- It is a step by step approach to solve a particular problem.
- It gives a crisp solution.
- It can be understood by non-technical people as well.
- It’s free of programming language constructs.
- There can be multiple possible algorithm to solve a given problem.

What is an Algorithmic Analysis?

- It is a step by step approach to solve a particular problem.
- It gives a crisp solution.
- It helps in the identification of optimal algorithm
- It helps in understanding the trade-off of time and space requirement.

Algorithmic Analysis- Types

- There are different types of algorithmic analysis
  - Best case
  - Average case
  - Worst case

- Worst case is computed always for the analysis as it dictates maximum resource requirements.

Time Complexity

- It determines the total number of unit operations to be undertaken to solve a particular problem.
- Unit operation is an operation that is independent and can’t be broken down in simpler operation.
- It is independent of architecture.
- It is computed on the basis of algorithm itself.
- It’s a high priority criteria in optimal algorithm selection.

Time Complexity- Example 1

Time Complexity- Example 2

Space Complexity

- It determines the total space to be allocated in order to solve a particular problem.
- It is the extra memory that an algorithm needs for its implementation.
- It involves the memory of computers.
- It’s a low priority criteria in optimal algorithm selection.

Space Complexity- Example 1

Space Complexity- Example 2

Recursion

What is Recursion?

- Recursion is about function calling itself.
- It comes intuitive when a function is breakable into subproblems.
- There are many data structures which are recursive in nature.

Steps of recursion

- Base condition
- Logic
- Recursive call

Recursion vs Iteration

- Recursion is about function calling itself whereas iteration can't call itself.
- There is always some space complexity associated with recursion.
- Recursion implicitly works with stack data structure. There is no such requirement in iteration.
- Recursion is slower than iteration.

Tail Recursion

- Here the recursive call is the last step of implementation.
- This is a faster approach of recursion.
- No activation record maintenance is required here.

Tail Recursion- Example

Non-Tail Recursion

- Here the recursive call is the not the last step of implementation.
- This is a slower approach of recursion.
- Activation record maintenance is required here.

Non-Tail Recursion- Example

Direct Recursion

- When call within function call is made on the same function.

- ```
  f(){
      f()
  }
  ```

- Only one function is involved here.
  ```

Direct Recursion- Example

Indirect Recursion

- When call within function call is made on the different function.

- ```
  f1(){
          f2()
  }

  f2(){
          f1()
  }
  ```

- More than one functions are involved here.
  ```

Indirect Recursion- Example

Recursion- Time Complexity

Recursion- Space Complexity

Binary Search Algorithm

What is Binary Search?

- Binary Search is one of the searching techniques.
- It can be used on sorted array.
- This searching technique follows the divide and conquer strategy and search space always reduces to half in every iteration.
- This is a very efficient technique for searching but it needs some order on which partition of the array will occur.

Binary Search - Iterative Algorithm

```
binarySearch(arr, size)
  loop until beg is not equal to end
      midIndex = (beg + end)/2
          if (item == arr[midIndex] )
              return midIndex
          else if (item > arr[midIndex] )
              beg = midIndex + 1
          else
              end = midIndex - 1
```

Binary Search - Recursive Algorithm

```
binarySearch(arr, item, beg, end)
        if beg<=end
          midIndex = (beg + end) / 2
          if item == arr[midIndex]
            return midIndex
          else if item < arr[midIndex]
            return binarySearch(arr, item, midIndex + 1, end)
          else
            return binarySearch(arr, item, beg, midIndex - 1)
        return -1
```

Binary Search - Demonstration

Item to be searched=20

input:

0	1	2	3	4
10	11	16	20	23

beg=0, end=4, mid=2

0	1	**2**	3	4
10	11	**16**	20	23

beg=3, end=4, mid=3

0	1	2	**3**	4
10	11	16	**20**	23

Element found at index 3, Hence 3 will get returned

Binary Search – Implementation 1

```
int binarySearch(int arr[], int item, int beg, int end) {
    while (beg <= end) {
      int midIndex = beg + (end - beg) / 2;
      if (arr[midIndex] == item)
        return midIndex;
      if (arr[midIndex] > item)
        beg = midIndex + 1;
      else
        end = midIndex - 1;
    }
    return -1;
  }
```

Binary Search – Implementation 2

```
int binarySearch(int arr[], int item, int beg, int end) {
    if (end >= beg) {
        int midIndex = beg + (end - beg) / 2;
            if (arr[midIndex] == item)
                return midIndex;
            if (arr[midIndex] < item)
                return binarySearch(arr, item, beg, midIndex - 1);
            return binarySearch(arr, item, midIndex + 1, end);
    }
    return -1;
}
```

Selection Sort Algorithm

What is Selection Sort?

- It is a simple sort algorithm that revolves around the ‘comparison’.
- In each iteration, one element gets placed.
- We choose the minimum element in the array and place is at the beginning of the array by swapping with the front element.
- We can also do this by choosing maximum element and placing it at the rear end.
- Selection sort basically selects an element in every iteration and place it at the appropriate position.

Selection Sort - Algorithm

```
selectionSort(arr, n)
    iterate (n - 1) times
    set the first unsorted element index as the min
        for each of the unsorted elements
            if element < currentMin
                set element's index as new min
        swap element at min with first unsorted position
end selectionSort
```

Selection Sort - Demonstration

Input:

0	1	2	3	4
23	10	16	11	20

First step - marking of sorted part

0	1	2	3	4
10	23	16	11	20

After i=1

0	1	2	3	4
10	11	16	23	20

Selection Sort - Demonstration Cont.

After i=2

0	1	2	3	4
10	11	16	23	20

After i=3

0	1	2	3	4
10	11	16	20	23

After i=4, no iteration is required as the last element is already sorted

0	1	2	3	4
10	11	16	20	23

Selection Sort - Implementation

```
void selectionSort(int arr[], int size) {
    for (int j = 0; j < size - 1; j++) {
    int min = j;
    for (int i = j + 1; i < size; i++) {
        if(arr[i]<arr[min])
            min = i;
    }
    int tmp = arr[j];
    arr[j] = arr[min];
    arr[min] = tmp;
  }
}
```

Insertion Sort Algorithm

What is Insertion Sort?

- It is one of the easiest and brute force sorting algorithms
- Insertion sort is used to sort elements in either ascending or descending order
- In insertion sort, we maintain a sorted part and unsorted part
- It works just like playing cards i.e picking one card and sorting it with the cards that we have in our hand already which in turn are sorted
- With every iteration, one item from unsorted is moved to the sorted part
- First element is picked and considered as sorted
- Then we start picking from 2nd elements onwards and start comparison with elements in sorted part.
- We shift the elements from sorted by one element until an appropriate location is not found for the picked element
- This continues till all the elements get exhausted

Insertion Sort - Algorithm

```
insertionSort(arr, size)
    consider 1st element as sorted part
        for each element from i=2 to n-1
            tmp = arr[i]
            for j=i-1 to 0
                If a[j]>tmp
                 Then right shift it by one position
            put tmp at current j
```

Insertion Sort - Demonstration

input:

0	1	2	3	4
23	10	16	11	20

First step - marking of sorted part

0	1	2	3	4
23	10	16	11	20

After i=1

0	1	2	3	4
10	23	16	11	20

After i=2

0	1	2	3	4
10	16	23	11	20

Insertion Sort - Demonstration Cont.

After i=3

0	1	2	3	4
10	11	16	23	20

After i=4

0	1	2	3	4
10	11	16	20	23

Insertion Sort - Implementation

```
void insertionSort(int arr[], int size) {
  for (int i = 1; i < size; i++) {
    int tmp = arr[i];
    int j = i - 1;
    while (j >= 0 && tmp < arr[j]) {
      arr[j + 1] = arr[j];
      --j;
      }
    arr[j + 1] = tmp;
   }
}
```

Quick Sort Algorithm

What is Quick Sort?

- It is one of the most widely used sorting algorithm
- It follows divide and conquer paradigm
- Recursion is used in quicksort implementation
- In each recursive call, a pivot is chosen then the array is partitioned in such a way that all the elements less than pivot lie to the left and all the elements greater than pivot lie to the right
- After every call, the chosen pivot occupies its correct position in the array which it is supposed to as in sorted array
- So with each step, our problem gets reduced by 2 which leads to quick sorting
- Pivot can be an element. Example: last element of current array, first element of current array, random pivot, etc.

Quick Sort - Algorithm

```
quickSort(arr, beg, end)
        if (beg < end)
          pivotIndex = partition(arr,beg, end)
          quickSort(arr, beg, pivotIndex -1)
          quickSort(arr, pivotIndex + 1, end)
```

Partition - Algorithm

```
partition(arr, beg, end)
    set end as pivotIndex
    pIndex = beg - 1
    for i = beg to end-1
      if arr[i] < pivot
       swap arr[i] and arr[pIndex]
       pIndex++
   swap pivot and arr[pIndex+1]
       return pIndex + 1
```

Quick Sort - Demonstration

Input:

0	1	2	3	4
5	10	9	6	7

beg pivot, end

Step 1

0	1	2	3	4
5	6	7	10	9

beg pivot, end beg pivot, end

Step 2

0	1	2	3	4
5	6	7	9	10

beg,pivot,end beg,pivot, end

Quick Sort - Implementation

```
void quickSort(int[] a,int p,int r)
    {
        if(p<r)
         {
            int q=partition(a,p,r);
            quickSort(a,p,q-1);
            quickSort(a,q+1,r);
         }
    }
```

Quick Sort – Implementation Cont.

```
int partition_function(int arr[], int l, int h){
 int pivot = arr[h];   // pivot is the last element
 int pIndex = (l - 1); // Index of smaller element
 for (int j = l; j <= h- 1; j++){
         if (arr[j] < p){
                 i++;
                 swap_elements(&arr[i], &arr[j]);
         }
 }
 swap_elements(&arr[i + 1], &arr[h]);
 return (i + 1);
}
```

Merge Sort Algorithm

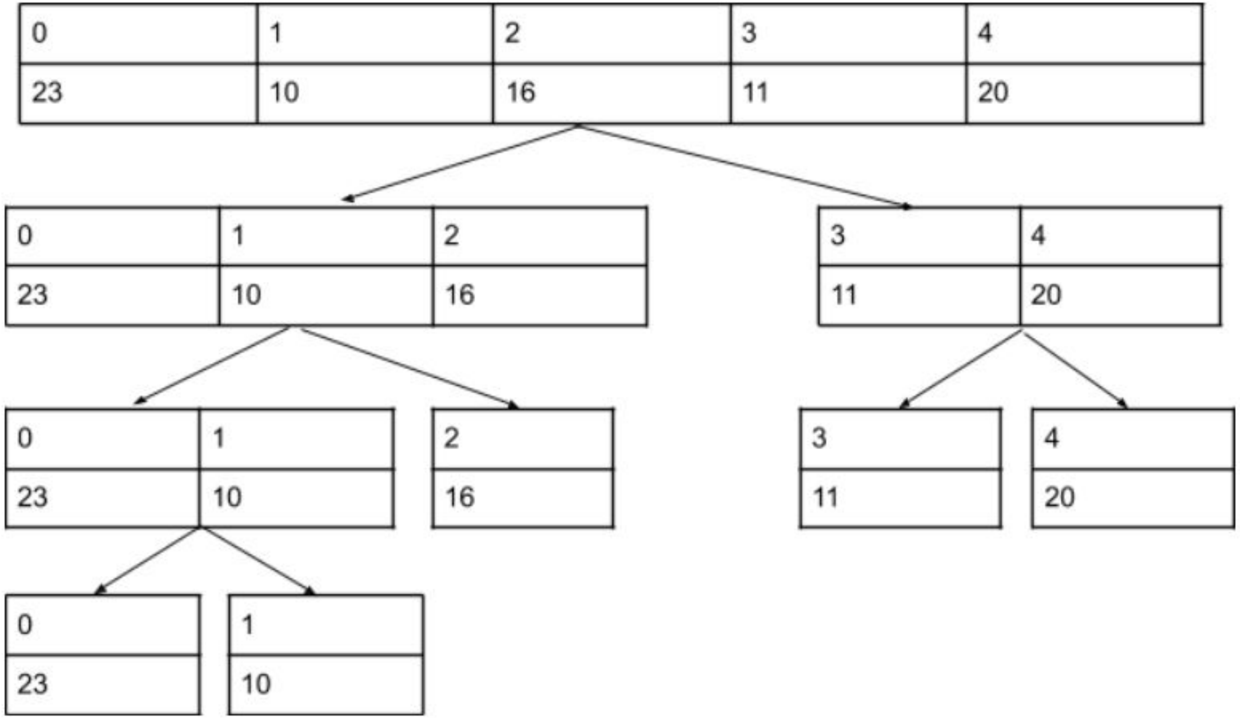
What is Merge Sort?

- In merge sort, the problem is divided into two sub problems in every iteration
- Hence efficiency is increased drastically
- It follows divide and conquer approach
- Divide break the problem into 2 sub problem which continues until problem set is left with one element only
- Conquer basically merges the 2 sorted array into the original array

Merge Sort - Algorithm

```
mergeSort(arr, left, right)
    if left > right
        return
    mid = (left+right)/2
    mergeSort(arr, left, mid)
    mergeSort(arr, mid+1, right)
    merge(arr, left, mid, right)
end
```

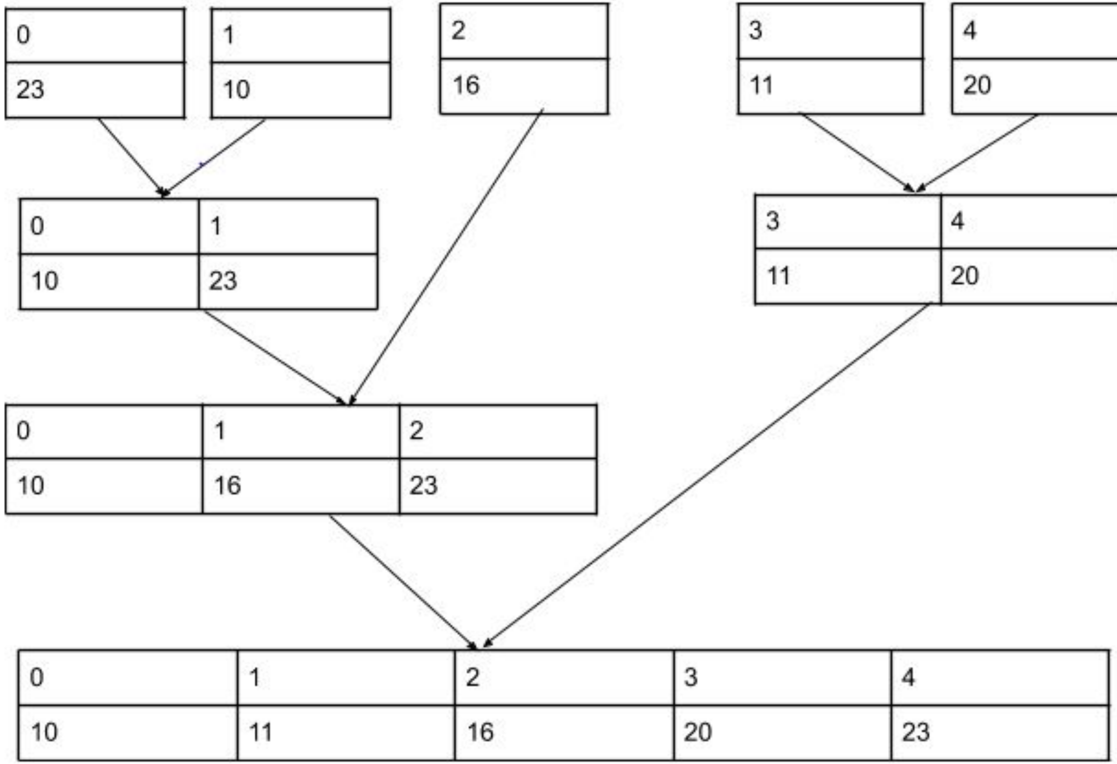
Merge Sort - Demonstration



Merge - Algorithm

- Create 2 subarrays Left and Right
- Create 3 iterators i, j and k
- Insert elements in Left and Right (i & j)
- k - Replace the values in the original array
- Pick the larger elements from Left and Right & place them in the correct position
- If there are no elements in either Left or Right, pick up the remaining elements either from Left or Right and insert in original array

Merge - Demonstration



Merge Sort - Implementation

```
void mergeSort(int arr[], int start, int right) {
 if (start < right) {
   int mid = (start + right) / 2;
   mergeSort(arr, start, mid);
   mergeSort(arr, mid + 1, right);
   merge(arr, start, mid, right);
  }
}
```

Merge Sort – Implementation Cont.

```
void merge(int arr[], int start, int mid, int end) {
  int len1 = mid - start + 1;
  int len2 = end - mid;

int leftArr[len1], rightArr[len2];

for (int i = 0; i < len1; i++)
    leftArr[i] = arr[start + i];
for (int j = 0; j < len2; j++)
    rightArr[j] = arr[mid + 1 + j];
```

Merge Sort – Implementation Cont.

```
int i, j, k;
i = 0;
j = 0;
k = start;
while (i < len1 && j < len2) {
  if (leftArr[i] <= rightArr[j]) {
    arr[k] = leftArr[i];
    i++;
  } else {
    arr[k] = rightArr[j];
    j++;
  }
  k++;
}
```

Merge Sort – Implementation Cont.

```
while (i < len1) {
   arr[k] = leftArr[i];
   i++;
   k++;
 }
 while (j < len2) {
   arr[k] = rightArr[j];
   j++;
   k++;
 }
}
```

Quick Sort Vs Merge Sort

QUICK SORT	MERGE SORT
Splitting of array depends on the value of pivot and other array elements	Splitting of array generally done on half
Worst case time complexity is O(n2)	Worst case time complexity is O(nlogn)
It takes less n space than merge sort	It takes more n space than quick sort
It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
It is in-place	It is out-place
Not stable	Stable

Summary

- Algorithm and its analysis
- What is Recursion?
- Binary Search
- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort