Searching and Sorting

ICT 4303

Content

Linear Search

Time Complexity?

O (n)

```
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
}
```

Binary Search: Iterative

Time Complexity?

O (log n)

```
int binarySearch(int arr[], int start, int end, int x)
{
   while (start <= end) {
            int m = start + (end - start) / 2;
            if (arr[m] == x)
                        return m;
            if (arr[m] < x)
                        start = m + 1;
            else
                        end = m - 1;
   }
}
```

Binary Search : Recursive

```
int binarySearch(int arr[], int start, int end, int x)
{
   if (end >= start) {
          int mid = start + (end - start) / 2;
          if (arr[mid] == x)
                   return mid;
          if (arr[mid] > x)
                    return binarySearch(arr, start, mid - 1, x);
          return binarySearch(arr, mid + 1, end, x);
   }
}
```

Insertion Sort

Time Complexity?

O (n): Best Case

O (n^2): Worst Case

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        curr = arr[i];
        j = i - 1;

while (j >= 0 && arr[j] > curr)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = curr;
    }
}
```

Quick Sort

Time Complexity?

O (n log n): Best and Average Case

O (n^2): Worst Case

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
            pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);  // Before pi
            quickSort(arr, pi + 1, high); // After pi
    }
}
```

Quick Sort

Partition code Snippet with last element as the pivot

```
partition (arr[], low, high)
{
    pivot = arr[high];

    i = (low - 1);

    for (j = low; j <= high- 1; j++)
    {

        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Quick Sort

Partition code Snippet with first element as the pivot

```
partition (arr[], low, high) {
        pivot = arr[low];
        j= low;
        i=high;
        while (j <i){
                while(arr[j]<=pivot){
                        j++;
                        }
                while(arr[i]>pivot){
                        i--;
                        }
                if(j < i){
                        swap (arr[j], arr[i]);
                }


                }
        swap(arr[low], arr[i]);
        return i;
  }
```

Merge Sort

Time Complexity?

O (n log n)

```
MergeSort(a[], lb,  ub)
If ub > lb
    1. Find the middle point to divide the array into two halves:
         middle m = lb+ ((ub-lb)/2)
    2. Call mergeSort for first half:
         Call mergeSort(a, lb, m)
    3. Call mergeSort for second half:
         Call mergeSort(a, m+1, ub)
    4. Merge the two halves sorted in step 2 and 3:
         Call merge(a, lb, m, ub)
```

Alternatively: middle m = (lb+ub)/2

Merge Sort

Merge function is at the core of Merge Sort.

```
merge (a, lb, m, ub) {
        i=lb;
        j=m+1;
        k=lb;
        while (i<=mid && j<=ub) {
                if(a [i] <= a[j]{
                        b[k] = a[i];
                        i++; k++;
                } else {
                        b[k] = a[j];
                        j++; k++;
                }
        }
        if (i>mid) {
                while (j<=ub){
                        b[k] = a[j];
                        j++; k++;
                }
        } else{
                while (i<=m){
                        b[k] = a[i];
                        i++; k++;
                }

        }
}
```

Heap Sort

Heapify () is at the core.

Time Complexity?

O (n log n)

```
void heapSort(int arr[], int n)
{

   // Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);

   // One by one extract an element from heap
   for (int i = n - 1; i > 0; i--) {

      // Move current root to end
      swap(arr[0], arr[i]);

      // call max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}
```

Heap Sort

Heapify () is at the core.

```
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

Shell Sort

A variation of Insertion Sort.

```
int shellSort(int arr[], int n)
{
 for (int gap = n/2; gap > 0; gap /= 2)
    {
     for (int i = gap; i < n; i += 1)
        {
            int temp = arr[i];

            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

           arr[j] = temp;
        }
    }
    return 0;
}
```