

Sorting

Topics:

1. Sorting Algorithms
2. Problems & Solutions
3. Notes

Sorting Algorithms

1. **Bubble Sort:** 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, 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.

```
for(int i=0; i<n; i++) {  
    for(int j=0; j<n-1; j++) {  
        if(ar[j] > ar[j+1])  
            ar[j] = ar[j+1]^ar[j]^ar[j+1] = ar[j]; //swapping ar[i] and ar[j]  
    }  
}
```

Time Complexity: $O(n^2)$

2. **Insertion Sort:** This 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.

```
for(int i=1; i<n; i++) {  
    int currentElement = ar[i];  
    int j = i-1;  
    while(j>=0 && ar[j]>currentElement) {  
        ar[j+1] = ar[j];  
        j--;  
    }  
    ar[j+1] = currentElement;  
}
```

Time Complexity: $O(n^2)$

3. **Selection Sort:** The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

```
for(int i=0; i<n-1; i++) {  
    int minimumElement = ar[i], index = i;  
    for(int j=i+1; j<n; j++) {  
        if(minimumElement<ar[j]) {  
            minimumElement = ar[j];  
            index = j;  
        }  
    }  
    ar[i] = ar[index]^ar[i]^ar[index] = ar[i]; //swapping ar[i] and ar[index]  
}
```

Time Complexity: $O(n^2)$

4. **Merge Sort:** Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. Split the array into 2 halves, sort the left part, sort the right part. It then merges each of the resulting sorted parts into the final sorted list.

```

void merge (int ar[], int begin, int end, int mid) {
    int br[end-begin+1];
    for (int i=begin, j=mid+1, k=0; k<end-begin+1; k++) {
        br[k] = (j==end+1) ? ar[i++] :
            (i==mid+1) ? ar[j++] :
            (ar[j]<ar[i]) ? ar[j++] :
            ar[i++];
    }
    for (int i=begin; i<end+1; i++)
        ar[i] = br[i-begin];
}

void merge_sort (int ar[], int begin, int end) {
    if(begin >= end)    return;
    int mid = (begin + end)/2;
    merge_sort(ar, begin, mid);
    merge_sort(ar, mid+1, end);
    merge(ar, begin, end, mid);
}

```

Time Complexity: $O(n \log n)$ [$T(N) = 2T(N/2) + N$]

5. **Quick Sort:** Pick an element (“pivot”) from the given array and partition the array based on the pivot, ie, move items which are less than pivot to the left and the greater elements to the right. The pivot goes in the middle. Repeat the algorithm on the left and right parts again, until you get a partition of size 1 (which is sorted).

```

int partition(int ar[], int begin, int end) {
    int pivot = ar[end];
    int ret = 0, idx = begin;
    for (int j=begin; j<end; j++) {
        if(ar[j]<=pivot) {
            ar[idx] = ar[j]^ar[idx]^ar[j]=ar[idx];
            idx++;
        }
    }
    ar[idx] = ar[end]^ar[idx]^ar[end]=ar[idx];
    return idx;
}

```

```

void quicksort (int ar[], int begin, int end) {
    if(begin >= end)    return;
    int pivot_position = partition(ar, begin, end);
    quicksort(ar, begin, pivot_position-1);
    quicksort(ar, pivot_position+1, end);
}

```

Time Complexity: $T(n) = T(k) + T(n-k-1) + n$

Worst Case($k=0$)	: $T(n) = T(n-1) + n$: $O(n^2)$
Best Case($k=n/2$)	: $T(n) = 2T(n/2) + n$: $O(n\log n)$
Average Case($k=n/10$)	: $T(n) = T(n/10) + T(9n/10) + n$: $O(n\log n)$

6. **Radix Sort:** The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

```

int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int i, count[10] = {0};
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%10]++;
    
    // Change count[i] so that it now contains actual position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i]/exp)%10] - 1] = arr[i];
        count[(arr[i]/exp)%10]--;
    }
    // Copy the output array to arr[] - contains numbers sorted by current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int arr[], int n) {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead of passing digit number,
    // exp is passed. exp is  $10^i$  where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

```

Time Complexity: $O(d*n)$

7. **Heap Sort:** Pop the smallest item from the heap and insert it at the end (final position). Repeat for all items. Time Complexity: $O(n \log n)$
8. **Counting Sort:** Assuming the data are integers, in a range of 0-k. Create an array of size K to keep track of how many times an item appears. We can use the count array to construct the final sorted list.

```
void counting_sort(int ar[], int n, int k) {
    int count[k+1];
    fill(count, count+k+1, 0);
    for(int i=0; i<n; i++)
        count[ar[i]]++;
    int idx = 0;
    for(int i=0; i<k+1; i++)
        for(int j=0; j<count[i]; j++)
            ar[idx++] = i;
}
```

Time Complexity: $O(n)$

Problems

1. Given an integer array of election votes having candidate IDs, WAP to find the winner of the election.
Solution: Counting Sort
2. Given an array of integers and a number K, WAP to check if there exists i,j such that $ar[i] + ar[j] = K$, $i \neq j$.
Solution: Sorting and 2 pointers
3. WAP to count Inversions in an array. Inversion is defined as a pair of indexes (i,j) such that $ar[i] > ar[j]$ and $i < j$.
Solution: Merge Sort
4. WAP to merge 2 sorted arrays. Array A is of size $n+m$, containing n elements, with m extra spaces at the end and array B is of size m. Merge B into A in a sorted order.
Solution: Merge in array A using 2 pointers, starting from right hand side.
5. WAP to find the K-th smallest element in an array of integers.
Solution: Quick Select
6. WAP to sort an array of 0s and 1s.
Solution: 2 pointers
7. WAP to sort an array of size n, having numbers $1 - n^2$.
Solution: Radix Sort

8. Median of two sorted array

Solution:

- a. <http://www.geeksforgeeks.org/median-of-two-sorted-arrays/>
- b. <http://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/>

9. WAP to find the minimum length subarray, sorting which sorts the complete array.

Solution: - Find already sorted array from left and right

- Find min/max in the remaining array
- Find position of min in the left sorted part (pos1)
- Find position of max in the right sorted part (pos2)
- We need to sort the subarray (ar[pos1], ar[pos2])

Notes for using qsort in C

```
1. #include <stdlib.h>
int compare (const void * a, const void * b) {
    return ( *(int*)a > *(int*)b );
}
qsort(ar, n, sizeof(int), compare);
```

Can use double, char or any other struct in place of int as well

Notes for using C++ STL sort function [use #include <algorithm>]

```
1. int ar[n];
sort(ar, ar+n);           //ascending order
sort(ar, ar+10, greater<int>()); //descending order

2. #include <vector>
vector<int> ar(n);
sort(ar.begin(), ar.end());
sort(ar.begin(), ar.end(), greater<int>());

3. vector<vector<int> > ar(n, vector<int>(m));
sort(ar.begin(), ar.end()); //sorts overall 2d array
//For sorting individual rows, use the below:
for(int i=0; i<n; i++)
    sort(ar[i].begin(), ar[i].end());

4. vector<pair<int, int> > ar(n);
//Sorts by first element, and in case of conflicts, sorts by second element.
sort(ar.begin(), ar.end());
```

```
5. class MyClass {  
    int a,b,c;  
    string name;  
}  
  
bool compare(MyClass x, MyClass y) {  
    if(x.a != y.a)      return x.a<y.a;  
    if(x.b != y.b)      return x.b<y.b;  
    if(x.c != y.c)      return x.c<y.c;  
    return x.name<y.name;  
}
```

Can use double, string or any other class in place of int as well [see 4 & 5]
“sort” library, by default, sorts in ascending order. We can override by passing a
manual compare function. [See 5]