Sorting Algorithms

by Harsh Govind

Analysis tools for a sorting algorithm

1. No of comparisons
2. No of swaps
3. Adaptive - if the array is already sorted, then it should take minimum time.
4. Stability - maintaining the relative positioning of the array.
5. Extra memory (in place means the algorithm does not use any extra memory)

Sorting algorithms

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick Sort
5. Merge sort
6. Heap sort
7. Count sort
8. Bin/bucket sort
9. Radix sort
10. Shell sort

1. Bubble sort

Check two consecutive elements. If the first element is greater than the second element, then swap it. The first greater element will come at the last position of the array after performing the first pass. The second greater element will come at the last second position on the second pass, and this process continues.

```
void bubbleSort(int arr[], int n)
    {
        for(int i=0; i<n-1; i++)
        {
            bool flag=true;
            for(int j=0; j<n-1-i; j++)
            {
                if(arr[j]>arr[j+1])
                {
                    flag=false;
                    swap(arr[j], arr[j+1]);
                }
            }
            if(flag)
            {
                break;
            }
        }
        return;
    }
```

- Time complexity: $o(n^2)$
- Space complexity: $o(1)$
- No of passes: $(n-1)$
- No. on comparisons (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Maximum no. of swaps (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Adaptive: Yes, by using a flag variable.
- Stable: Yes, by nature.
- Extra memory: None.
- If we perform k passes you will get k sorted elements (largest elements at the last), i.e. intermediate results are useful.

2. Insertion sort

Select an element and insert it into its sorted position.

```
void insertionSort(int arr[], int n)
    {
        for(int i=1;i<n; i++)
        {
            int x=arr[i], j=i-1;
            while(j>=0 and arr[j]>x)
```

```
            {
                arr[j+1]=arr[j];
                j--;
            }
            arr[j+1]=x;
        }
        return;
    }
```

- Time complexity: $o(n^2)$
- Space complexity: $o(1)$
- No of passes: $(n-1)$
- No. on comparisons (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Maximum no. of swaps (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Adaptive: Yes, by nature.
- Stable: Yes, by nature.
- Extra memory: None.
- Can be applied to the linked list.

3. Selection sort

Select a position and search for the element for that position.

```
void selectionSort(int arr[], int n)
    {
        for(int i=0;i<n-1; i++)
        {
            int j, k;
            for(j=k=i; j<n; j++)
            {
                if(arr[j]<arr[k])
                {
                    k=j;
                }
            }
            swap(arr[i], arr[k]);
        }
        return;
    }
```

- Time complexity: $o(n^2)$

- Space complexity: $o(1)$
- No of passes: $(n-1)$
- No. on comparisons (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Maximum no. of swaps (at most): $(1+1+1+1....) = n$
- Adaptive: No.
- Stable: No.
- Extra memory: None.
- If we perform k passes you will get k sorted elements (smallest elements at the beginning) i.e. intermediate results are useful.

4. Quick sort

Select an element and search for the position for that element. It is also known as Selection Exchange Sort and Partition Exchange Sort.

> 💡 **How do you find the element is at a sorted position?** All the elements before that element should be smaller or equal and all the elements after that should be greater. The position is known as the partitioning position. The elements before and after partitioning position may or may not be sorted.

```
class Solution
{
    public:
    void quickSort(int arr[], int low, int high)
    {
        if(low<high)
        {
            int p=partition(arr, low, high);
            quickSort(arr, low, p-1);
            quickSort(arr, p+1, high);
        }
    }

    public:
    int partition (int arr[], int low, int high)
    {
        int pivot=arr[low], count=0;
        for(int i=low+1; i<=high; i++)
        {
            if(arr[i]<=pivot)
```

```
            {
                count++;
            }
        }

        int pivotIndex=count+low;

        swap(arr[low], arr[pivotIndex]);
        int start=low, end=high;
        while(start < pivotIndex and pivotIndex<end)
        {
            while(arr[start]<=pivot) start++;
            while(arr[end]>pivot) end--;
            if(start < pivotIndex and pivotIndex<end) swap(arr[start++], arr[end--]);
        }

        return pivotIndex;
    }
};
```

- Time complexity:
  - Best: $o(nlog(n))$
  - Average: $o(nlog(n))$
  - Worst: $o(n^2)$
- Space complexity: $o(1)$, recursive stack
- No of passes: $log(n)$ in the best case
- No. on comparisons (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Maximum no. of swaps (at most): $(1+2+3+4....) = (n(n+1))/2 = o(n^2)$
- Adaptive: No.
- Stable: No.
- Extra memory: Recursive stack only.

💡 **Why is Quick Sort preferred for arrays?**

- Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm.
- Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.
- Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
- Quick Sort is also a cache friendly sorting algorithm as it has good **locality of reference** when used for arrays.
- Quick Sort is also **tail recursive**, therefore tail call optimizations is done.

5. Merge sort

Consider each element of an array as a separate list. Each list size is one so the single element is always sorted itself. Merge all the lists using the merge algorithm.

```
//https://practice.geeksforgeeks.org/problems/merge-sort/1
class Solution
{
    public:
    void merge(int arr[], int low, int mid, int high)
    {
        int ans[high-low+1], i=low, j=mid+1, k=0;
        while(i<=mid and j<=high)
        {
            if(arr[i]<arr[j])
            {
                ans[k++]=arr[i++];
            }
            else
            {
                ans[k++]=arr[j++];
```

```
            }
        }
        while(i<=mid)
        {
            ans[k++]=arr[i++];
        }
        while(j<=high)
        {
            ans[k++]=arr[j++];
        }
        k=0;
        for(int i=low; i<=high; i++)
        {
            arr[i]=ans[k++];
        }
    }
    public:
    void mergeSort(int arr[], int low, int high)
    {
        if(low<high)
        {
            int mid=low+(high-low)/2;
            mergeSort(arr, low, mid);
            mergeSort(arr, mid+1, high);
            merge(arr, low, mid, high);
        }
    }
};

//Iterative merge sort
void iMergeSort(int a[], int n)
{
  int p, i, l, mid, h;
  for(p=2; p<=n; p*=2)
  {
    for(i=0; i+p-i<n; i=i+p)
    {
      l=i;
      h=i+p-1;
      mid=(l+h)/2;
      merge(a, l, mid, h);
    }
  }
  if(p/2<n)
  {
    merge(a, 0, p/2, n-1);
  }
}
```

- Time complexity: $o(nlog(n))$
- Space complexity: $o(n)$, recursive stack (maybe optimised)
- No of passes: $log(n)$ in the best case
- Adaptive: No.

- Stable: No.
- Extra memory: Recursive stack and merge function (maybe optimized).

> 💡 Merged sort is preferred for linked lists because it does not require random access to elements in the list. Instead, it recursively splits the list into smaller sublists, sorts them, and then merges them back together. This can be done efficiently with pointers in a linked list, unlike other sorting algorithms that require random access to elements. Additionally, merge sort has a stable time complexity of $o(nlog(n))$, making it an efficient choice for sorting larger linked lists.

6. Heap sort

Introduction

Heap: a complete binary tree which is either min or max heap.

Complete binary tree: all the levels except the last level should be completely filled. The last level should be filled from left to right without a gap.

Max heap: a binary tree in which all the child nodes are smaller than or equal to the parent node.

Min heap: a binary tree in which all the child nodes are greater than or equal to the parent node.

Heapify: converting regular unsorted array to heap array.

▼ Leaf nodes are considered heapified at the beginning, i.e. if we use 1-based indexing than nodes from (n/2 + 1) to the last node are considered to be heapified.

Below is the code for a heap using an array. Insert, delete, and heapify functions are implemented.

```
#include <bits/stdc++.h>
using namespace std;
class heap
{
public:
    int arr[100], size;
    heap()
    {
        arr[0] = -1;
        size = 0;
```

```
    }
    int insert(int x)
    {
        size++;
        arr[size] = x;

        int idx = size;
        while (idx > 1)
        {
            int parent = idx / 2;
            if (arr[parent] < arr[idx])
            {
                swap(arr[parent], arr[idx]);
                idx /= 2;
            }
            else
            {
                return x;
            }
        }
        return -1;
    }
    int deleteFromHeap()
    {
        if (size == 0)
        {
            cout << "Unable to delete from empty heap";
            return -1;
        }

        int x = arr[1];
        arr[1] = arr[size];
        size--;
        int i = 1;
        while (i < size)
        {
            int left = i * 2, right = i * 2 + 1;
            if (left < size and arr[left] > arr[i])
            {
                swap(arr[left], arr[i]);
            }
            else if (right < size and arr[right] > arr[i])
            {
                swap(arr[i], arr[right]);
            }
            else
            {
                break;
            }
        }
        return x;
    }
    void print()
    {
        for (int i = 1; i <= size; i++)
        {
            cout << arr[i] << " ";
        }
```

```
        cout << endl;
    }
};
void heapify(int arr[], int n, int i)
{
    int largest = i, left = i * 2, right = i * 2 + 1;

    if (left <= n and arr[left] > arr[largest])
    {
        largest = left;
    }
    if (right <= n and arr[right] > arr[largest])
    {
        largest = right;
    }

    if (i != largest)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
int main()
{
    heap h;
    h.insert(50);
    h.insert(55);
    h.insert(53);
    h.insert(52);
    h.insert(54);
    h.print();
    cout << h.deleteFromHeap() << endl;
    h.print();

    int arr[] = {-1, 54, 53, 55, 52, 50};
    int n = 5;
    for (int i = n / 2; i > 0; i--)
    {
        heapify(arr, n, i);
    }

    for (int i = 1; i <= n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

- Time complexity
  - Insert n elements: $o(n * log(n))$
  - Delete an element: $o(nlog(n))$

    - Heapify: $o(nlog(n))$
- Space complexity
    - o(1), recursive stack in heapify

Sorting using heapify (heap sort)

https://leetcode.com/problems/sort-an-array/

- Heapify the unsorted array.
- The top node will contain the maximum element.
- Swap the top node with the last node. (first element to the last element)
- We will get the first sorted element (max element) at the last position.
- Heapify the array from start to before sorted array (last one).

```
class Solution
{
    public:
        void heapify(vector<int> &arr, int n, int i)
        {
            int largest = i, left = i *2 + 1, right = i *2 + 2;

            if (left < n and arr[left] > arr[largest])
            {
                largest = left;
            }
            if (right < n and arr[right] > arr[largest])
            {
                largest = right;
            }

            if (i != largest)
            {
                swap(arr[i], arr[largest]);
                heapify(arr, n, largest);
            }
        }
    void heapSort(vector<int> &arr, int n)
    {
        int t = n - 1;
        while (t > 0)
        {
            swap(arr[t], arr[0]);
            heapify(arr, t, 0);
            t--;
        }
    }
    vector<int> sortArray(vector<int> &nums)
    {
        int n = nums.size();
```

```
        for (int i = n / 2 - 1; i >= 0; i--)
        {
            heapify(nums, n, i);
        }

        heapSort(nums, n);
        return nums;
    }
};
```

- Time complexity: $o(nlog(n))$
- Space complexity: $o(1)$, only recursive stack

7. Count sort

Count the occurrences of each element in the input array and use this information to determine the correct position of each element in the sorted output array.

```
void countSort(vector<int> &arr) {
    int max_element = *max_element(arr.begin(), arr.end());
    vector<int> count(max_element + 1, 0);

    for (int &num : arr) {
        count[num]++;
    }

    int index = 0;
    for (int i = 0; i < count.size(); i++) {
        while (count[i] > 0) {
            arr[index] = i;
            index++;
            count[i]--;
        }
    }
}
```

- Time complexity: $o(n)$
- Space complexity: $o(n)$
- Not suitable if the arrays contain negative and huge numbers.

8. Bucket sort

Divide the input into buckets, each representing a range of values. Use the same principle as above to convert the buckets to a sorted array.

```
#include <iostream>

using namespace std;

template <class T>
void Print(T& vec, int n, string s){
    cout << s << ": [" << flush;
    for (int i=0; i<n; i++){
        cout << vec[i] << flush;
        if (i < n-1){
            cout << ", " << flush;
        }
    }
    cout << "]" << endl;
}

int Max(int A[], int n){
    int max = -32768;
    for (int i=0; i<n; i++){
        if (A[i] > max){
            max = A[i];
        }
    }
    return max;
}

// Linked List node
class Node{
public:
    int value;
    Node* next;
};

void Insert(Node** ptrBins, int idx){
    Node* temp = new Node;
    temp->value = idx;
    temp->next = nullptr;

    if (ptrBins[idx] == nullptr){ // ptrBins[idx] is head ptr
        ptrBins[idx] = temp;
    } else {
        Node* p = ptrBins[idx];
        while (p->next != nullptr){
            p = p->next;
        }
        p->next = temp;
    }
}

int Delete(Node** ptrBins, int idx){
    Node* p = ptrBins[idx];  // ptrBins[idx] is head ptr
    ptrBins[idx] = ptrBins[idx]->next;
    int x = p->value;
    delete p;
    return x;
}
```

```
void BinSort(int A[], int n){
    int max = Max(A, n);

    // Create bins array
    Node** bins = new Node* [max + 1];

    // Initialize bins array with nullptr
    for (int i=0; i<max+1; i++){
        bins[i] = nullptr;
    }

    // Update count array values based on A values
    for (int i=0; i<n; i++){
        Insert(bins, A[i]);
    }

    // Update A with sorted elements
    int i = 0;
    int j = 0;
    while (i < max+1){
        while (bins[i] != nullptr){
            A[j++] = Delete(bins, i);
        }
        i++;
    }

    // Delete heap memory
    delete [] bins;
}

int main() {

    int A[] = {2, 5, 8, 12, 3, 6, 7, 10};
    int n = sizeof(A)/sizeof(A[0]);

    Print(A, n, "\t\tA");
    BinSort(A, n);
    Print(A, n, " Sorted A");
    cout << endl;
    return 0;
}
```

- Time complexity: $o(n)$
- Space complexity: $o(n)$
- Not suitable if the arrays contain negative and huge numbers.

9. Radix sort

Sort the elements by processing them digit by digit from the least significant digit to the most significant digit, using counting sort or any stable sorting algorithm at each digit. Repeat this process for all digits to achieve a fully sorted array.

```
#include <iostream>
#include <cmath>

using namespace std;

template <class T>
void Print(T& vec, int n, string s){
    cout << s << ": [" << flush;
    for (int i=0; i<n; i++){
        cout << vec[i] << flush;
        if (i < n-1){
            cout << ", " << flush;
        }
    }
    cout << "]" << endl;
}

int Max(int A[], int n){
    int max = -32768;
    for (int i=0; i<n; i++){
        if (A[i] > max){
            max = A[i];
        }
    }
    return max;
}

// Linked List node
class Node{
public:
    int value;
    Node* next;
};

int countDigits(int x){
    int count = 0;
    while (x != 0){
        x = x / 10;
        count++;
    }
    return count;
}

void initializeBins(Node** p, int n){
    for (int i=0; i<n; i++){
        p[i] = nullptr;
    }
}

void Insert(Node** ptrBins, int value, int idx){
    Node* temp = new Node;
    temp->value = value;
    temp->next = nullptr;

    if (ptrBins[idx] == nullptr){
        ptrBins[idx] = temp;  // ptrBins[idx] is head ptr
```

```
    } else {
        Node* p = ptrBins[idx];
        while (p->next != nullptr){
            p = p->next;
        }
        p->next = temp;
    }
}

int Delete(Node** ptrBins, int idx){
    Node* p = ptrBins[idx];  // ptrBins[idx] is head ptr
    ptrBins[idx] = ptrBins[idx]->next;
    int x = p->value;
    delete p;
    return x;
}

int getBinIndex(int x, int idx){
    return (int)(x / pow(10, idx)) % 10;
}

void RadixSort(int A[], int n){
    int max = Max(A, n);
    int nPass = countDigits(max);

    // Create bins array
    Node** bins = new Node* [10];

    // Initialize bins array with nullptr
    initializeBins(bins, 10);

    // Update bins and A for nPass times
    for (int pass=0; pass<nPass; pass++){

        // Update bins based on A values
        for (int i=0; i<n; i++){
            int binIdx = getBinIndex(A[i], pass);
            Insert(bins, A[i], binIdx);
        }

        // Update A with sorted elements from bin
        int i = 0;
        int j = 0;
        while (i < 10){
            while (bins[i] != nullptr){
                A[j++] = Delete(bins, i);
            }
            i++;
        }
        // Initialize bins with nullptr again
        initializeBins(bins, 10);
    }

    // Delete heap memory
    delete [] bins;
}

int main() {
```

```
    int A[] = {237, 146, 259, 348, 152, 163, 235, 48, 36, 62};
    int n = sizeof(A)/sizeof(A[0]);

    Print(A, n, "\t\tA");
    RadixSort(A, n);
    Print(A, n, " Sorted A");

    return 0;
}
```

- Time complexity: $o(n * d)$ d is the length of the maximum digit
- Space complexity: $o(1)$
- Not suitable if the arrays contain negative and huge numbers.

10. Shell sort

Divide the input into multiple subarrays and sort them individually using an insertion sort with decreasing gap sizes, gradually reducing the gap until it becomes 1, resulting in a partially sorted array that is then sorted using a final pass of the insertion sort.

```
// Code is similar to Insertion Sort with some modifications
void ShellSort(int A[], int n){
    for (int gap=n/2; gap>=1; gap/=2){
        for (int j=gap; j<n; j++){
            int temp = A[j];
            int i = j - gap;
            while (i >= 0 && A[i] > temp){
                A[i+gap] = A[i];
                i = i-gap;
            }
            A[i+gap] = temp;
        }
    }
}
```

- Time Complexity: Depends on the gap sequence used. For the original Shell's sequence (n/2, n/4, ..., 1), the worst-case time complexity is not clearly defined but is generally considered to be sub-quadratic. With the best-known gap sequence, the Sedgewick sequence, the time complexity is O(n^1.3).
- Space Complexity: O(1)

> Thanks for reading.