Sorting

Jarif Rahman

14th Januray 2022

সূচীপত্র

1 Bubble Sort 3

2 Merge Sort 3

2.1 Merge Algorithm 3

2.2 Merge Sort 4

3 Counting Sort 7

4 C++ functions 8

4.1 std::sort 8

4.2 Custom Comparators 9

Compatitive Programming এর সবচেয়ে বেসিক বিষয় বস্তুর একটা হলো সর্টিং। সর্টিং Compatitive Programming অহরহ লাগে। সর্টিং এর মুল বিষয় হলো একটা অ্যারেকে কোনো এক ক্রমে সাজানো বা সর্ট করা। সর্টিং এর জন্য অনেক অ্যালগরিদম আছে। এর মধ্যে গুরুত্বপুর্ণ কয়েকটা দেখানো হলো। আমরা আপাতত শুধু ছোট থেকে বড় সর্ট করা নিয়েই মাথা ঘামাবো।

1 Bubble Sort

Bubble Sort বোধায় সবচেয়ে সহজ সর্টিং অ্যালগরিদম। এটি $O(n^2)$ এ কাজ করে। ধর n সাইজ এর একটা অ্যারে a কে সর্ট করতে চাই। আমরা অ্যারেটার ওপর n বার ইটারেট করবো। প্রত্যেকবার যখন $a[i] > a[i+1]$ হবে আমরা এলেমেন্ট দুইটাকে swap করে দিবো। প্রথম ইটারেশনের পর সবচেয়ে বড় সংখ্যা তার সঠিক জায়াগায় আসবে, দ্বিতীয় ইটারেশনের পর দ্বিতীয় সবচেয়ে বড় সংখ্যা তার সঠিক জায়াগায় আসবে, $\cdots$, n টা ইটারেশনের পরে সব গুলা সংখ্যা নিজের সঠিক জায়গায় চলে আসবে।

```
for(int i = 0; i < n; i++) for(int j = 0; j+1 < n; j++)
    if(a[j] > a[j+1]) swap(a[j], a[j+1]);
```

2 Merge Sort

Merge Sort অ্যালগরিদম সম্পূর্ণ ভাবে megre নামের অন্য এক এলগরিদমের ওপর নির্ভরশীল। তাই merge sort শিখার আগে আমাদের merge অ্যালগরিদম শিখা লাগবে।

2.1 Merge Algorithm

Merge অ্যালগরিদম দুইটা sorted অ্যারেকে ইনপুট হিসেবে নেয়, আর ওই দুটা অ্যারেকে combine করে একটা নতুন sorted অ্যারে আউটপুট হিসেবে দেয়। যেমন: ইনপুট $[1, 2, 4, 10, 13]$ আর $[2, 3, 5]$ হলে আউটপুট হবে $[1, 2, 2, 3, 4, 5, 10, 13]$। Merge অ্যালগরিদম $O(n + m)$ এ কাজ করে যেখানে n, m যথাক্রমে অ্যারে দুইটার সাইজ।

ধরো আমাদের দুইটা অ্যারে a আর b দেওয়া আছে। আমরা দুইটাকে merge করে c বানাতে চাই। অ্যারে a sorted, অর্থাৎ যেকোনো $i, j(i \leq j)$ এর জন্য $a_i \leq a_j$। তাই a অ্যারের আগের সংখ্যাগুলা c অ্যারেতেও আগে থাকবে, আর পরের গুলা c অ্যারেতেও পরে থাকবে। একই কথা b অ্যারের জন্যও সত্য।

এই ফ্যাক্টটা ব্যাবহার করে merge অ্যালগরিদম কাজ করে। Merge করার জন্য আমরা দুইটা পয়েন্টার রাখবো (আসলে সত্যিকারের পয়েন্টার না রাখলেও হবে, ইনডেক্স দিয়েও কাজ চলে যাবে, implementation দেখ)। ধর পয়েন্টার দুইটা i আর j। শুরুতে $i = j = 0$। যদি $a_i \leq b_j$ হয় আমরা c অ্যারেতে a_i add দিব, এবং i এর মান এক বাড়ায় দিব। $a_i > b_j$ হলে c অ্যারেতে b_j add করে দিব, এবং j এর মান এক বাড়ায় দিব। নিচের উদাহরণ দেখলেই বুঝতে পারবা এটা কেন কাজ করে।

- $a = [1, 2, 4, 10, 13], b = [2, 3, 5], c = []$
- $a = [1, 2, 4, 10, 13], b = [2, 3, 5], c = [1], i = 0, j = 0 \ (a_i \leq b_j)$

- $a = [\cancel{1}, 2, 4, 10, 13], b = [2, 3, 5], c = [1, 2], i = 1, j = 0 \ (a_i \leq b_j)$
- $a = [\cancel{1}, \cancel{2}, 4, 10, 13], b = [2, 3, 5], c = [1, 2, 2], i = 2, j = 0 \ (a_i > b_j)$
- $a = [\cancel{1}, \cancel{2}, 4, 10, 13], b = [\cancel{2}, 3, 5], c = [1, 2, 2, 3], i = 2, j = 1 \ (a_i > b_j)$
- $a = [\cancel{1}, \cancel{2}, 4, 10, 13], b = [\cancel{2}, \cancel{3}, 5], c = [1, 2, 2, 3, 4], i = 2, j = 2 \ (a_i \leq b_j)$
- $a = [\cancel{1}, \cancel{2}, \cancel{4}, 10, 13], b = [\cancel{2}, \cancel{3}, 5], c = [1, 2, 2, 3, 4, 5], i = 3, j = 2 \ (a_i > b_j)$
- $a = [\cancel{1}, \cancel{2}, \cancel{4}, 10, 13], b = [\cancel{2}, \cancel{3}, \cancel{5}], c = [1, 2, 2, 3, 4, 5, 10], i = 3, j = 3 > |b|$ (10 is the only option)
- $a = [\cancel{1}, \cancel{2}, \cancel{4}, \cancel{10}, 13], b = [\cancel{2}, \cancel{3}, \cancel{5}], c = [1, 2, 2, 3, 4, 5, 10, 13], i = 4, j = 3 > |b|$ (13 is the only option)
- $a = [\cancel{1}, \cancel{2}, \cancel{4}, \cancel{10}, \cancel{13}], b = [\cancel{2}, \cancel{3}, \cancel{5}], c = [1, 2, 2, 3, 4, 5, 10, 13]$ (done)

এই উদাহরণটার একটা gif এখানে পাওয়া যাবে।

Implementation

```
vector<int> merge(vector<int> a, vector<int> b){
    vector<int> c(a.size()+b.size());
    int i = 0, j = 0;
    while(i < a.size() || j < b.size()){
        if(j == b.size()) c[i+j] = a[i], i++;
        else if(i == a.size()) c[i+j] = b[j], j++;
        else if(a[i] <= b[j]) c[i+j] = a[i], i++;
        else c[i+j] = b[j], j++;
    }
    return c;
}
```

উল্লেখ্য যে C++ এর নিজস্ব merge অ্যালগরিদমের implementation আছে।

```
vector<int> a = {1, 2, 4, 10, 13};
vector<int> b = {2, 3, 5};
vector<int> c(8);
merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());
//c = {1, 2, 2, 3, 4, 5, 10, 13}
```

2.2 Merge Sort

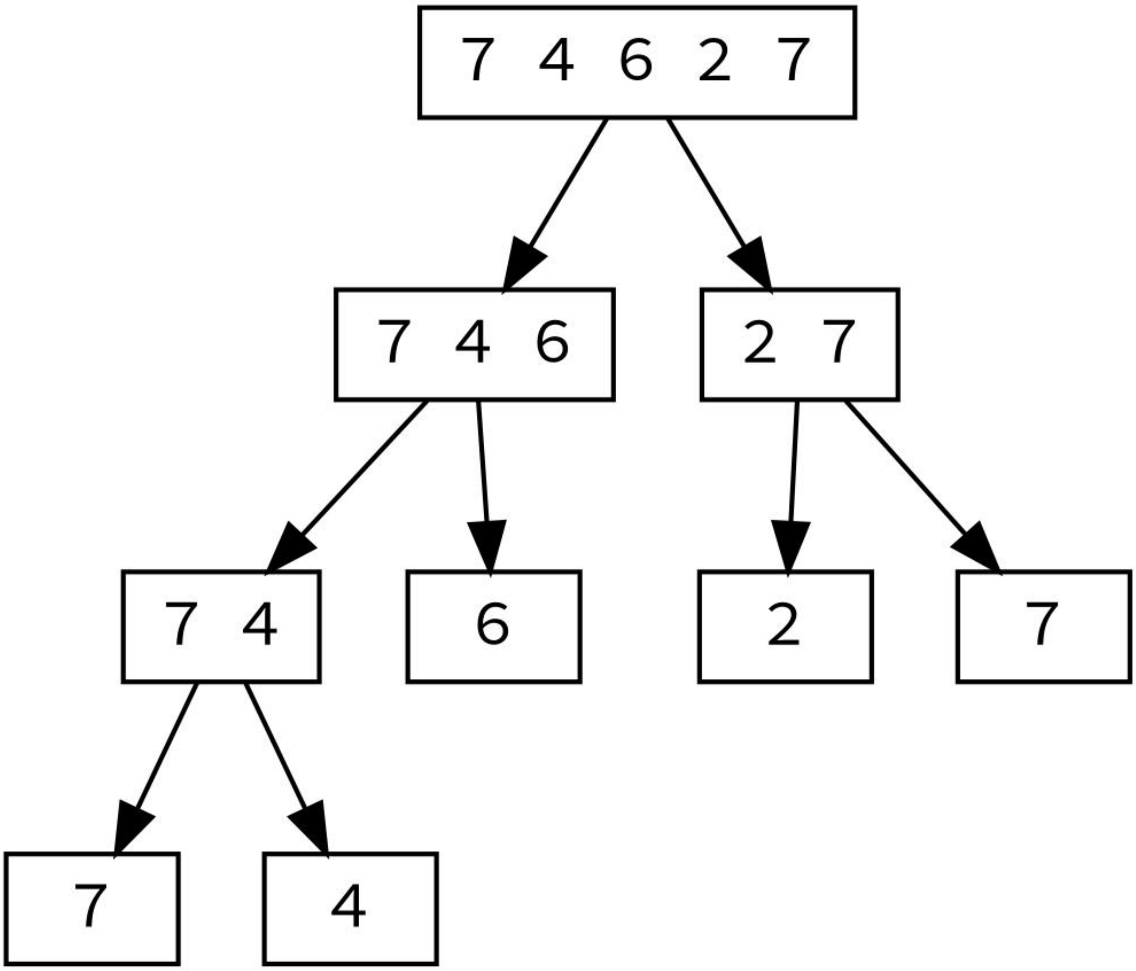
Merge Sort একটা সর্টিং অ্যালগরিদম যেটা একটা অ্যারেকে $O(n \log n)$ এ সর্ট করতে পারে। এটা recursively কাজ করে। ধর আমরা `merge_sort(a, l, r)` নামের একটা ফাংকশান বানাবো যেটা অ্যারে `a` কে `a[l]` থেকে `a[r]` পর্যন্ত sort করবে (পুরাটা sort করতে আমরা `merge_sort(a, 0, a.size()-1)` call করব)। `merge_sort` ফাংকশানটা এভাবে কাজ করবে:

- যদি `l == r` হয় তাহলে এই range এ শুধু একটা সংখ্যা আছে। একটা সংখ্যা নিজেই sorted। তাই আর কিছু করা লাগবে না।

- যদি `l != r` হয় তাহলে আমরা এই range কে দুইটা সমানভাগে ভাগ করব। যদি সমান ভাগে ভাগ না করা যায় তাহলে এক এলেমেন্ট কম বেশি করা যাবে (যেমন 7 টা এলেমেন্ট থাকলে প্রথম ভাগে 4 আর দ্বিতীয় ভাগে 3 এলেমেন্ট দেয়া যাবে)। আমরা এই দুই ভাগের ওপর `merge_sort` call করে এদেরকে সর্ট করে নিব (আবার মনে করিয়ে দেই Merge Sort কিন্তু recursively কাজ করে)। তারপর এই দুইটা range কে merge অ্যালগরিদম দ্বারা merge করে নিব।
- Simple :D

এখানে $[7, 4, 6, 2, 7]$ অ্যারের ওপর merge sort করার একটা gif পাওয়া যাবে। সবাই এটা দেখে আসো। কারণ আমি প্রায় sure যে শুধু মাত্র কথা দিয়ে megre sort বুঝান সম্ভব হবে না।

gif এ যেই ট্রি টা দেখানো হয়েছে সেটাকে বলে merge sort tree।



চিত্র 1: Merge Sort Tree

দেখ এখানে প্রত্যেকটা লেভেল এ অ্যারেটা দুইভাগে ভাগ হচ্ছে। তাই এই ট্রিটার height প্রায় $\log n$ যেখানে n হলো অ্যারেটার উপাদান সংখ্যা। প্রত্যেক লেভেল এ আমরা সর্বোচ্চ n বার merge করব (প্রত্যেক লেভেলে উপাদানই তো সর্বোচ্চ n টা :P)। তাই Merge Sort এর time complexity $O(n \log n)$।

Implementation

```
vector<int> merge(vector<int> a, vector<int> b){
```

```
    vector<int> c(a.size()+b.size());
    int i = 0, j = 0;
    while(i < a.size() || j < b.size()){
        if(j == b.size()) c[i+j] = a[i], i++;
        else if(i == a.size()) c[i+j] = b[j], j++;
        else if(a[i] <= b[j]) c[i+j] = a[i], i++;
        else c[i+j] = b[j], j++;
    }
    return c;
}

void merge_sort(vector<int> &v, int l, int r){
    if(l == r) return;
    int md = (l+r)/2;
    merge_sort(v, l, md);
    merge_sort(v, md+1, r);
    vector<int> a, b, c;
    for(int i = l; i <= md; i++) a.push_back(v[i]);
    for(int i = md+1; i <= r; i++) b.push_back(v[i]);
    c = merge(a, b);
    for(int i = l; i <= r; i++) v[i] = c[i-l];
}

int main(){
    vector<int> v = {7, 4, 6, 2, 7};
    megre_sort(v, 0, 4);
    //v = {2, 4, 6, 7, 7}
}
```

আরেকভাবে merge sort implement করা যায়:

```
template<typename it>
void merge_sort(it l, it r){
    int n = r-l;
    if(n <= 1) return;
    int md = n/2;
    merge_sort(l, l+md);
    merge_sort(l+md, r);
    typename remove_reference<decltype(*l)>::type* sth
        = new typename remove_reference<decltype(*l)>::type[n];
    for(int i = 0; i < n; i++) sth[i] = *(l+i);
    merge(sth, sth+md, sth+md, sth+n, l); //std::merge
    delete[] sth;
}

int main(){
    vector<int> v = {7, 4, 6, 2, 7};
    merge_sort(v.begin(), v.end()); //v = {2, 4, 6, 7, 7}
    array<int, 5> a = {7, 4, 6, 2, 7};
    merge_sort(a.begin(), a.end()); //a = {2, 4, 6, 7, 7}
}
```

এটা আগের চেয়ে কঠিন কিন্তু এটা বেশি STL-ish, flexible আর fast।

3 Counting Sort

আমরা এখন পর্যন্ত যত সর্টিং অ্যালগরিদম দেখেছি সবই comparision sort। আমরা সর্ট করার সময় ছোট থেকে বড় সর্ট করেছিলাম। কিন্তু একটু চিন্তা করলেই বুঝতে পারবে আমরা ছোট থেকে বড় বাদে অন্য কোনো ক্রম ও ব্যবহার করতে পারতাম। যেমন: বড় থেকে ছোট, lexographically ইত্যাদি। Merge Sort আমদের শুধু merge ফাংকশানটা একটু পরিবর্তন করতে হতো। যেমন দুইটা string কে যখন compare করা হয় তখন lexographically compare করা হয়। তাই এই কোডটা

```
vector<string> merge(vector<string> a, vector<string> b){
    vector<string> c(a.size()+b.size());
    int i = 0, j = 0;
    while(i < a.size() || j < b.size()){
        if(j == b.size()) c[i+j] = a[i], i++;
        else if(i == a.size()) c[i+j] = b[j], j++;
        else if(a[i] <= b[j]) c[i+j] = a[i], i++;
        else c[i+j] = b[j], j++;
    }
    return c;
}
```

string কে lexographically সর্ট করবে। কিন্তু আমরা যদি string কে সাইজ অনুযায়ী সর্ট করতে চাই তাহলে ৭ম লাইনে `a[i] <= b[j]` এর জায়গায় `a[i].size() <= b[j].size()` দিলেই হয়ে যাবে।

```
vector<string> merge(vector<string> a, vector<string> b){
    vector<string> c(a.size()+b.size());
    int i = 0, j = 0;
    while(i < a.size() || j < b.size()){
        if(j == b.size()) c[i+j] = a[i], i++;
        else if(i == a.size()) c[i+j] = b[j], j++;
        else if(a[i].size() <= b[j].size()) c[i+j] = a[i], i++;
        else c[i+j] = b[j], j++;
    }
    return c;
}
```

আমরা সর্টিং এর জন্য যেকোনো রকম ক্রম ব্যবহার করতে পারি যতক্ষণ সেই ক্রম নিচের সর্তটা মানে:

- যদি ওই ক্রম অনুযায়ী $a \le b$ হয় এবং $b \le c$ হয় তাহলে ওই ক্রম অনুযায়ী $a \le c$ হবে।

এটা প্রমাণিত যে কোনো comparision sorting algorithm $O(n \log n)$ এর চেয়ে efficient হতে পারে না।

Counting Sort একটা non-comparision sorting algorithm। এখানে ক্রম যেকোনো হতে পারে না। সাধারণত আমরা শুধু পুর্ণসংখ্যাকে ছোট থেকে বড় বা বড় থেকে ছোট সর্ট করতে এটাকে ব্যবহার করে থাকি। Counting Sort এ শর্ত হলো ইনপুট অ্যারের সংখ্যাগুলাকে ছোট ছোট হওয়া লাগবে। যদি ইনপুট অ্যারেতে n টা সংখ্যা থাকে এবং ইনপুট অ্যারের সকল সংখ্যা 0 থেকে m এর মাঝে হয় তাহলে এর time complexity হবে $O(n+m)$।

ধর আমাদের ইনপুট অ্যারে a। এবং m সাইজের নতুন এক অ্যারে c বানাব যেখানে c_i হলো a অ্যারেতে i কয়বার আছে তার সংখ্যায়। অন্য ভাবে বললে আমরা c অ্যারের c_i তে a অ্যারেতে i কয়বার আছে তা

count করে রাখব (বুঝাই যাচ্ছে কেনো একে Counting Sort বলা হয়)। ধর আমরা a অ্যারেকে সর্ট করে b অ্যারে বানাতে চাচ্ছি, আমরা b অ্যারেতে c_0 টা 0, c_1 টা 1, c_2 টা 2, $\cdots$, c_{m-1} টা $m-1$ add দিব।

যেমন: $a = [2, 3, 1, 5, 5, 1, 5]$ হলে $c = [0, 2, 1, 1, 0, 3]$। এরপর আমরা b তে 0 টা 0, 2 টা 1, 1 টা 2, 1 টা 3, 0 টা 4 এবং 3 টা 5 add দিব। তাহলে $b = [1, 1, 2, 3, 5, 5, 5]$ হবে।

Implementation

```
vector<int> counting_sort(vector<int> a, int m){
    int n = a.size();
    vector<int> c(m, 0);
    for(int i = 0; i < n; i++) c[a[i]]++;
    vector<int> b;
    for(int i = 0; i < m; i++) for(int j = 0; j < c[i]; j++) b.push_back(i);
    return b;
}

int main(){
    vector<int> v = {2, 3, 1, 5, 5, 1, 5};
    auto s = counting_sort(v, 6); // s = {1, 1, 2, 3, 5, 5, 5}
}
```

4 C++ functions

4.1 std::sort

C++ এও নিজস্ব সর্টিং ফাংকশান আছে। এটা হলো `std::sort`।

```
vector<int> v = {7, 4, 6, 2, 7};
sort(v.begin(), v.end()); // v ={2, 4, 6, 7, 7}
```

C++ এ `std::sort` অনেক efficiently implement করা। তাই আমরা সাধারণত ওটাই ব্যবহার করে থাকি। `std::sort` introsort নামের এক অ্যালগরিদমকে implement করে। introsort ও $O(n \log n)$ এ কাজ করে। কিন্তু এখানে সুবিধার কথা হলো এর memory complexity $O(1)$ যেখানে Merge Sort এর memory complexity $O(n)$ কারণ প্রত্যেকবার merge করার সময় আমাদের এলেমেন্ট গুলাকে অন্য অ্যারেতে কপি করতে হয়। আমরা একই অ্যারেতে merge করতে পারি না। তাই আমরা সাধারণত `std::sort` ই ব্যবহার করে থাকি। Merge Sort তেমন একটা কাজে আসে না। এটা শিখানো হয়েছে শুধু জানার জন্য। কিন্তু কখনো কখনো merge sort কাজে আসতে পারে। Custom Comparators এ দেখবে আমরা ছোট থেকে বড় বাদেও অন্য ফাংকশান ব্যবহার করতে পারি। যদি কোনো interactive প্রব্লেম এ আমাদের কোনো এলেমেন্ট কে compare করতে প্রত্যেকবার query করা লাগে এবং আমাদের সীমিত সংখ্যক query থাকে তাহলে Merge Sort ব্যবহার করাই বেশি ভালো হবে। কারণ Merge Sort কিভাবে কাজ করে সেটা দেখলেই বুঝতে পারবা এখানে $n(\lceil \log n \rceil + 1)$ এর চেয়ে বেশি query করা হবে না। কিন্তু `std::sort` এর time complexity $O(n \log n)$ হলেও এখানে হালকা constant factor আছে। তাই এটাতে query এর সংখ্যা $n \log n$ পার হয়ে যেতে পারে।

4.2 Custom Comparators

আগেই বলা হয়েছে যে comparision sorting algorithm এ আমরা বড় থেকে ছোট বাদেও অন্য কোনো ক্রম ব্যবহার করতে পারব। C++ `std::sort` এর default comparator হলো `<`। যদি কোনো comparator না বলে দেওয়া হয় তাহলে এলেমেন্টগুলাকে `<` দ্বারা অর্থাৎ ছোট থেকে বড় ক্রমে সাজানো হয়। আমরা `std::sort` এ custom comparator ব্যবহার করতে পারি। Custom comparator হবে একটা ফাংকশান (বা ল্যাম্বডা হলেও হবে) যার পারামিটার হবে অ্যারের দুইটা এলেমেন্ট আর ফাংকশান একটা boolean (true অথবা false) রিটার্ন করবে। যদি comparator অনুযায়ী প্রথমটা ছোট হয় তাহলে false রিটার্ন করতে হবে। বড় হলে true রিটার্ন করতে হবে। দুইটা সমান হলেও false রিটার্ন করতে হবে। নাহলে runtime error খাবে।

```
bool comp(int a, int b){
    return a > b;
}
int main(){
    vector<int> v = {7, 4, 6, 2, 7};
    sort(v.begin(), v.end(), comp); //v = {7, 7, 6, 4 2}
}
```

Custom comparator এ ল্যাম্বডা ফাংকশান ব্যবহার করাই আমার কাছে বেশি সুবিধাজনক মনে হয়। ল্যাম্বডা ব্যবহার করলে বার বার ফাংকশান ডিক্লেয়ার করা লাগে না।

```
vector<int> v = {7, 4, 6, 2, 7};
sort(v.begin(), v.end(), [](int a, int b){
    return a > b;
});
//v = {7, 7, 6, 4 2}
```

Custom comparator আরো বিভিন্ন ভাবে ব্যবহার করা যায়। যেমন ধর তোমার কাছে একটা অ্যারে a আছে আর তুমি জানতে চাও a কে সর্ট করার পর iতম এলেমেন্ট কথায় যাবে। তুমি এটাকে b অ্যারেতে সেভ করে রাখতে চাও। এটাকে এভাবে implement করা যাবে:

```
vector<int> a = {7, 4, 6, 2, 7};
vector<int> b = {0, 1, 2, 3, 4};
sort(b.begin(), b.end(), [&](int x, int y){
    return a[x] < a[y];
});
//b = {3, 1, 2, 0, 4}
```