

CP- EITC

Lecture 4

CP Cell Leads:

Idriss EL ABIDI

Souhail LOUZI

Complete search

January, 2025



- 1 Generating subsets
- 2 Generating permutations
- 3 Prefix Sums - Sum Queries

Introduction

Complete search is a general method that can be used to solve almost any algorithm problem. The idea is to generate all possible solutions to the problem using brute force, and then select the best solution or count the number of solutions, depending on the problem.

Complete search is a good technique if there is enough time to go through all the solutions, because the search is usually easy to implement and it always gives the correct answer. If complete search is too slow, other techniques, such as greedy algorithms or dynamic programming, may be needed.

- 1 Generating subsets
- 2 Generating permutations
- 3 Prefix Sums - Sum Queries

What is Competitive Programming?

We first consider the problem of generating all subsets of a set of n elements. For example, the subsets of 0,1,2 are ; 0, 1, 2, 0,1, 0,2, 1,2 and 0,1,2.

There are two common methods to generate subsets: we can either perform a recursive search or exploit the bit representation of integers.

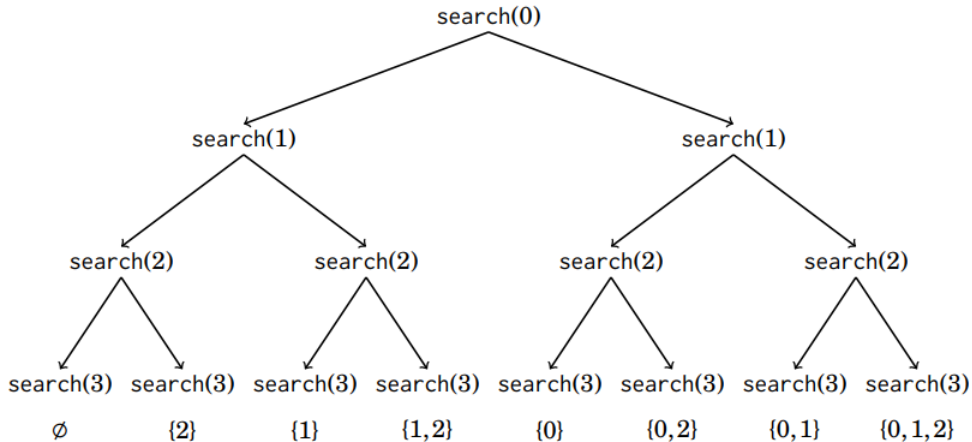
Generating subsets : Recursion

Definiton:

An elegant way to go through all subsets of a set is to use recursion. The following function search generates the subsets of the set $0, 1, \dots, n-1$. The function maintains a vector subset that will contain the elements of each subset. The search begins when the function is called with parameter 0.

```
void search(int k) {  
    if (k == n) {  
        // process subset  
    } else {  
        search(k+1);  
        subset.push_back(k);  
        search(k+1);  
        subset.pop_back();  
    }  
}
```

Generating subsets : Recursion



Generating subsets : bit representation

Another way to generate subsets is based on the bit representation of integers. Each subset of a set of n elements can be represented as a sequence of n bits, which corresponds to an integer between $0 \dots 2^n - 1$. The ones in the bit sequence indicate which elements are included in the subset.

The usual convention is that the last bit corresponds to element 0, the second last bit corresponds to element 1, and so on. For example, the bit representation of 25 is 11001, which corresponds to the subset 0,3,4.

```
for (int b = 0; b < (1<<n); b++) {  
    vector<int> subset;  
    for (int i = 0; i < n; i++) {  
        if (b & (1<<i)) subset.push_back(i);  
    }  
}
```


- 1 Generating subsets
- 2 Generating permutations**
- 3 Prefix Sums - Sum Queries

Generating permutations : Recursion

Like subsets, permutations can be generated using recursion. The following function search goes through the permutations of the set $0, 1, \dots, n-1$. The function builds a vector permutation that contains the permutation, and the search begins when the function is called without parameters.

```
void search() {  
    if (permutation.size() == n) {  
        // process permutation  
    } else {  
        for (int i = 0; i < n; i++) {  
            if (chosen[i]) continue;  
            chosen[i] = true;  
            permutation.push_back(i);  
            search();  
            chosen[i] = false;  
            permutation.pop_back();  
        }  
    }  
}
```

- 1 Generating subsets
- 2 Generating permutations
- 3 Prefix Sums - Sum Queries**

Sum queries - 1D

We can easily process sum queries on a static array by constructing a prefix sum array. Each value in the prefix sum array equals the sum of values in the original array up to that position, i.e., the value at position k is $\text{sumq}(0,k)$. The prefix sum array can be constructed in $O(n)$ time.

For example, consider the following array:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

The corresponding prefix sum array is as follows:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Sum queries - 1D

Since the prefix sum array contains all values of $\text{sum}_q(0,k)$, we can calculate any value of $\text{sum}_q(a,b)$ in $O(1)$ time as follows:

$$\text{sum}_q(a,b) = \text{sum}_q(0,b) - \text{sum}_q(0,a-1)$$

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Now, after an $\mathcal{O}(N)$ preprocessing to calculate the prefix sum array, each of the Q queries takes $\mathcal{O}(1)$ time.

Max Subarray Sum

Given an array `arr[]`, the task is to find the elements of a contiguous subarray of numbers that has the largest sum.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6 using ll = long long;
7
8 int main() {
9     int n;
10    cin >> n;
11    vector<long long> pfx(n + 1); // prefix sum array initially filled with 0's
12    for (int i = 1; i <= n; i++) {
13        ll x;
14        cin >> x;
15        pfx[i] = pfx[i - 1] + x; // compute the prefix sum at each element
16    }
17
18    ll max_subarray_sum = pfx[1];
19    ll min_prefix_sum = pfx[0];
20    for (int i = 1; i <= n; i++) {
21        // max subarray sum is the maximum difference between two prefix sums
22        max_subarray_sum = max(max_subarray_sum, pfx[i] - min_prefix_sum);
23        min_prefix_sum = min(min_prefix_sum, pfx[i]);
24    }
25    cout << max_subarray_sum << endl;
26 }
```

Copy CPP

Max Subarray Sum

Other approach:

```
void SubarrayWithMaxSum(vector<int> nums)
{
    // Initialize currMax and globalMax
    // with first value of nums
    int currMax = nums[0], globalMax = nums[0];
    // Initialize endIndex startIndex, globalStartIndex
    int endIndex = 0;
    int startIndex = 0, globalMaxStartIndex = 0;

    // Iterate for all the elements of the array
    for (int i = 1; i < nums.size(); ++i) {

        // Update currMax and startIndex
        if (nums[i] > nums[i] + currMax) {
            currMax = nums[i];
            startIndex = i; // update the new startIndex
        }

        // Update currMax
        else if (nums[i] < nums[i] + currMax) {
            currMax = nums[i] + currMax;
        }

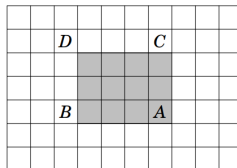
        // Update globalMax and globalMaxStartIndex
        if (currMax > globalMax) {
            globalMax = currMax;
            endIndex = i;
            globalMaxStartIndex = startIndex;
        }
    }

    // Printing the elements of subarray with max sum
    for (int i = globalMaxStartIndex; i <= endIndex; ++i) {
        cout << nums[i] << " ";
    }
}
```

Sum queries - 2D

It is also possible to generalize this idea to higher dimensions. For example, we can construct a two-dimensional prefix sum array that can be used to calculate the sum of any rectangular subarray in $O(1)$ time. Each sum in such an array corresponds to a subarray that begins at the upper-left corner of the array

The following picture illustrates the idea:



The sum of the gray subarray can be calculated using the formula

$$S(A) - S(B) - S(C) + S(D),$$

where $S(X)$ denotes the sum of values in a rectangular subarray from the upper left corner to the position of X .

Sum queries - 2D

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

0	0	0	0	0	0
0	1	6	12	23	31
0	2	14	31	51	63
0	6	24	42	65	79
0	13	36	58	83	100

Since no matter the size of the submatrix we are summing, we only need to access four values of the 2D prefix sum array, this runs in $\mathcal{O}(1)$ per query after an $\mathcal{O}(NM)$ preprocessing.

Thank you for listening !

Idriss El Abidi
Souhail Louzi

Special thanks to the former CP leads:
Hamza BA-MOHAMMED, Hachim Hassani Lahsinui and mohamed ghacham amrani
for their insightful review and precious remarks.