CP- EITC

Lecture 4

CP Cell Leads:
Idriss EL ABIDI
Souhail LOUZI

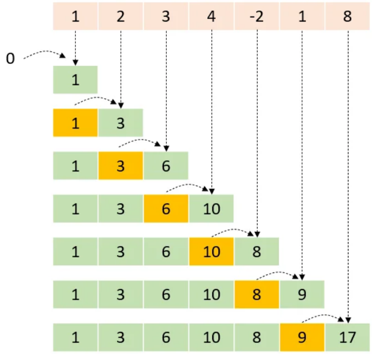
Key Search Techniques: Prefix Sums, Sliding Window, Subsets & Permutations

January, 2025

1. Prefix Sums
2. Sliding Window
3. Generating subsets & permutations
4. Practice

Sum queries - 1D

Let's say we have a one-indexed integer array `arr` of size N and we want to compute the value of: **arr[a]+arr[a+1]+...+arr[b]**. for Q different pairs (a, b) satisfying $1 \leq a \leq b \leq N$.



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 $\texttt{sumq}(0,k)$. The prefix sum array can be constructed in $O(n)$ time.

Note: In prefix sum, we are 1-indexing the array, so we need to set $\texttt{prefix}[0] = 0$. Then, for indices k such that $1 \leq k \leq N$, define the prefix sum array with:

$$\texttt{prefix}[k] = \texttt{prefix}[k-1] + \texttt{arr}[k]$$

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 sumq(0,k), we can calculate any value of sumq(a,b) in O(1) time as follows:

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

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. CSES Problem

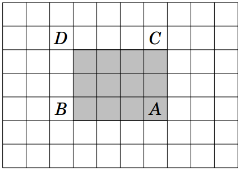
Copy CPP

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
using ll = long long;

int main() {
    int n;
    cin >> n;
    vector<long long> pfx(n + 1);  // prefix sum array initially filled with 0's
    for (int i = 1; i <= n; i++) {
        ll x;
        cin >> x;
        pfx[i] = pfx[i - 1] + x;  // compute the prefix sum at each element
    }

    ll max_subarray_sum = pfx[1];
    ll min_prefix_sum = pfx[0];
    for (int i = 1; i <= n; i++) {
        // max subarray sum is the maximum difference between two prefix sums
        max_subarray_sum = max(max_subarray_sum, pfx[i] - min_prefix_sum);
        min_prefix_sum = min(min_prefix_sum, pfx[i]);
    }
    cout << max_subarray_sum << endl;
}
```

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

$$\boxed{\texttt{prefix}[a][b] = \sum_{i=1}^{a}\sum_{j=1}^{b} \texttt{arr}[i][j].}$$

$$\boxed{\begin{aligned}\texttt{prefix}[i][j] = {} & \texttt{prefix}[i-1][j] + \texttt{prefix}[i][j-1] \\ & - \texttt{prefix}[i-1][j-1] + \texttt{arr}[i][j]\end{aligned}}$$

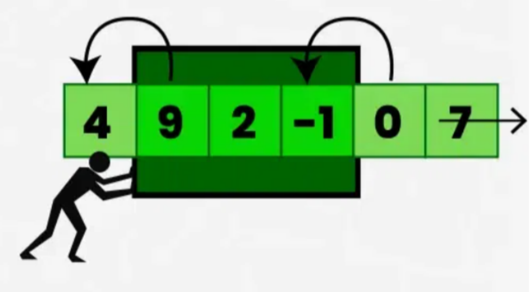
0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

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.

Introduction

Sliding Window is a method used to efficiently solve problems that involve defining a **window** or **range** in the input data (arrays or strings) and then moving that window across the data to perform some operation within the window.



Types of sliding window

Fixed Size Sliding Window

- Compute the sum (or required value) for the first window of size k.
- Slide the window right by removing the first element and adding the next element.
- Maintain and update the required values efficiently.

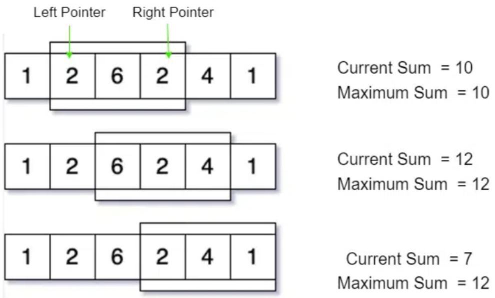
Variable Size Sliding Window

- Start with two pointers (typically left and right) both at the beginning of the array.
- Expand the window by moving the right pointer and add values to the window.
- Once the window violates a condition (e.g., the sum exceeds a threshold), move the left pointer to shrink the window until the condition is met again.
- Continue sliding the window by adjusting left and right pointers.

Find the maximum sum of all subarrays of size K

Problem:

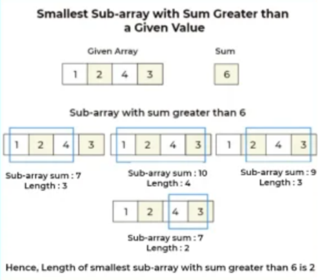
Given an array of integers, find the maximum sum of any contiguous subarray of size k.



Smallest subarray with sum greater than a given value

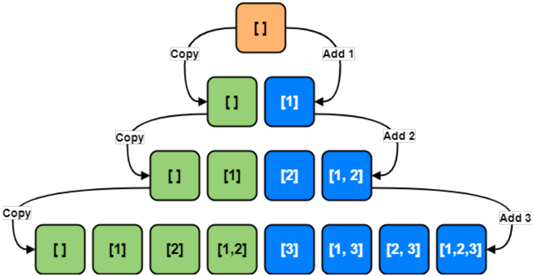
Problem:

Given an array of integers and a target sum S, find the smallest contiguous subarray whose sum is greater than S.



Introduction

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.



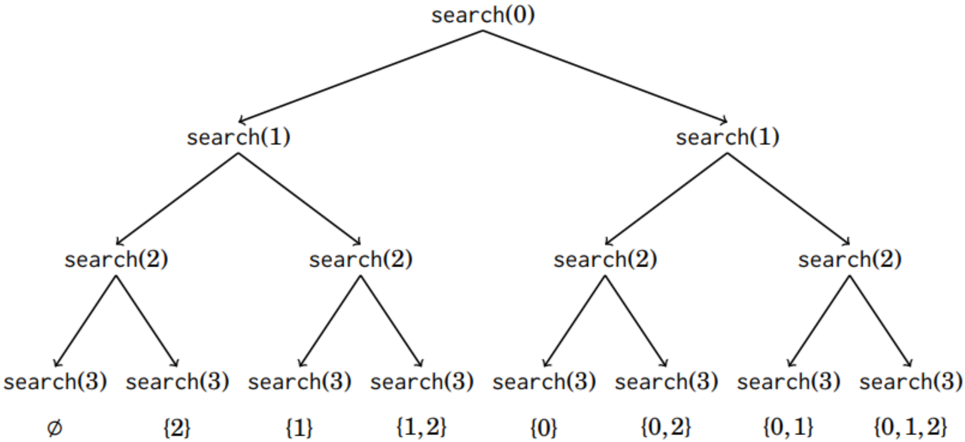
Generating subsets : Recursion

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,...,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 $0 \ldots 2^n - 1$. 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);
    }
}
```

Generating permutations : Recursion

Like subsets, permutations can be generated using recursion. The following function search goes through the permutations of the set 0,1,...,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();
        }
    }
}
```

Problems

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.