Last update: June 8, 2022 Translated From: e-maxx.ru

Search the subarray with the maximum/minimum sum

Here, we consider the problem of finding a subarray with maximum sum, as well as some of its variations (including the algorithm for solving this problem online).

Problem statement

Given an array of numbers $a[1 \ldots n]$. It is required to find a subarray $a[l \ldots r]$ with the maximal sum:

$$\max_{1 \le l \le r \le n} \sum_{i=l}^{r} a[i].$$

For example, if all integers in array $a[]$ were non-negative, then the answer would be the array itself. However, the solution is non-trivial when the array can contain both positive and negative numbers.

It is clear that the problem of finding the **minimum** subarray is essentially the same, you just need to change the signs of all numbers.

Algorithm 1

Here we consider an almost obvious algorithm. (Next, we'll look at another algorithm, which is a little harder to come up with, but its implementation is even shorter.)

Algorithm description

The algorithm is very simple.

We introduce for convenience the **notation**: $s[i] = \sum_{j=1}^{i} a[j]$. That is, the array $s[i]$ is an array of partial sums of array $a[]$. Also, set $s[0] = 0$.

Let us now iterate over the index $r = 1 \ldots n$, and learn how to quickly find the optimal l for each current value r, at which the maximum sum is reached on the subarray $[l, r]$.

Formally, this means that for the current r we need to find an l (not exceeding r), so that the value of $s[r] - s[l-1]$ is maximal. After a trivial transformation, we can see that we need to find in the array $s[]$ a minimum on the segment $[0, r-1]$.

From here, we immediately obtain a solution: we simply store where the current minimum is in the array $s[]$. Using this minimum, we find the current optimal index l in $O(1)$, and when moving from the current index r to the next one, we simply update this minimum.

Obviously, this algorithm works in $O(n)$ and is asymptotically optimal.

Implementation

To implement it, we don't even need to explicitly store an array of partial sums $s[]$ — we will only need the current element from it.

The implementation is given in 0-indexed arrays, not in 1-numbering as described above.

We first give a solution that finds a simple numerical answer without finding the indices of the desired segment:

```
int ans = a[0], sum = 0, min_sum = 0;

for (int r = 0; r < n; ++r) {
    sum += a[r];
    ans = max(ans, sum - min_sum);
    min_sum = min(min_sum, sum);
}
```

Now we give a full version of the solution, which additionally also finds the boundaries of the desired segment:

```
int ans = a[0], ans_l = 0, ans_r = 0;
int sum = 0, min_sum = 0, min_pos = -1;

for (int r = 0; r < n; ++r) {
    sum += a[r];
    int cur = sum - min_sum;
    if (cur > ans) {
        ans = cur;
        ans_l = min_pos + 1;
        ans_r = r;
    }
    if (sum < min_sum) {
        min_sum = sum;
        min_pos = r;
    }
}
```

Algorithm 2

Here we consider a different algorithm. It is a little more difficult to understand, but it is more elegant than the above, and its implementation is a little bit shorter. This algorithm was proposed by Jay Kadane in 1984.

Algorithm description

The algorithm itself is as follows. Let's go through the array and accumulate the current partial sum in some variable s. If at some point s is negative, we just assign $s = 0$. It is argued that the maximum all the values that the variable s is assigned to during the algorithm will be the answer to the problem.

Proof:

Consider the first index when the sum of s becomes negative. This means that starting with a zero partial sum, we eventually obtain a negative partial sum — so this whole prefix of the array, as well as any suffix, has a negative sum. Therefore, this subarray never contributes to the partial sum of any subarray of which it is a prefix, and can simply be dropped.

However, this is not enough to prove the algorithm. In the algorithm, we are actually limited in finding the answer only to such segments that begin immediately after the places when $s < 0$ happened.

But, in fact, consider an arbitrary segment $[l, r]$, and l is not in such a "critical" position (i.e. $l > p+1$, where p is the last such position, in which $s < 0$). Since the last critical position is strictly earlier than in $l-1$, it turns out that the sum of $a[p+1 \ldots l-1]$ is non-negative. This means that by moving l to position $p+1$, we will increase the answer or, in extreme cases, we will not change it.

One way or another, it turns out that when searching for an answer, you can limit yourself to only segments that begin immediately after the positions in which $s < 0$ appeared. This proves that the algorithm is correct.

Implementation

As in algorithm 1, we first gave a simplified implementation that looks for only a numerical answer without finding the boundaries of the desired segment:

```
int ans = a[0], sum = 0;

for (int r = 0; r < n; ++r) {
    sum += a[r];
    ans = max(ans, sum);
    sum = max(sum, 0);
}
```

A complete solution, maintaining the indexes of the boundaries of the corresponding segment:

```
int ans = a[0], ans_l = 0, ans_r = 0;
int sum = 0, minus_pos = -1;

for (int r = 0; r < n; ++r) {
    sum += a[r];
    if (sum > ans) {
        ans = sum;
        ans_l = minus_pos + 1;
        ans_r = r;
    }
    if (sum < 0) {
        sum = 0;
        minus_pos = r;
    }
}
```

Related tasks

Finding the maximum/minimum subarray with constraints

If the problem condition imposes additional restrictions on the required segment $[l, r]$ (for example, that the length $r-l+1$ of the segment must be within the specified limits), then the described algorithm is likely to be easily generalized to these cases — anyway, the problem will still be to find the minimum in the array $s[]$ with the specified additional restrictions.

Two-dimensional case of the problem: search for maximum/minimum submatrix

The problem described in this article is naturally generalized to large dimensions. For example, in a two-dimensional case, it turns into a search for such a submatrix $[l_1 \ldots r_1, l_2 \ldots r_2]$ of a given matrix, which has the maximum sum of numbers in it.

Using the solution for the one-dimensional case, it is easy to obtain a solution in $O(n^3)$ for the two-dimensions case: we iterate over all possible values of l_1 and r_1, and calculate the sums from l_1 to r_1 in each row of the matrix. Now we have the one-dimensional problem of finding the indices l_2 and r_2 in this array, which can already be solved in linear time.

Faster algorithms for solving this problem are known, but they are not much faster than $O(n^3)$, and are very complex (so complex that many of them are inferior to the trivial algorithm for all reasonable constraints by the hidden constant). Currently, the best known algorithm works in $O\left(n^3 \frac{\log^3 \log n}{\log^2 n}\right)$ time (T. Chan 2007 "More algorithms for all-pairs shortest paths in weighted graphs")

This algorithm by Chan, as well as many other results in this area, actually describe **fast matrix multiplication** (where matrix multiplication means modified multiplication: minimum is used instead of addition, and addition is used instead of multiplication). The problem of finding the submatrix with the largest sum can be reduced to the problem of finding the shortest paths between all pairs of vertices, and this problem, in turn, can be reduced to such a multiplication of matrices.

Search for a subarray with a maximum/minimum average

This problem lies in finding such a segment $a[l, r]$, such that the average value is maximal:

$$\max_{l \le r} \frac{1}{r-l+1} \sum_{i=l}^{r} a[i].$$

Of course, if no other conditions are imposed on the required segment $[l, r]$, then the solution will always be a segment of length 1 at the maximum element of the array. The problem only makes sense, if there are additional restrictions (for example, the length of the desired segment is bounded below).

In this case, we apply the **standard technique** when working with the problems of the average value: we will select the desired maximum average value by **binary search**.

To do this, we need to learn how to solve the following subproblem: given the number x, and we need to check whether there is a subarray of array $a[]$ (of course, satisfying all additional constraints of the problem), where the average value is greater than x.

To solve this subproblem, subtract x from each element of array $a[]$. Then our subproblem actually turns into this one: whether or not there are positive sum subarrays in this array. And we already know how to solve this problem.

Thus, we obtained the solution for the asymptotic $O(T(n) \log W)$, where W is the required accuracy, $T(n)$ is the time of solving the subtask for an array of length n (which may vary depending on the specific additional restrictions imposed).

Solving the online problem

The condition of the problem is as follows: given an array of n numbers, and a number L. There are queries of the form (l, r), and in response to each query, it is required to find a subarray of the segment $[l, r]$ of length not less than L with the maximum possible arithmetic mean.

The algorithm for solving this problem is quite complex. KADR (Yaroslav Tverdokhleb) described his algorithm on the Russian forum.