Notes on the Maximum Sum Subarray Problem

Conrado Martínez[1]

March 30, 2022

[1]Departament of Computer Science. Universitat Politècnica de Catalunya. Barcelona, Spain. E-mail: `conrado - cs upc edu`

Introduction

Given an array $A[0..n-1]$ of $n > 0$ integers, we are asked to find a non-empty *segment* or subarray $A[i..j]$ with maximum sum. That is, if we define

$$\sigma_{ij} := \sum_{k=i}^{j} A[k]$$

then we want to locate a subarray such that its sum is

$$\sigma^* = \max_{0 \le i \le j < n} \sigma_{ij},$$

and we would also be able to find the indices that delimit such subarray.

Solution #1

We just apply the definition and solve the problem by *brute force*.

Algorithm 1 Brute force algorithm

```
int n = A.size();
int sum_max = A[0]; int imax = 0, jmax = 0;
for(int i = 0; i < n; ++i) {
    for (int j = i; j < n; ++j) {
        int sum = 0;
        for (int k = i; k <= j; ++k)
            sum += A[k];
        // sum = σ_ij
        if (sum > sum_max) {
            sum_max = sum;
            imax = i; jmax = j;
        }
    }
}
// Post: A[imax..jmax] is a segment of maximum sum;
// the sum of its elements is sum_max
```

This algorithm is obviously correct but also clearly very inefficent: its cost is $\Theta(n^3)$.

Solution #2

An easy refinement of our previous solution is immediate after realizng that

$$\sigma_{ij} = \sigma_{i,j-1} + A[j], \qquad i \le j,$$

with the convention that $\sigma_{ij} = 0$ whenever $j < i$.

Algorithm 2 A first refinement

```
int n = A.size();
int sum_max = A[0]; int imax = 0, jmax = 0;
for(int i = 0; i < n; ++i) {
  int sum = 0;
  for (int j = i; j < n; ++j) {
    sum += A[j];
    // sum = σ_ij
    if (sum > sum_max) {
      sum_max = sum;
      imax = i; jmax = j;
    }
  }
}
// Post: A[imax..jmax] is a segment of maximum sum;
// the sum of its elements is sum_max
```

Solution #3

For our third solution, we will devise a *divide & conquer* algorithm. Let σ^*_{ij} denote the maximum sum of any segment of the subarray $A[i..j]$. We are thus interested in $\sigma^* \equiv \sigma^*_{0,n-1}$. In the D&C approach, recursive calls on the subarrays $A[i..m]$ and $A[m+1..j]$ will return $\sigma^*_{i,m}$ and $\sigma^*_{m+1,j}$. The maximum sum σ^*_{ij} might be one of those two, but it might be the sum of a segment across the two subarrays, that is, it might be that $\sigma^*_{ij} = \sigma_{i',j'}$ for some $i' \leq m$ and some $j' > m$. In that case the segment is composed of two special segments: $A[i'..m]$ has maximal sum among all segments of $A[i..m]$ ending at m, and $A[m+1..j']$ has maximal sum among all segments of $A[m+1..j]$ starting at $m+1$. If some of these two claims were not true we would arrive at a contradiction, because then $\sigma_{i',j'}$ could not be maximal among all the segments starting before or at m and ending after or at $m+1$.

Finding i' and j' can be easily done by scanning the array from m to the right and from $m+1$ to the left. The cost of this function is $\Theta(n)$.

```
// Returns the maximum sum of a segment of A[i..j] across
// index m, and the beginning and end of such a segment,
// that is, A[ip..jp] has maximal sum among all segments of
// A[i..j] that start before or at m and end after or at m+1
int maximum_segment_across(const vector<int>& A, int i, int j,
                           int m, int& ip, int& jp) {
  ip = m; int sum_max_left = A[m]; int sum = 0;
  for (int k = m; k >= i; --k) {
    sum += A[k];
    // sum = σ_{k,m}
```

```
      if (sum > sum_max_left) {
        sum_max_left = sum; ip = k;
      }
    }
    jp = m+1; int sum_max_right = A[m+1]; sum = 0;
    for (int k = m+1; k <= j; ++k) {
      sum += A[k];
      // sum = σ_{m+1,k}
      if (sum > sum_max_right) {
        sum_max_right = sum; jp = k;
      }
    }
    return sum_max_left + sum_max_right;
}
```

Algorithm 3 is the resulting divide & conquer algorithm using the function above. The algorithm works no matter what the value of m we pick in the recursive case, as long as $i \leq m < j$; the best choice is to take the middle point of the subarray $A[i..j]$.

The cost $S(n)$ of Algorithm 3 satisfies then the following recurrence:

$$S(n) = \Theta(n) + 2S(n/2),$$

and the solution is $S(n) = \Theta(n \log n)$.

Solution #4

The previous D&C solution can be improved by making the recursive calls do some more work for us. Thus, given a subarray $A[i..j]$, we will design a recursive function which returns as a function result or via reference parameters:

- the sum (σ^*_{ij}) of a segment with maximal sum in $A[i..j]$
- the indices ($imax, jmax$) delimiting such a segment
- the sum $\overrightarrow{\sigma}^*_{ij}$ of a segment starting at i with maximal sum among all the segments of the form $A[i..k]$ with $k \leq j$
- the index $\overrightarrow{\jmath}$ in which the above segment ends
- the sum $\overleftarrow{\sigma}^*_{ij}$ of a segment ending at j with maximal sum among all the segments of the form $A[k..j]$ with $k \geq i$
- the index $\overleftarrow{\imath}$ in which the above segment starts
- the sum of the segment σ_{ij}

If we have all this information for $A[i..m]$ and for $A[m+1..j]$, obtained by recursive calls in each of the two subarrays of $A[i..j]$, we can get the sought information for $A[i..j]$ quite easily and efficiently:

Algorithm 3 A divide& conquer algorithm

```
// Returns the maximum sum of a segment of A[i..j]
// and the beginning and end of such a segment
// that is, A[imax..jmax] has maximal sum among all segments of
// A[i..j]
int maximum_sum_segment(const vector<int>& A, int i, int j,
                        int& imax, int& jmax) {
  if (i+1 >= j) { // i+1 <= j ==> one or two elements
    if (i == j) { // one element
      imax = jmax = i;
      return A[i];
    } else { // two elements, i+1 == j
      if (A[i] < A[j] and A[i] + A[j] < A[j]) {
        imax = jmax = j; return A[j];
      } else if (A[i] > A[j] and A[i] + A[j] > A[i]) {
        imax = jmax = i; return A[i];
      } else {
        imax = i; jmax = j; return A[i] + A[j];
      }
    }
  } else { // i + 1 < j ==> three or more elements
    int m = (i+j)/2;
    int i_left, j_left;
    int sum_max_left = maximum_sum_segment(A, i, m, i_left, j_left);
    int i_right, j_right;
    int sum_max_right = maximum_sum_segment(A, m+1, j, i_right, j_right);
    int i_across, j_across;
    int sum_max_across =
          maximum_segment_across(A, i, j, m, i_across, j_across);
    if (sum_max_left <= sum_max_right
        and sum_max_left <= sum_max_across) {
      imax = i_left; jmax = j_left; return sum_max_left;
    }
    if (sum_max_right <= sum_max_left
        and sum_max_right <= sum_max_across) {
      imax = i_right; jmax = j_right; return sum_max_right;
    }
    // sum_max_across is maximum
    imax = i_across; jmax = j_across;
    return sum_max_across;
  }
}
```

1. σ^*_{ij} is the maximum of $\sigma^*_{i,m}$, $\sigma^*_{m+1,j}$ and $\overleftarrow{\sigma}^*_{i,m} + \overrightarrow{\sigma}^*_{m+1,j}$.
2. depending on which was the maximum value above, the indices $(imax, jmax)$ will be the ones from the recursive call in $A[i..m]$, from the recursive call in $A[m+1..j]$, or $imax = \overleftarrow{\imath}$ from the call with $A[i..m]$ and $jmax = \overrightarrow{\jmath}$ from the call with $A[m+1..j]$.
3. $\overrightarrow{\sigma}^*_{ij}$ is the maximum of $\overrightarrow{\sigma}^*_{i,m}$ and $\sigma_{i,m} + \overrightarrow{\sigma}^*_{m+1,j}$
4. the index $\overrightarrow{\jmath}$ is the value given by the recursive call on $A[i..m]$ or the value given by the call on $A[m+1..j]$, depending on which case gives us the maximum $\overrightarrow{\sigma}^*_{ij}$
5. $\overleftarrow{\sigma}^*_{ij}$ is the maximum of $\overleftarrow{\sigma}^*_{m+1,j}$ and $\overleftarrow{\sigma}^*_{i,m} + \sigma_{m+1,j}$, and the index $\overleftarrow{\imath}$ is either coming from the call on $A[m+1..j]$ or from $A[i..m]$ accordingly
6. $\sigma_{ij} = \sigma_{i,m} + \sigma_{m+1,j}$

In this new D & C algortihm there are many more results to be computed in the non-recursive part of the function than in our previous D&C algorithm, but the non-recursive cost will be $\Theta(1)$ instead of $\Theta(n)$. Then the recurrence is

$$S'(n) = \Theta(1) + 2S'(n/2),$$

and now the solution is $\Theta(n)$, which can't be improved—any solution to the problem has cost $\Omega(n)$ in the worst-case as every element in A must be inspected at least once.

Solution #5

We are now going to develop an alternative solution based on *dynamic programming*. Its cost will be $\Theta(n)$ which is not better than that of our last solution; however, the final algorithm will be considerably simpler, and very elegant too.

Define $\lambda_m \equiv \overleftarrow{\sigma}^*_{0,m} = \max_{0\le k\le m} \sigma_{k,m}$, that is the maximum sum of any segment ending at m. Then we have that the solution we are looking for is

$$\sigma^*_{0,n-1} = \max_{0\le m<n} \lambda_m.$$

Here comes the optimality principle:

$$\lambda_m = \begin{cases} A[0] & \text{if } m = 0, \\ \max\{A[m], A[m] + \lambda_{m-1}\} & \text{if } m > 0. \end{cases}$$

If we have an array L and we want $L[m] = \lambda_m$ at the end of the execution it is clear that we just need to fill it from left to right, and we will do it in linear time. To recover a segment that achieves the maximal sum we just need to store for each m the value $I[m]$ of the index that gives the maximum (m or $I[m-1]$). This will give, for each m, the left limit of the segment that achieves the maximum. The right limit of the segment that achieves the maximal sum is the value m for which λ_m is maximal.

A moment's thought makes it clear that neither of the two arrays (L and I) is necessary since to calculate $L[m]$ we only need $L[m-1]$ and likewise $I[m]$ is the same as $I[m-1]$ or changes to m. Moreover, we can maintain a current maximum $\mathit{sum\_max} = \max_{0\leq p<m}\lambda_p$, so that after the last iteration $\mathit{sum\_max} = \sigma^*_{0,n-1} = \max_{0\leq p<n}\lambda_p$. On the other hand, we will find convenient to write the recurrence as

$$\lambda_m = A[m] + \max\{0, \lambda_{m-1}\}$$

Writing down the dynamic programming algorithm, the memoization is accomplished with just a couple of auxiliary variables, so the cost in (extra) space is $\Theta(1)$, the cost of the algorithm is $\Theta(n)$ and both $\sigma^*_{0,n-1}$ and the limits of a segment with such maximal sum can be obtained in a single pass. This is a very fortunate circumstance, not so typical of dynamic programming algorithms. This algorithm is well known in the literature as Kaldane's algorithm for the *maximum sum subarray problem*.

Algorithm 4 A dynamic programming algorithm

```
int maximum_sum_segment(const vector<int>& A,
                        int& imax, int& jmax) {
  sum = A[0]; sum_max = A[0]; imax = jmax = 0;
  for (int m = 1; m < A.size(); ++m) {
    // sum = λ_{m-1}
    // sum_max = max_{0≤p<m} λ_p
    if (sum < 0) {
      sum = 0; imax = m;
    }
    sum += v[m];
    if (sum > sum_max) {
      sum_max = sum; jmax = m;
    }
  }
  return sum_max;
}
```