Design and Analysis of Algorithms (Lab)

Sakaya Milton R
Department of CSE, SSN College of Engineering

Session 1: Maximum Subarray
7 January 2024

1 Maximum Subarray Sum

You are given an array `A[0:n-1]` of integers which may be positive, negative, or zero. Find the maximum sum of elements in a subarray `A[i:j]`. To be precise, find two indices `i` and `j`, $0 \leq$ `i` $\leq$ `j` $\leq$ `n`, that maximizes $\sum_{k=i}^{j}$ `A[k]`. This is referred to as the *maximum subarray sum* of `A`. `A[i:j]` is the *maximum sum subarray*. In the special case where all elements are negative, the maximum sum subarray is empty and maximum subarray sum is zero (sum of elements of an empty subarray is zero).

Example: In the array shown below, the maximum sum subarray is `A[2:5]`, and the maximum subarray sum is 13.

-2	-4	3	-1	5	6	-7	-2	4	3	2
0	1	2	3	4	5	6	7	8	9	10

maximum subarray is `A[2:5]`
maximum subarray sum $= 3-1+5+6 = 13$

1. Design a exhaustive enumeration algorithm and implement it. Name it `max_subarray_sum_cubic`. The algorithm keeps track of the maximum subarray sum in an accumulator. It enumerates all possible subarrays, computes the sum for each subarray, and updates the maximum subarray sum.

   Design an algorithm `sum(A, i, j)` for computing the sum of the items of a subarray `A[i:j]` and use it in `max_subarray_sum_cubic`. You may implement `sum(A, i, j)` as a function and call it in `max_subarray_sum_cubic`. Or you may use the algorithm directly inside `max_subarray_sum_cubic`. Show that the algorithm runs in cubic time, $O(n^3)$.

   Empirical analysis: Observe the running times of `max_subarray_sum_cubic` for increasing sizes of input. To observe the running time of a function for an input size n, call the function

m times with same input and take the average. For each input size, record the actual running time, and the ratio of the actual running time to its asymptotic running time in a table as shown below:

Input size n	Running time	Ratio
10		
50		
100		
1000		
5000		
⋮		

2. Algorithm in Question 1 calculates the sum of each subarray in linear time ($O(n)$). By calculating the sums of all prefixes `A[0:i]`, for $0 \leq$ `i` $\leq n$, of the array once and saving them in a table `prefix_sum`, we can calculate `sum(A, i, j` in constant time, using

   ```
   sum(A, i, j) = prefix_sum[j] - prefix_sum[i]
   ```

   Design an algorithm `max_subarray_sum_quadratic` using this idea and implement it. Show that the algorithm runs in quadratic time, $O(n^2)$. Do an empirical analysis of the algorithm.

3. Design a divide and conquer algorithm for the maximum subarray sum: Divide the array `A[low:high]` into halves `A[low:mid-1]` and `A[mid:high]` where `mid` is roughly in the middle of `low` and `high`.

   Three cases arise and the largest of the three is the maximum sum subarray of `A[low:high]`:

   1. The maximum sum subarray is entirely in the first half `A[i:mid-1]`;
   2. The maximum sum subarray is entirely in the second half `A[mid:j]`;
   3. The maximum sum subarray overlaps both halves.

   The first two cases can be solved through recursive calls on each of the halves. For the third case, compute the maximum sum suffix of the first half `A[i:mid-1]` and the maximum sum prefix of the second half `A[mid:j . A[i:j]` is the maximum sum subarray overlapping the two halves. We should note that the maximum sum suffix (or maximum sum prefix) of a subarray can be found in linear time.

   Design algorithm `max_subarray_sum_linearithmic` using divide-and-conquer and implement it. Show that the algorithm runs in linearithmic time, $O(n \log n)$. Do the empirical analysis of the algorithm.

4. Suppose an algorithm maintains the maximum suffix sum of all subarrays `A[0:j]` for `j` $= 0 \ldots$ `n` $- 1$ in a table `max_suffix_sum`. We can compute them in a single sweep of the array.

   ```
   max_suffix_sum[j] = max (max_suffix_sum[j-1] + A[j], 0)
   ```

   Using table `suffix_max`, we can compute the maximum subarray sum of `A[0:j]` in a single sweep of the array using

   ```
   subarray_max (A, j) = max (subarray_max (A,j-1),  suffix_max [j])
   ```

   Construct algorithm `max_subarray_sum_linear` using this idea and implement it. Show that the algorithm runs in linear time, $O(n)$. Do empirical analysis of the algorithm.

5. Implement algorithm `max_subarray_sum_linear` with a single sweep of the array `A`. Since `subarray_max (A, j)` calculation needs only `suffix_max [j])`, there is no need to use an array `suffix_max` to explicitly store the maximum suffix sums of `A[0:1], A[0:2, ..., A[0:j-1]]`. Instead, maintain the maximum suffix sum of `A[0:j]` in a single variable and use it to calculate `subarray_max (A, j)`.