Design and Analysis of Algorithms — Lab

R S Milton, C Aravindan, T T Mirnalinee
Department of CSE, SSN College of Engineering

Session 2: Empirical Analysis

1 Maximum Subarray Sum

You are given an array `A[0:n]` 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 $\sum_{k=i}^{j-1}$ `A[k]`. This is referred to as the *maximum subarray sum* of `A`. `A[i:j]` is the maximal subarray. In the special case where all elements are negative, the maximal subarray is empty and maximum subarray sum is zero (sum of elements of an empty Subarray is zero).

Example: In the array shown, the maximal subarray is `A[2:6]`, 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[2:6]`
maximum subarray sum = 13

1. Design a exhaustive enumeration/brute-force 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. 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

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

time, and the ratio of the actual running time to its asymptotic running time in a table as shown below:

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 empirical analysis of the algorithm.

3. Design a divide and conquer algorithm for maximum subarray sum: Divide the array `A[low:high]` into halves `A[low:mid]` and `A[mid:high]`. Find the maximum subarrays of the two halves recursively. The maximum subarray of `A[low:high]` is one of the three:

   1. Maximum subarray of `A[low:mid]`,
   2. Maximum subarray of `A[mid:high]`,
   3. Maximum subarray overlapping both halves.

   To calculate the maximum subarray overlapping the two halves, compute the maximum suffix `A[i:mid]` of the first half and the maximum prefix `A[mid:j` of the second half. `A[i:j]` is the maximum subarray overlapping the two halves.

   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 empirical analysis of the algorithm.

4. Suppose an algorithm maintains the maximum suffix sum of all subarrays `A[0:j]` for $0 \leq$ `j` $\leq$ `n` in a table `suffix_max`. We can compute them in a single scan of the array.

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

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