Design and Analysis of Algorithms — Lab

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

Session 3: Empirical Analysis

1 Maximum Subarray Sum

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

Example: In the array shown, the maximum sum subarray is `A[2:6]`, and the maximum 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. Implement a brute-force algorithm. The algorithm constructs all possible subarrays and computes the sum for each subarray. It maintains the maximum subarray sum in an accumulator.

Algorithm: MaxSubarraySumCubic A[0:n]

Input: An n-element array A[0:n] of integers.
Output: The maximum subarray sum of array A.

```
1 m ← 0
2 for i ← 0 to n do
3 |   for j ← i to n do
4 |   |   if sum (A, i, j) > m then  m ← s
5 |   end
6 end
7 return m
```

2. Implement an algorithm that uses dynamic programming to compute sum of a subarray.

Algorithm: MaxSubarraySumQuadratic a[0:n]

Input: An n-element array a[0:n] of integers.
Output: The maximum subarray sum of array a.

```
1  S[0] ← 0
2  for i ← 1 to n do
3  |   S[i] ← S[i-1] + A[i]
4  end
5  m ← 0
6  for j ← 1 to n do
7  |   for k ← j to n do
8  |   |   s ← S[k] - S[j-1]
9  |   |   if s > m then  m ← s
10 |   end
11 end
12 return m
```

3. Implement an algorithm that does not examine all the subarrays.

Algorithm: MaxSubarraySumLinear A[0:n]

Input: An n-element array A[0:n] of integers.
Output: The maximum subarray sum of array A.

```
1 M[1] ← 0
2 for t ← 2 to n-1 do
3 |   M[t] ← max (0, M[t-1] + A[t])
4 end
5 m ← 0
6 for t ← 0 to n-1 do
7 |   m ← max (m, M[t])
8 end
9 return m
```

4. Implement an divide and conquer algorithm: Divide the array `A` into halves. Three cases arise:

   1. The maximum sum subarray is entirely in the first half;
   2. The maximum sum subarray is entirely in the second half;
   3. The maximum sum subarray contains elements from both halves.

   The first two cases are solved through simple recursive calls on the halves. For the third case, compute the maximum sum subarray that starts at the first element of the second half. Symmetrically, compute the maximum sum subarray that ends with the last element of the first half. The union of these two intervals is the maximum sum subarray that contains elements from both halves.

5. Perform empirical analysis of run time of all the three versions: Execute the functions for different values of sequence size n and tabulate the results (note that each entry should be an average over several runs, say m). Perform ratio analysis with well known complexity classes to confirm the growth rate of running times of all the three versions.

6. (OPTIONAL) Predict running times (in suitable units) for very large values of n that are not in your table. Execute and find real running times for those values of n (average over several runs). What is your prediction error? Use paired t-test to show that deviations in your predictions are not significant.