CHAPTER 4

Q1. What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

Ans.

Firstly let us begin by explaining the algorithm in our hands as steps in order to fully understand the output in that special case.

Our algorithm goes as follows:

1- the base case is when there is an array of one element; return its value and index.

2- otherwise, divide the array into two sub-arrays.

3- find the maximum sub-array of each of two sub-arrays recursively.

4- find the maximum sub-array that has some elements in the two sides, contiguous elements crossing midpoint.

5- return the maximum of the three categories: completely left , completely right or array crossing midpoint.

Thus when all the elements are negative, it will try to find the maximum sub-array, that is the least negative sub-array. Now, because there are no elements with value ≥ 0, any sub-array of elements more than 1 would be ***less*** than the single element. Thus, we expect algorithm to return a single element, which is least negative.

Q2. Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in Θ(n2) time.

Ans. Firstly let us try to explain what this brute force algorithm demand. The algorithm will try to evaluate every possible pair from the array and check that whether it can be a possible solution.

So, out algorithm goes as follow:

1. Loop for every possible start index s ϵ [1,n]
2. Within each iteration, initialize the sub-array with sum 0.
3. Loop for every possible ending index e ϵ [s ,n], each iteration within this loop represent a unique sub-array.
4. Update the sub-array sum.
5. If it’s bigger than the previous maximum record. Update it.

4. finally, return the maximum record after the end of outer loop.

Pseudocode:-

BRUTE-FORCE-FIND-MAXIMUM-SUBARRAY(A)

n = A.length

max-sum = -∞

for l = 1 to n

sum = 0

for h = l to n

sum = sum + A[h]

if sum > max-sum

max-sum = sum

low = l

high = h

return (low, high, max-sum)

Let us analyze this algorithm

T(n) = Θ(1) +

=

= Θ(1)

= Θ(1) n (n+1) -

= Θ(1) = Θ(n2)

Q3. Implement both the brute-force and recursive algorithms for the maximum subarray problem on your own computer. What problem size n0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n0. Does that change the crossover point?

Ans. Continuing explanation of algorithm from above question.

On my computer,n0​  is 25. After input size greater than 25 recursive approach outcast in performance.

If the algorithm is modified to use divide and conquer and the brute-force approach when n  is less than threshold, the hybrid algorithm performs better than the other two algorithms. Till 28 hybrid model oscillates but after 29 it outcast other at every value of n.

What I find interesting is that if we set n0 = 20 ​and used the mixed approach to sort elements, it is faster than both.

Q4. Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 00. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

Ans. There can be many approaches to do so

a. If the function return a negative value. Then simply discard it and return zero.

b. We can also run a linear search in the array. If we can’t find any positive element in the array. Return the function with sum as 0.

c. We can also modify initial values of left sum and right sum as 0 instead of -∞ in the algorithm.

Q5. Use the following ideas to develop a non-recursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of A[1…j] extend the answer to find a maximum subarray ending at index j+1 by using the following observation: a maximum subarray of A[1…j+1] is either a maximum subarray of A[1…j] or a subarray A[1…j+1], for some 1 ≤ i ≤ j+1. Determine a maximum subarray of the form A[1..j+1] in constant time based on knowing a maximum subarray ending at index j .

Ans.

M = −∞

low, high = null

curr\_sum = 0

temp\_low = 0

**for** i from 1 to A.length do

curr\_sum + = A[i]

if curr\_sum > M then

low = temp\_low

high = i

M = curr\_sum

**if** curr\_sum < 0 then

curr\_sum = 0

temp\_low = i + 1

**return** (M,low,high)

Q1. Use Strassen’s algorithm to compute the matrix product.

1 3 6 8

7 5 4 2

Ans.

Q2. Write pseudocode for Strassen’s algorithm.

Ans.

STRASSEN(A, B)

n = A.rows

if n == 1

return a[1, 1] \* b[1, 1]

let C be a new n × n matrix

A[1, 1] = A[1..n / 2][1..n / 2]

A[1, 2] = A[1..n / 2][n / 2 + 1..n]

A[2, 1] = A[n / 2 + 1..n][1..n / 2]

A[2, 2] = A[n / 2 + 1..n][n / 2 + 1..n]

B[1, 1] = B[1..n / 2][1..n / 2]

B[1, 2] = B[1..n / 2][n / 2 + 1..n]

B[2, 1] = B[n / 2 + 1..n][1..n / 2]

B[2, 2] = B[n / 2 + 1..n][n / 2 + 1..n]

S[1] = B[1, 2] - B[2, 2]

S[2] = A[1, 1] + A[1, 2]

S[3] = A[2, 1] + A[2, 2]

S[4] = B[2, 1] - B[1, 1]

S[5] = A[1, 1] + A[2, 2]

S[6] = B[1, 1] + B[2, 2]

S[7] = A[1, 2] - A[2, 2]

S[8] = B[2, 1] + B[2, 2]

S[9] = A[1, 1] - A[2, 1]

S[10] = B[1, 1] + B[1, 2]

P[1] = STRASSEN(A[1, 1], S[1])

P[2] = STRASSEN(S[2], B[2, 2])

P[3] = STRASSEN(S[3], B[1, 1])

P[4] = STRASSEN(A[2, 2], S[4])

P[5] = STRASSEN(S[5], S[6])

P[6] = STRASSEN(S[7], S[8])

P[7] = STRASSEN(S[9], S[10])

C[1..n / 2][1..n / 2] = P[5] + P[4] - P[2] + P[6]

C[1..n / 2][n / 2 + 1..n] = P[1] + P[2]

C[n / 2 + 1..n][1..n / 2] = P[3] + P[4]

C[n / 2 + 1..n][n / 2 + 1..n] = P[5] + P[1] - P[3] - P[7]

return C

Q3. How would you modify Strassen’s algorithm to multiply n x n matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time Θ(nlg 7).

Ans. Strassen’s algorithm can be applied to n x n matrix multiplications where n is not an exact power of 2 by padding the operands with 0’s. Let m = 2k such that 2k - 1 < n < 2k (m equals 2⌈lgn⌉). Create m x m matrices A’ and B’ by padding A and B respectively.

Applying Strassen’s algorithm, the resulting matrices C’, A’ and B’ appear as follows,

where C’ is the matrix product of A’ and B’:

To obtain the product, we simply extract the matrix C from C’.

The runtime for this method is Θ(mlg7). Since 2k – 1 < n, it follows that m < 2n. Therefore,

the runtime becomes Θ((2n)lg7) = Θ(2 lg7 ⋅ n lg7) = Θ(n lg7).

e.g.

C’= C 0 A’ = A 0 B’ = B 0

0 0 0 0 0 0

Q4. What is the largest k such that if you can multiply 3 x 3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply n x n matrices in time Θ(nlg 7)? What would the running time of this algorithm be?

Ans. Strassen’s algorithm takes the approach of a recursive multiply with a base case condition of 2 x2 matrices. We are asked to apply an algorithm using a base case of a 3 x 3 matrix and told that it will take k multiplication.

Consider the recursions:-

Strassen T(n) = 7T(n/2) + Θ(n2)

3 x 3 : T(n) = kT(n/3) + Θ(n2)

Now , case 1 of master’s theorem applies and recursive term dominates here. Concentrating on the 3 x3 recursion, we want to solve for k such that the number of multiplies will be less than nlg 7. We do so as follows:

Θ(nlg 7) ≥ Θ(n[lg]k) , so

nlg 7 ≥ nlg[3]k

lg 7 ≥ log3 k

Solving we get 21.8499 ≥ k.

Therefore the largest possible k , while doing better than Θ(nlg 7) using 3 x 3 method is 21.

Q5. V. Pan has discovered a way of multiplying 68 x 68 matrices using 132,464 multiplications, a way of multiplying 70 x 70 matrices using 143,640 multiplications, and a way of multiplying 72 x 72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen’s algorithm?

Ans.

Applying same approach as we did in above method replacing 3x3 by

a. 68x68

b. 70 x 70

c. 72 x 72

in all 3 cases. Or in general using T(n) = Θ(nlogm k) where k is no. of multiplication in base case and m is base case.

We get

T1­(n) = Θ(nlog68 132464) = n2.795128

T2(n) = Θ(nlog70 143640) = n2.795123

T3(n) = Θ(nlog72 155424) = n2.795147

Thus T3 > T1 >T2

Hence T2 is the fastest one. (70 x 70 )

70 x 70 using 143,640 is better than Strassen’s algorithm.

Q6. How quickly can you multiply a k n x n matrix by an n x kn matrix, using Strassen’s

algorithm as a subroutine? Answer the same question with the order of the input

matrices reversed.

Ans.

a. (kn x n)(n x kn) produces a kn x kn matrix. This produces k2 multiplications. Hence using Strassen method we can do it in Θ(k2 nlg7).

b. (n x kn)(kn x n) produces a n x n matrix. This produces k multiplications. Hence using Strassen method we can do it in Θ(knlg7).

Q7. Show how to multiply the complex numbers a + bi and c + di using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component ac - bd and the imaginary component ad + bc separately.

Ans. There can be many ways to do it.

a. Let

P1  = (a+b)c

P2 = b(c+d)

P3 = (a-b)d

Now real part can be P1 – P2

Img part can be P2 + P3

b. You can also use Strassen’s algorithm to solve this.

Think of a,b,c,d as

a 0 c d

b 0 0 0

Multiplying both above will give us

ac ad

bc bd

So all we need to calculate is

C11 – C22 [ Real Part]

C12 + C21 [ Img part]

Doing further calculation you will find that we only need multiplication for P1, P3 and P7. Hence 3.

Then we can calculate C11 , C12 , C21 , C22. And then finally

(ac-bd) = C11 – C22

(ad + bc) = C12+ C21