Sean
Blanchard
12/8/20
COMP 354

Final: Written

3-1) Proof of correctness

list J of Jobs - each takes the same amount of time to complete
↳ Schedule jobs in S[0 to n]
↳ Each Job occupies a single index in the array (because they all take same amount of time to complete)
Each job $j \in J$ has:
• deadline j.deadline from 0 to n, denoting the last index in S at which it can be scheduled
• profit j.profit, denoting the profit or gain from completing job j.

☆ The goal is to maximize the sum of profits of the jobs scheduled in S. |J| can be greater than |S|, so it might not be possible to schedule all jobs. No penalty for undone jobs.

To test the job scheduling algorithm I will plug in values into my List "J" of jobs
(Job, deadline, Profit)

J[] = {{'J1', 1, 2}, {'J2', 1, 5}, {'J3', 2, 7}, {'J4', 3, 9}, {'J5', 1, 1}, {'J6', 3, 2}, {'J7', 3, 12}};
n = sizeof(J[]);

↓

visually filled

Job	deadline	Profit
J1	1	2
J2	1	5
J3	2	7
J4	3	9
J5	1	1
J6	3	3
J7	3	12

Algorithm: Sort J in non-increasing order of profits
After sorting

Job	deadline	Profit
J7	3	12
J4	3	9
J3	2	7
J2	1	5
J6	3	3
J1	1	2
J5	1	1

- First select J7, as it is completed within its deadline. This will give us the max profit.
- Second J4 cannot be selected because deadline is over.
- J3 is selected and is executed within the deadline
- J2 is selected and is executed within the deadline
- J6 cannot be selected because deadline is over
- J1 cannot be selected because deadline is over
- J5 cannot be selected because deadline is over

hence sequence <J7, J3, J2> give max profit @ deadline

$12 + 7 + 5 = 24$ ✓ true ✓

3-2) Time Complexity

Derive time complexity of the following algorithm.

x = 2 // 2 bit

y = 2 // 2 bit

Recurrence Relation:

The algorithm calls itself 3 times on the input twice T(n/2)

Since the return statement is actually bit shift it takes O(n) time.

$$T(n) = T(n/2) + T(n/2) + T(n/2) + O(n)$$

Then we must use the Master Theorem for divide-and-conquer recurrences. to get the asymptotic analysis (Big O) for recurrence relations.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a = 3,\ b = 2,\ f(n) = O(n)$

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

a = number of subproblems in the recursion

b = factor by which subproblem size is reduced in each recursive call.

Satisfy case 1:

$\log_b a = \log_2 3 = 1.584963 > c$ ✓

$T(n) = O(n^{\log_b a}) = \boxed{O(n^{\log_2 3})}$ ✓

$\boxed{\text{Time complexity is } O(n^{\log_2 3})}$

33) Algorithmic design

Write an algorithm which takes a sequence of real numbers R as an input and effeciently finds a maximum sum S_{ij}.

Kadane's Algorithm

$R = \{1, -2, \underline{1, 2, 4}, -9, 6\}$

expected: 7 (1+2+4)

We could essentially start from position 0 in the array, and run Kadane's Algorithm which looks for all positive contiguous segments of the array, and keeps track of maximum sum contiguous segments among all positive segments. Each time a positive sum is retreived, compare with maxSoFar if it is greater than maxSoFar, update value.

maxSoFar = 0
maxEnding = 0

Loop for each element of array
(a) maxEnding = maxEnding + a[i]
(b) if (maxSoFar < maxEnding)
maxSoFar = maxEnding
(c) if (maxEnding < 0)
maxEnding = 0
return maxSoFar

~~MORE EFFICIENT~~
a) stays same
b) if (maxEnding < 0)
maxEnding = 0;
c) elseif (maxSoFar < maxEnding)
maxSoFar = maxEnding

R = {1, -2, 1, 2, 4, -9, 6}

i = 0, a[0] = 1, maxSoFar = 1 0 < 1 ✓
i = 1, a[1] = -2, maxSoFar = 1 -2 < 1
i = 2, a[2] = 1, maxSoFar = 1 ← get re updated w/ 1
i = 3, a[3] = 2, maxSoFar = 3 2 < 3 ✓
i = 4, a[4] = 4, maxSoFar = 7 4 < 7 ✓
i = 5, a[5] = -9, maxSoFar = 7 -9 < 7 ← stays 7 → reset maxEnding
i = 6, a[6] = 6, maxSoFar = 7 6 < 7
$\boxed{7}$

This works but not fully optimized, If we compare maxSoFar with maxEnding if maxEnding > 0, also handle case when all numbers in array are negative. This will not compare for all elements, but only when maxEnding > 0