Elijah Chou

Dr. Michelangelo Grigni

CS 326

2/26/2022

HW 3

1. (a): Use the book sample input:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| di | 4 | 2 | 4 | 3 | 1 | 4 | 6 |
| wi | 70 | 60 | 50 | 40 | 30 | 20 | 10 |

After phase 1, the schedule is as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| T[j] | 4 | 2 | 3 | 1 |  | 7 |  |

Note that all filled spots in T are “early.” After phase 2, the schedule is as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| T[j] | 4 | 2 | 3 | 1 | 5 | 7 | 6 |

Both 5 and 6 are late since they were added during phase 2. This also means that this is not early-first because the task 5 is late but still appears before the early task 7. It can also be seen here that the early tasks are not in deadline order because task 4, which has a deadline of 3, appears before task 2, which has a deadline of 2.  
  
 (b): Suppose that w1 ≥ w2 ≥ … ≥ wn, and ai = i, then both this algorithm and the textbook algorithm look at the tasks in the same order. This should be enough to show that they find exactly the same early tasks. Then, it must be shown that whenever the textbook algorithm adds ai to A, the alternate algorithm also schedules ai in Phase 1, and vice versa.

To deduce this, it can be reasoned that if the alternative algorithm schedules ai in Phase 1, then there is an early schedule for A ∪ {ai}, so the textbook algorithm will also add ai to A. On the other hand, if the alternative algorithm doesn’t schedule ai with a deadline of di, this is because the previously scheduled jobs in A have taken all time slots from i to di. This means that Nt(A) = t, Nt(A ∪ {ai}) > t for t = di, so A ∪ {ai} is not independent, and the textbook algorithm would not have scheduled ai to A.

1. (c): Given a set of coin denominations of {1, 3, 4} and given the problem of finding the minimum number of coins needed for a total change of 6, then the greedy solution would choose {1, 1, 4}, but the actual optimal solution should be {3, 3}  
     
   (d): Let coinsRequired and coin be empty arrays of length v, and any attempt to access them at indeces in the range of -max(S) and -1 should return infinity.

CHANGE(S, v):

for i from 1 to v:

bestCoin = null

bestNum = infinity

for c in S:

if coinsRequired[i – c] + 1 < bestNum:

bestNum = coinsRequired[i – c]

bestCoin = c

let totalChange be empty set

iter = v

while iter > 0:

add coin[iter] to totalChange

iter = iter – coin[iter]

return change

1. There is a Q = (S1, S2) where S1 and S2 are empty stacks.  
     
   Enqueue(Q, x):

S1.push(x)

Size(Q):  
 return size(S1) + size(S2)

Dequeue(Q):

If S2 is empty:

While S1 is not empty:

S2.push(S1.pop())

Return S2.pop()  
 For bounding the time, it should be enough to count both push and pop operations. The potential function should be as follows: Φ(Q) = 2 \* size(S1). Φ should equal to 0 initially and Φ≥0 all the time.

|  |  |  |
| --- | --- | --- |
|  | Actual Cost | Amortized Cost |
| Enqueue | 1 | 3 |
| Dequeue | 1 or 1 + 2 \* size(S1) | 1 |

Based on this table, it can be seen that all operations take amortized time O(1).

1. Suppose S is an unordered linked list of integer values with size S.N. With the SELECT function listed in section 9.3, it is possible to implement DELETELARGERHALF(S) in time with at most c\*N time for N ≥ no where c and no are constants. It is also possible to execute INSERT(S, x) in O(1) time when just inserting to the front of the list. To define the potential function, it is as follows: Φ(S) = 2 \* c \* N where N is the size of S. With this, we get Θ(1) + 2c = Θ(1) as the amortized cost of the INSERT function and Θ(N) – 2 \* c \* ≤ Θ(N) – c \* N for the amortized cost of DELETELARGERHALF. For small N where N < no the amortized cost of DELETELARGERHALF is actually ≤ CN – CN = 0. Therefore, all operations are implemented in amortized cost of O(1).
2. (a): One could do this by doing a binary search in each empty Ai, from i = k – 1 down to 0, until *x* is found. The worst case time should be Θ(k + (k – 1) + (k – 2) + … + 1) = Θ(k2) = Θ(lg2n)

(b): The insert function could be done as follows, if there is an array T that can:  
  
INSERT(x):

T[0] = x

T.N = 1

k = 0

while Ak is not empty:

merge all items from Ak and T into array T

Ak.N = 0; T.N = 2k+1

k=k+1

copy all 2k items from T to Ak

T.N = 0

In this insert function, the total time for m of these insertions, when starting from empty, is O(m lg m). This is because m is the final N, so k is approximately equal to lg m. In order to bound the time of the operations, it should be enough to bound the total size of all the merges that occurred in the function. Each item x is involved in at most lg N merges, since each merge takes x to a larger array, so therefore:

It can also be deduced here that the amortized time per insert is Θ(lg m).