William Randall

Final

805 (6) 986

```
make_list ( k, n, m, helpers):
    initialize list solution
    let final = k[n][m]
    for i=n ; where i > 0 and final >0 ; i-- :        if value above it is the same then it was not included
        if final is k[i-1][m] then continue
        else
            Add R_{i-1} to solution        // value above was different so we add it to the list
            final = final - P_{i-1}        // decrement final by the profit
            m = m - time_required(R_{i-1})  // decrement months by the time needed for R_{i-1}
        end if
    end for
    return pair (helpers, solution)
end def
```

Problem 1 (35 pts). A software company has n requests for developing software systems $\{R_1, R_2, \ldots, R_n\}$. For each system R_i, the company will get $\$p_i$ and will need m_i months for the development of R_i. On all projects, the company can work not more than m months and it is necessary to finish one system before starting another one. However, if the company will hire 10 more programmers, the time for each project will be half of the initial time but the company will have to pay each programmer $\$a$ per month. Note that the company either hire all additional programmers for all m months or does not hire them at all. Design an efficient algorithm allowing the company to get the largest profit developing these systems and estimate the complexity of this algorithm.

n requests $\{R_1, R_2, \ldots, R_n\}$

$\forall R_i$ company gets $\$p_i$

+ needs m_i months

will hire or won't hire

cost $\$10am + \frac{m_i}{2}$ months

knapsack

Weight = m_i months

Value = $\$p_i$

if hired total profit

$m_i/2 + (\Sigma p_i) - 10am$

Let R be requests, n = # of requests, m = total amnt of months

initialize 2d array k [n+1][m+1]

```
for i in range(n):
    for j in range(m):
        if (i is 0) or (j is 0):
            k[i][j] = 0    // if no requests or no time profit is = 0
        elif time_required(R_{i-1}) ≤ j :
            k[i][j] = max(P_{i-1} + K[i-1][j - time_required(R_{i-1})], K[i-1][j])   // get max of adding or not adding the request
        else   // if time required is more than current alloted time
            k[i][j] = K[i-1][j]   // don't add the request
        end if
    endfor
endfor
```

initialize 2d array kh [n+1][2*m+1] // 2*m+1 so all values are whole numbers

```
for i in range(n):
    for j in range(2*m): // by halving all times by 2 this essentially doubles the total time
        if (i is 0) or (j is 0):
            kh[i][j] = 0
        elif time_required(R_{i-1}) ≤ j :
            kh[i][j] = max(P_{i-1} + kh[i-1][j - time_required(R_{i-1})], kh[i-1][j])
        else
            kh[i][j] = kh[i-1][j]
        end if
    endfor
endfor
```

↓ continue

↓ continue

```
if k[n][m] > kh[n][2*m] - 10*a*m: // max of profit without programmers vs with programmers minus their cost
|   return make_list(k, n, m, False) // helper fxn seen above
else
|   return make_list(kh, n, 2*m, True)
end if

// end
```

Explanation for Problem 1

This is essentially 2 knapsack problems. First you create a 2d array k which is n+1 by m+1. It is 0 if there is no profit or if there is no months. Then we loop through k and if the current project will take longer than the amm ount of months given then we will just take the profit from the part of the array which is above the current position (because this represents what the profit is if we had the same time limit but did not include the current project). Otherwise, we will take the max profit between doing the current project and not doing the current project. We will make this whole process faster by looking backwards in the array at previous answers to save time. We repeat the above process to make the kh array which is for if we hired the additional programmers. To account for the time of each project taking half the amm ount of time, I doubled the total time limit for this array. Finally we check whether the profit without the extra programmers + their cost is more than if we hire them. Then we return by using the helper function make_list which walks back through either 2d array and creates the solution list of which requests will be taken. Then we return a pair of a boolean value which is true if we hired the programmers and false otherwise and the list of all requests which we took

The time complexity for this algorithm would be $O(n \cdot m)$ because of the two loops that make the two arrays.

$O(n \cdot m) + O(2 \cdot m \cdot n) \Rightarrow O(m \cdot n)$

William Randall 805 167 986

Problem 2 (35 pts). A computing facility has to perform n processing jobs $\{J_1, J_2, \ldots, J_n\}$. Each J_i demands three types of computers: at first, a supercomputer for the time r_i, then a regular desktop for the time q_i and after this, a specialized computer for the time t_i. The facility has only one supercomputer that can perform only one job after another but it has more than n desktops and more than n specialized computers. Design an efficient algorithm allowing finding the schedule to do all jobs with the minimal time and estimate the complexity of this algorithm.

n processing jobs $\{J_1, J_2, \ldots, J_n\}$

quantity

1) suc: r_i 1
2) rd: q_i more than n
3) spc: t_i more than n

each job must
① use supercomputer for r_i
② use regular desktop for q_i ← these can be parallelized because there are more than n of each
③ use specialized computer for t_i ← these can be parallelized because there are more than n of each

the minimum ammount of time necessary to complete all jobs is Σr_i
We need to minimize the ammount of time taken after the super computer finishes all jobs.
To do this we should start all jobs with large q_i & t_i's first so they can have the longest running times in the regular desktops & the specialized computers.
We would order the jobs from biggest to smallest based on their combined q_i & t_i.

```
find_best_order(J):
    initialize list called Order of size n which will contain pairs
    for i in range(n):
        Order[i] = pair(q_i + t_i, J_i)
    end for
    Use merge sort to sort array Order from largest to smallest based on the q_i + t_i value in the pair
    return Order
end fxn
```

The time complexity will be $O(n) + O(n \log n) \Rightarrow \boxed{O(n \log n)}$

$O(n)$: first loop; $O(n \log n)$: Merge Sort

We created an array called Order out of pairs which was then sorted based on each index's value of $q_i + t_i$. Then we returned the sorted list. This will minimize the running time after the supercomputer finishes processing all requests because it will start the tasks with long parallelizeable steps first.

William Randall 805 167 986

Problem 3 (30 pts). Let G be an arbitrary set of n companies and their brand values are also given. Some of these companies compete with one another. A set A of companies from G is perfect if no two of them compete with one another. The problem MPP of finding a perfect set of companies with the maximal total brand value (that is, the sum of their brand values) is considered intractable although solvable in the general case. That is why a special case of G is considered.

Assume that the structure of G is cyclic (that is, each company competes exactly with two another companies) and find an efficient algorithm for solving MPP estimating the complexity of this algorithm. An example of a cyclic structure with brand values as weights is given in Figure 1.

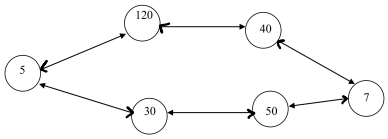

Figure 1. Brand values are given in millions.

```
competition(G):
    if n=0 then return 0
    if n=1 then return G[0]
    if n=2 then return max(G[0], G[1])
    initialize list D length n
    D[0] = G[0]
    D[1] = Max(G[0], G[1])
    for i=2 in range(n):
        D[i] = max(G[i] + D[i-2], D[i-1]) // either add company or dont
    end for
    Initialize list A
    for (i=n; i>0; i--):
        if D[i] = D[i-1] then continue
        else
            Add G_i to A list
        end if
    end for
    return A
```

$O(2 \cdot n)$

This algorithm has a time complexity of $O(n)$ because it only needs to loop through twice

This is an example of dynamic programming where we make a list D which stores values of our previous computations & continuously checks if it is worth it to include the current company or to just use the previously calculated best value. Then I loop back through the list and if two adjacent values are the same then it is not included but if they are different then we include it in A.