[03–60–454] Assignment 3

Stephen Nusko
103693282

March 31, 2015

1 Question 1

1.1 Problem Description and Notation

You have a machine to do n jobs $wi, w2, \ldots, wn$. Job w_i, $1 \leq i \leq n$, requires t_i units of service time and you will receive a service charge p_i if the job is done by time d_i (the deadline) but receive nothing (I.E. The service is free) if the job is done after time t_i. The machine can only do one job at a time. Moreover, the machine must do each job w_i uninterruptedly for t_i consecutive time units.

We now define some helpful notation and sets.
Let $T = \{s | s$ is a possible ordering of all jobs$\}$
Let $R(s)$ be the set of jobs which we *receive* a service charge. I.E. $R(s) = \{w_i|$ We receive $w_i's$ service charge for a schedule $s \in T\}$
Let $R(s)'$ be the complement of $R(s)$.
I.E. All the jobs we do not receive a service charge for for a schedule 's'.
Let $profit(s)$ be the profit of a given schedule $s \in T$.
Let $s = (w_i, w_j, \ldots, w_k)$ denote a schedule $s \in T$ where w_i is the first job, and so on where $i, j, k \in [1, n]$ with $i \neq j \neq k$
Let $idx(w_i)$ be the index for job w_i in the schedule under discussion.

1.2 Part a

We are to prove that there exists an optimal solution in which all paying jobs come before non-paying jobs.

1.2.1 Prove the existence of an optimal solution

First we will prove that there is at least one optimal solution.
Let A be the set of ordered pairs such that the first element is a schedule $s \in T$ and the second element is the $profit(s)$
I.E. $A = \{(s, p) | s \in T \wedge p = profit(s)\}$
Choose s' as a linear ordering of jobs I.E. $s' = (w_1, w_2, \ldots, w_n)$.
Since s' is a possible ordering of all jobs $s' \in T$
Therefore s' has a profit we denote by $x = profit(s)$
Therefore $(s', x) \in A$
Therefore $A \neq \emptyset$
If we order the set A by the second element in the pair, then by the well-ordering theorem there exists a maximum element.
Let (s_o, p_o) be the maximum element.
Then $\forall (x, y) \in A, y \leq p_o$
$\therefore s_o$ maximizes the profit received.

$\therefore s_o$ is a optimal schedule, and therefore an optimal solution.

1.2.2 Prove the existence of an optimal solution in which all paying jobs come before non-paying jobs

With the proof of an optimal solution existing, we now proceed and prove the existence of a optimal solution in which all paying jobs come before non-paying jobs.
Since we have already proven the existence of an optimal solution s_o we separate our proof into two cases based on the ordering of s_o, one where all paying jobs come before non-paying jobs, and secondly where they do not.
Case 1: All paying jobs come before non-paying jobs.
Since s_o is an optimal solution in which all paying jobs come before non-paying jobs, we conclude that there exists a optimal solution in which all paying jobs come before non-paying jobs.

Case 2: Some non-paying jobs come before paying jobs.
Let $W(s) = \{w_i | w_i \in R(s)' \wedge (\exists w_j \in R(s), idx(w_i) < idx(w_j))\}$
I.E. $W(s)$ is the set of jobs for which we do not receive a service charge, and that there is some paying job that comes after them in the ordering for a schedule 's'.
Since by the assumption of the case we are in, there are some non-paying jobs that come before paying jobs in s_o. $W(s_o) \neq \emptyset$.
Therefore let $x \in W(s_o)$ denote the farthest right job in $W(s_o)$, and let $r \in R(s_o)$ denote the farthest right job that comes after x.
We are guarantied the existence of r, because W requires a paying job to come after any of it's members, and we know that $W(s_o) \neq \emptyset$.
We shift all jobs to the right of x to the left, and then place x into the old position of r. Generating a schedule s'_o. For illustration let $r_i(s')$ represents the i^{th} job we receive a service charge for, and $r_j(s')'$ is the j^{th} job we don't receive a service charge for,
and then imagine that $s' = (r_1(s')', r_1(s'), r_2(s'), r_2(s')')$.
Then $r_1(s')'$ is the is the only element of $W(s')$ and is therefore the farthest right member of $W(s')$, while $r_2(s')$ is the farthest right paying job after $r_1(s')'$.
We shift all the jobs after $r_1(s')'$ one to the left and then place $r_1(s')'$ in the place of the farthest right paying job in this case $r_2(s')$.
This results in $(s')' = (r_1(s'), r_2(s'), r_1(s')', r_2(s')')$.

Now that we have illustrated with a simple example we proceed with proving for all cases. We will show that the $profit(s'_o) = profit(s_o)$.
Let $B = \{idx(w_i) | idx(w_i) \leq idx(r)$ before the shift$\}$
Let $A = \{idx(w_i) | idx(w_i) \leq idx(x)$ after the shift$\}$
Consider all jobs to the right of r, we note that since x only moves to r's position, and the $\sum_{i \in B} t_i = \sum_{i \in A} t_i$, Nothing has changed and all those jobs remain non-paying.
We now consider the set of jobs that have an index i that is $\leq idx(r)$ and $> idx(x)$.
Each job in this set, must be a paying job, this is because if there was a non-paying job then it would be to the right of x but the left of r, and would actually be the farthest right non-paying job that has a paying job after it, which contradicts how we picked x and r.
Now since it is moving earlier in the queue while they maintain their relative ordering to the other jobs in the set. Let old_t_k represent the time the k^{th} member used to complete, it now completes at an earlier time represented by new_t_k with $new\_tk \leq old\_k$.
However each job in this set completed before the shift, so for all k, $old\_t_k \leq d_k$.
By transitivity $new\_tk \leq d_k$.
Therefore each job still completes, and $profit(s'_o)$ is unchanged.
Now we consider the job x, this job did not complete before it's deadline, and has now been moved later in the queue. It now completes at a later time, but since that is still greater then its deadline it remains belonging to the set of non-paying jobs.

Therefore by the arguments above we can say $R(s'_o) = R(s_o) \wedge R(s'_o)' = R(s_o)'$.
That is to say, $profit(s'_o) = profit(s_o)$.
If the set $W(s'_o) = \emptyset$ then we have proven that there exists a optimal solution in which all paying jobs come before non-paying jobs.
Otherwise if the set $W(s'_o) \neq \emptyset$ we can repeat the steps described above to generate s''_o, again if $W(s''_o) \neq \emptyset$ we can repeat the steps again and again till all non-paying jobs have been moved. This will take $|W(s_o)|$ steps.
Therefore let $s_o^{|W(s_o)|}$ be the solution after $|W(s_o)|$ steps. Then by the above arguments $profit(s_o^{|W(s_o)|}) = profit(s_o)$, and all paying jobs in $s_o^{|W(s_o)|}$ come before non-paying jobs.
Therefore there exists a optimal solution in which all paying jobs come before non-paying jobs.

Therefore in all cases we can prove the existence of a optimal solution in which all paying jobs come before non-paying jobs. □

1.3 Part b

We are to prove that there exists an optimal schedule in which every job for which we receive a service charge is sorted in ascending order by the deadline of the jobs.
First we prove the following lemma.

1.3.1 Lemma *

Prove that placing a list of jobs that all complete in time (I.E. Before their deadlines) in ascending order of their deadlines will not cause any jobs to not complete.
We do a proof by induction on the number of jobs misplaced.
Base Case: $n = 0$
In this case no jobs are misplaced in order of their deadlines, therefore all jobs are unmoved, and therefore all jobs still complete.

Base Case: $n = 1$
In this case we have a schedule s in which all jobs complete, but one job is not in the correct position according to ascending order based on deadlines.
a Let $Right = \{w_i | w_i$'s position is to the right of $x\}$
Let $Left = \{w_i | w_i$'s position is to the left of $x\}$
Let x denote the job that is out of order. We have two cases either x's deadline $d_x > d_y$ for some jobs $y \in Right$, or $d_x < d_z$ for some jobs $z \in Left$.
Sub Case: $d_x > d_y$ for some jobs $y \in Right$
Remove x from the list
Then all jobs $y \in Right$ will be shifted to the left to to ensure the ascending order is not changed.
They now complete earlier then before, so will continue to complete as before.
let r_i be the i^{th} job in the ordering, and let k be the index of the job right before where x should be.
Then we note that $\sum_{j=1}^{k} t_j + t_x \leq d_k$,
this is because w_k was completed in its old position so the sum of all completion time plus t_x must be less than d_k.
We now reinserting x into its proper position.
This makes the time until x completes equal to $\sum_{j=1}^{idx(x)-1} t_j + t_x$ however this is the same as $\sum_{j=1}^{k} t_j + t_x$.
Since x comes after r_k when sorted by ascending deadlines. We have that $d_k < d_x$.
therefore by transitivity the time that x takes to complete is less than its deadline d_x.
All jobs that come after x are unchanged in position in the list, and will complete at the same time as before.
Therefore all jobs before x complete in time, x completes in time, and all jobs after x complete in time.
Therefore in this sub case ordering the list leaves the set of jobs that compete unchanged.

Sub Case: $d_x < d_z$ for some jobs $z \in Left$.

Let $beginning = \{w_i | idx(w_i) \leq idx(x)\}$

Since all jobs complete in the schedule s, we know that $\sum_{i \in beginning} t_i \leq d_x$ Remove x from the list, and place it in the correct position.

However since we are we are only shifting jobs in the first $idx(x)$ jobs the set beginning will not change.

By transitivity, since by our case assumption $d_x < d_{z_{\min}}$, $\sum_{i \in beginning} t_i \leq d_{z_{\min}}$ where $d_{z_{\min}}$ is the minimum deadline in $Left$ that is greater than d_x.

Therefore since all jobs complete before the minimum deadline $d_{z_{\min}}$ all jobs that have changed sides in relation to x, will still complete.

Any job that is to the left of x after it has been moved, has just remained where it was, and thus will still complete.

Therefore all jobs before x complete in time, x completes in time, and all jobs after x complete in time.

Therefore in this sub case ordering the list leaves the set of jobs that compete unchanged.

Therefore in both sub cases we found that the set of jobs that complete are unchanged by correcting the job that is misplaced in terms of ascending order by deadline.
Therefore the base case of $n = 1$ is proven.

Inductive step: Assume for a ordering with k misplaced jobs $k < n$ we can correct their positions, without effecting which jobs complete.

Now given a ordering with n misplaced jobs we will show that correcting their positions will not effect which jobs complete.
Since the set of all misplaced jobs is not empty. If we order the set by the deadline, the well-ordering theorem guarantees the existence of a maximum element.
Let max = the misplaced job that has the maximum deadline, that is $(\forall d_i) d_i \leq d_{\max}$.
Move max to its correct position.
Any job to the right of max's new position is unmoved, and will therefore will take the same amount of time, and will still complete.
Any job that changed from the right of max to the left of max has moved to the left in the queue, and will now finish earlier.
Thus any jobs that changed sides will still complete.
Since all jobs completed before, any job that remained to the left of max was unmoved, and will thus still complete.
Now we consider if max will still complete.
Let $c_{\max}$ be the job that after the move is one index behind max.
The time it took to complete $c_{\max} was = \sum_{i=1 \leq idx(c_{\max})} t_i$.
Since $c_{\max}$ completed we know that $\sum_{i=1 \leq idx(c_{\max})} t_i \leq d_{c_{\max}}$.
And since after the move $c_{\max}$ comes before max we know that $d_{c_{\max}} < d_{\max}$.
Therefore since the set of the first $idx(c_{\max})$ jobs before the move is unchanged from the set of the first $idx(\max)$ jobs after the move.
We know by transitivity that the time that max now takes is $\sum_{i=1 \leq idx(c_{\max})} t_i$ which is less than $\leq d_{c_{\max}}$ which is less than $d_{\max}$.
Therefore max will still complete.
Therefore after doing one move we know that all jobs to the left of max will still complete, max will still complete, and all jobs to the right of max will still complete.
Therefore after correctly placing one job all jobs will still complete.
Therefore we know have a list of jobs that all complete, with only $n - 1$ misplaced jobs.
Since $n - 1 < n$ by the inductive hypothesis we can correct the order without changing the set of jobs that complete.

Therefore we have proven that the base cases for $n = 0$ and $n = 1$ that we can correct the position of any misplaced jobs, into ascending order of deadline without effecting the set of jobs that complete before

their deadlines.
We have also improved the inductive step where a list of jobs with n misplaced elements can also be corrected into ascending order.
Therefore Lemma * is proven. □

1.3.2 Proof of the existence of a optimal solution in ascending order of deadline

We now prove the existence of a optimal solution in which all paying jobs are sorted in ascending order of deadlines.
In part 1 we proved the existence of a optimal solution s_o where all paying jobs come before non-paying jobs. If we restrict our efforts to just the paying jobs at the beginning of the schedule, we have a list of size $|R(s_o)|$ in which all jobs complete.
By lemma *, we can rearrange the order creating a solution s'_o so that the jobs are sorted in ascending order by their deadlines, without changing the set of completing jobs.
That is $R(s'_o) = R(s_o)$.
It follows from the definition of complement and that a job can not both be paying and non-paying that $R(s'_o)' = R(s_o)'$.
Therefore $R(s'_o) = R(s_o) \wedge R(s'_o)' = R(s_o)'$.
Therefore $profit(s'_o) = profit(s_o)$.
Since s_o is optimal, therefore s'_o is also optimal.
Therefore their exists a optimal solution in which the jobs in which we receive a service charge are completed in ascending order of their deadlines. □

1.4 Part c

1.4.1 English description of the Algorithm

The algorithm is simple. The algorithm focuses on just selecting the order of the jobs we recieve service charges for, all other jobs are just done in any order. We start by solving the best choice to fit in a single time slot. We say that this solotion takes s_t time and s_p profit. From then on the number of time slots keep expanding by one say until we have passed the max deadline of any job or the all jobs have been completed. Each iteration we have t_i time to create a solution to fit in. The algorithm considers two solutions to be considered. The first solution is the previous solution plus any job that fits in $t_i - s_t$. The second solution is the most profitable job that fits in t_i time, plus the previously found optimal solution for any remaining time. Which we can add because part b allows use to reorder jobs that complete in terms of their deadlines. The algorithm will store which ever was the maximum profit of the two as the optimal choice for that t_i. This will repeat as mentioned until the maximum deadline is passed or all jobs have been completed.

1.4.2 Pseudocode

Algorithm max_fit_in_t(W, t, d);
Input: A list of jobs W of size n, and a time t to fit a job in. And the time al jobs will be completed in d.
Output: A pair where the first element is the job which maximizes the profit in time t or is a non-existent job with all fields set to zero, and the second element is the index the job was found in list W or -1 if no job was found.

begin
Let max_job be a job with zero time, zero profit, and zero deadline.
let index = -1
for i = 1 to n do
    if W[i].time $\leq t$ and W[i].deadline > d then
        if index == -1 or W[i].profit > max_job.profit then
            max_job = W[i]
            index = i
        end if

```
    end if
end for
return (max_job, index)
end
```

Algorithm Job_Selection(W);
Input: A list of jobs 1 to n where each $w_i \in W$ has a t_i for time, a d_i for deadline, and a p_i for payment. Given a job w_i we will access their time, deadline, and payment like $w_i.time$, $w_i.deadline$, and $w_i.payment$
Output: A schedule of jobs, that will maximize the profit received from the payments.
We let a schedule have the following form
{
paying_jobs
non-paying_jobs
profit
time_required
}
Where all jobs in paying_jobs, are done before non-paying_jobs, profit is the payments we recieve from paying-jobs, and time_required is the time all paying_jobs take to complete.

```
begin
let max_deadline = 0
let max_timecompletation = 0
for i = 1 to n do
    if W[i].deadline > max_deadline then
        max_deadline = W[i].deadline
    end if
    max_timecompletation = max_timecompletation + W[i].time
end for
if max_deadline > max_timecompletation then
    let counter = max_timecompleation
else
    let counter = max_deadline
end if
let previous_schedule = { paying_jobs = {}, non_paying_jobs = W, profit = 0, time_required = 0 }
let choices = {previous_schudule}
for i = 1 to counter do
    let (best_addition_to_previous, idx_best) =
        max_fit_in_t(previous_schedule.non_paying_jobs, i − previous_schedule.time_required, i)
    let (total_best, total_best_idx) = max_fit_in_t(W, i, i)
    if i − total_best.time > 0
        let next_optimal = i − total_best.time
    else
        let next_optimal = 1
    end if
    if total_best ∈ choices[next_optimal].paying_jobs then
        let (total_best, total_best_idx) =
            max_fit_in_t(choices[next_optimal].non_paying_jobs, i − choices[next_optimal].time_required)
    end if      let prev_profit = best_addition_to_previous.profit + previous_schedule.profit
    let next_profit = total_best.profit + choices[next_optimal].profit      if prev_profit ≥ next_profit and idx_best
≠ -1 then
        move previous_schedule.non_paying_jobs[idx_best] into previous_schedule.paying_jobs
        increase previous_schedule.time_required by best_addition_to_previous.time
        increase previous_schedule.profit by best_addition_to_previous.profit
    else if total_best_idx ≠ −1 then
```

```
        replace previous_schedule.paying with {total_best}
        replace previous_schudule.non_paying with W − {total_best}
        replace previous_schedule.profit with total_best.profit
        replace previous_schedule.time_required with total_best.time
    end if
    choices[i + 1] = previous_schedule
end for
return previous_schedule
end
```

1.4.3 Correctness

max_fit_in_t
First if the list of jobs W passed in is empty that then the for loop will never return, in that case the return value will be a pair of a job with zero time, zero profit, and zero deadline, and an index set to -1 because only line 1 and 2 are run and that is the values set on those lines.
However if there is a job we will prove the following about the loop.
At the end of every iteration m of the loop max_job represents the job that has the most profit that can be completed in time t for the first m^{th} jobs and index is the location of that job in the passed in list $\vee$ they are the sentinal value of a job with zero time, zero profit, and zero deadline, and index is -1.
Proof by induction
Base case: n = 1
In this case at the start of the loop index is -1. We have two cases, if the first job in list takes $\leq t$ time and the deadline is greater then d or the previous statement is fales.
sub-case 1: $W[1].time \leq t \wedge W[1].deadline \geq d$

In this sub-case We will pass the first if statement, and then check the next if statement.

the second if statement will evaulate to true, because index is equal to -1.

we will therefore set max_job to $W[1]$ and index = 1.

Therefore at the end of the first loop max_job is set to the job in W which maximizes the profit in $\leq t$ time and deadline greater then d, and index is the location in the list this job is found.

Therefore at the end of the first loop max_job is set to the job in W which maximizes the profit in $\leq t$ time, and index is the location in the list this job is found $\vee$ they are sentinal value of a job with zero time, zero profit, and zero deadline, and index is -1.

sub-case 2: $W[1].time > t \vee W[1].deadline < d$

In this sub-case, the first if statement will fail and the loop will end.

Therefore the values of max_job is unchanged from a job of all zeros, and index is still -1.

Therefore at the end of the first loop max_job is set to the job in W which maximimizes the profit in $\leq t$ time, and index is the location in the list of this job $\vee$ they are sential value of a job with zero time, zero profit, and zero deadlin,e and index is -1.

Therefore in both sub cases at the end of the first loop max_job is set to the job in W which maximimizes the profit in $\leq t$ time, and index is the location in the list of this job $\vee$ they are sential value of a job with zero time, zero profit, and zero deadlin,e and index is -1.

Inductive step Assume after $k < m$ iterations that at the end of the loop max_job is set to the job in $W[1, \ldots, k]$ which maximimizes the profit in $\leq t$ time, and index is the location in the list of this job $\vee$ they are sential value of a job with zero time, zero profit, and zero deadlin,e and index is -1.
Now consider iteration m, we have two possible results from the previous $m - 1$ iterations. We found a job that takes $\leq t$ time in which case index $\neq -1$, or we didn't and index $= -1$ we consider each case below.
sub-case 1: $index = -1$

In this case we are in exactly the same position as a list of size 1, because our inital values are max_job is all zeros, and index is -1.

Therefore as proven above in the base case this iteration will end correctly.

Therefore at the end of the first loop max_job is set to the job in W which maximizes the profit in $\leq t$ time, and index is the location in the list this job is found $\vee$ they are sentinal value of a job with zero time, zero profit, and zero deadline, and index is -1.

sub-case 2: $index \neq -1$

In this case max_job represents the maximum profit for a job in the first k jobs in W which take $< t$ time, and index is the location of that job in the list W.

We have two cases to consider if job $W[m].time \leq t$ or $W[m].time > t$.

sub-sub-case 1: $W[m].time \leq t \wedge W[m].deadline \geq d$

If $W[m]$.profit <max_job.profit then the second if statement fails, and we are left with max_job and index unchanged, but because $W[m]$.profit is less than max_job.profit.

We have that max_job still represents the maximum profit for a job in $\leq t$ time, and index is its location, and it can be completed in in d time.

however if $W[m]$.profit > max_job.profit, then the if statement is true, and we set max_job to $W[m]$ and index to m. We therefore have that max_job still represents the maximum profit for a job in $\leq t$ time, and index is its location.

sub-sub-case 2: $W[m].time > t \vee W[m].deadline < d$

In this case the first if statement fails, and we go to the end of the loop, leaving max_job and index unchanged.

Therefore we have that max_job still represents the maximum profit for a job in $\leq t$ time, and index is its location, and max_job can be completed in d time.

Therefore in both sub-sub-cases, we found that max_job represents the maximum job for a job in $\leq t$ time, and index is its location, and max_job can be completed in in d time.

Therefore at the end of the loop max_job is set to the job in W which maximizes the profit in $\leq t$ time, and can be completed in d time, and index is the location in the list this job is found $\vee$ they are sentinal value of a job with zero time, zero profit, and zero deadline, and index is -1.

Therefore in both the base case and the inductive step we find that at the end of every iteration m of the loop max_job represents the job that has the most profit that can be completed in time t for the first m^{th} jobs and index is the location of that job in the passed in list $\vee$ they are the sentinal value of a job with zero time, zero profit, and zero deadline, and index is -1.

Job_Selection

We need to prove the properties of two for loops we will split them into lemmas.

Lemma *

Lemma * will prove that the first for loop in Job_Selection will end with max_deadline being the maximum deadline in the list, and max_timecompletation being the sum of all jobs time requirements.

Proof by induction

Base Case: n = 0

The claim is vacuously true.

Therefore in this base case at the end of the loop max_deadline is the maximum deadline in the list, and max_timecompletation is the sum of all jobs time requirements.

Base case: n = 1

In this case if the single job has a deadline > 0 then max_deadline will be correctly updated by the if statement. If the single job has a deadline of 0 then no change of max_deadline is needed because of the assignment on line 1.

The final statement in the loop will add the job's time to the current value which is zero.

Thus max_timecompletation will be the total time needed for all jobs.

Therefore the base cases are proven.

Inductive step Assume for iteration $k < m$ that max_deadline is the maximum deadline of the first k jobs, and max_timecompletation is the sum of all time requirements up to the first k jobs.

Consider the m^{th} iteration. The value in max_deadline represents the maximum deadline for $m-1$ jobs, and max_timecompletation is the sum of all $m-1$ job time requirements.

In the if statement if $W[m]$.deadline > max_deadline, we change max_deadline to it, otherwise we leave it alone.
Therefore max_deadline will represent the maximum deadline for the m^{th} first jobs.
next the loop will add $W[m]$.time to max_timecompletation, and since it was the summation of all the first $m-1$ jobs and we are adding the m job.
Therefore max_timecompletation will represent the sum of all the first m job time requirements.

Therefore both the base case and the inductive step is proven.
Therefore since the length of the list is in $\mathbb{N}$ by lemma * the first for loop in Job_Selection will end with max_deadline being the maximum deadline in the list, and max_timecompletation being the sum of all jobs time requirements.

lemma @
Lemma @ will prove that the end of each iteration i that choices[i] will contain the optimal schedule that maximizes profit in $i-1$ time slots.

Proof by induction
Base Case: n = 0
In this case before the start of the loop previous_schedule is set to have no paying_jobs, no non_paying_jobs and profit and time of zero.
Also choices[1] is set to previous_schedule current value, and since no jobs fits in a time of 0, it is vacuously true that choices[1] is the optimal schedule for $1-1$ time slots.

Base Case: n = 1
In this case the for loop is only ran once, and as seen in the $n=0$ case choices[1] is the optimal solution for 0 time slots.
Since we have proven the correctness of max_fit_in_t, we know that the first line of the for loop we let best_addition_to_previous be the job if it exists that will maximize the profit for previous_schedule.non_paying_jobs (that is all jobs) in $1-0$ time slot.
On the next line we let total_best be the job that will maximize the profit of all jobs, in 1 time slot.
We set prev_profit to choices[1].profit + the job that optimizes for 1 time slot.
We also set next_profit to choices[1].profit + the job that optimizes for 1 time slot.
Since these are the same the profit is also the same.
If a job was found that fits in 1 time slot, we update previous_schedule to include that job in paying_jobs, and increase the profit and time required by the job's profit and time respectively.
Since we have only 1 time slot and we have the optimal job for one time slot.
Therefore previous_schedule is updated to be the optimal choice for 1 time slot. The loop will then add the current value of previous_schedule to choices[i+1].
Since i = 1, choices[2] is set to the optimal solution for 1 time slot, we have that choices[n] $\forall n$ is the optimal solution for $i-1$ time slots.
Therefore this base case is proven.
Inductive step Assume for iteration $k < m$ that all values of $1 \leq i \leq k$ of choices[i] is the optimal solution for $i-1$ time slots.

We select best_addition_to_previous to be the job that maximizes the left over time from the previous solution.
We let total_best be the job that maximizes all possible jobs in the current iterations time slots.
We fill up any remaining time with the optimal solution for $m-$ total_best.time.
We then compare the profit of the two, and choose the one that generates the maximum profit.
This value is then added to choices at choices[m + 1].
This value is optimal because the optimal solution has to be in one of those two choices.
From part a we know that we only need to consider the jobs which can be completed in a time slots.
Since every choices[i] can be completed in i time, since max_fit_in_t chooses based on deadline of iteration.
and we know that the job we added has a deadline $\geq$ than the current iteration.

From part b we know we can rearrange the order of the jobs into order of ascending deadline.
Which means either adding a single job that fits within the deadline onto the previous solution, or taking a maximum profitable job that fits in the total current time slot and then adds the optimal solution of the remaining time, which from part b can be rearranged so all jobs complete.
Therefore we have choosen choices[m] to be the optimal solution for $m-1$ time slots.
Therefore both the base cases, and the inductive step is proven. □

1.4.4 Time Complexity

max_fit_in_t
The first two lines each take constant time so say c_1 operations.
The for loop runs from 1 to n so takes n operations.
Each statement inside the for loop takes constant time say c_2.
So the for loop takes in total $c_2 n$ time.

Therefore the total time this function takes is $c_1 + c_2 n \leq (|c_1| + |c_2|)n$
Therefore max_fit_in_t $\in O(n)$

Job_Selection
The first two lines each take constant time so say c_1 operations.
The for loop runs from 1 to n so takes n operations.
Each statement inside the for loop is constant time so say c_2 operations.
Therefore the loop in total takes $c_2 n$ time.
The if else statements plus the initilization of previous_schedule takes constant time say c_3 operations.
The for loop runs from 1 to counter, and counter was set to the $\min(\max(d_i), \sum_{i=1}^{n} t_i)$ so it runs $\min(\max(d_i), \sum_{i=1}^{n} t_i)$ times.
Since max_fit_in_t $\in O(n)$ each call to the function in the first two lines take a constant times n operations. Say $c_4 n + c_5 n$ operations.
The selection of the variable next_optimal through the if else statement takes constant time thus c_6 operations.
The if statement takes cn operations naively to check the in operator. While the call to max_fit_in_t also takes some constant times n.
Therefore the if statement takes $c_7 n$ operations. All remaining statements besides one are constant time, say c_8 operations.
The statement that is not constant time is replacing non_paying_jobs with $W - \{\text{total\_best}\}$ this would take $n-1$ operations to copy the list over however we can say it takes $c_9 n$ time because it belongs to $O(n)$.
Therefore the for loop runs in $\min(\max(d_i), \sum_{i=1}^{n} t_i) * (c_4 n + c_5 n + c_6 + c_7 n + c_8 + c_9 n)$

Therefore the total time this function takes is
$c_1 + c_2 n + c_3 + \min(\max(d_i), \sum_{i=1}^{n} t_i) * (c_4 n + c_5 n + + c_6 + c_7 n + c_8 + c_9 n) \leq (|c_1| + |c_2| + |c_3| + |c_4| + |c_5| + |c_6| + |c_7| + |c_8| + c_9|)n * \min(\max(d_i), \sum_{i=1}^{n} t_i)$
Therefore job_Selection $\in O(n * \min(\max(d_i), \sum_{i=1}^{n} t_i))$
a

2 Question 2

Given n brands of cookie B_i, $1 \leq i \leq n$. Each brand of cookie B_i has a price p_i and comes with a coupon that carries a value c_i for another brand of cookie. If the coupon is for brand B_j, then $c_i \leq p_j$. Moreover, no two different brands of cookie carry their coupon for the same brand of cookie. The objective is to buy one bag of cookie of each brand (with a coupon if available) so that the total price is minimum.
Present a greedy algorithm which when presented with: (i) p_i, $1 \leq i \leq n$, (ii) c_i, $1 \leq i \leq n$, and (iii) $B_i^{'}$, $1 \leq i \leq n$, such that c_i is for brand $B_i^{'}$, determines an order to buy the cookies so that the total price is

minimum. You algorithm must run in $O(n)$ time.

2.1 English Description

The algorithm finds the coupon of minimum value and buys the brand the minimum coupon is for, it then proceeds to buy based on the coupon that Brand carries. If all the cookies are not yet purchased then it repeats this process. That is it finds the minimum value of a coupon for a cookie we haven't bought, and buy the cookie that the minimum coupon was for.

2.2 Pseudocode

Algorithm buy_cookies(B)
Input: A list of n brands of cookie B, each brand b_i comes with a price p_i and a coupon c_i which tells us which brand b_j to buy and the value we get off b_j.
Output: A ordering of purchases which minimizes the total price paid with coupon values taken off.

```
begin
let purchased = 0
let order = {}
let list_of_cycles = {}
let num_cycles = 1
while B ≠ ∅
    let first = B[1]
    add first to list_of_cycles[num_cycles]
    remove first from B
    let second = brandOf(coupon(first))
    while second ≠ first do
        add second to list_of_cycles[num_cycles]
        remove second from B
        let second = brandOf(coupon(second))
    end while
    num_cycles = num_cycles + 1
end while
for i = 1 to num_cycles do
    let curr_cycle = list_of_cycles[i]
    let min = couponValue(curr_cycle[1])
    let min_brand = brandOf(coupon(curr_cycle[1]))
    k = 1 to size(curr_cycle) do
        if min > couponValue(curr_cycle[k]) then
            min = couponValue(curr_cycle[k])
            min_brand = brandOf(coupon(curr_cycle[k]))
        end if
    end for
    let first = min_brand
    add first to order
    let second = brandOf(coupon(first))
    while second ≠ first do
        add second to order
        let second = brandOf(coupon(second))
    end while
end for
return order
```

end

2.3 correctness

After the first while loop we have separated the list of brands into connected sub-components.
Because no brand can hold a coupon of itself there must be at least 2 brands of cookies.
We start with that as a base case.
Proof by induction
Base Case: n = 2
In this case we grab the first element in the first statement.
It is added to the current cycle, and is removed from B.
We set second to the next brand of the coupon entering the while loop.
We add second to the current cycle, and remove it from B.
We update second to point to the brand that the current coupon is for.
In this case that is back to the first brand, therefore the loop ends.
the only thing left in the outer loop is increasing num_cycles to 1.
Therefore at the end of the loop B is empty, list_of_cycles contains each cycle that can be found in B is now in list_of_cycles.
Therefore we have separated the list of brands into connected sub-components.

Inductive step Assume that after iterations $k < m$ brands that the list has separated the brands into connected sub-components.

At the start of iteration m we have separated the list of brands into $m - 1$ connected componts, and num_cycles is equal to $m - 1$.
Since B is not empty we grab the first brand in the list.
We then add first to the current cycle and then remove it from B. let the second equal the brandOf(coupon(first)).
That is second is the brand that the coupon of first is for.
We then add second to the current cycle and then remove it from B.
We then update second to point to the brand that the current coupon is for.
If this is not the first one we added to the cycle we repeat this process until then.
This will cause the entire cycle that is generated by first to be removed from B and placed in the current cycle.
The while loop will end, and we will increase the value of num_cycles by 1.
Therefore at the end of loop we will have separated another connected sub-component, and the num_cycles will be equal to m.
Therefore the base case is proven, and the inductive step is also proven.
Therefore the first while loop will separate the list of brands B into num_cycles connected sub-components.

The second while loop is simple in execution, it goes through each cycle grabs the minimum coupon value, and then buys the brand that coupon is for. Then purchases the rest of the cycle.
The first for loop in the cycle will select the minimum value of a coupon seen thus far setting min to it.
at the same time it sets the brand the coupon is for to min_brand.
After that we use the same way that we used in the first while loop to go through the cycles.
but this time we add each brand to the order.
Since we repeat this for all the cycles in the original B. We will order all brands based on the starting with the brand that has the minimum coupon for it.

We will prove for the case of a single cycle of size n that buying the brand that has the minimum coupon value for it. Will maximize the the savings of buying all the brands.

We can think of buying all brands as paying the cost $\sum_{i=1}^{n} brand_i$, and since we can not use the coupon for the first brand we buy.

We can only have the coupon reduce the price up by $\sum_{i=1 \wedge i \neq first}^{n} coupon_i$ where first is the brand we bought to start the cycle.
Therefore we want to minimize $\sum_{i=1}^{n} brand_i - \sum_{i=1 \wedge i \neq first}^{n} coupon_i$.
However we must purchase all brands we can not decrease $\sum_{i=1}^{n} brand_i$, therefore we have to maximize $\sum_{i=1 \wedge i \neq first}^{n} coupon_i$.
Since the only difference we can make is the choice of first brand we buy.
We therefore minimize the value of the coupon we will lose, so the sum of all the coupons we get to use is the highest.
Therefore for a single cycle we must buy the brand that has the minimum coupon value.

Since each cycle must be purchased and has to start with a first, to minimize the cost for all cycles we must purchase each cycles minimum brand savings.
Therefore from our method described above, we will correctly minmize the amount of money we pay.

2.4 time complexity

The first four statements take constant time say c_1
The while loop will run until B is empty, and each loop as proven above removes one connected sub-component.
Therefore the loop will run r times where r is the number of connected sub-components.
Inside the loop the first four statements is constant time so c_2.
The inner loop will go through the current cycle and do constant operations so it takes $c_3|r_i|$ where $|r_i|$ is the size of the connected sub-component.
However since we run over it for all connected sub-components and $\sum_{i=1}^{r} |r_i| = n$
Therefore the inner loop is $c_4 n$ operations.
Therefore the first loop runs $(|c_3| + |c_4|)n$ operations.
The second loop runs through each cycle so r times.
Each loop we go do constant work c_5 plus a loop that executes $|r_i|$ times, doing constant work c_6.
Since we do it r times and it is over all sub-components the the amount of operations in the loop is n.
Therefore the second loop runs in $(|c_5| + |c_6|)n$.
Therefore the function buy_cookies operates in $c_1 + c_2 + (|c_3| + |c_4| + |c_5| + |c_6|)n \leq (|c_1| + |c_2| + |c_3| + |c_4| + |c_5| + |c_6|)n$
Therefore buy_cookies $\in O(n)$. $\square$.

3 Question 3

3.1 English Description

This algorithm is simple, it proceeds by doing a recursive depth first search of the inputted tree. It keeps track of all parent values above a certain node in an array *parents*, if there is $> g(v)$ elements in *parents* then we set the value of that node to the value in *parents* otherwise to undefined.

3.2 Pseudocode

Algorithm replace_tree(T)
Input: A rooted tree T from which every node v has a nonnegative value $g(v)$.
Output: The values of $g(v)$ for all nodes v are replaced with their $g(v)$th parent's value.

begin
let parents = []
let n = 0

replace_tree(T, parents, n)
end

Algorithm replace_tree_internal(T, *parents*, n)
Input: A rooted tree T from which every node v has a nonnegative value $g(v)$, the values of all ancestors *parents*, and the number of ancestors n in *parents*.
Output: The values of $g(v)$ for all nodes v are replaced with their $g(v)$th parent's value.

begin
let $val = g(T)$
$parents[n+1] = val$
$new\_n = n + 1$
if $new\_n - val > 0$ then
    $g(T) = parents[new\_n - val]$
else
    $g(T) = \perp$
end if
for each child v of T do
    replace_tree_internal(v, *parents*, new_n)
end for
end

3.3 Correctness

replace_tree_internal
The function replace_tree_internal takes three arguments, the current vertex, the value of g for all ancestors in order stored in *parents* and the number of ancestors n.
Assuming the correct information passed in, we evaluate the the correctness of replace_tree_internal under these assumptions.
We get the current value of the vertex T.
we store that value into the next position of *parents* and set a variable to the new number of elements in *parents*.
If there was more than $g(T)$ ancestors that is number of elements in minus the value of $g(T)$, we set the current value to the $g(T)$th ancestor, that is $parents[new\_n - g(T)]$.
Otherwise we set the current value to $\perp$.
This correctly updates the current value, and then for each child we call replace_tree_internal with the new_n and the modified parents.
We know that if our assumption of the correct input holds that we have now passed in the current node's value onto all children.
And since the input is a tree we know that there is no cycles and we will only visit each node once.
Therefore replace_tree_internal is correct assuming the ancestor values are passed in are all there and in the correct order, and n is the correct number of elements in *parents*.
□

replace_tree
This function sets up the correct values for the start of the recursive internal function.
Since there are no ancestors of the root, we leave parent empty, and therefore the number of elements in parents is 0 which we store in n. Therefore the function replace_tree then calls replace_tree_internal, which was proven correct assuming that all the values passed in were correct. Therefore since replace_tree_internal is correct, we have replace_tree is correct. □

3.4 Time complexity

replace_tree_internal
Since the input is a tree we know there is no cycles, no disconnected sub components and we only ever call the function on each of its children.
The depth first search will only finish once every vertex is seen.
Therefore the complexity will be of the order of the number of vertexs, that is replace_tree_internal $\in O(|V|)$

replace_tree the first two lines take constant time, and then replace_tree_internal takes $O(|V|)$ time.
Therefore replace_tree also takes $O(|V|)$ time.