1. **(Chapter 4 Problem 13 of the text book)**
   A greedy solution:
   Order the jobs in the order $a_1, .., a_n$ s.t. $\frac{t_1}{w_1} \leq \frac{t_2}{w_2} \leq ... \leq \frac{t_n}{w_n}$.

   Running time: $O(n \log(n))$: (sorting array of length n)

   Prove:
   Based on the proof in the lecture, we have two properties:

   - No idle time
   - No inversion on $\frac{t_i}{w_i}$

   We prove that the optimal schedule has no inversion.
   let S: $a_1, .., a_n$ be an optimal schedule with no idle time (by claim one). If S has no inversion, the claim holds. Assume S has at least one inversion. Then there is pair a_i and a_{i+1} s.t $\frac{t_i}{w_i} > \frac{ti+1}{wi+1}$. Let S' be the schedule optained by exchanging the order of a_i and a_{i+1}. Then S' has one less inversion than S. We prove $l(S') \leq l(S)$:

$$l(S) = \sum_{i=1}^{n} w_i C_i$$

$$l(S') = \sum_{i=1, i \notin i, i+1}^{n} w_i C_i + w_{i+1}(C_{i-1} + t_{i+1}) + w_i C_{i+1}$$

   so that $C_i = \sum_{q=1}^{i} t_q$,

$$l(S') - l(S) = w_{i+1}(C_{i-1} + t_{i+1}) + w_i C_{i+1} - w_i C_i - w_{i+1} C_{i+1}$$

$$= w_{i+1}(C_{i-1} + t_{i+1} - C_{i+1}) + w_i(C_{i+1} - C_i)$$

$$= -w_{i+1} t_i + w_i t_{i+1} < 0$$

   The last inequality comes from

$$\frac{t_i}{w_i} > \frac{ti+1}{wi+1} \Rightarrow w_{i+1} t_i > w_i t_{i+1}$$

2. I Suggest twp greedy algorithm, with similar running times.

   Algorithm 1 (assign jobs to the first resource, then the rest of the jobs to the next resource, etc):

   Sort array S by f_i
   $NI := S$
   for $i = 1$ to p, **do**:
   $EFTF(NI)$
   $P_i = P$
   $NP = NO \setminus P$
   End for;

Return P_1 to P_p.

EFTF$(NI)\{$
$P = \emptyset$
$S = NI$
$j = 1$
while $S \neq \emptyset$, **do** {
pick a_m as the first element in sorted array S, with smallest f_i
$S = \{a_i \in NI | s_i \geq f_j\}$
update $j = m$.
$P = P \cup \{a_j\}$
End while
Rerutn P
$\}$

Prove
For P_1, by the proof of Earliest-Finish-Time-First algorithm in the lecture note, maximum number of jobs is assigned to resource number 1, so $|P_1|$ is maximized

let $S_k^1 = \{a_i \in S | s_i \geq f_k\}$, and $S_k^p = \{a_i \in S \setminus \{\cup_{j=1}^{p-1} P_J\} | s_i \geq f_k\}$ for $p > 1$ and P_j the set of jobs assigned to resource j.

let a_m be the element in S_k^p with smallest f_i. then a_m is included in maximum-size subset of mutually compatible jobs in S_k^p. Let A_k be a maximum-size subset of mutually compatible jobs in $\{a_i \in S | s_i \geq f_k\}$ and a_j be the job in A_k with the smallest f_j. if $a_j \in \cup_{j=1}^{p-1} P_J$, then, theorem is proved. So imagine $a_j \notin \cup_{j=1}^{p-1} P_J$, thus in a_j if $a_j = a_m$ then the theorem is proved. Assume $a_j ! = a_m$. Let $A'_k = (A_k\ a_j) \cup a_m$. Since $a_j \in A_k$ and for any $a_i \in A_k$ with $a_i \neq a_j$, $f_j \leq f_i$ anda_j is compatible with $a_i, f_j \leq s_i$. Therefore and from $f_m \leq f - j, f_m \leq s_i$for every $a_i \in A_k$. Thus, jobs in A'_k are mutually compatible.

Algorithm 2: Assign jobs in order of finish times to the available resource with latest finish time

Sort array S by f_i /* $O(n \log(n))$ */
$F_j := 0$, for $i = 1, ..., p$
$P_j := \emptyset$, for $j = 1, ..., p$
for $i = 1$ to n, **do** /* n iterations */
$l = LFT(F, s_i)$ /* $O(p)$*/
if $l \neq 0$, **then**
$P_l = P_j \cup \{s_i\}$ /* $O(1)$ */
$F[l] = f_i$ /* $O(1)$ */
Break
End if
End for
Return P_j, for $j = 1, ..., p$

Running time is $O(n \times \max\{\log(n), p\})$

LFT(F, s_i)
Input:F, s_i
Output: l (index of job with latest finish time before s_i) { /* $O(p)$*/
$l = 0$ /* $O(1)$*/
for $j = 1$ to p, **do** /* p iterations */
if $F_j < s_i$, and $F_j > F_l$, **then** /* $O(1)$*/
$l = j$ /* $O(1)$*/
end if
return l
}

prove
The algorithm assign a_1 with f_1, the smallest finish time to resource 1. Assume after i iterations, jobs a_{m_j}, $j = 1, ..., p$ are assigned to each resource j and $f_{m_1} \leq f_{m_2} \leq ... \leq f_{m_p}$. At this point the algorithm is optimal.

Assume a_i is the next not-tried job with the smallest f_i. If it is not compatible with any jobs a_{m_j}, then, in there is an optimal solution that it is not assigned to any resource. (prove similar to what discussed in the class)

If a_i is compatible with one resource m_k, then similar prove is used to show in any optimal solution, a_i is assigned to resource m_k, with largest index or largest finish time.

If a_i is compatible with more than one resource m_k and $m_{k'}$ and $f_{m_{k'}} \leq f_{m_k}$ Then we prove that there is an optimal solution in which a_i is assigned to the resource m_k

Assume there is an optimal solution in which a_i is assigned to $m_{k'}$ Then, let a'_i be the next job with the smallest finish time. If $f_{m'_k} \leq s'_i \leq f_{m_k}$, then we could have assigned i' to $m_{k'}$ and a_i to m_k and increase number of matched jobs. In other case, such as $s'_i \leq f_{m'_k} \leq f_{m_k}$ and $f_{m_{k'}} \leq f_{m_k} \leq s'_i$ there is no reduction in the number of assigned jobs by doing so.

3. (Chapter 5 Problem 2 of the text book) 15 points

I base on MergSort algorithm to find the number of significant inversions too. The final answer is as follows:

Initialize: $l = 1, r = n, m = 0$
MeergSort(A, l, r, m);
Return m;

So that:
MergeSort(A, l, r, m) {
Input: Array A and positions l, r
Output: Array A s.t. $A[l], A[l+1], .., A[r]$ are sorted. Number m, which is the number of significant inversion between $A[l]$ and $A[r]$.
if $l < r$ **then**
$p := [(l + r)/2]$
MergeSort(A, l, p, m_l)
MergeSort$(A, p + 1, r, m_r)$

$m = m_l + m_r$ /* $O(1)$ */
Merge(A, l, p, r, m)
$n_l = p - l + 1$; $n_r = r - p$; $L[n_l + 1] = \infty$; $R[n_r + 1] = \infty$;
for $i = 1$ to n_l **do** $L[i] = A[l + i - 1]$;
for $j = 1$ to n_r **do** $R[j] = A[p + j]$;

$i = 1; j = 1$;
for $k = l$ to r **do**
if $L[i] \leq R[j]$ **then** $[A[k] = L[i]]; i = i + 1$
End if
else $A[k] = R[j]; j = j + 1$
if $L[i] > 2 \times R[j]$ **then** $m = m + (n_l - (i - 1))$ /* $O(1)$ */
End if
End for
}
Running time is $O(n \log(n))$, which is similar to original Mergsort. Only two lines are added which run in constant times.

Prove:
Given the number of inversion in $A[l]$ to $A[p]$ and $A[p+1]$ to $A[r]$, we only need to count number of significant inversions between an element in the first array and another element in the second array. If element A$[l']$ is larger than $2 \times A[p']$, then all of the elements after $A[l']$ are also larger than $2 \times A[p']$, thus there will be $(p - (l' - 1))$ significant inversions. The algorithm counts this number.

4. (Chapter 5 Problem 3 of text book) 15 points

I use a divide-and-conquer algorithm to find the elements which is the candidate for being equivalent to at least $n/2$ elements. Then test if this elements is equivalent with at least $n/2$ elements.

Initialize: $l = 1, r = n, m = 0$
$u =$ **MergeCandidate**(C, l, r) { /* $O(n \log(n))$ (see bellow) */
if u= 0 then, print("There isn't more than $n/2$ cards in C that all equivalent to another")
/*$O(1)$*/
else
compare $v(u)$ with $v(C[1])$ to $v(C[n])$ /* $O(n)$ comparison */
if $v(u)$ matches with at least $[\frac{n}{2}] + 1$ elements, **then**
print("there are more than $n/2$ cards in C that all equivalent to u")
else
print("There isn't more than $n/2$ cards in C that all equivalent to another")
end if
end if
};
Total Running time is $O(n \log(n))$

The functions are defined as bellow:
u = **MergeCandidate**(C, l, r)

Input: Array C and positions l, r
Output: Element u as potential candidate between $C[l], C[l+1], .., C[r]$ with which the strong majority of elements are equivalent. {
if $l < r$ **then**
$p := [(l+r)/2]$ / $*$ $\theta(1)$ $*$ /
$u_l =$ **MergeCandidate**(C, l, p)
$u_r =$ **MergeCandidate**$(C, p+1, r)$
$u =$ **Merge**(C, l, p, r, u_l, u_r)

Merge(C, l, p, r, u_l, u_r) { /* $\theta(n)$ */
if $u_l = 0$ and $u_r = 0$, **then** $u = 0$
if $u_l = 0$ and $u_r \neq 0$, **then** $u = u_l$
if $u_l \neq 0$ and $u_r = 0$, **then** $u = u_r$
if $u_l \neq 0$ and $u_r \neq 0$, **then**:
compare $v(u_l)$ with $v(C[l])$ to $v(C[r])$ /* $(r - l + 1)$ comparison , thus $\theta(n)$ */
if $v(u_l)$ matches with at least $[\frac{l-r}{2}] + 1$ elements, **then** $u = u_l$
else $u = u_r$
end if
end if

}

Running time of MergeCandidate() function:
$T(n) = 2T(n/2) + O(n)$, which based on the master theorem, means the algorithm has running time of $\theta(n \log(n))$,

Prove:
Although it has not explicitly mentioned in the question, I assume that the equivalence relationship is transitive. That is, if $v(u_1) = v(u_2)$, and $v(u_1) = v(u_3)$, then $v(u_1) = v(u_2)$, which means if an element is equivalent to two elements as a result of a direct comparison, then these two elements are equivalent.
If an element u is equivalent to more than $n/2$ elements in A, then dividing A into two equal size group C_l and C_r, u must be equal to more than half of the elements in either C_l or C_r or both. Similarly, if dividing each A_l and A_r into two equal size group, then u is equivalent to more than half of each set. By recursion, we can conclude that at each layer, there must be a at least an element u, which is equivalent to more than $n/2$ elements in one of the sets in that layer. Thus, u always exist.

In case there are two majority candidate for the two groups, then at most one of them is the majority candidate for the union of the two group, We use brute force to check which if the first one has majority when the groups are merged. If not, it the second candidate is the only candidate.

If there is no majority candidate for the two groups, then as proved before, there is no candidate from this set, so no strong majority in general.

By induction, u is going to be the only candidate that form strong majority.

Finally, we manually check if u is actually equivalent to more than $n/2$ elements.

5. (Chapter 6 Problem 1 of the text book) 15 points
(a) Consider the following counter-example with $n = 5$ and $w_1 = 1, w_2 = 4, w_3 = 5, w_4 = 4, w_5 = 1$. The algorithm select $S = \{1, 3, 5\}$, and $w(S) = 7$. But we have $S' = \{2, 4\}$ and $w(S') = 8$, which has higher weight than the output of this "heaviest-first" algorithm

(b) Consider the following counter-example with $n = 5$ and $w_1 = 5, w_2 = 2, w_3 = 2, w_4 = 5, w_5 = 1$. The algorithm select $S = \{1, 3, 5\}$, and $w(S) = 8$. But we have $S' = \{1, 4\}$ and $w(S') = 10$, which has higher weight than the output of this algorithm

(c) Consider the following Bellman equation, for the subgraph of $V_i = \{v_1, v_2, .., v_i\}$

$$\text{opt}(i) = \begin{cases} \max\{\text{opt}(i-2) + w_i, \text{opt}(i-1)\} & \text{for } i \geq 3 \\ w_1 & \text{for } i = 1 \\ \max\{w_1, w_2\} & \text{for } i = 2 \end{cases} \quad (1)$$

Input: Array W and scholar n as the length of the path.
Output: element opt[n] and S_t with size n. {
Initialize: $S_{t-2} = 1, opt[1] = w_1$
If $w_1 > w_2$, **then** $S_{t-1} = \{1\}, \text{opt}[2] = w_1$
Else, **then** $S_{t-1} = \{2\}, \text{opt}[2] = w_2$
End if
for $i = 3$ to n, **do** /*n times*/
If $\text{opt}(i-2) + w_i > \text{opt}(i-1)$ **then** /*$O(1)$*/
$\text{opt}(i) = \text{opt}(i-2) + w_i$, /*$O(1)$*/
$S_t = S_{t-2} \cup \{i\}$ /*$O(1)$*/
Else, **then**
$\text{opt}(i) = \text{opt}(i-1)$ /*$O(1)$*/
$S_t = S_{t-1}$ /*$O(1)$*/
End if
$S_{t-2} = S_{t-1}$ /*$O(1)$ (if defined as a passing.)*/
$S_{t-1} = S_t$ /*$O(1)$*/
End for
Return $S_t, \text{opt}[n]$

Running time is $O(n)$ and space is $O(n)$

Prove by induction
It is trivial that for a sub-problem with size $i = 1$ and $i = 2$ the algorithm returns the optimal answer. By induction, assume that for a sub-problem with size $i - 1$ the algorithm returns the optimal solution, we show that for the sub-problem with size i, the algorithm returns the optimal solution too.

Imagine, the last node of the optimal solution of sub-problem with size i is s. if $s = i$, then $i - 1$ must not be included. Thus $i - 2$ can be included. By induction assumption, $\text{opt}(i-2)$ returns the maximum weight for nodes from 1 to $i - 2$. Thus $\text{opt}(i-2) + w_i$ is maximum.

If $s \leq i-2$, we can add node i and since weight of each node is greater than zero, we can add node i to this sequence and have better optimal solution.

If $s = i-1$, then by induction hypothesis, we already now that the optimal solution is $opt(i-1)$, and the algorithm returns $\text{opt}(i) = \text{opt}(i-1)$, thus, again we reach the maximum value.

6. (Chapter 6 Problem 5 of the text book) 15 points

My logic is to find optimal value of word(s) using letters i to j, we can recursive compare the optimal value of word(s) with i to q and $q+1$ to j, for all q. The optimal result is when $i = 1$ and $j = n$.

Definev Bellman equation as:

$$\text{opt}(i,j) = \begin{cases} \max\{\text{opt}(i,q') + \text{opt}(q'+1,j), v(i,j)\} & \text{for } i < j \\ \text{Qty}(y_i) & \text{for } i = j \end{cases} \tag{2}$$

So that:

$$q' = \underset{q}{\text{argmax}}\{\text{opt}(i,q) + \text{opt}(q+1,j) | i \leq q \leq (j-1)\}$$

$$v(i,j) = \text{Qty}(y_i ... y_j)$$

Prove of the algorithm by induction:
It is trivial that when $i = j$ and $i = J-1$, $\text{opt}(i,j)$ is the maximum quality of the segmentation of $y_i..y_j$. Assuming that $\text{opt}(i,j)$ returns the optimal value, we want to show that $\text{opt}(i,j+1)$ is the maximum quality of segmentation of $y_i..y_{j+1}$. By contradiction, imagine $\text{opt}(i,j+1)$ is not the quality of the maximum segmentation. Thus, this segmentation is either one word, or more than one word. If it is one word, then its quality is $v(i,j+1) \leq \text{opt}(i,j+1)$ Which is a contradiction. If it is more than one word, let q be the last letter of the first word. Then the quality of the first word is less than or equal to opti,q and the quality of the rest of the words is less than or equal to opt$q+1, j+1$. Thus the total quality is less than or equal to $\text{opt}(i,j+1)$. Which is a contradiction. We can show $\text{opt}(i+,j)$ is the maximum quality of segmentation similarly.

(Assuming Qty(x) return value in constant time)

Input: function Qty(x), array y.
Output: opt$[1,n], L$ {
let opt be $n \times n$ arrays with $\text{opt}(i,i) = \text{Qty}(y_i)$, let be an $n \times n$ arrays S and let $L = \varnothing$ /* $O(n)$ */
for $m = 1$ to n, **do** /* n times */
for $i = 1$ to $n-(m-1)$ **do** /* maximum of n times */
$j = i + m$
$q' = \text{argmax}_q\{\text{opt}[i,q] + \text{opt}[q+1,j] | i \leq q \leq (j-1)\}$ /* $O(n)$ (max of n possible q) */

$\text{opt}[i,j] = \text{opt}[i,q'] + \text{opt}[q'+1,j]$ /* $O(1)$ */

if $v(i,j) > \text{opt}[i,j]$, **then** /* $O(1)$ */
$q' = 0$

opt$[i, j] = v(i, j)$
end if
$S[i, j] = q'$
end for
end for
DFS(1, n) /* $O(n^2)$ as visits each element of S at most one time.*/
Return opt$[1, n], L$

The running time is $O(n^3)$. We have:

DFS(i, j) {
if $i \neq j$ and $S[i, j] \neq 0$, **then**
$q := S[i, j]$
$L = L \cup q$
DFS(i, q)
DFS$(q + 1, j)$
end if
Total Running time is $O(n^3)$, space is $\Theta(n^2)$

Segmentation is in L, so that if $j \in L$, there is a word ending with y_j and the next word starts from y_{j+1}
opt$[1, n]$ is the maximum quality of the segmentation.

7. 10 points

(a)

$$B = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 3 & 6 & 6 & 6 & 9 & 9 & 9 & 9 & 9 \\ 0 & 0 & 0 & 3 & 6 & 18 & 18 & 18 & 21 & 24 & 24 & 24 \\ 0 & 0 & 0 & 3 & 6 & 18 & 22 & 22 & 22 & 25 & 28 & 40 \\ 0 & 0 & 0 & 3 & 6 & 18 & 22 & 28 & 28 & 28 & 31 & 40 \end{matrix}$$

(b)

j	A (value, weight)
1	$\{(0, 0), (3, 3)\}$
2	$\{(0, 0), (3, 3), (6, 4), (9, 7)\}$
3	$\{(0, 0), (3, 3), (6, 4), (18, 5), (21, 8), (24, 9)\}$
4	$\{(0, 0), (3, 3), (6, 4), (18, 5), (22, 6), (25, 9), (28, 10), (40, 11)\}$
5	$\{(0, 0), (3, 3), (6, 4), (18, 5), (22, 6), (28, 7), (31, 10), (40, 11)\}$

(c) I used random integers between 1 to 100 for values and weights.:

Algorithm	**n**	**Running Time (seconds)**	**Array Elements Created**
Knapsack	100	0.4268860817	505,101
Knapsack1	100	0.0479755402	11,749
Knapsack	500	10.944418199	12,525,501
Knapsack1	500	2.5533194542	257,921