# Problem 1

1. Sort the intervals based on their ending points in non-decreasing order.
2. Initialize an empty set of points, let's call it P.
3. Iterate through each interval in the sorted order.
4. For each interval, if its starting point is greater than the last point in P (this interval does not intersect at any current point we have in P) or P is empty, add its ending point to P.
5. Repeat step 4 for all intervals.

prove optimal:

Define $P*$ to be and optimal schedule that has the fewest number of selected points.

Define $P$ to be the selected points using our greedy.

Suppose $P^*$ and $P$ selected the same first $k-1$ points, that is $p_1^* = p_1, p_2^* = p_2, \ldots, p_{k-1}^* = p_{k-1}$.

Now, let's consider the $k$-th interval $I_k = [s_k, t_k]$ in the sorted order.

Since $P^*$ is optimal, it must intersect $I_k$ as well. There are two possibilities:

1. If $p_k^*$ is already selected by $P^*$ to intersect $I_k$, then $P^*$ and $P$ select the same point for $I_k$.
2. If $p_k^*$ is not selected to intersect $I_k$, then $P^*$ must select another point $p_k^*$ to intersect $I_k$.

In either case, $P^*$ and $P$ select the same number of points up to the $k$-th interval. Therefore, if $P^*$ and $P$ select the same first $k-1$ points, they will also select the same number of points for the first $k$ intervals.

Since this argument holds for any $k$, it implies that $P^*$ and $P$ select the same number of points for all intervals. Hence, $P$ is optimal.

---

# Problem 2

1. Calculate the priority $p_i = \frac{w_i}{t_i}$ of each task based on its importance and processing time. –$O(n)$
2. Sort the tasks in descending order of priority. –$O(n \log n)$
3. Perform the tasks in the sorted order. –$O(n)$

Therefore the total time complexity for this algorithm is $O(n \log n)$

Proof:

Suppose the weight after sorted in priority is $w_1, w_2, \ldots w_n$

$$\frac{w_1}{t_1} \geq \frac{w_2}{t_2} \geq \cdots \geq \frac{w_n}{t_n}$$

$$\sum_{i=1}^{n} w_i C_i = w_1 t_1 + w_2(t_1 + t_2) + \ldots w_i(t_1 + \cdots + t_i) + \ldots w_n(t_1 + \cdots + t_n)$$

If there exist any two tasks that is not in descending order, with $w_m$ and $w_n$ ,and WLOG we suppose $m < n$. Name this schedule $S*$. Name the schedule from our algorithm $S$.

$$\frac{w_m}{t_m} \geq \frac{w_n}{t_n}$$
$$w_m t_n \geq w_n t_m$$

Then the weighted completion time differs from the former one in:

$$S : w_m(t_1 + \cdots + t_m) + w_n(t_1 + \cdots + t_n)$$
$$S* : w_n(t_1 + \cdots + t_{m-1} + t_n) + w_m(t_1 + \cdots + t_{m-1} + t_n + \cdots + t_{n-1} + t_m)$$

$S*$ has a larger weighted completion time.

Therefore, our algorithm is optimal.

---

# Problem 3

Suppose there are $n$ houses and the $i$-th house has money $m_i$ for $0 \leq i \leq n$.

Let $OPT(i)$ be the amount of money the robber gets after robbing the first $i$ houses.

Then,

$$OPT(0) = 0, \quad OPT(1) = m_1$$
$$OPT(i) = \max\{OPT(i-1), OPT(i-2) + m_i\}, \text{ for } i \geq 2$$

Iterate through $OPT$ from $i = 0$ to $i = n$.

The result will be $OPT(n)$.

---

# Problem 4

Suppose the length of $L_1$ is $m$ and the length of $L_2$ is $n$.

Let $L_1[i]$ represent the $i$-th char in $L_1$, and similar for $L_2[j]$ and $L[k]$.

Let $dp[i][j] = 1$ represent that the first $i$ characters of $L_1$ and the first $j$ characters of $L_2$ can be interleaved to form the first $i + j$ characters of $L$, otherwise $dp[i][j] = 0$.

Then,

$$dp[i][j] = \begin{cases} 1 & \text{if } i = 1 \text{ and } j = 1 \\ 1 & \text{if } i > 1 \text{ and } dp[i-1][j] = 1 \text{ and } L[i+j-2] = L_1[i-1] \\ 1 & \text{if } j > 1 \text{ and } dp[i][j-1] = 1 \text{ and } L[i+j-2] = L_2[j-1] \\ 0 & \text{otherwise} \end{cases}$$

The result will be $dp[m+1][n+1]$.

The space complexity is $O(mn)$ since the dynamic programming matrix requires $O(mn)$ space.

The time complexity is also $O(mn)$, since we have to iterate through the dp matrix and each iteration requires $O(1)$ time.

---

# Problem 5

Suppose there are totally $n$ T and F.

Name the propositional logic formula $l$,

Let $dp[i][j]$ represent the number of ways to parenthesize the substring $l(i, j)$.

Let $T[i][j]$ represent the number of ways to **correctly** parenthesize the substring $l(i, j)$.

Then,

$$dp[i][i] = 1$$
$$dp[i][j] = dp[i][k] * dp[k+1][j]$$
$$T[i][i] = 1 \text{ if the corresponding symbol is 'T' or } 0 \text{ if it is 'F'}$$
$$T[i][j] = \sum_{i \le k < j} t(i, k, j)$$
$$t(i, k, j) = \begin{cases} [i][k] * T[k+1][j], & \text{if the term connecting the 2 parts is '}\wedge\text{'} \\ (dp[i][k] - T[i][k])(dp[k+1][j] - T[k+1][j], & \text{otherwise} \end{cases}$$

The result will be $T[1][n]$.

The recursive function $T$ correctly counts the number of ways to parenthesize the formula such that it evaluates to true by considering all possible splits and evaluating the number of ways for each subformula.

The time complexity of the dynamic programming algorithm is $O(n^3)$ since there are $O(n^2)$ subproblems, and each subproblem takes $O(n)$ time to solve.

---

# Problem 6

Let $dp[i][j]$ represent the number of walks of length i from vertex s to vertex j.

$dp[0][s] = 1$

For each $len \le L$, iterate over all edges (u, v) in the graph

- for each edge $(u, v)$, update

$$dp[len][v] = \begin{cases} dp[len][v] + dp[len - weight(u, v)][u] & \text{, if } len - weight(u, v) \ge 0 \\ 0 & \text{, otherwise} \end{cases}$$

The result will be $dp[L][t]$.

The outer loop goes $O(L)$ times, and the inner loop goes $O(m)$ times where $m$ is the number of edges in the graph. Therefore, the time complexity of the algorithm is $O(mL)$.