UNIVERSITY OF WASHINGTON
SCHOOL OF SCIENCE, TECHNOLOGY, ENGINEERING AND MATHEMATICS
DIVISION OF COMPUTING AND SOFTWARE SYSTEMS

**ALGORITHM DESIGN AND ANALYSIS**

**Problem Set 2**

# Greedy Algorithms II, Divide-and-Conquer and Dynamic Programming

**Professor:** Clark Olson
**Student:** Tran Tien Phat
phattt@u.washington.edu

WASHINGTON, FEBRUARY 2024

# Contents

# Problem 1

Consider the graph $G$ in Figure 1:

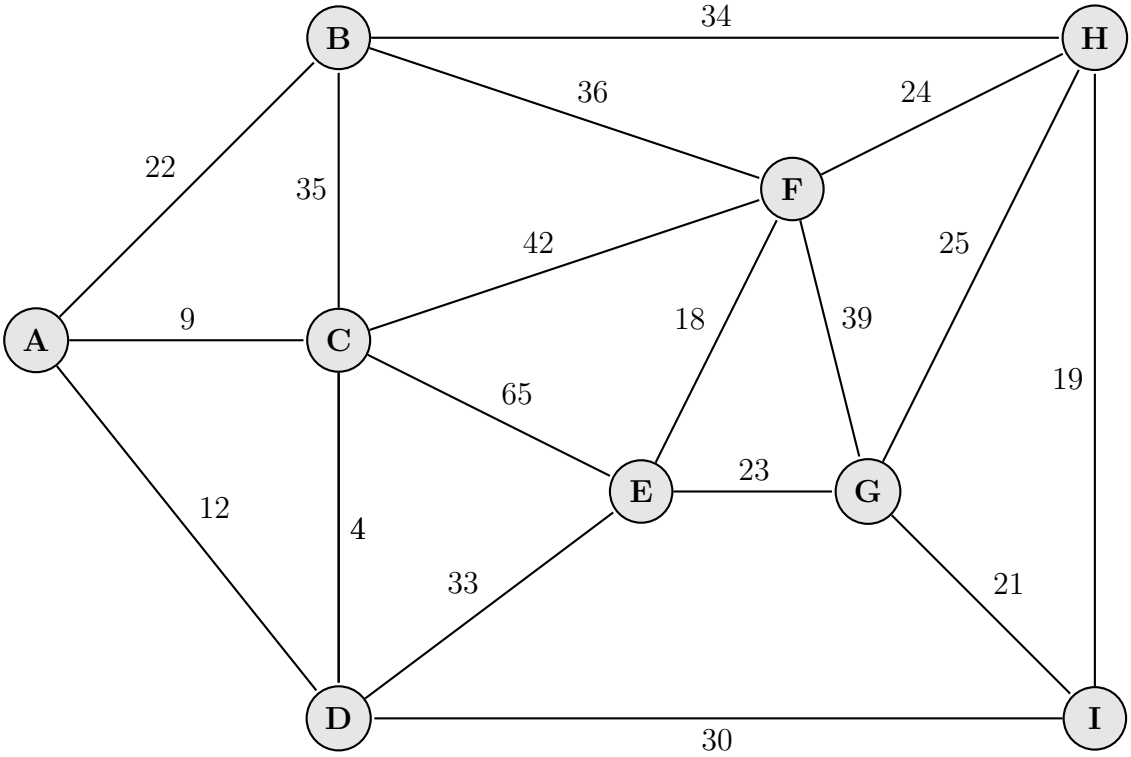


Figure 1: Graph $G$.

(a) What order does Kruskal's algorithm add edges to the minimum spanning tree (include only those edges that are in the minimum spanning tree)? Denote the edges by the two nodes they attach. So, the edge with weight 9 attaching $A$ to $C$ is $AC$. Always put the letters in alphabetical order when denoting an edge.

(b) What order does Prim's algorithm add edges to the minimum spanning tree (again include only edges in the minimum spanning tree)? Start the algorithm at node $A$.

(c) Now assume that every edge is directed from the node lower in the alphabet to the node higher in the alphabet. For example, edge $AC$ leaves $A$ and enters $C$. Use the minimum cost arborescence algorithm to find the minimum cost arborescence rooted at A. List the edges in the result. Show your work.

**Solution.** Consider a weighted graph $G = (V, E)$, where $V$ represents the set of vertices or nodes, and $E$ denotes the set of edges, which are the connections established between these vertices. Each edge $e \in E$ is assigned a specific weight, indicating the cost or value associated with that particular connection.

## Kruskal's algorithm

Kruskal's algorithm adds edges to the minimum spanning tree in increasing order of their weights.

Here is a step-by-step guide to Kruskal's algorithm:

1. Remove Loops and Parallel Edges: Eliminate all loops and parallel edges from the graph to ensure simplicity.
2. Sort Edges: Arrange all edges in non-decreasing order based on their weights to facilitate the edge selection process.
    - $CD - 4$
    - $AC - 9$
    - $AD - 12$
    - ...
3. Edge Selection: Choose the smallest edge from the sorted list. Check if adding this edge forms a cycle with the existing spanning tree. If no cycle is formed, include this edge in the spanning tree; otherwise, discard it.
    - 1st iteration: Add $CD - 4$.
    - 2rd iteration: Add $AC - 9$.
    - 3th iteration: Skip $AD - 12$ because it forms a cycle with $AC$ and $CD$.
    - ...
4. Repetition: Repeat the edge selection process until the spanning tree contains $V - 1$ edges.

Based on the weights provided in Figure 1, the order of edges added to the minimum spanning tree using the Kruskal's algorithm would be:

1. $CD - 4$
2. $AC - 9$

3. $EF - 18$
4. $HI - 19$
5. $GI - 21$
6. $AB - 22$
7. $EG - 23$
8. $DI - 30$

The sum of the weights of the edges in the provided order is:

$$4 + 9 + 18 + 19 + 21 + 22 + 23 + 30 = 146$$

## Prim's algorithm

Prim's algorithm is a method to find a minimum spanning tree in a weighted graph. It starts with a single vertex and adds the smallest edge connecting the tree to a new vertex at each step. The order of edge addition depends on edge weights. For example, starting at node $v$, the first edge added is the smallest one connected to $v$.

Here is a detailed explanation of Prim's algorithm:

1. Initialization: Start with a graph $G = (V, E)$. Choose an arbitrary vertex as the starting vertex of the minimum spanning tree.
2. Edge Selection: Find all the edges that connect the tree to new vertices. Pick the edge with the minimum weight and add it to the tree.
3. Repetition: Keep repeating step 2 until all the vertices are included in the minimum spanning tree.
4. Cycle Check: Ensure that the chosen edge does not form a cycle with the minimum spanning tree obtained so far.
5. Completion: The algorithm stops when all the vertices are included in the minimum spanning tree.

Detailed the algorithm step by step:

1. Start at node $A$. The edges connected to A are $AB - 22$, $AC - 9$, and $AD - 12$. Add the smallest one $AC - 9$ to the minimum spanning tree.

2. Consider the edges connected to $A$ and $C$. They are $AB - 22$, $AD - 12$, $CD - 4$, $BC - 35$, and $CF - 42$. Add the smallest one $CD - 4$ to the minimum spanning tree.
3. Consider the edges connected to $A$, $C$, and $D$. They are ...
4. ...

Based on the weights provided in Figure 1, the order of edges added to the minimum spanning tree using the Prim's algorithm would be:

1. $AC - 9$
2. $CD - 4$
3. $AB - 22$
4. $DI - 30$
5. $HI - 19$
6. $GI - 21$
7. $EG - 23$
8. $EF - 18$

The sum of the weights of the edges in the provided order is:

$$9 + 4 + 22 + 30 + 19 + 21 + 23 + 18 = 146$$

## Edmonds's algorithm

The minimum cost arborescence algorithm is used to find the minimum cost arborescence (a directed, rooted tree where all edges are directed away from the root) in a directed graph. This is similar to finding a minimum spanning tree in an undirected graph, but it is for directed graphs.

- Input:
  - Directed graph $G = (V, E)$ with weights on each edge.
  - Root node $r$ where the arborescence starts.
- Output:
  - Minimum cost arborescence rooted at $r$.

The steps of this algorithm are as follows [1]:

1. Input: The algorithm takes as input a directed graph $G = (V, E)$ where $V$ is the set of nodes and $E$ is the set of directed edges, a distinguished vertex called the root, and a real-valued weight $w(e)$ for each edge $e$.

2. Initialization: Remove any edge from $E$ whose destination is the root. Replace any set of parallel edges (edges between the same pair of vertices in the same direction) by a single edge with weight equal to the minimum of the weights of these parallel edges.

3. Minimum Incoming Edge: For each node other than the root, find the edge incoming to of lowest weight (with ties broken arbitrarily). Denote the source of this edge by $\pi(v)$. For example, in the graph $G$, for vertex $B$, select the edge from $A$ to $B$ with a weight of 22.

4. Check for Cycles: If the set of edges $\pi(v)$ does not contain any cycles, then it is the minimum spanning arborescence. Otherwise, $\pi(v)$ contains at least one cycle.

5. Cycle Contraction: Arbitrarily choose one of these cycles and call it $C$. Define a new weighted directed graph in which the cycle is "*contracted*" into one node.

6. Recursive Call: Find a minimum spanning arborescence of the new graph using a recursive call to the algorithm.

7. Expand Cycle: Since the result of the recursive call is a spanning arborescence, each vertex has exactly one incoming edge. Let $e$ be the unique incoming edge to $v_C$ in the result. This edge corresponds to an edge $e'$ with $\pi(v) = u$. Remove the edge $e'$ from $C$, breaking the cycle. Mark each remaining edge in $C$.

Apply step by step to the graph $G$:

1. Initialization: Start with an empty set of edges for the arborescence node $A$.

2. Edge Selection: For each node other than the root $A$, select the incoming edge with the smallest weight:

   - For $B$, select edge $AB$ with weight 22.
   - For $C$, select edge $AC$ with weight 9.
   - For $D$, select edge $CD$ with weight 4.

- For $F$, select edge $EF$ with weight 18.
- For $E$, select edge $DE$ with weight 33.
- For $G$, select edge $EG$ with weight 23.
- For $H$, select edge $HI$ with weight 19.
- For $I$, select edge $GI$ with weight 21.

3. Cycle Detection: Check if there are any cycles. There are no cycles in the selected edges so the algorithm can stop after this step, otherwise go to the next step.

4. Cycle Contraction: For each cycle found, contract it into a single vertex and adjust the weights of the edges adjacent to the cycle. The weight of each edge entering the cycle is reduced by the weight of the minimum-weight edge in the cycle, and the weight of each edge leaving the cycle is unchanged.

5. Repeat: Go back to the Edge Selection step with the contracted graph.

Based on the weights provided in Figure 1, the order of edges added to the minimum spanning tree using the minimum cost arborescence algorithm would be:

1. $AB - 22$
2. $AC - 9$
3. $CD - 4$
4. $DE - 33$
5. $EF - 18$
6. $EG - 23$
7. $GI - 21$
8. $HI - 19$

These are the edges of the minimum cost arborescence rooted at $A$. The sum of the costs is:

$$9 + 4 + 42 + 33 + 18 + 23 + 21 + 19 = 169$$

# Problem 2

Consider recursive divide-and-conquer algorithms with the following descriptions. For each, determine the running time in Big Theta $\Theta$ notation. You may use the Master Theorem. You do not need to prove your result.

(a) Performs 5 recursive calls on problems half the size of the input and performs $\Theta(n^2)$ work per recursive call.

(b) Performs 2 recursive calls on problems a quarter the size of the input and performs $\Theta(n)$ work per recursive call.

(c) Performs 8 recursive calls on problems half the size of the input and performs $\Theta(n^3)$ work per recursive call.

**Solution.** The Master Theorem is a powerful tool that we can use to solve recurrence relations, which are common in divide-and-conquer algorithms. It applies specifically to relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $n$ is the size of the problem.
- $a$ is the number of subproblems in the recursion.
- $\frac{n}{b}$ is the size of each subproblem. All subproblems are assumed to have the same size.
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions.

The base cases are defined as $T(0) = 0$ and $T(1) = \Theta(1)$. The Master Theorem states that the solution is [2]:

- $T(n) = \Theta(n^{\log_b a})$ if $f(n) = O(n^c)$ where $c < \log_b a$,
- $T(n) = \Theta(n^{\log_b a} \log n)$ if $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$ and $k \geq 0$,
- $T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^c)$ where $c > \log_b a$, if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ and sufficiently large $n$.

By applying the Master Theorem, we can find the time complexity of divide-and-conquer algorithms without the need for extensive analysis or the use of induction. This makes it a valuable tool in the study of algorithm efficiency.

(a) Performs 5 recursive calls on problems half the size of the input and performs $\Theta(n^2)$ work per recursive call.

In this case, the running time $T(n)$ can be expressed as follows:

$$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2)$$

From this equation, we can identify the following parameters:

- $a = 5$,
- $b = 2$,
- $c = 2$.

Here, $log_b a = log_2 5 \approx 2.3219$. Since $f(n) = n^2$ and $2 < log_b a$, this is the first case of the Master Theorem. Therefore, the running time of the algorithm is $T(n) = \Theta(n^{log_b a}) = \Theta(n^{\log_2 5})$.

(b) Performs 2 recursive calls on problems a quarter the size of the input and performs $\Theta(n)$ work per recursive call.

In this case, the running time $T(n)$ can be expressed as follows:

$$T(n) = 2T\left(\frac{n}{4}\right) + \Theta(n)$$

From this equation, identify the following parameters:

- $a = 2$,
- $b = 4$,
- $c = 1$.

Here, $log_b a = log_4 2 = 0.5$. Since $f(n) = n$ and $1 > log_b a$, this is the third case of the Master Theorem. Therefore, the running time of the algorithm is $T(n) = \Theta(f(n)) = \Theta(n)$.

(c) Performs 8 recursive calls on problems half the size of the input and performs $\Theta(n^3)$ work per recursive call.

In this case, the running time $T(n)$ can be expressed as follows:

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^3)$$

From this equation, identify the following parameters:

- $a = 8$,
- $b = 2$,
- $c = 3$.

Here, $log_b a = log_2 8 = 3$. Since $f(n) = n^3$ and $3 = log_b a$, this is the second case of the Master Theorem. Therefore, the running time of the algorithm is $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^3 \log n)$.

# Problem 3

Consider the algorithm to find the closest pair of points in the plane. Suppose you wanted to generalize the algorithm to find the two closest pairs of points in the plane given a set of (unsorted) points $p_1, \ldots, p_n$. Give an algorithm for finding the two distances for this pair. In the step to combine the two subproblems, you must explain why your algorithm is guaranteed to find the correct results. You do not need to specify the best possible algorithm, but your overall algorithm must run in $O(n \log n)$ time. Full pseudocode is not necessary, but your explanation must describe how or why the algorithm works.

**Solution.** Firstly, analyze and identify the inputs and outputs of the problem:

- Input:
  - A set of points in a 2-dimensional space plane, represented as $p_1, p_2, \ldots, p_n$, where each point is a pair of coordinates $(x, y)$.
  - The points are not necessarily sorted.
- Output:
  - Two distances of two pairs of points $(p_i, p_j)$ and $(p_k, p_l)$, such that the distance between $p_i$ and $p_j$ is the smallest distance among all pairs, and the distance between $p_k$ and $p_l$ is the second smallest distance.

To find the two closest pairs of points in the plane, a modified divide-and-conquer algorithm is employed, building upon the classic approach for finding the closest pair of points. The following steps outline the algorithm:

1. **Sort the points according to their $x$-coordinates and $y$-coordinates**
   - Create two separate arrays from the given set of points, one sorted by $x$-coordinates and the other by $y$-coordinates.
   - This sorting can be efficiently performed using a standard sorting algorithm such as merge sort or quicksort. The time complexity for this step is $O(n \log n)$.
2. **Divide the set of sorted points**
   - Partition the sorted set of points into two equal-sized subsets by drawing a vertical line through the median $x$-coordinate (the midpoint of the array sorted by $x$-coordinates).

- Divide the array of points arranged by $y$-coordinate into two equal halves along a vertical line at the $x$-median coordinate.
- This step has a linear time complexity of $O(n)$.

3. **Recursively find the closest pairs in each subset**
   - Recursively find the closest pair of points in each of the two subsets, resulting in two lists for each subset: one sorted by $x$-coordinates and the other by $y$-coordinates.
   - This step takes $T(\frac{n}{2})$ time for each half, so $2T(\frac{n}{2})$ time in total.
4. **Combine using a merge operation**
   - Utilize a merge operation to combine the two pre-sorted lists of points sorted by $y$-coordinates from the recursive calls.
   - Efficiently merge the points within the strip, leading to a time complexity of $O(n)$.
5. **Find the minimum distance within the strip**
   - Iterate through the sorted list of points within the strip of width $2 * d$ (centered at the median $x$-coordinate). Here, $d$ is the minimum distance found in the recursive calls, and the strip extends $d$ units to the left and right of the median.
   - Identify and update the two closest pair of points (if any) within this strip.
   - This step takes linear time, $O(n)$.
6. **Return the minimum distances**
   - Return the two minimum distance among the distances found in the recursive calls and the distances found within the strip.

During the initiation phase, the algorithm performs a sorting operation on both $x$ and $y$ coordinates. This operation has a time complexity of $O(n \log n)$. In the phases that follow, the algorithm merely filters these pre-sorted lists, eliminating the need for any additional sorting.

The total time complexity of the algorithm is:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(n) + O(n) + O(n) \\ &= 2T\left(\frac{n}{2}\right) + 3O(n) \\ &= 2T\left(\frac{n}{2}\right) + O(n) \end{aligned}$$

In order to analyze this using the Master Theorem, identify the parameters:

- $a = 2$,
- $b = 2$,
- $c = 1$.

Here, $\log_b a = log_2 2 = 1$. Since $f(n) = n$ and $1 = log_b a$, this is the second case of the Master Theorem. Therefore, the running time of the algorithm is $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

Therefore, for the provided algorithm, the time complexity is $\Theta(n \log n)$.

The algorithm's effectiveness is grounded in the observation that any pair of points must fall into one of three categories: both in the left half, both in the right half, or one in the left half and one in the right half. The recursive calls handle the first two cases, while the strip check effectively handles the last case. The use of pre-sorted lists and merge operations during the combination step optimizes the algorithm further. Hence, the algorithm is guaranteed to correctly find the two closest pairs of points in the plane.

# Problem 4

Your company is able to take on two types of jobs – long jobs that require two days and short jobs that require one day. You are given a schedule for the upcoming $d$ days that contains one long job and one short job that you may perform each day. If you perform the long job, you are unable to take a new job the next day. If you take a short job, you are able to take a job the next day. You cannot take a long job on the last day of the schedule. Each job has a value, the value of the long job on day $i$ is $l_i$ and the value of the short job on day $i$ is $s_i$. Jobs must be taken on the day they are available or they cannot be performed. Your goal is to develop an algorithm that maximizes the value of all jobs taken.

(a) Give a recurrence relation that expresses the optimal value of the jobs completed through day $i$. For example, see Theorem 6.11 on page 272 for a recurrence relation for the knapsack problem.

(b) Give an efficient algorithm (with respect to both time and space) for computing the maximum value that can be achieved.

(c) What is the running time of your algorithm in Big Theta $\Theta$ notation? How much memory is required?

**Solution.** Let $OPT(i)$ represent the maximum value of jobs completed through day $i$. Assume that the values $l_i$ and $s_i$ are non-negative. For each day $i$, there are two potential job options:

- Short job $s$ on day $i$:
  - If a short job is selected on day $i$, it allows for the possibility of undertaking a job on day $i+1$.
  - The cumulative value of jobs completed through day $i$ in this scenario is represented by $OPT(i-1)+s_i$.
- Long job $l$ on day $i$:
  - If a long job is selected on day $i$, it implies that no job can be undertaken on day $i+1$ (provided $i$ is not the final day).
  - The cumulative value of jobs completed through day $i$ in this scenario is represented by $OPT(i-2)+l_i$.

In its basic form, the recurrence relation can be expressed as:

$$OPT(i) = \max\{OPT(i-1)+s_i, OPT(i-2)+l_i\}$$

Besides, this recurrence relation also takes into account the constraints that:

- If a long job is performed, no new job can be taken the next day.
- A short job allows for a new job the next day.
- A long job cannot be taken on the last day of the schedule.

Considered generally, this recurrence relation $OPT(i)$ represents the maximum value that can be obtained by scheduling jobs up to day $i$.

- $OPT(i) = 0$ if $i = 0$. This represents the case where there are no jobs to schedule, so the maximum value is 0.
- $OPT(i) = \max\{s_1, l_1\}$ if $i = 1$. This represents the case where there is a choice between a long job and a short job on the first day. The one with the higher value is chosen.
- $OPT(i) = OPT(i-1)$ if $i = 2$ and $OPT(1) = l_1$. This represents the case where a long job was chosen on the first day, so no job can be scheduled on the second day.
- $OPT(i) = \max\{OPT(i-1) + s_i, OPT(i-2) + l_i\}$ if ($i > 1$ and $i < d-1$ and $i \neq 2$) or ($i = 2$ and $OPT(1) = s_1$). This represents the case where there is a choice between:
  - Taking a short job on day $i$ and adding its value to the maximum value obtained up to day $i-1$, or
  - Taking a long job on day $i$ and adding its value to the maximum value obtained up to day $i-2$ (since no job can be scheduled on day $i+1$ if a long job is taken on day $i$).
- $OPT(i) = OPT(i-2) + l_i$ if $i = d$ and $flag = true$. This represents the case where a long job was taken on day $d-1$, so no job can be scheduled on the last day.
- $OPT(i) = OPT(i-1) + s_i$ if $i = d$ and $flag = false$. This represents the case where a short job was taken on day $d-1$, so another short job can be scheduled on the last day.

The $flag$ variable is used to track whether a long job or a short job was taken on day $d-1$. If $flag = true$, a long job was taken. If $flag = false$, a short job was taken. This is important because no job can be scheduled on day $d$ if a long job was taken on day $d-1$.

Given these options, the recurrence relation for this problem can be formulated as follows:

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\{s_1, l_1\} & \text{if } i = 1 \\ OPT(i-1) & \text{if } i = 2 \text{ and } OPT(1) = l_1 \\ \max\{OPT(i-1) + s_i,\ OPT(i-2) + l_i\} & \text{if (1)} \\ OPT(i-2) + l_i & \text{if (2)} \\ OPT(i-1) + s_i & \text{if (3)} \end{cases}$$

$$\begin{aligned} (1): &\quad (i > 1 \text{ and } i < d-1 \text{ and } i \neq 2) \text{ or } (i = 2 \text{ and } OPT(1) = s_1) \\ (2): &\quad i = d \text{ and } flag = true \\ (3): &\quad i = d \text{ and } flag = false \end{aligned}$$

Assume that the arrays $s[]$ and $l[]$, which signify the values of short and long jobs respectively, are indexed starting from 1. This implies that the value of the job on the first day is situated at $s[1]$ or $l[1]$, the value of the job on the subsequent day is at $s[2]$ or $l[2]$, and this pattern continues up to the $d$th day. A similar indexing approach is assumed for the $OPT[]$ array.

The algorithm operates by initializing an array $OPT[]$ of size $d + 1$ to store the maximum value that can be achieved up to each day. It then iterates over each day from 2 to $d - 1$, deciding whether to take a short job or a long job based on which choice would yield the maximum value.

A special flag variable is used to track if a long job is taken on the second last day, as this would prevent a job from being taken on the last day. If a long job is taken on the second last day, the maximum value on the last day is the same as the second last day. Otherwise, a short job can be taken on the last day, and its value is added to the maximum value of the second last day.

The algorithm returns the maximum value that can be achieved over the $d$ days, which is stored in $OPT[d]$.

The running time of the algorithm in Big Theta notation is $\Theta(d)$, where $d$ is the number of days. This is because the algorithm iterates over the range from 2 to $d - 1$ once, performing a constant amount of work in each iteration.

**Algorithm 1:** Maximize Job Value with $\Theta(d)$ Space

**Data:** Arrays $s$ (short job values), $l$ (long job values), and integer $d$ (number of days)
**Result:** Maximum value that can be achieved

```
if d = 0 then
    return 0;
end
OPT ← array of size d + 1;
OPT[0] ← 0;
OPT[1] ← max(s[1], l[1]);
flag ← false;
for i ← 2 to d − 1 do
    if i = 2 and OPT[1] = l[1] then
        OPT[i] ← OPT[i − 1];
    end
    else
        if i = d − 1 and OPT[i − 1] + s[i] < OPT[i − 2] + l[i] then
            flag ← true;
            OPT[i] ← OPT[i − 2] + l[i];
        end
        else
            OPT[i] ← max(OPT[i − 1] + s[i], OPT[i − 2] + l[i]);
        end
    end
end
if flag then
    OPT[d] ← OPT[d − 1]
end
else
    OPT[d] ← OPT[d − 1] + s[d];
end
return OPT[d];
```

The space complexity of the algorithm is also $\Theta(d)$, as it uses an additional array of size $d + 1$ to store the maximum values. This stems from the utilization of a dynamic programming table, a global array whose size is directly proportional to the number of days. Each entry in the table requires a constant amount of space. This implies that the algorithm's memory needs increase in a linear fashion with the size of the input parameter $d$.

However, since each $OPT(i)$ depends only on $OPT(i-1)$ and $OPT(i-2)$, the solution can be optimized for the space complexity to $\Theta(1)$ by keeping only the necessary values.

In the revised version of the algorithm, only the maximum values for the current day ($curr$) and the preceding day ($prev$) are retained. The maximum value for the subsequent day is computed, following which $prev$ and $curr$ are updated for the next iteration. This modification reduces the space complexity to $\Theta(1)$, signifying that only a constant number of variables are stored. The time complexity, however, remains $\Theta(d)$. This is indicative of the fact that the execution time of the algorithm is directly proportional to the number of days, $d$.

**Algorithm 2:** Maximize Job Value with $\Theta(1)$ Space

**Data:** Arrays $s$ (short job values), $l$ (long job values), and integer $d$ (number of days)
**Result:** Maximum value that can be achieved

```
if d = 0 then
    return 0;
end
prev ← −∞;
next ← −∞;
curr ← max(s[1], l[1]);
flag ← false;
for i ← 2 to d − 1 do
    if i = 2 and curr = l[1] then
        next ← curr;
    end
    else
        if i = d − 1 and curr + s[i] < prev + l[i] then
            flag ← true;
            next ← prev + l[i];
        end
        else
            next ← max(curr + s[i], prev + l[i]);
        end
    end
    prev ← curr;
    curr ← next;
end
if flag then
    curr ← prev
end
else
    curr ← curr + s[d];
end
return curr;
```

# Problem 5

Suppose you have $n$ identical pieces of candy that you want to sell for as much as possible. Oddly, you can sometimes make more by selling a subset together, rather than as individual pieces. You are given a table that says for each $0 < i \leq n$ how much profit you can achieve for a set of $i$ pieces of candy. For $i$ pieces, your profit is $p_i$. Note that selling a set with $i$ pieces of candy does not prevent you from selling another set with the same number of candies for the same price (if you still have enough left). Your goal is to develop an algorithm that maximizes the profit of selling all of the candy by determining the highest value division of the n pieces.

(a) Give a recurrence relation that expresses the optimal value that can be obtained.

(b) Give an efficient algorithm (with respect to both time and space) for computing the maximum value that can be achieved.

(c) What is the running time of your algorithm in Big Theta $\Theta$ notation? How much memory is required?

**Solution.** Simplify by considering this example:

- Input:
  - Suppose there are $n = 5$ pieces of candy.
  - The profit table is given as $p = [0, 1, 5, 7, 8, 10]$, where $p_i$ is the profit for selling $i$ pieces of candy.
- Output:
  - The maximum profit can achieve is 12 by selling one set of 2 pieces and one set of 3 pieces (or vice versa), as $5 + 7 = 12$.

This problem can be solved using dynamic programming. Let $OPT(n)$ denote the maximum profit that can be obtained by selling $n$ pieces of candy. The recurrence relation for the optimal value that can be obtained is as follows:

$$OPT(n) = \begin{cases} 0 & \text{if } n = 0 \\ p_n & \text{if } n = 1 \\ \max\limits_{1 \leq k \leq n} \{p_k + OPT(n-k)\} & \text{if } n > 1 \end{cases}$$

The base case is $OPT(0) = 0$, which means if there are no candies, the profit is zero. This relation states that the optimal profit for $n$ pieces of candy is the

maximum profit over all possible first pieces of candy to sell (from 1 to $n$), where $p_k$ is the profit for selling $k$ pieces, and $OPT(n-k)$ is the optimal profit for the remaining pieces.

The maximum over $1 \leq k \leq n$ is taken in the recurrence relation to consider all possible divisions of the $n$ pieces of candy. Since it is not possible to sell less than one candy or more than the available quantity, $k$ must lie between 1 and $n$ (inclusive).

For each $k$, the profit $p_k$ obtained by selling $k$ candies, plus the optimal profit $OPT(n-k)$ for the remaining $n-k$ candies, is calculated.

The maximum is taken over all possible $k$ to find the division that yields the highest total profit. This ensures all possible combinations of selling candies are considered, leading to the maximum profit.

**Algorithm 3:** Maximize Candy Profit

**Data:** Array $p$ (profit per candy quantity)
**Result:** Maximum profit that can be achieved

$n \leftarrow$ length of $p$;
**if** $n = 0$ **then**
  **return** 0;
**end**
$OPT \leftarrow$ array of size $n+1$;
$OPT[0] \leftarrow 0$;
**for** $i \leftarrow 1$ **to** $n$ **do**
  $OPT[i] \leftarrow -\infty$;
  **for** $j \leftarrow 1$ **to** $i$ **do**
    $OPT[i] \leftarrow \max(OPT[i], p[j] + OPT[i-j])$;
  **end**
**end**
**return** $OPT[n]$;

The algorithm uses an array $OPT[]$ to store the maximum profit that can be achieved for each quantity of candies from 0 to $n$. For each quantity $i$, it considers all possible numbers of candies $j$ that can be sold together (from 1 to $i$), and updates $OPT[i]$ with the maximum value of the current $OPT[i]$ and the profit of selling $j$ candies together plus the maximum profit of the remaining $i-j$ candies.

The running time of the algorithm in Big Theta notation is $\Theta(n^2)$, where $n$ is

the number of pieces of candy. This is because the algorithm contains two nested loops that each run up to $n$ times.

The space complexity of the algorithm is $\Theta(n)$, as it uses an additional array of size $n + 1$ to store the maximum profits. This means that the amount of memory required by the algorithm grows linearly with the input size $n$. This is quite efficient in terms of memory usage.

# References

[1] Wikipedia, *Edmonds' algorithm*, Accessed 17 February 2024. [Online]. Available: https://en.wikipedia.org/wiki/Edmonds%27_algorithm.

[2] Wikipedia, *Master theorem (analysis of algorithms)*, Accessed 22 February 2024. [Online]. Available: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms).