CSCI-SHU 220: Algorithms
Dynamic Programming II

NYU Shanghai
Spring 2025

Dynamic programming

- We have seen some problems that can be solved efficiently using dynamic programming.

Dynamic programming

- We have seen some problems that can be solved efficiently using dynamic programming.
  - Shortest paths (in an acyclic graph)
  - Maximum subarray
  - Weighted activity selection
  - Longest increasing subsequence
  - Longest common subsequence

Dynamic programming

- We have seen some problems that can be solved efficiently using dynamic programming.
  - Shortest paths (in an acyclic graph)
  - Maximum subarray
  - Weighted activity selection
  - Longest increasing subsequence
  - Longest common subsequence
- In all these problems, when we consider a subproblem $P \in \mathcal{S}$, each choice in $C(P)$ reduces P to one (smaller) subproblem in $\mathcal{S}$.

Dynamic programming

- We have seen some problems that can be solved efficiently using dynamic programming.
  - Shortest paths (in an acyclic graph)
  - Maximum subarray
  - Weighted activity selection
  - Longest increasing subsequence
  - Longest common subsequence
- In all these problems, when we consider a subproblem $P \in \mathcal{S}$, each choice in $C(P)$ reduces P to one (smaller) subproblem in $\mathcal{S}$.
- In some more complicated problems, it can happen that after taking a choice in $C(P)$, the problem is reduced to a combination of multiple subproblems in $\mathcal{S}$ that are independent.

Cut the cake!

Problem (cut the cake!)

We have a cake which is a rectangle consisting of $n \times m$ square cells. Each cell might be empty or contain a cherry. Now we want to cut the cake into pieces where each piece contains at most one cherry. The cutting procedure is done as follows.

- In each step, we pick one piece of cake and cut it into two.
- Each cut should be straight, either horizontal or vertical, and along the boundary lines of the cells.

Our goal is to minimize the total length of the cuts.

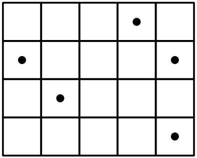
Cut the cake!

Problem (cut the cake!)

We have a cake which is a rectangle consisting of $n \times m$ square cells. Each cell might be empty or contain a cherry. Now we want to cut the cake into pieces where each piece contains at most one cherry. The cutting procedure is done as follows.

- In each step, we pick one piece of cake and cut it into two.
- Each cut should be straight, either horizontal or vertical, and along the boundary lines of the cells.

Our goal is to minimize the total length of the cuts.



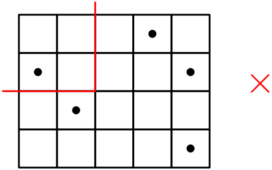
Cut the cake!

Problem (cut the cake!)

We have a cake which is a rectangle consisting of $n \times m$ square cells. Each cell might be empty or contain a cherry. Now we want to cut the cake into pieces where each piece contains at most one cherry. The cutting procedure is done as follows.

- In each step, we pick one piece of cake and cut it into two.
- Each cut should be straight, either horizontal or vertical, and along the boundary lines of the cells.

Our goal is to minimize the total length of the cuts.



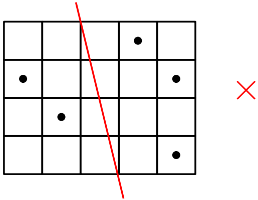
Cut the cake!

Problem (cut the cake!)

We have a cake which is a rectangle consisting of $n \times m$ square cells. Each cell might be empty or contain a cherry. Now we want to cut the cake into pieces where each piece contains at most one cherry. The cutting procedure is done as follows.

- In each step, we pick one piece of cake and cut it into two.
- Each cut should be straight, either horizontal or vertical, and along the boundary lines of the cells.

Our goal is to minimize the total length of the cuts.



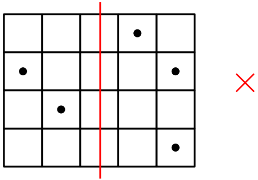
Cut the cake!

Problem (cut the cake!)

We have a cake which is a rectangle consisting of $n \times m$ square cells. Each cell might be empty or contain a cherry. Now we want to cut the cake into pieces where each piece contains at most one cherry. The cutting procedure is done as follows.

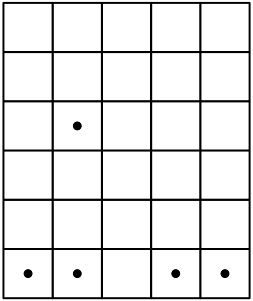
- In each step, we pick one piece of cake and cut it into two.
- Each cut should be straight, either horizontal or vertical, and along the boundary lines of the cells.

Our goal is to minimize the total length of the cuts.



Cut the cake!

- Different ways to cut the cake have different total length.

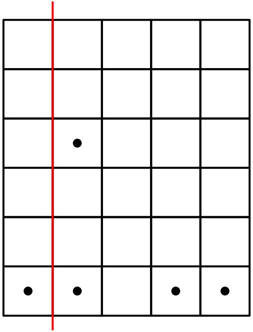


Total length

0

Cut the cake!

- Different ways to cut the cake have different total length.

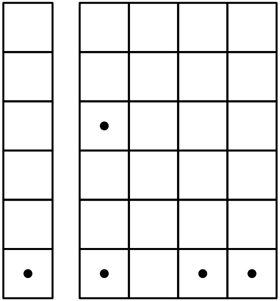


Total length

0

Cut the cake!

- Different ways to cut the cake have different total length.

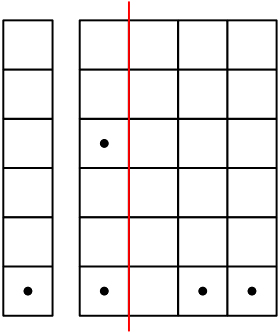


Total length

6

Cut the cake!

- Different ways to cut the cake have different total length.

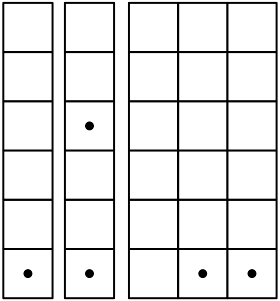


Total length

6

Cut the cake!

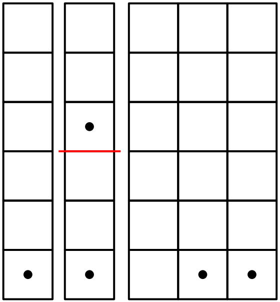
- Different ways to cut the cake have different total length.



Total length
12

Cut the cake!

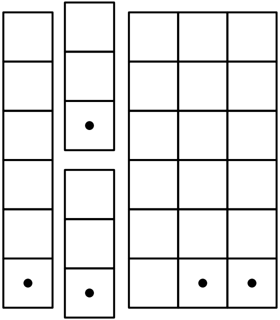
- Different ways to cut the cake have different total length.



Total length
12

Cut the cake!

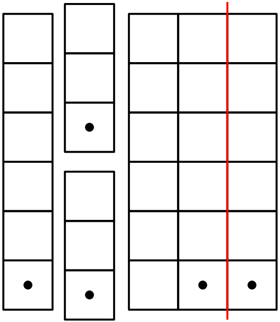
- Different ways to cut the cake have different total length.



Total length
13

Cut the cake!

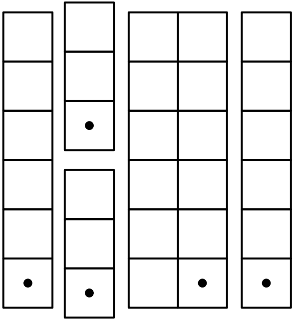
- Different ways to cut the cake have different total length.



Total length
13

Cut the cake!

- Different ways to cut the cake have different total length.

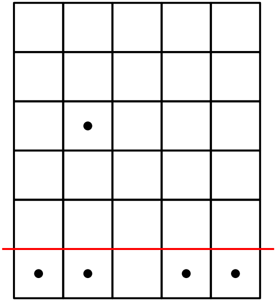


Total length
19

Done!

Cut the cake!

- Different ways to cut the cake have different total length.

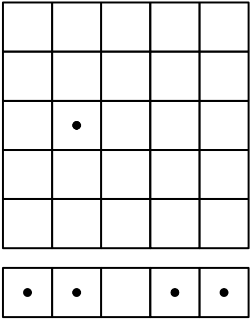


Total length

0

Cut the cake!

- Different ways to cut the cake have different total length.

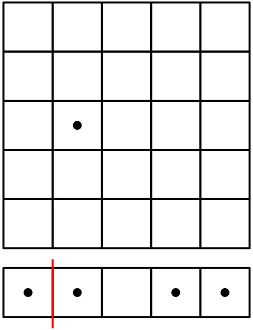


Total length

5

Cut the cake!

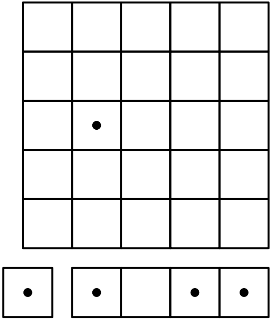
- Different ways to cut the cake have different total length.



Total length
5

Cut the cake!

- Different ways to cut the cake have different total length.

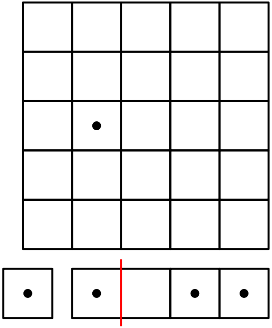


Total length

6

Cut the cake!

- Different ways to cut the cake have different total length.

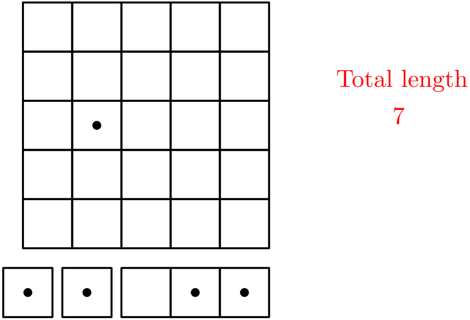


Total length

6

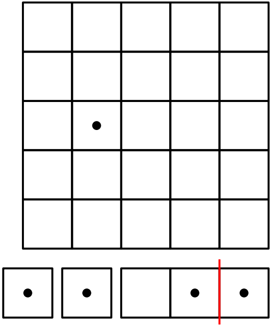
Cut the cake!

- Different ways to cut the cake have different total length.



Cut the cake!

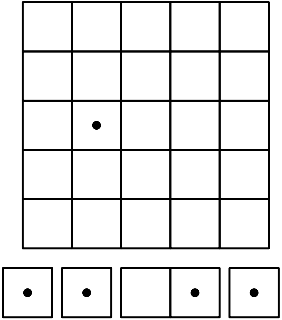
- Different ways to cut the cake have different total length.



Total length
7

Cut the cake!

- Different ways to cut the cake have different total length.



Total length
8

Done!

Cut the cake!

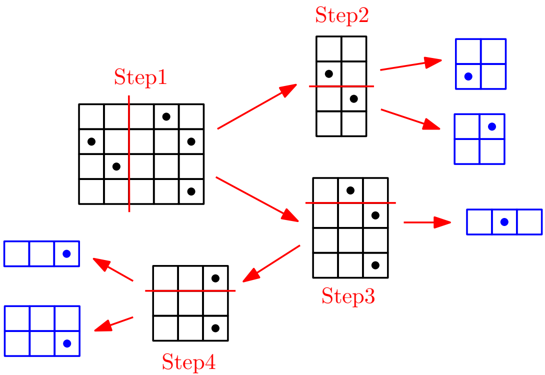
- To use DP, we need to define the subproblems and the dependency.

Cut the cake!

- To use DP, we need to define the subproblems and the dependency.
- To this end, let's consider a cutting procedure.

Cut the cake!

- To use DP, we need to define the subproblems and the dependency.
- To this end, let's consider a cutting procedure.

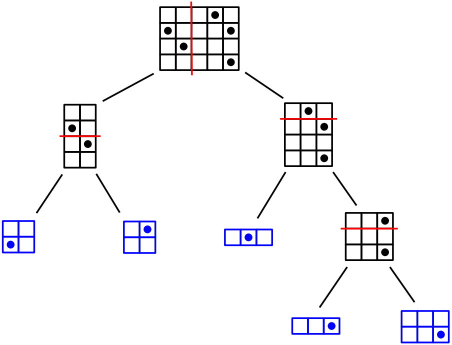


Cut the cake!

- We can use a binary tree to describe a cutting procedure.

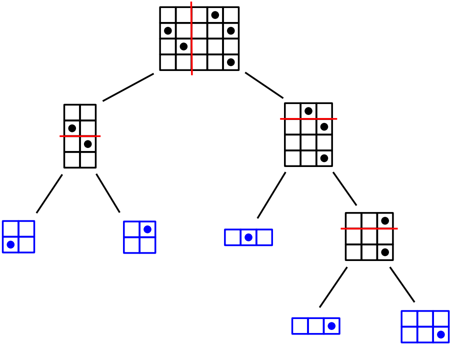
Cut the cake!

- We can use a binary tree to describe a cutting procedure.



Cut the cake!

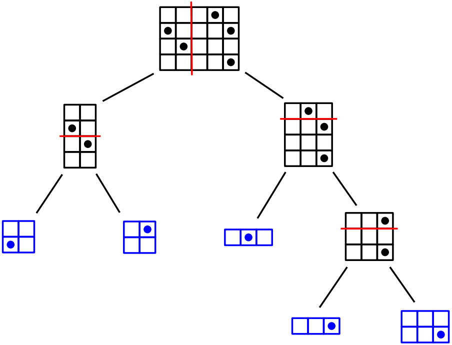
- We can use a binary tree to describe a cutting procedure.



- Each internal node corresponds to one cut and each leaf corresponds to a final piece (which contains at most one cherry).

Cut the cake!

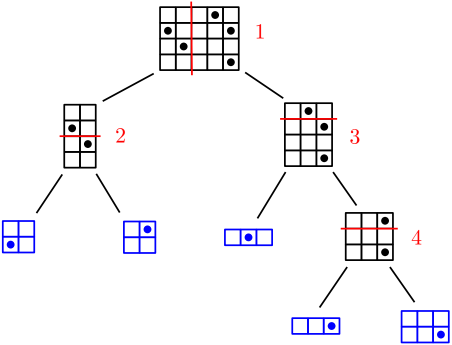
- We can use a binary tree to describe a cutting procedure.



- Each internal node corresponds to one cut and each leaf corresponds to a final piece (which contains at most one cherry).
- There are multiple ways to order the cuts, with the same total length.

Cut the cake!

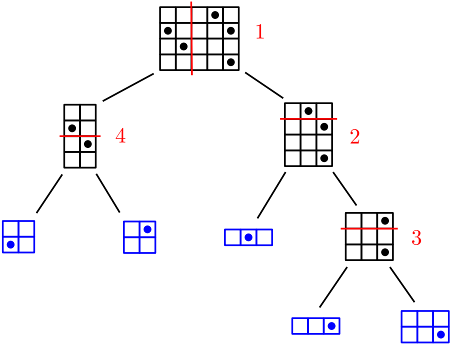
- We can use a binary tree to describe a cutting procedure.



- Each internal node corresponds to one cut and each leaf corresponds to a final piece (which contains at most one cherry).
- There are multiple ways to order the cuts, with the same total length.

Cut the cake!

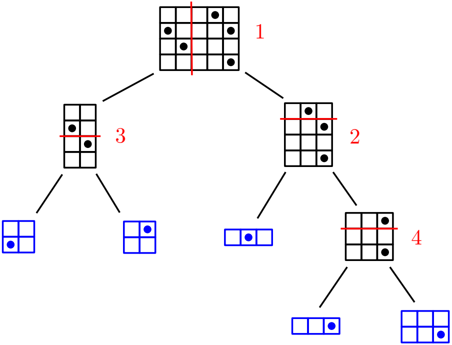
- We can use a binary tree to describe a cutting procedure.



- Each internal node corresponds to one cut and each leaf corresponds to a final piece (which contains at most one cherry).
- There are multiple ways to order the cuts, with the same total length.

Cut the cake!

- We can use a binary tree to describe a cutting procedure.



- Each internal node corresponds to one cut and each leaf corresponds to a final piece (which contains at most one cherry).
- There are multiple ways to order the cuts, with the same total length.

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.
- What is the total length of all cuts in the cutting procedure?

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.
- What is the total length of all cuts in the cutting procedure?
  length of the first cut + length for cutting A + length for cutting B

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.
- What is the total length of all cuts in the cutting procedure?
  length of the first cut $+$ length for cutting A $+$ length for cutting B
- In order to make the cutting procedure optimal, we have to cut the two smaller pieces A and B optimally.

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.
- What is the total length of all cuts in the cutting procedure?
  length of the first cut + length for cutting A + length for cutting B
- In order to make the cutting procedure optimal, we have to cut the two smaller pieces A and B optimally.
- Note that A and B are both smaller than the original piece.

Cut the cake!

- We can always finish the cuts on the left subtree first, and then do the cuts on the right subtree. In other words, after the first cut, we can cut the two smaller pieces one by one.
- Let A and B be the two smaller pieces obtained by the first cut.
- What is the total length of all cuts in the cutting procedure?
  length of the first cut + length for cutting A + length for cutting B
- In order to make the cutting procedure optimal, we have to cut the two smaller pieces A and B optimally.
- Note that A and B are both smaller than the original piece.
- We know how to cut a piece optimally as long as we know how to cut all smaller pieces optimally.

Cut the cake!

- $\mathcal{K}$ = set of all cake pieces that can appear in a cutting procedure

Cut the cake!

- $\mathcal{K}$ = set of all cake pieces that can appear in a cutting procedure
- What are the pieces contained in $\mathcal{K}$?

Cut the cake!

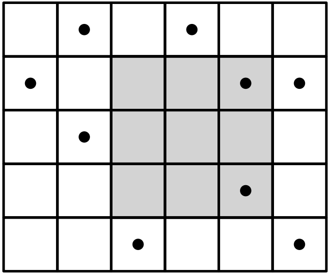
- $\mathcal{K}$ = set of all cake pieces that can appear in a cutting procedure
- What are the pieces contained in $\mathcal{K}$?
    - A cake piece must be a rectangle.

Cut the cake!

- $\mathcal{K}$ = set of all cake pieces that can appear in a cutting procedure
- What are the pieces contained in $\mathcal{K}$?
  - A cake piece must be a rectangle.
  - The boundary of cake piece must consist of boundary lines of cells.

Cut the cake!

- $\mathcal{K}$ = set of all cake pieces that can appear in a cutting procedure
- What are the pieces contained in $\mathcal{K}$?
  - A cake piece must be a rectangle.
  - The boundary of cake piece must consist of boundary lines of cells.



Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  $i^-/i^+ =$ index of the topmost/bottommost row
  $j^-/j^+ =$ index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?
  Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?
  Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.
  Vertical: (i^-, i^+, j^-, j) and $(i^-, i^+, j+1, j^+)$ for some $j \in [j^-, j^+)$.

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?
  Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.
  Vertical: (i^-, i^+, j^-, j) and $(i^-, i^+, j+1, j^+)$ for some $j \in [j^-, j^+)$.
- $\mathrm{opt}^{\mathbf{h}}_{i^-,i^+,j^-,j^+} = \min_{i\in[i^-,i^+)}(j^+ - j^- + 1 + \mathrm{opt}_{i^-,i,j^-,j^+} + \mathrm{opt}_{i+1,i^+,j^-,j^+})$
  $\mathrm{opt}^{\mathbf{v}}_{i^-,i^+,j^-,j^+} = \min_{j\in[i^-,i^+)}(i^+ - i^- + 1 + \mathrm{opt}_{i^-,i^+,j^-,j} + \mathrm{opt}_{i^-,i^+,j+1,j^+})$

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?
  Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.
  Vertical: (i^-, i^+, j^-, j) and $(i^-, i^+, j+1, j^+)$ for some $j \in [j^-, j^+)$.
- $\mathrm{opt}^{\mathbf{h}}_{i^-,i^+,j^-,j^+} = \min_{i\in[i^-,i^+)}(j^+ - j^- + 1 + \mathrm{opt}_{i^-,i,j^-,j^+} + \mathrm{opt}_{i+1,i^+,j^-,j^+})$
  $\mathrm{opt}^{\mathbf{v}}_{i^-,i^+,j^-,j^+} = \min_{j\in[i^-,i^+)}(i^+ - i^- + 1 + \mathrm{opt}_{i^-,i^+,j^-,j} + \mathrm{opt}_{i^-,i^+,j+1,j^+})$
- $\mathrm{opt}_{i^-,i^+,j^-,j^+} = \min\{\mathrm{opt}^{\mathbf{h}}_{i^-,i^+,j^-,j^+}, \mathrm{opt}^{\mathbf{v}}_{i^-,i^+,j^-,j^+}\}$

Cut the cake!

- Each piece in $\mathcal{K}$ can be described by a tuple (i^-, i^+, j^-, j^+).
  i^-/i^+ = index of the topmost/bottommost row
  j^-/j^+ = index of the leftmost/rightmost column
- The original cake is $(1, n, 1, m)$.
- If we cut a piece (i^-, i^+, j^-, j^+), what are the two resulting pieces?
  Horizontal: (i^-, i, j^-, j^+) and $(i+1, i^+, j^-, j^+)$ for some $i \in [i^-, i^+)$.
  Vertical: (i^-, i^+, j^-, j) and $(i^-, i^+, j+1, j^+)$ for some $j \in [j^-, j^+)$.
- $\mathrm{opt}^{\mathbf{h}}_{i^-,i^+,j^-,j^+} = \min_{i\in[i^-,i^+)}(j^+ - j^- + 1 + \mathrm{opt}_{i^-,i,j^-,j^+} + \mathrm{opt}_{i+1,i^+,j^-,j^+})$
  $\mathrm{opt}^{\mathbf{v}}_{i^-,i^+,j^-,j^+} = \min_{j\in[i^-,i^+)}(i^+ - i^- + 1 + \mathrm{opt}_{i^-,i^+,j^-,j} + \mathrm{opt}_{i^-,i^+,j+1,j^+})$
- $\mathrm{opt}_{i^-,i^+,j^-,j^+} = \min\{\mathrm{opt}^{\mathbf{h}}_{i^-,i^+,j^-,j^+}, \mathrm{opt}^{\mathbf{v}}_{i^-,i^+,j^-,j^+}\}$
- **Boundary case**
  $\mathrm{opt}_{i^-,i^+,j^-,j^+} = 0$ if (i^-, i^+, j^-, j^+) contains at most one cherry.

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x, y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x,y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\textsc{cost}(x,y) = xy$

$$7 + 3 + 8 + 11 + 4$$

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x,y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\textsc{cost}(x,y) = xy$

$$\begin{array}{ccccccccc} 7 & + & 3 & + & 8 & + & 11 & + & 4 \\ & 2 & & 4 & & 3 & & 1 & \end{array}$$

$$\text{Total cost} = 0$$

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\text{COST}(x, y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\text{COST}(x, y) = xy$

$$\underset{}{7} \underset{2}{+} 3 \underset{4}{+} 8 \underset{3}{+} \quad 15$$

$$\text{Total cost} = 44$$

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x,y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\textsc{cost}(x,y) = xy$

$$\begin{array}{ccccc} 10 & \underset{4}{+} & 8 & \underset{3}{+} & 15 \end{array}$$

$$\text{Total cost} = 65$$

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x,y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\textsc{cost}(x,y) = xy$

$$\underset{4}{10 \quad + \quad 23}$$

$$\text{Total cost} = 185$$

Expression evaluation

Problem (expression evaluation)

We have an expression with n numbers and $n-1$ "$+$" operations. In order to evaluate the expression, we can perform these $n-1$ "$+$" operations in any order and, the outcome is always the sum of the n numbers. Suppose computing the sum $x+y$ of two numbers x and y has a cost, which can be known from a given oracle $\textsc{cost}(x, y)$. Our goal is to find an optimal order to perform the "$+$" operations which minimizes the total cost.

- **Example:** $\textsc{cost}(x, y) = xy$

$$33$$

$$\text{Total cost } = 415$$

Expression evaluation

- To use DP, let's consider the first step to solve the problem.

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.
- E_i = the resulting expression after doing $x_i + x_{i+1}$

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.
- E_i = the resulting expression after doing $x_i + x_{i+1}$
- $\text{opt}(E) = \min_{i=1}^{n-1}(\textsc{Cost}(x_i, x_{i+1}) + \text{opt}(E_i))$

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.
- E_i = the resulting expression after doing $x_i + x_{i+1}$
- $\text{opt}(E) = \min_{i=1}^{n-1}(\textsc{Cost}(x_i, x_{i+1}) + \text{opt}(E_i))$
- This gives you a DP algorithm, but it is very inefficient...

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.
- E_i = the resulting expression after doing $x_i + x_{i+1}$
- $\mathrm{opt}(E) = \min_{i=1}^{n-1}(\textsc{Cost}(x_i, x_{i+1}) + \mathrm{opt}(E_i))$
- This gives you a DP algorithm, but it is very inefficient...
- We should consider the subproblems of all possible expressions that can appear in an evaluation procedure.

Expression evaluation

- To use DP, let's consider the first step to solve the problem.
- E = original expression $x_1 + \cdots + x_n$
- In the first step, we can perform $x_i + x_{i+1}$ for any $1 \leq i \leq n-1$.
- E_i = the resulting expression after doing $x_i + x_{i+1}$
- $\text{opt}(E) = \min_{i=1}^{n-1}(\textsc{Cost}(x_i, x_{i+1}) + \text{opt}(E_i))$
- This gives you a DP algorithm, but it is very inefficient...
- We should consider the subproblems of all possible expressions that can appear in an evaluation procedure.
- There are exponentially many such expressions.

Expression evaluation

- Key idea to solve this problem: consider the last step.

Expression evaluation

- Key idea to solve this problem: consider the last step.
- Suppose the last step performs the i-th "$+$" in the original expression.

Expression evaluation

- Key idea to solve this problem: consider the last step.
- Suppose the last step performs the i-th "+" in the original expression.

$$\overbrace{\boxed{x_1 + \cdots + x_i}}^{\sum_{j=1}^{i} x_j} + \overbrace{\boxed{x_{i+1} + \cdots + x_n}}^{\sum_{j=i+1}^{n} x_j}$$

- The cost of the last "+" is $\textsc{Cost}(\sum_{j=1}^{i} x_j, \sum_{j=i+1}^{n} x_j)$.

Expression evaluation

- Key idea to solve this problem: consider the last step.
- Suppose the last step performs the i-th "+" in the original expression.

$$\overbrace{\boxed{x_1 + \cdots + x_i}}^{\sum_{j=1}^{i} x_j} + \overbrace{\boxed{x_{i+1} + \cdots + x_n}}^{\sum_{j=i+1}^{n} x_j}$$

- The cost of the last "+" is $\textsc{Cost}(\sum_{j=1}^{i} x_j, \sum_{j=i+1}^{n} x_j)$.
- Before the last step, need to evaluate $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$.

Expression evaluation

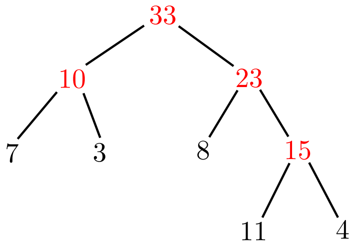
- The evaluations of $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$ are independent.

Expression evaluation

- The evaluations of $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$ are independent.
- Why? Let's represent the evaluation by a binary tree.

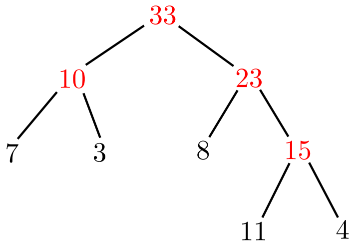
Expression evaluation

- The evaluations of $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$ are independent.
- Why? Let's represent the evaluation by a binary tree.



Expression evaluation

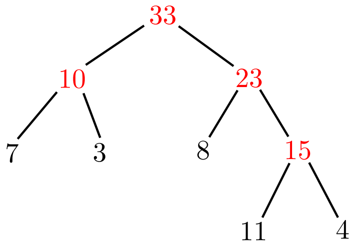
- The evaluations of $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$ are independent.
- Why? Let's represent the evaluation by a binary tree.



- The leaves from left to right correspond to the numbers $x_1, \ldots, x_n$, and each internal node represents a "$+$" operation.

Expression evaluation

- The evaluations of $x_1 + \cdots + x_i$ and $x_{i+1} + \cdots + x_n$ are independent.
- Why? Let's represent the evaluation by a binary tree.



- The leaves from left to right correspond to the numbers $x_1, \ldots, x_n$, and each internal node represents a "+" operation.
- The tree determines the total cost. So we can always evaluate the left subtree first, and then the right subtree.

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \le i \le j \le n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.
- $\text{opt}(P_{i,j}) = \min_{k=i}^{j-1}(\textsc{Cost}(\Sigma_i^k, \Sigma_{k+1}^j) + \text{opt}(P_{i,k}) + \text{opt}(P_{k+1,j}))$

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.
- $\text{opt}(P_{i,j}) = \text{min}_{k=i}^{j-1}(\textsc{Cost}(\Sigma_i^k, \Sigma_{k+1}^j) + \text{opt}(P_{i,k}) + \text{opt}(P_{k+1,j}))$
- **Boundary case**
  $\text{opt}(P_{i,j}) = 0$ if $i = j$ (no "+" operation is needed)

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \le i \le j \le n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.
- $\text{opt}(P_{i,j}) = \min_{k=i}^{j-1}(\text{Cost}(\Sigma_i^k, \Sigma_{k+1}^j) + \text{opt}(P_{i,k}) + \text{opt}(P_{k+1,j}))$
- **Boundary case**
  $\text{opt}(P_{i,j}) = 0$ if $i = j$ (no "+" operation is needed)
- Time complexity = $O(n^3)$

Expression evaluation

- $P_{i,j}$ = evaluating the expression $x_i + \cdots + x_j$
  $\mathcal{S} = \{P_{i,j} : 1 \leq i \leq j \leq n\}$
- How many subproblems are there? $1 + \cdots + n = O(n^2)$
- Let's write $\Sigma_i^j = x_i + \cdots + x_j$.
- $\text{opt}(P_{i,j}) = \min_{k=i}^{j-1}(\textsc{Cost}(\Sigma_i^k, \Sigma_{k+1}^j) + \text{opt}(P_{i,k}) + \text{opt}(P_{k+1,j}))$
- **Boundary case**
  $\text{opt}(P_{i,j}) = 0$ if $i = j$ (no "+" operation is needed)
- Time complexity = $O(n^3)$

More challenging DP problems

- We have seen that the key for designing DP algorithms is to properly define subproblems and figure out the recursive dependency.

More challenging DP problems

- We have seen that the key for designing DP algorithms is to properly define subproblems and figure out the recursive dependency.
- However, for some problems, it is challenging to find the correct way to define subproblems.

More challenging DP problems

- We have seen that the key for designing DP algorithms is to properly define subproblems and figure out the recursive dependency.
- However, for some problems, it is challenging to find the correct way to define subproblems.
- There is no universal approach to do this. You can only improve by practicing more and becoming more experienced.

More challenging DP problems

- We have seen that the key for designing DP algorithms is to properly define subproblems and figure out the recursive dependency.
- However, for some problems, it is challenging to find the correct way to define subproblems.
- There is no universal approach to do this. You can only improve by practicing more and becoming more experienced.
- **A typical way to do this**
  Start from the most trivial definition of subproblems, and try to work out the recursive dependency. If you fail, you will find a more proper way to define subproblems. Repeat this procedure until success.

Generalized (weighted) activity selection

Problem (generalized activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. The room is big enough and can hold c activities at the same time for a constant c. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of c-compatible activities with maximum total weight.

Generalized (weighted) activity selection

Problem (generalized activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. The room is big enough and can hold c activities at the same time for a constant c. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of c-compatible activities with maximum total weight.

- c-compatible: at any time, there are at most c activities.

Generalized (weighted) activity selection

Problem (generalized activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. The room is big enough and can hold c activities at the same time for a constant c. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of c-compatible activities with maximum total weight.

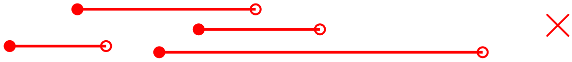
- c-compatible: at any time, there are at most c activities.
- **Example:** $c = 2$

Generalized (weighted) activity selection

Problem (generalized activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. The room is big enough and can hold c activities at the same time for a constant c. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of c-compatible activities with maximum total weight.

- c-compatible: at any time, there are at most c activities.
- **Example:** $c = 2$



Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.

Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.

Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.

Generalized (weighted) activity selection

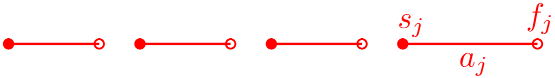
- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.
- P_i = activity selection on $\{a_1, \ldots, a_i\}$

Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.
- $P_i =$ activity selection on $\{a_1, \ldots, a_i\}$
- We consider the rightmost activity in the solution.

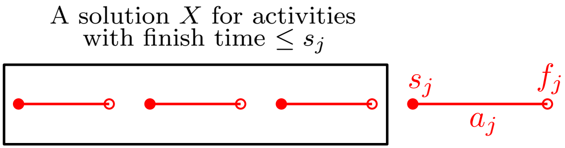
Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.
- P_i = activity selection on $\{a_1, \ldots, a_i\}$
- We consider the rightmost activity in the solution.



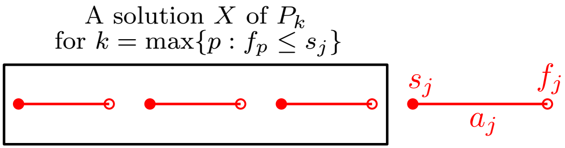
Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.
- P_i = activity selection on $\{a_1, \ldots, a_i\}$
- We consider the rightmost activity in the solution.



Generalized (weighted) activity selection

- In order to get some idea, let's simply assume $c = 2$.
- Again, let's sort all activities such that $f_1 \leq \cdots \leq f_n$.
- Recall how we solve the problem when $c = 1$.
- P_i = activity selection on $\{a_1, \ldots, a_i\}$
- We consider the rightmost activity in the solution.



- **First issue**

  When $c > 1$, the activities in a solution need not to be disjoint. So what do we mean by the rightmost activity?

Generalized (weighted) activity selection

- **First issue**

  When $c > 1$, the activities in a solution need not to be disjoint. So what do we mean by the rightmost activity?

- We can define "rightmost" in terms of finish time.

Generalized (weighted) activity selection

- **First issue**

  When $c > 1$, the activities in a solution need not to be disjoint. So what do we mean by the rightmost activity?

- We can define "rightmost" in terms of finish time.

- **Second issue**

  Even if we fix the rightmost activity in our solution, we are not able to reduce the problem to a subproblem P_k for some $k < j$.

Generalized (weighted) activity selection

- **First issue**

  When $c > 1$, the activities in a solution need not to be disjoint. So what do we mean by the rightmost activity?

- We can define "rightmost" in terms of finish time.

- **Second issue**

  Even if we fix the rightmost activity in our solution, we are not able to reduce the problem to a subproblem P_k for some $k < j$.

- Every activity in $\{a_1, \ldots, a_{j-1}\}$ can possibly appear in the solution.

Generalized (weighted) activity selection

- **First issue**

  When $c > 1$, the activities in a solution need not to be disjoint. So what do we mean by the rightmost activity?

- We can define "rightmost" in terms of finish time.

- **Second issue**

  Even if we fix the rightmost activity in our solution, we are not able to reduce the problem to a subproblem P_k for some $k < j$.

- Every activity in $\{a_1, \ldots, a_{j-1}\}$ can possibly appear in the solution.

- However, not every feasible solution of P_{j-1} remains feasible after adding the activity a_j.

Generalized (weighted) activity selection

- To handle the second issue, let's consider the following question: Which solutions of P_{j-1} remains feasible after we add the activity a_j?

Generalized (weighted) activity selection

- To handle the second issue, let's consider the following question: Which solutions of P_{j-1} remains feasible after we add the activity a_j?
- Consider a (feasible) solution $S \subseteq \{a_1, \ldots, a_{j-1}\}$ of P_{j-1}.

Generalized (weighted) activity selection

- To handle the second issue, let's consider the following question: Which solutions of P_{j-1} remains feasible after we add the activity a_j?
- Consider a (feasible) solution $S \subseteq \{a_1, \ldots, a_{j-1}\}$ of P_{j-1}.
- $S \cup \{a_j\}$ is 2-compatible iff for any $t \geq s_j$, at most one activity in S contains the time point t.

Generalized (weighted) activity selection

- To handle the second issue, let's consider the following question: Which solutions of P_{j-1} remains feasible after we add the activity a_j?
- Consider a (feasible) solution $S \subseteq \{a_1, \ldots, a_{j-1}\}$ of P_{j-1}.
- $S \cup \{a_j\}$ is 2-compatible iff for any $t \geq s_j$, at most one activity in S contains the time point t.
- If $S \cup \{a_j\}$ is optimal, then S must be optimal among all 2-compatible subsets of $\{a_1, \ldots, a_{j-1}\}$ that are 1-compatible in $[s_j, \infty)$.

Generalized (weighted) activity selection

- To handle the second issue, let's consider the following question: Which solutions of P_{j-1} remains feasible after we add the activity a_j?
- Consider a (feasible) solution $S \subseteq \{a_1, \ldots, a_{j-1}\}$ of P_{j-1}.
- $S \cup \{a_j\}$ is 2-compatible iff for any $t \geq s_j$, at most one activity in S contains the time point t.
- If $S \cup \{a_j\}$ is optimal, then S must be optimal among all 2-compatible subsets of $\{a_1, \ldots, a_{j-1}\}$ that are 1-compatible in $[s_j, \infty)$.
- **New problem**

  Given an activity set A and a number x, compute the max-weighted 2-compatible subset of A that is 1-compatible in $[x, \infty)$.

Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$

Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.

Generalized (weighted) activity selection

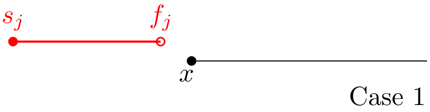
- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.
- Again, we consider the rightmost activity (say a_j) in the solution.

Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.
- Again, we consider the rightmost activity (say a_j) in the solution.
- Locationally, there are three cases for a_j and the range $[x, \infty)$.
  (1) $s_j < f_j \leq x$ (2) $s_j \leq x < f_j$ (3) $x < s_j < f_j$

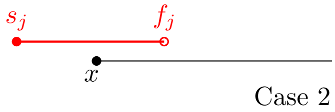
Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.
- Again, we consider the rightmost activity (say a_j) in the solution.
- Locationally, there are three cases for a_j and the range $[x, \infty)$.
  (1) $s_j < f_j \leq x$ (2) $s_j \leq x < f_j$ (3) $x < s_j < f_j$



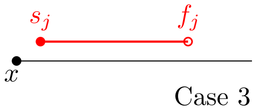
Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.
- Again, we consider the rightmost activity (say a_j) in the solution.
- Locationally, there are three cases for a_j and the range $[x, \infty)$.
  (1) $s_j < f_j \leq x$ (2) $s_j \leq x < f_j$ (3) $x < s_j < f_j$



Generalized (weighted) activity selection

- $P_{i,x}$ = activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility in $[x, \infty)$
- Now let's try to solve each subproblem $P_{i,x}$.
- Again, we consider the rightmost activity (say a_j) in the solution.
- Locationally, there are three cases for a_j and the range $[x, \infty)$.
  (1) $s_j < f_j \leq x$ (2) $s_j \leq x < f_j$ (3) $x < s_j < f_j$



Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$
  - **Case 3:** $k = \max\{p : f_p \leq s_j\}$ and $y = x$

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$
  - **Case 3:** $k = \max\{p : f_p \leq s_j\}$ and $y = x$
- Knowing the optimal solutions of all subproblems $P_{k,y}$ with $k < i$, we can solve the subproblem $P_{i,x}$.

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$
  - **Case 3:** $k = \max\{p : f_p \leq s_j\}$ and $y = x$
- Knowing the optimal solutions of all subproblems $P_{k,y}$ with $k < i$, we can solve the subproblem $P_{i,x}$.
- Clearly, we can't solve $P_{i,x}$ for all $x \in \mathbb{R}$.

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$
  - **Case 3:** $k = \max\{p : f_p \leq s_j\}$ and $y = x$
- Knowing the optimal solutions of all subproblems $P_{k,y}$ with $k < i$, we can solve the subproblem $P_{i,x}$.
- Clearly, we can't solve $P_{i,x}$ for all $x \in \mathbb{R}$.
- But do we really need to consider all $x \in \mathbb{R}$? Not exactly...

Generalized (weighted) activity selection

- In each case, we reduce to a subproblem $P_{k,y}$ for some $k < j$.
  - **Case 1:** $k = j - 1$ and $y = s_j$
  - **Case 2:** $k = \max\{p : f_p \leq x\}$ and $y = s_j$
  - **Case 3:** $k = \max\{p : f_p \leq s_j\}$ and $y = x$
- Knowing the optimal solutions of all subproblems $P_{k,y}$ with $k < i$, we can solve the subproblem $P_{i,x}$.
- Clearly, we can't solve $P_{i,x}$ for all $x \in \mathbb{R}$.
- But do we really need to consider all $x \in \mathbb{R}$? Not exactly...
- Only the time points in $\{s_1, \ldots, s_n, \infty\}$ are relevant.

Generalized (weighted) activity selection

- $\mathcal{S} = \{P_{i,s_t} : 1 \leq i, t \leq n\}$

Generalized (weighted) activity selection

- $\mathcal{S} = \{P_{i,s_t} : 1 \leq i, t \leq n\}$
- $\text{opt}(P_{i,s_t}) = \max_{j=1}^{i}(\text{opt}_j(P_{i,s_t})) = \max\{\text{opt}(P_{i-1,s_t}), \text{opt}_i(P_{i,s_t})\}$

Generalized (weighted) activity selection

- $\mathcal{S} = \{P_{i,s_t} : 1 \leq i, t \leq n\}$
- $\text{opt}(P_{i,s_t}) = \max_{j=1}^{i}(\text{opt}_j(P_{i,s_t})) = \max\{\text{opt}(P_{i-1,s_t}), \text{opt}_i(P_{i,s_t})\}$
- $\text{opt}_i(P_{i,s_t}) = \text{opt}(P_{k,y}) + w_i$, where
  - if $s_i < f_i \leq s_t$, $k = i - 1$ and $y = s_i$,
  - if $s_i \leq s_t < f_i$, $k = \max\{p : f_p \leq s_t\}$ and $y = s_i$,
  - if $s_t < s_i < f_i$, $k = \max\{p : f_p \leq s_i\}$ and $y = s_t$.

Generalized (weighted) activity selection

- $\mathcal{S} = \{P_{i,s_t} : 1 \leq i, t \leq n\}$
- $\text{opt}(P_{i,s_t}) = \max_{j=1}^{i}(\text{opt}_j(P_{i,s_t})) = \max\{\text{opt}(P_{i-1,s_t}), \text{opt}_i(P_{i,s_t})\}$
- $\text{opt}_i(P_{i,s_t}) = \text{opt}(P_{k,y}) + w_i$, where
  - if $s_i < f_i \leq s_t$, $k = i - 1$ and $y = s_i$,
  - if $s_i \leq s_t < f_i$, $k = \max\{p : f_p \leq s_t\}$ and $y = s_i$,
  - if $s_t < s_i < f_i$, $k = \max\{p : f_p \leq s_i\}$ and $y = s_t$.
- Time complexity $= O(n^2)$ by precomputing $\max\{p : f_p \leq s_t\}$ for all t

Generalized (weighted) activity selection

- $\mathcal{S} = \{P_{i,s_t} : 1 \le i, t \le n\}$
- $\text{opt}(P_{i,s_t}) = \max_{j=1}^{i}(\text{opt}_j(P_{i,s_t})) = \max\{\text{opt}(P_{i-1,s_t}), \text{opt}_i(P_{i,s_t})\}$
- $\text{opt}_i(P_{i,s_t}) = \text{opt}(P_{k,y}) + w_i$, where
  - if $s_i < f_i \le s_t$, $k = i - 1$ and $y = s_i$,
  - if $s_i \le s_t < f_i$, $k = \max\{p : f_p \le s_t\}$ and $y = s_i$,
  - if $s_t < s_i < f_i$, $k = \max\{p : f_p \le s_i\}$ and $y = s_t$.
- Time complexity $= O(n^2)$ by precomputing $\max\{p : f_p \le s_t\}$ for all t
- What if $c > 2$?

Generalized (weighted) activity selection

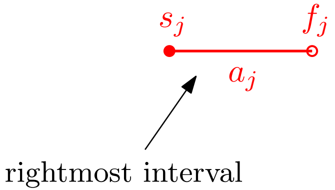
- Let's see what happens when $c = 3$.

Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.

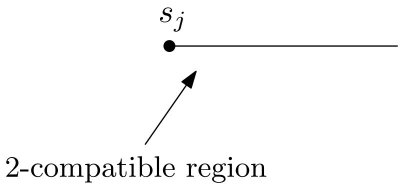
Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



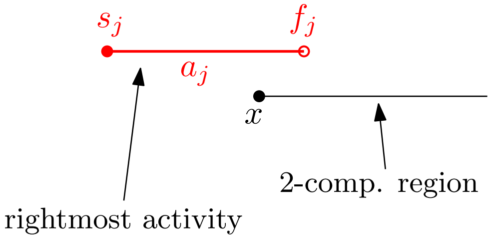
Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



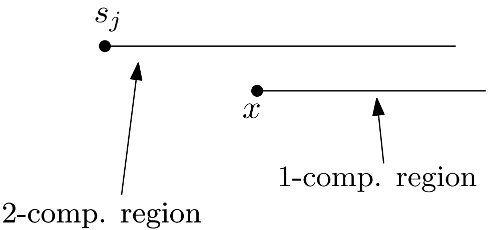
Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



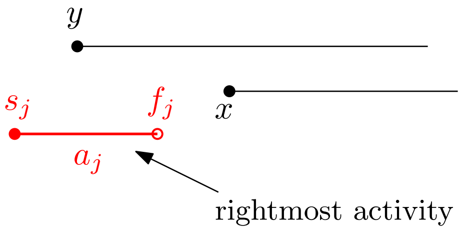
Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



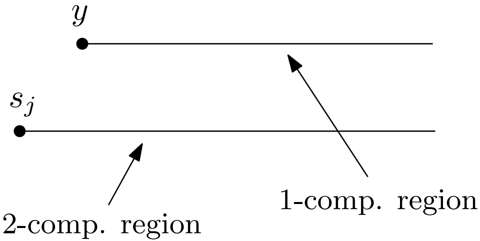
Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



Generalized (weighted) activity selection

- Let's see what happens when $c = 3$.
- We can define $P_{i,x,y}$ with $x \geq y$ be activity selection on $\{a_1, \ldots, a_i\}$ with 1-compatibility on $[x, \infty)$ and 2-compatibility on $[y, \infty)$.



Generalized (weighted) activity selection

- For a general c?

Generalized (weighted) activity selection

- For a general c?
- We can define $P_{i,x_1,\ldots,x_{c-1}}$ with $x_1 \geq \cdots \geq x_{c-1}$ be activity selection on $\{a_1, \ldots, a_i\}$ with t-compatibility on $[x_t, \infty)$ for all $t \leq c-1$.

Generalized (weighted) activity selection

- For a general c?
- We can define $P_{i,x_1,\ldots,x_{c-1}}$ with $x_1 \geq \cdots \geq x_{c-1}$ be activity selection on $\{a_1, \ldots, a_i\}$ with t-compatibility on $[x_t, \infty)$ for all $t \leq c-1$.
- Suppose a_j is the rightmost activity of your solution of $P_{i,x_1,\ldots,x_{c-1}}$.

Generalized (weighted) activity selection

- For a general c?
- We can define $P_{i,x_1,\ldots,x_{c-1}}$ with $x_1 \geq \cdots \geq x_{c-1}$ be activity selection on $\{a_1, \ldots, a_i\}$ with t-compatibility on $[x_t, \infty)$ for all $t \leq c-1$.
- Suppose a_j is the rightmost activity of your solution of $P_{i,x_1,\ldots,x_{c-1}}$.
- Which subproblem do you reduce to?

Generalized (weighted) activity selection

- For a general c?
- We can define $P_{i,x_1,\ldots,x_{c-1}}$ with $x_1 \geq \cdots \geq x_{c-1}$ be activity selection on $\{a_1, \ldots, a_i\}$ with t-compatibility on $[x_t, \infty)$ for all $t \leq c-1$.
- Suppose a_j is the rightmost activity of your solution of $P_{i,x_1,\ldots,x_{c-1}}$.
- Which subproblem do you reduce to?
- $P_{i',y_1,\ldots,y_{c-1}}$ where $i' = \min\{j-1, \max\{k : f_k \leq \max\{s_j, x_1\}\}\}$ and $y's$ are the $c-1$ smallest numbers in $\{x_1, \ldots, x_{c-1}, s_j\}$, remaining ascending.

Generalized (weighted) activity selection

- For a general c?
- We can define $P_{i,x_1,\ldots,x_{c-1}}$ with $x_1 \geq \cdots \geq x_{c-1}$ be activity selection on $\{a_1,\ldots,a_i\}$ with t-compatibility on $[x_t,\infty)$ for all $t \leq c-1$.
- Suppose a_j is the rightmost activity of your solution of $P_{i,x_1,\ldots,x_{c-1}}$.
- Which subproblem do you reduce to?
- $P_{i',y_1,\ldots,y_{c-1}}$ where $i' = \min\{j-1, \max\{k : f_k \leq \max\{s_j, x_1\}\}\}$ and $y's$ are the $c-1$ smallest numbers in $\{x_1,\ldots,x_{c-1},s_j\}$, remaining ascending.
- Time complexity $= O(n^c)$
  (We don't need to enumerate the rightmost activity a_j. Instead, we only consider $P_{i-1,x_1,\ldots,x_{c-1}}$ and the case a_i is the rightmost activity.)

Picking non-adjacent entries

Problem (picking non-adjacent entries)

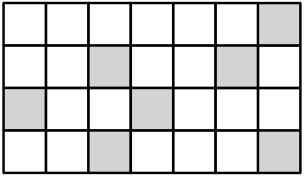
Suppose we have a $4 \times n$ matrix M where each entry contains a number. Our goal is to pick some non-adjacent entries in M such that the sum of these entries is maximized.

Picking non-adjacent entries

Problem (picking non-adjacent entries)

Suppose we have a $4 \times n$ matrix M where each entry contains a number. Our goal is to pick some non-adjacent entries in M such that the sum of these entries is maximized.

- **Example:** $n = 7$



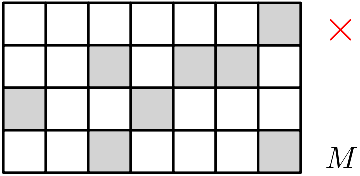
M

Picking non-adjacent entries

Problem (picking non-adjacent entries)

Suppose we have a $4 \times n$ matrix M where each entry contains a number. Our goal is to pick some non-adjacent entries in M such that the sum of these entries is maximized.

- **Example:** $n = 7$



Picking non-adjacent entries

- To get some ideas, let's consider the case where M is $1 \times n$.
- In other words, M is an array of length n.

Picking non-adjacent entries

- To get some ideas, let's consider the case where M is $1 \times n$.
- In other words, M is an array of length n.
- Try to design an DP algorithm for this case?

Picking non-adjacent entries

- To get some ideas, let's consider the case where M is $1 \times n$.
- In other words, M is an array of length n.
- Try to design an DP algorithm for this case?
- $P_i =$ the subproblem of picking non-adjacent entries in $M[1 \ldots i]$

Picking non-adjacent entries

- To get some ideas, let's consider the case where M is $1 \times n$.
- In other words, M is an array of length n.
- Try to design an DP algorithm for this case?
- P_i = the subproblem of picking non-adjacent entries in $M[1 \dots i]$
- $\text{opt}(P_i) = \max\{\text{opt}(P_{i-1}), \text{opt}(P_{i-2}) + M[i]\}$

Picking non-adjacent entries

- To get some ideas, let's consider the case where M is $1 \times n$.
- In other words, M is an array of length n.
- Try to design an DP algorithm for this case?
- $P_i =$ the subproblem of picking non-adjacent entries in $M[1 \dots i]$
- $\text{opt}(P_i) = \max\{\text{opt}(P_{i-1}), \text{opt}(P_{i-2}) + M[i]\}$
- When M is $4 \times n$, we can define P_i as the subproblem of picking non-adjacent entries in the first i column of M.

Picking non-adjacent entries

- To solve the subproblem P_i, we consider which entries in the i-th column are contained in the optimal solution.

Picking non-adjacent entries

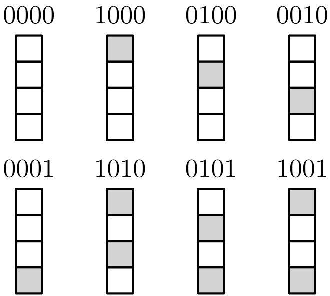
- To solve the subproblem P_i, we consider which entries in the i-th column are contained in the optimal solution.
- The i-th column has 4 entries, resulting in $2^4 = 16$ configurations.

Picking non-adjacent entries

- To solve the subproblem P_i, we consider which entries in the *i*-th column are contained in the optimal solution.
- The *i*-th column has 4 entries, resulting in $2^4 = 16$ configurations.
- Only 8 configurations are valid.

Picking non-adjacent entries

- To solve the subproblem P_i, we consider which entries in the i-th column are contained in the optimal solution.
- The i-th column has 4 entries, resulting in $2^4 = 16$ configurations.
- Only 8 configurations are valid.



Picking non-adjacent entries

- Suppose a solution uses the configuration 0101 in the i-th column.

Picking non-adjacent entries

- Suppose a solution uses the configuration 0101 in the i-th column.
- What does the part in the first $i-1$ columns look like?

Picking non-adjacent entries

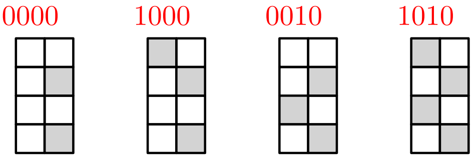
- Suppose a solution uses the configuration 0101 in the i-th column.
- What does the part in the first $i-1$ columns look like?
- The part in the first $i-1$ columns is a solution of P_{i-1}.

Picking non-adjacent entries

- Suppose a solution uses the configuration 0101 in the i-th column.
- What does the part in the first $i-1$ columns look like?
- The part in the first $i-1$ columns is a solution of P_{i-1}.
- However, not every solution of P_{i-1} can be combined with the 0101 in the i-th column to obtain a solution of P_i.

Picking non-adjacent entries

- Suppose a solution uses the configuration 0101 in the i-th column.
- What does the part in the first $i - 1$ columns look like?
- The part in the first $i - 1$ columns is a solution of P_{i-1}.
- However, not every solution of P_{i-1} can be combined with the 0101 in the i-th column to obtain a solution of P_i.
- Only possible configurations for the $(i - 1)$-th column:



Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
  $\mathcal{S} = \{P_{i,x} : i \in \{1, \ldots, n\} \text{ and } x \in \Gamma\}$

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
  $\mathcal{S} = \{P_{i,x} : i \in \{1, \ldots, n\}$ and $x \in \Gamma\}$
- $\text{opt}(P_{i,x}) = \max_{y \in \Gamma, y \wedge x = 0000}(\text{opt}(P_{i-1,y}) + \sum_{k=1}^{4} x_i M_{k,i})$

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
  $\mathcal{S} = \{P_{i,x} : i \in \{1, \ldots, n\}$ and $x \in \Gamma\}$
- $\mathrm{opt}(P_{i,x}) = \max_{y \in \Gamma, y \wedge x = 0000}(\mathrm{opt}(P_{i-1,y}) + \sum_{k=1}^{4} x_i M_{k,i})$
- Final optimum $= \max_{P_{i,x} \in \mathcal{S}} \mathrm{opt}(P_{i,x})$

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
  $\mathcal{S} = \{P_{i,x} : i \in \{1, \ldots, n\}$ and $x \in \Gamma\}$
- $\text{opt}(P_{i,x}) = \max_{y \in \Gamma, y \wedge x = 0000}(\text{opt}(P_{i-1,y}) + \sum_{k=1}^{4} x_i M_{k,i})$
- Final optimum $= \max_{P_{i,x} \in \mathcal{S}} \text{opt}(P_{i,x})$
- Time complexity $= O(n)$

Picking non-adjacent entries

- Why not consider the configurations of the last column when defining the subproblems?
- Let $P_{i,x}$ be the problem of picking non-adjacent entries in the first i column of M such that the i-th column has configuration x.
- $\Gamma = \{0000, 1000, 0100, 0010, 0001, 1010, 0101\}$
  $\mathcal{S} = \{P_{i,x} : i \in \{1, \ldots, n\}$ and $x \in \Gamma\}$
- $\text{opt}(P_{i,x}) = \max_{y \in \Gamma, y \wedge x = 0000}(\text{opt}(P_{i-1,y}) + \sum_{k=1}^{4} x_i M_{k,i})$
- Final optimum $= \max_{P_{i,x} \in \mathcal{S}} \text{opt}(P_{i,x})$
- Time complexity $= O(n)$
- Can be generalized to matrix of size $c \times n$ for a constant c.