Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Dynamic Programming (DP)

– A ***Powerful Paradigm*** for Solving

Combinatorial Optimization Problems

Example 1.

Rod-Cutting Problem

The Rod-Cutting Problem

- You have a rod of length N, where N is a positive integer.

  You're given $\boldsymbol{p_1, p_2, \ldots, p_N}$, where $\boldsymbol{p_i}$ denotes ***the value of a rod with length i***. Determine a way to cut the rod to *maximize the total value*.

  - For example, if $N = 7$ and p_i as follows,

i	1	2	3	4	5	6	7
p_i	1	5	8	9	10	17	17

then, cutting it into $3 + 4$ or just 7 gives a total value of 17.

On the other hand, $1 + 6$ or $2 + 2 + 3$ has a total value of 18.

Observation

- For any $i \geq 0$, let $A(i)$ denote the maximum value obtainable from a rod with length i.
  - Then, $A(0) = 0$.
  - For any $0 < i \leq N$,

The optimal solution comes from ***one of the scenarios*** we have considered.

$$A(i) \; = \; \max_{1 \leq k \leq i} (\, p_k \; + \; A(i-k) \,) \; .$$

worth p_k

worth $A(i-k)$ at max

Observation

- For any $i \geq 0$, let $A(i)$ denote the maximum value obtainable from a rod with length i.
  - Then,

$$A(i) = \begin{cases} 0\,, & \text{if } i = 0, \\ \max\limits_{1 \le k \le i} \left(p_k \, + \, A(i-k) \right)\,, & \text{if } i > 0. \end{cases}$$

  - The above is known as ***a recurrence formula** for $\boldsymbol{A(i)}$*.
    - With the formula, we can compute $A(i)$ for all i either in a ***bottom-up*** or a ***top-down*** manner.

Solving the Problem

$$A(i) = \begin{cases} 0\,, & \text{if } i = 0, \\ \max\limits_{1 \le k \le i} (\, p_k \, + \, A(i-k) \,)\,, & \text{if } i > 0. \end{cases}$$

- Declare an array A of size N.
- We can compute $A(i)$ for all i.

  1. ***<u>Bottom-up</u>*** – Based on the formula, compute $A(0), A(1), \ldots, A(N)$ in order.
  2. ***<u>Top-down</u>*** – Use a recursion function to compute $A(N)$. During the process, ***recurse on*** $A(i-k)$ ***<u>only if</u>*** *<u>it has not been computed yet</u>*.

- The computation takes $O(N^2)$.

Solving the Problem

$$A(i) = \begin{cases} 0\,, & \text{if } i = 0, \\ \max_{1 \le k \le i} \left(p_k + A(i-k) \right)\,, & \text{if } i > 0. \end{cases}$$

- Declare an array A of size N.
- We can compute $A(i)$ for all i.
  1. ***Bottom-up***
  2. ***Top-down***
- The computation takes $O\left(N^2\right)$.
- By recording the choices made during the computation process, we can construct the solution backward.

That is, which k results in the maximum value for $A(i)$.

The Dynamic Programming Paradigm

Dynamic Programming Paradigm

- To apply the dynamic programming technique, we proceed in following steps.

Requires *observation* & *creativity*.

1. Define a ***suitable subproblem*** that is *expressed* with ***a few indexes***.

2. Write down ***the recurrence formula*** for the solution of the subproblem, using solutions for instances of smaller sizes.

3. Compute the answer according to the recurrence formula.

Elements of Dynamic Programming

- Problems that can be solved via dynamic programming exhibits the following properties.

  1. **Optimal Substructure** – An optimal solution to the problem contains within it optimal solutions to subproblems.
  2. Overlapping Subproblems.
  3. Memorization.

- With the above, suitable problems can be defined, and recurrence formulas can be written down.

Example 2.

Matrix Chain Multiplication

See also ProgHW-IV-D

Matrix Chain Multiplication

- Suppose that for any $A \in \mathbb{R}^{p\times q}$ and any $B \in \mathbb{R}^{q\times r}$, computing $A \times B$ takes $p \times q \times r$ number of multiplications.

- Given $n+1$ positive integers $p_1, p_2, \dots, p_{n+1}$, consider the scenario that we are to compute

$$M_1 \times M_2 \times \cdots \times M_n ,$$

  where $M_i \in \mathbb{R}^{p_i \times p_{i+1}}$ is a $p_i \times p_{i+1}$ matrix.

- Find the ***optimal way*** to computing $M_1 \times M_2 \times \cdots \times M_n$ such that the total *number of multiplications* is *minimized*.

Matrix Chain Multiplication

- For example, for $M_1 \times M_2 \times M_3 \times M_4$,
  there are 5 different ways to do it.
  - $(M_1(M_2 (M_3M_4)))$, $(M_1 ((M_2M_3) M_4))$, $((M_1M_2) (M_3M_4))$,
  - $((M_1 (M_2M_3)) M_4)$, $(((M_1M_2) M_3) M_4)$.

- If $(p_1, \ldots, p_5) = (13, 5, 89, 3, 34)$, then
  - $(M_1(M_2 (M_3M_4)))$ takes $(89 * 3 * 34) + (5 * 89 * 34) + (13 * 5 * 34) = 26418$ multiplications.

Matrix Chain Multiplication

- For example, for $M_1 \times M_2 \times M_3 \times M_4$,
  there are 5 different ways to do it.
  - $(\,M_1(\,M_2\,(M_3M_4)\,)\,)$, $(\,M_1\,(\,(M_2M_3)\,M_4\,)\,)$, $(\,(M_1M_2)\,(M_3M_4)\,)$,
  - $(\,(\,M_1\,(M_2M_3)\,)\,M_4\,)$, $(\,(\,(M_1M_2)\,M_3\,)\,M_4\,)$.
- If $(p_1, \dots, p_5) = (13, 5, 89, 3, 34)$, then
  - The 5 different ways take 26418, 4055, 54201, $\mathbf{2856}$, and 10582 multiplications, respectively.
  - $(\,(\,M_1\,(M_2M_3)\,)\,M_4\,)$ is the optimal way.

Define a Proper Subproblem

- Given $n+1$ positive integers $p_1, p_2, \ldots, p_{n+1}$, consider the scenario that we are to compute

$$M_1 \times M_2 \times \cdots \times M_n \,,$$

where $M_i \in \mathbb{R}^{p_i \times p_{i+1}}$ is a $p_i \times p_{i+1}$ matrix.

- For any $[\ell, r]$ with $1 \leq \ell \leq r \leq n$, let $m[\ell, r]$ denote the minimum number of multiplications required by

$$M_\ell \times M_{\ell+1} \times \cdots \times M_r \,.$$

Derive the Recurrence Formula

- For any $[\ell, r]$ with $1 \le \ell \le r \le n$,
  let $m[\ell, r]$ denote the minimum number of multiplications required by

$$M_\ell \times M_{\ell+1} \times \cdots \times M_r \; .$$

- For $1 \le \ell = r \le n$, we have $m[\ell, r] = 0$.
- For $\ell < r$,

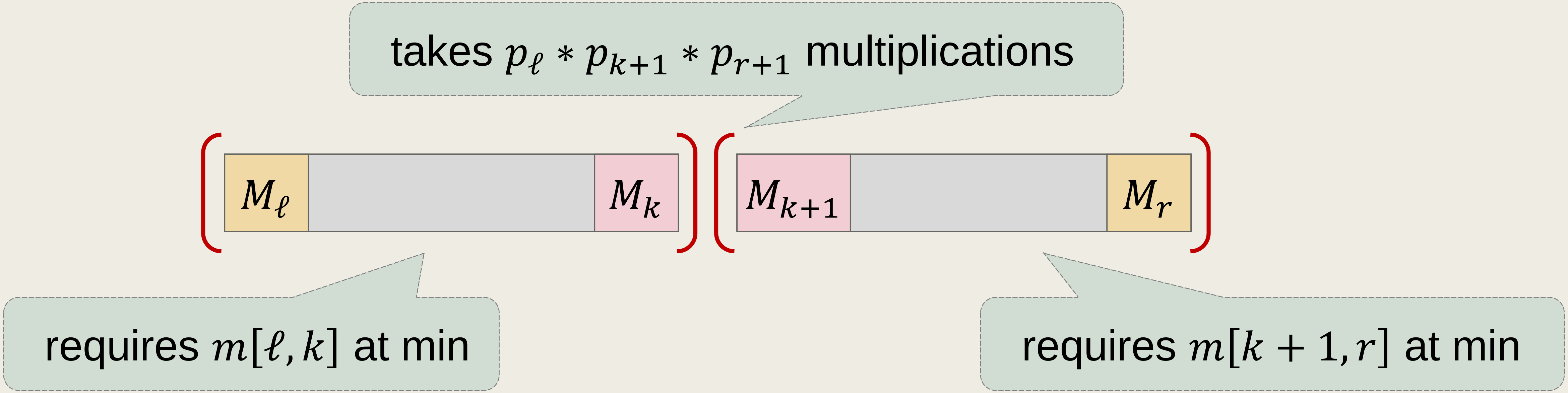
$$m[\ell, r] \; = \; \min_{\ell \le k < r} \left(\; m[\ell, k] \; + \; m[k+1, r] \; + \; p_\ell * p_{k+1} * p_{r+1} \; \right) \; .$$

- For any $[\ell, r]$ with $1 \le \ell \le r \le n$,
  let $m[\ell, r]$ denote the minimum number of multiplications required by

$$M_\ell \times M_{\ell+1} \times \cdots \times M_r \ .$$

- For $\ell < r$,

$$m[\ell, r] \ = \ \min_{\ell \le k < r} \left(\ m[\ell, k] \ + \ m[k+1, r] \ + \ p_\ell * p_{k+1} * p_{r+1} \ \right) .$$



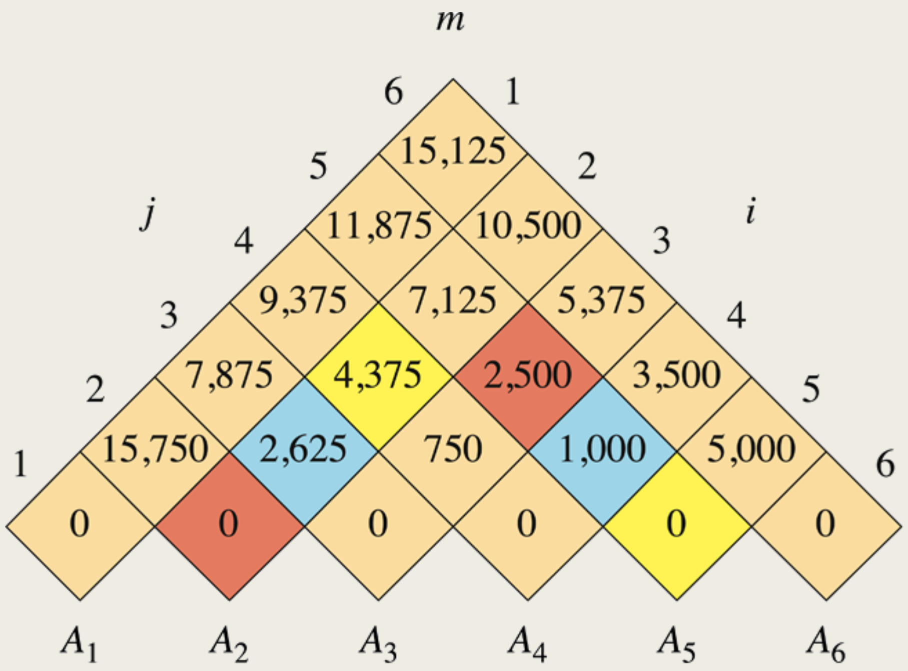
Fill-in the Table

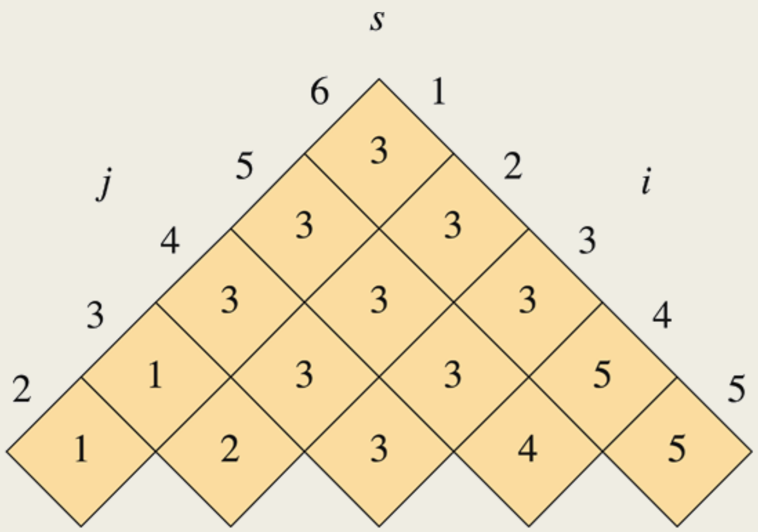
- Declare a matrix m with size $n \times n$.

- For any segment $I = [\ell, r]$,
  computing $m[I]$ requires the values of $m[I']$ for all I' with $|I'| < |I|$.
  - Note - Top-down computation using recursion is easier.

- The time it takes is $O\left(n^3\right)$.

- Declare a matrix m with size $n \times n$.
- For any segment $I = [\ell, r]$,
  computing $m[I]$ requires the values of $m[I']$ for all I' with $|I'| < |I|$.





matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Example 3.

The Knapsack Problem

The Knapsack Problem

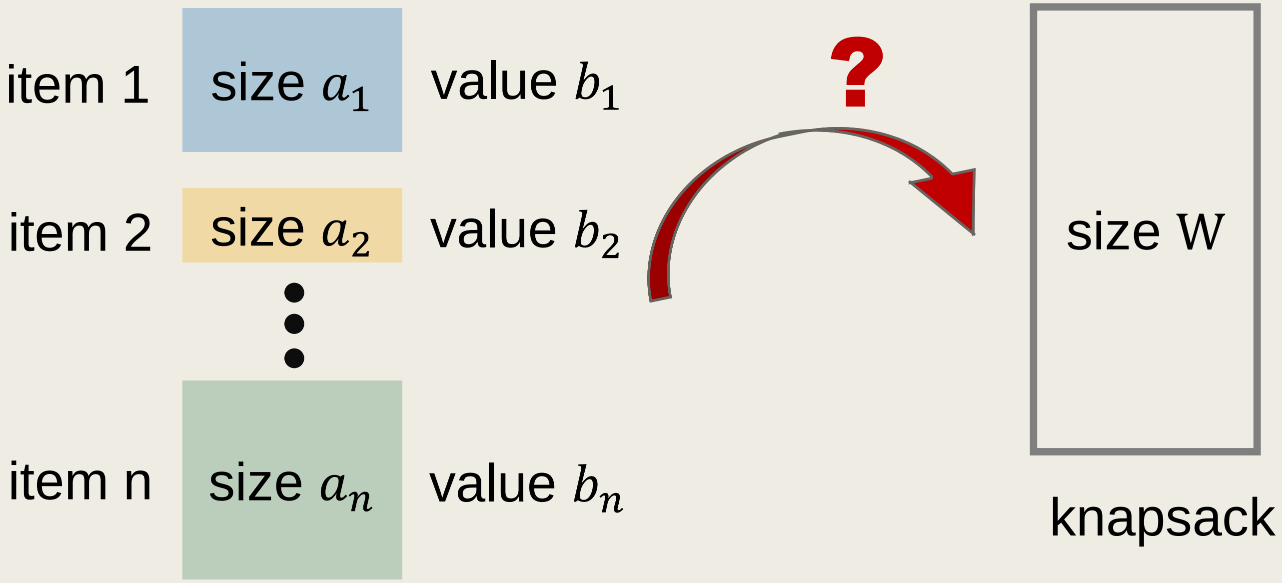
- Given n items $I_1 = (a_1, b_1), I_2 = (a_2, b_2), \ldots, I_n = (a_n, b_n)$, where $\boldsymbol{a_i}$ **and** $\boldsymbol{b_i}$ are ***the size*** and ***the value*** of the i^{th}-item, and a ***knapsack size*** $\boldsymbol{W}$,

  compute a subset $A \subseteq \{1,2, \ldots, n\}$ such that $\sum_{i \in A} a_i \leq W$ and $\sum_{i \in A} b_i$ is maximized.

  – That is, select a subset of items that have ***size at most W*** such that the ***total value*** of the selected items is ***maximized***.

The Knapsack Problem

– That is, select a subset of items that have ***size at most W*** such that the ***total value*** of the selected items is ***maximized***.



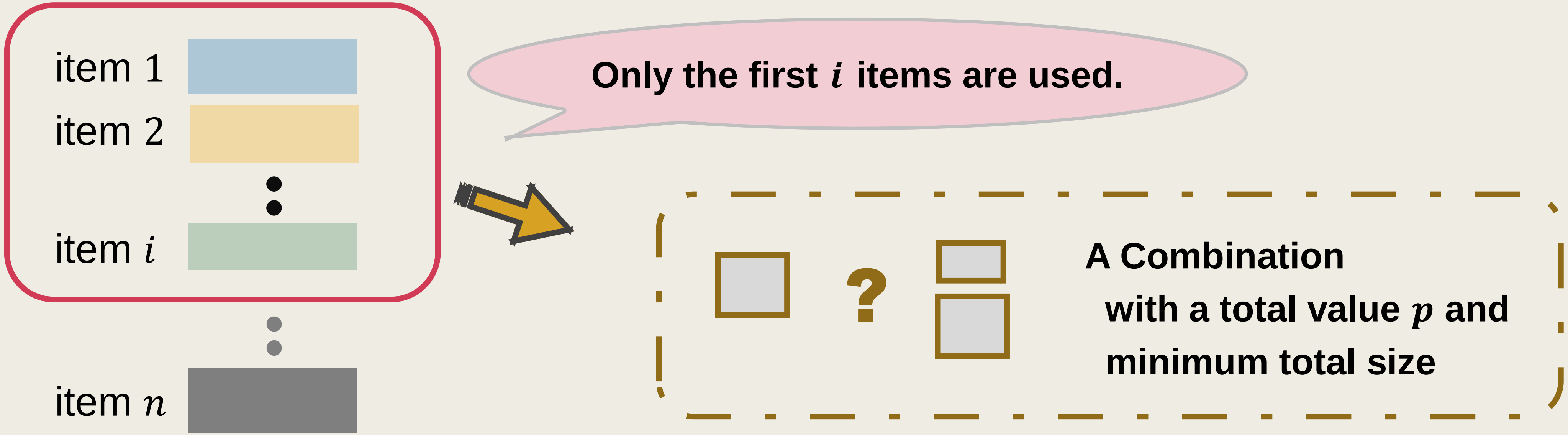
To maximize the **total value** to be put in the knapsack

Define a Proper Subproblem

- For any $0 \le i \le n$ and $p \ge 0$,

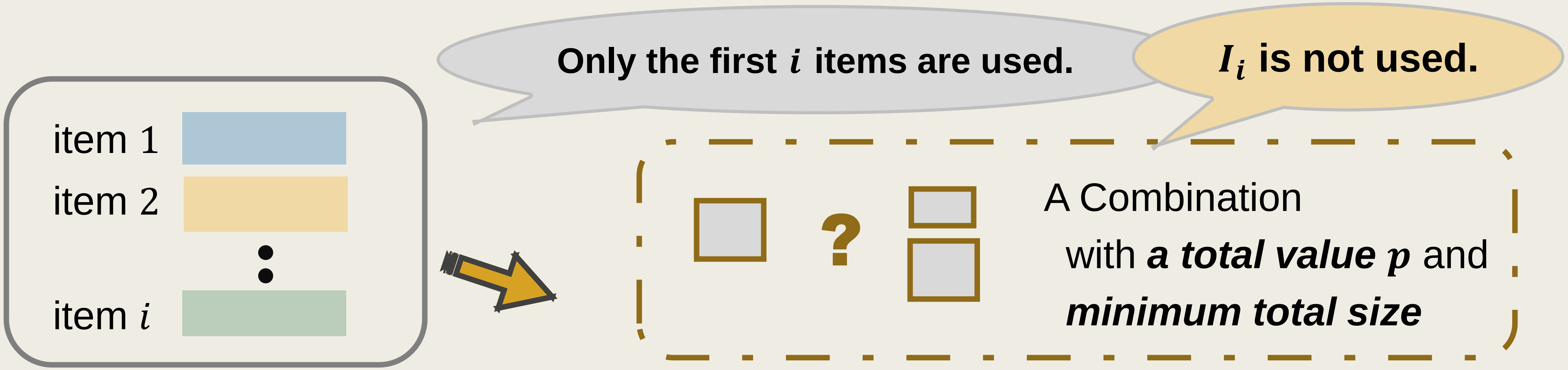
  let $A(i, p)$ denote the **minimum total size** it requires to get a **total value of p using only the first i items**.

  - $A(i, p)$ is defined to be ∞ *if no such combination* exists.



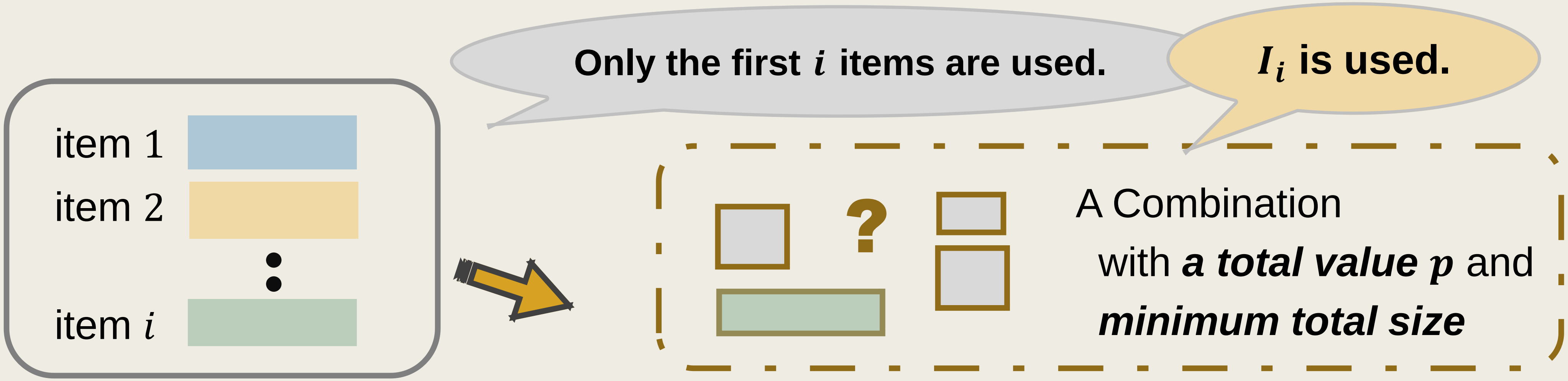
Derive the Recurrence Formula

- Consider an "***optimal combination***" for $A(i, p)$.
  - There are *only two possibilities* – it either **contains $\boldsymbol{I_i}$** or **excludes $\boldsymbol{I_i}$**.
  - If I_i is not contained,
    then it must be an ***optimal combination*** for $A(i-1, p)$.



Derive the Recurrence Formula

- Consider an "***optimal combination***" for $A(i, p)$.
  - There are *only two possibilities* – it either **contains I_i** or **excludes I_i**.
  - If I_i is contained, then it consists of an ***optimal combination*** for $A(i-1, p-b_i)$ and I_i.



The Recurrence Formula for $A(i, p)$

- Based on the observation,
  we can write down the recurrence for $A(i, p)$ as follows.

$$A(i,p) = \begin{cases} \begin{cases} \infty, & \text{if } p < 0 \\ \min\{ A(i\text{-}1,p),\ A(i\text{-}1, p\text{-}b_i) + a_i \}, & \text{if } p \geq 0 \end{cases}, & \text{for } i > 0, \\ \\ \begin{cases} 0, & \text{if } p = 0 \\ \infty, & \text{if } p \neq 0 \end{cases}, & \text{for } i = 0. \end{cases}$$

Solving the Knapsack Problem via DP

- Based on the recurrence formula,
  we can compute $A(i, p)$ for all $0 \leq i \leq n$ and $0 \leq p \leq P$,
  where $\mathrm{P} \coloneqq \sum_{1 \leq i \leq n} b_i$ is the total value of the items.

- The answer is then given by the maximum p such that $A(n, p) \leq W$.

- The time complexity is $O(n \cdot P)$.
  - Note that, this is ***<u>not a polynomial-time algorithm</u>***.
  - We call it a "pseudo-polynomial time" algorithm.

It is not efficient.

Recurrence Formula is not Unique

- The following is an alternative way to defining a proper subproblem.
- For any $0 \leq i \leq n$ and $w \geq 0$,
  let $B(i, w)$ denote the **maximum total value** we can get
  with a **total size $\boldsymbol{w}$ <u>using only the first $\boldsymbol{i}$ items</u>**.
  - $B(i, w)$ is defined to be $-\infty$ *if no such combination* exists.
- As an exercise, write down the recurrence formula for $B(i, w)$ so that the Knapsack problem can be solved in $O(n^2W)$ time.
  - Also describe & explain what the answer is.

Example 4.

The Longest Common Subsequence (LCS) Problem

String Alignment in DNA Sequence

- Suppose that we are given two DNA sequences,
  each of which is a string consisting of the characters ‘C’, ‘G’, ‘T’, ‘A’.
  - For example, $s_1 = AGCAT$ and $s_2 = GAC$.

- We want to compute a string s *with a maximum length* such that s is a subsequence of both s_1 and s_2.
  - For example, both GC and GA are common subsequences of s_1 and s_2.

The ***longer*** *a common subsequence* is,
the ***more similar*** the two DNA sequences are.

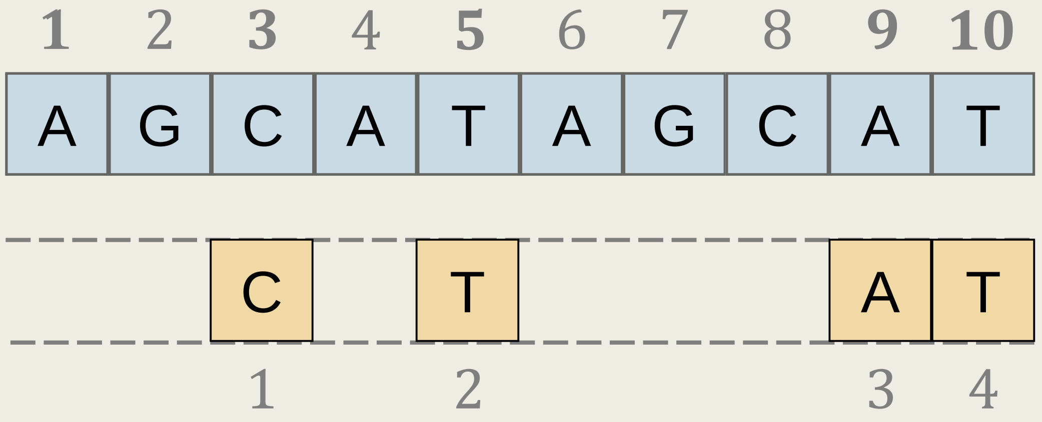
Sequence and Subsequence

- Let $s = s_1 s_2 \cdots s_n$ be a string of length n.
- We say that a string $t = t_1 t_2 \cdots t_k$ is a ***<u>subsequence</u>*** of s, if there exists *<u>indexes</u>* $j_1, j_2, \ldots, j_k$ with $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ such that

$$t_i = s_{j_i} \text{ for all } 1 \leq i \leq k.$$

$s = AGCATAGCAT$

$t = CTAT$



There is a way to align t with s.

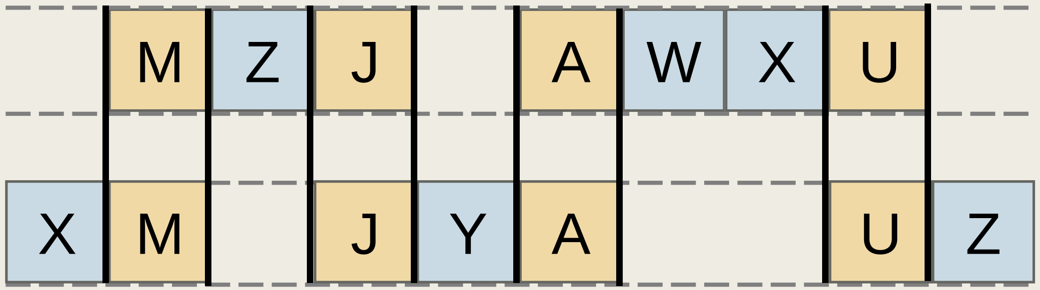
The Longest Common Subsequence (LCS) Problem

- In the LCS problem, we are given two strings s_1 and s_2.
  The goal is to compute a common subsequence t of s_1 and s_2 such that the length of t is the longest possible.
  - For example, if $s_1 = MZJAWXU$ and $s_2 = XMJYAUZ$, then one optimal solution is $t = MJAU$.

Find an optimal way to align the two strings.

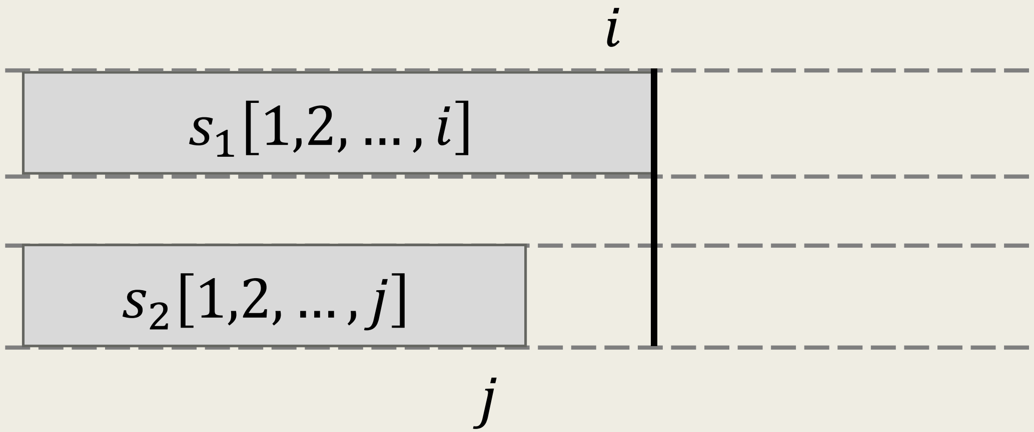
$s_1 = MZJAWXU$

$s_2 = XMJYAUZ$



Define a Suitable Subproblem

- Let $n = |s_1|$ and $m = |s_2|$
- For any $1 \leq i \leq n$ and $1 \leq j \leq m$,
  define $L(i, j)$ to be the length of the ***optimal alignment*** of
  $\boldsymbol{s_1[1 \ldots i]}$ and $\boldsymbol{s_2[1 \ldots j]}$.



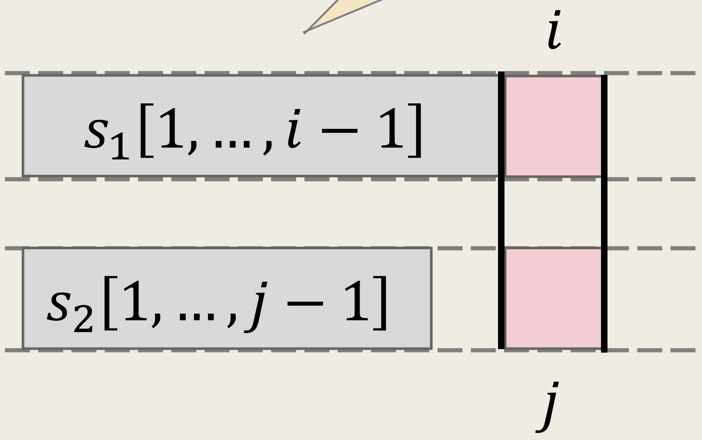
Make Observations on the Optimal Solution

- For any $1 \leq i \leq n$ and $1 \leq j \leq m$, define $L(i, j)$ to be the length of the ***optimal alignment*** of $\boldsymbol{s_1[1 \ldots i]}$ and $\boldsymbol{s_2[1 \ldots j]}$.

- The optimal alignment must be one of the following 3 cases.

  1. If $s_1[i] = s_2[j]$, then *there exists an optimal solution* that align $s_1[i]$ with $s_2[j]$.
     - The rest is the optimal alignment between $s_1[1 \ldots i-1]$ and $s_2[1 \ldots j-1]$.
     - That is, $L(i-1, j-1)$.

$L(i-1,\ j-1)$.



Make Observations on the Optimal Solution

- For any $1 \leq i \leq n$ and $1 \leq j \leq m$, define $L(i, j)$ to be the length of the ***optimal alignment*** of $\boldsymbol{s_1[1 \dots i]}$ and $\boldsymbol{s_2[1 \dots j]}$.

- The optimal alignment must be one of the following 3 cases.

2. If $s_1[i] \neq s_2[j]$, then either $s_1[i]$ or $s_2[j]$ is not aligned in *the optimal solution.*

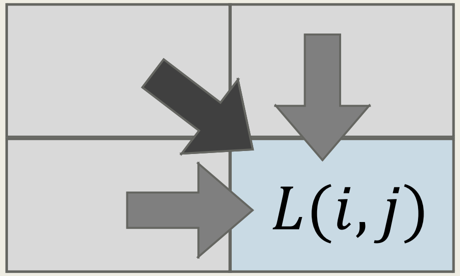
    - The optimal alignment is either $L(i-1, j)$ or $L(i, j-1)$.

The Recurrence Formula for $L(i,j)$

- Based on the observation,
  we can write down the recurrence for $L(i,j)$ as follows.

$$L(i,j) = \begin{cases} 0, & \text{if } \min(i,j) = 0, \\ \begin{cases} L(i-1, j-1) + 1, & \text{if } s_1[i] = s_2[j] \\ \max\{\, L(i-1,j),\ L(\, i, j-1\,)\,\}, & \text{if } s_1[i] \neq s_2[j] \end{cases}, & \text{otherwise.} \end{cases}$$

- By the recurrence formula, we can compute $L(i,j)$ for all i and j in $O(nm)$ time.
- The answer is $L(n,m)$.



Example 5.

Optimal Binary Search Tree

The Scenario

- Suppose that you have a set of keywords

$$k_1 \le k_2 \le \ldots \le k_n .$$

Furthermore, consider $I_0 = (-\infty, k_1), I_1 = (k_1, k_2), \ldots, I_n = (k_n, \infty)$.

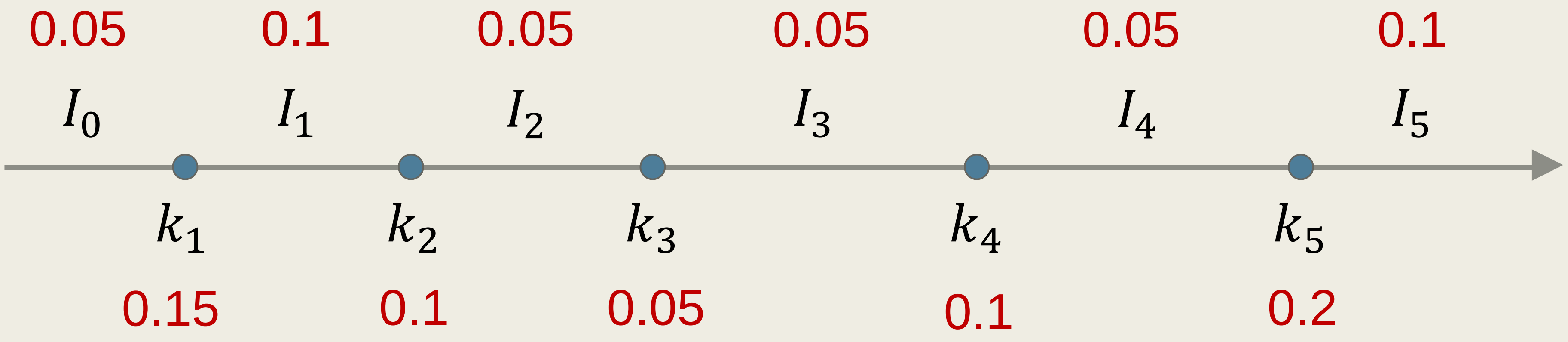
- Suppose that you are given the probability distribution that a key is to be searched.
  - p_i : the probability that k_i is to be searched.
  - q_i : the probability that a key $k \in I_i$ is to be searched.
- Furthermore, $$\sum_{1 \le i \le n} p_i + \sum_{0 \le i \le n} q_i = 1\,.$$

- Build a BST that *minimizes* ***the expected search time***.

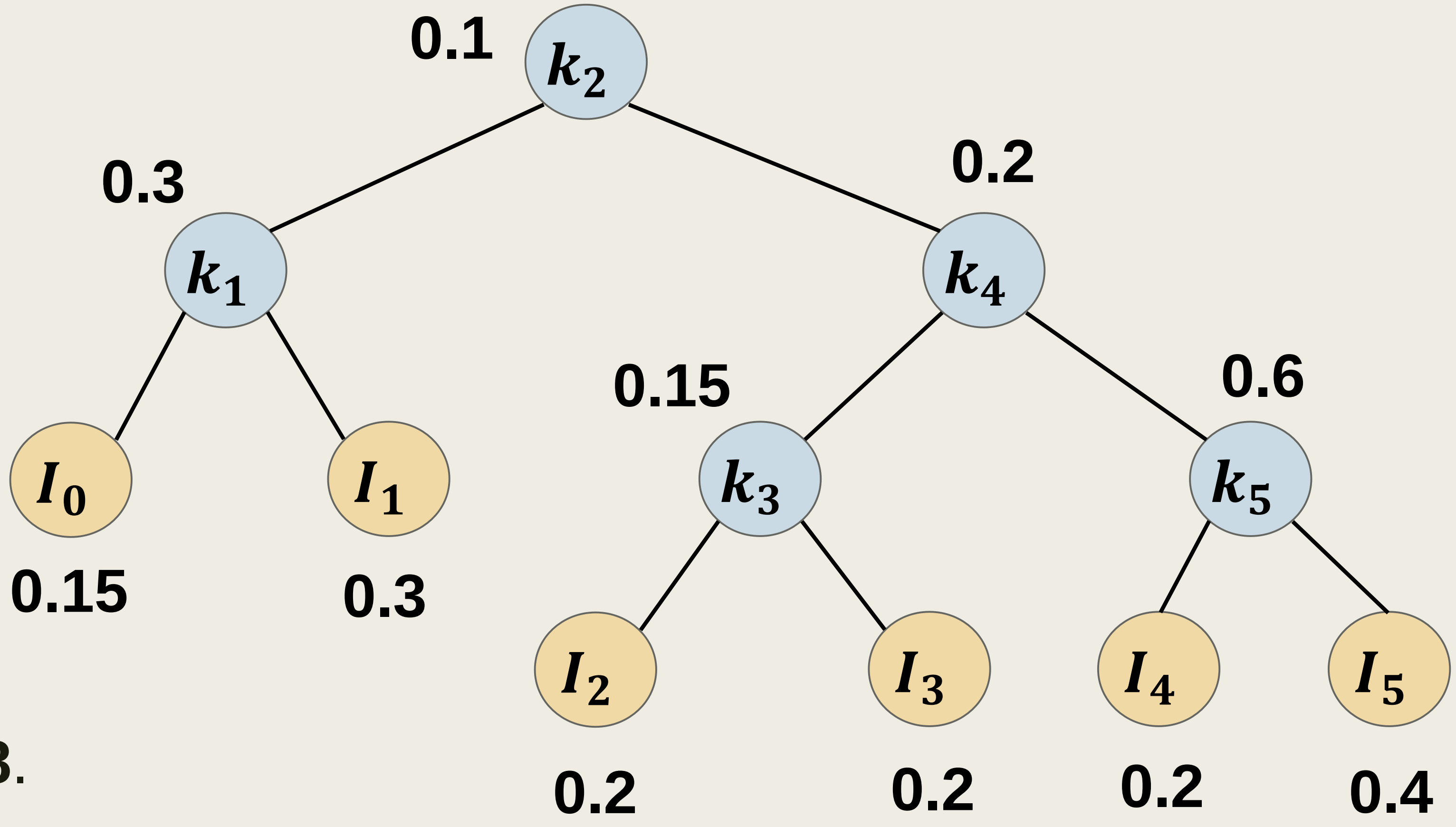
Optimal BST

- For example, consider the following distribution.

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1



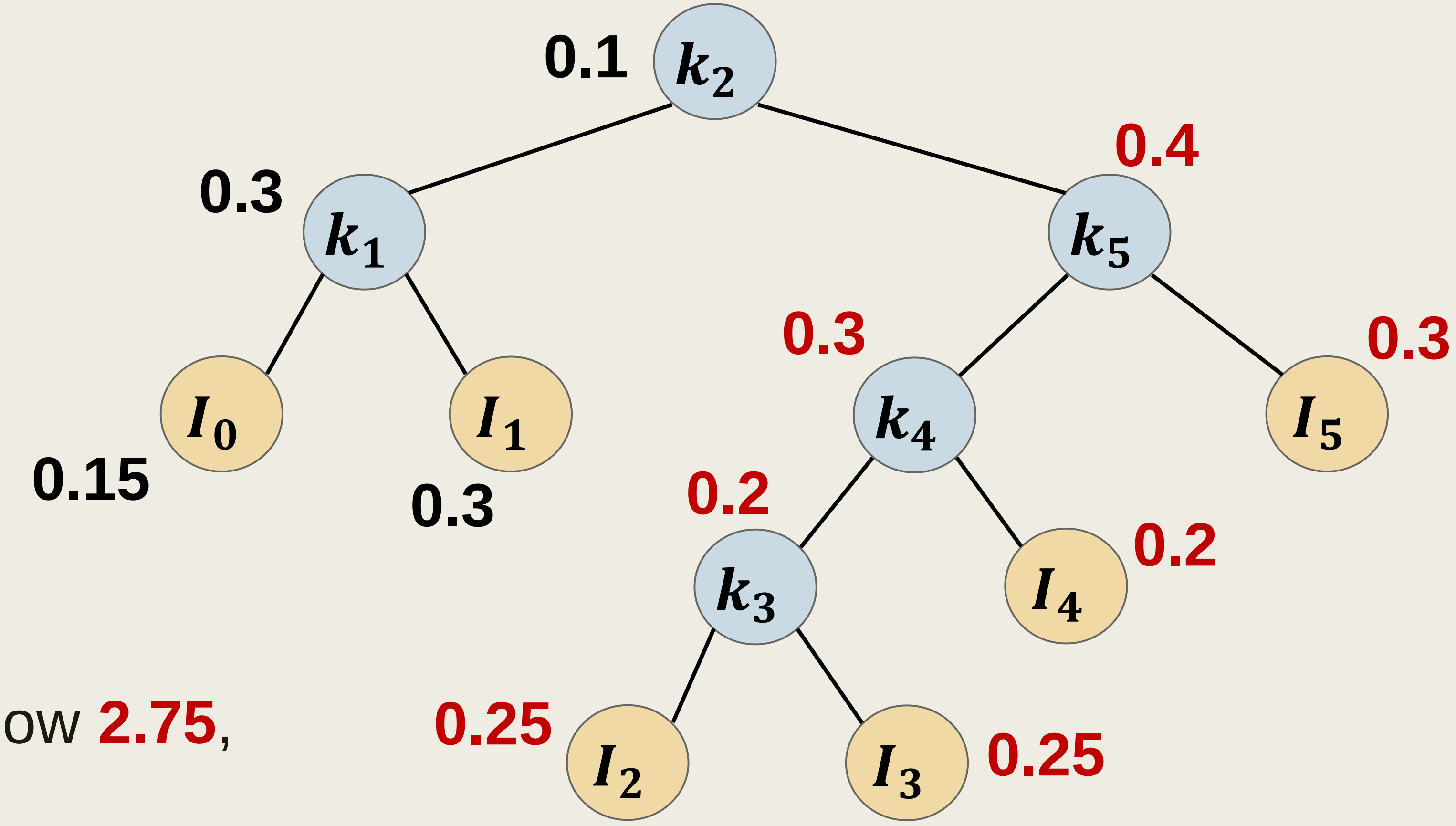
Expected Cost



- The overall cost is **2.8**.

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

Expected Cost



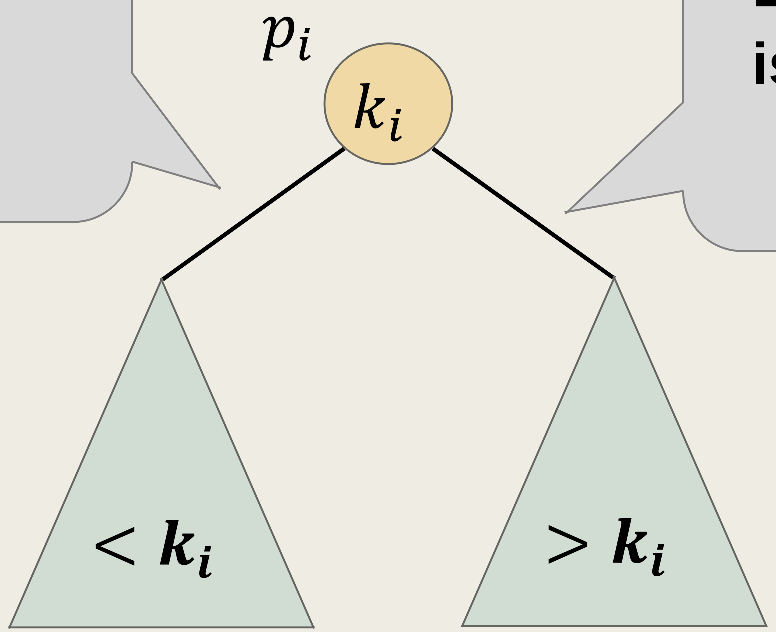
- The overall cost is now **2.75**, instead of **2.8**.

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

Observation and Optimal Substructure

- Since a BST is to be built,
  one of the key k_i has to be the root of the BST.

Expected cost to the subtree is $\displaystyle\sum_{0 \le j < i} p_j + \sum_{0 \le j \le i} q_j$.



Expected cost to the subtree is $\displaystyle\sum_{i < j \le n} p_j + \sum_{i \le j \le n} q_j$.

Optimal BST
for $k_1, \ldots, k_{i-1}$.
(recursive problem)

Optimal BST
for $k_{i+1}, \ldots, k_n$.
(recursive problem)

Define a Suitable Subproblem

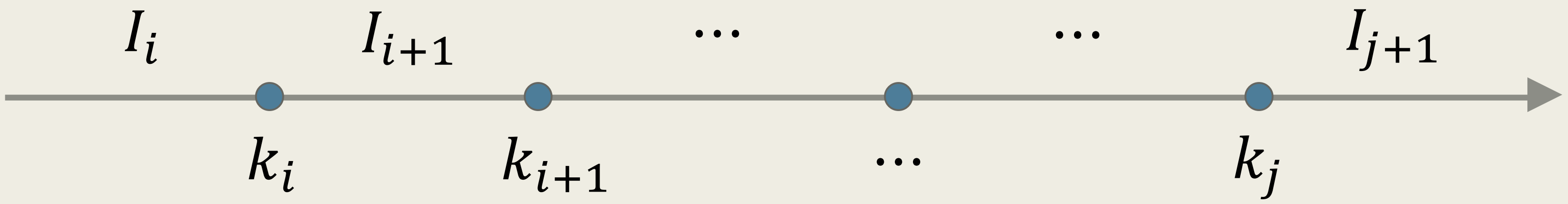
- For any i, j with $1 \leq i \leq j \leq n$,

  let $E[i, j]$ be the expected cost of the optimal BST for $k_i, \ldots, k_j$.

  - Also let

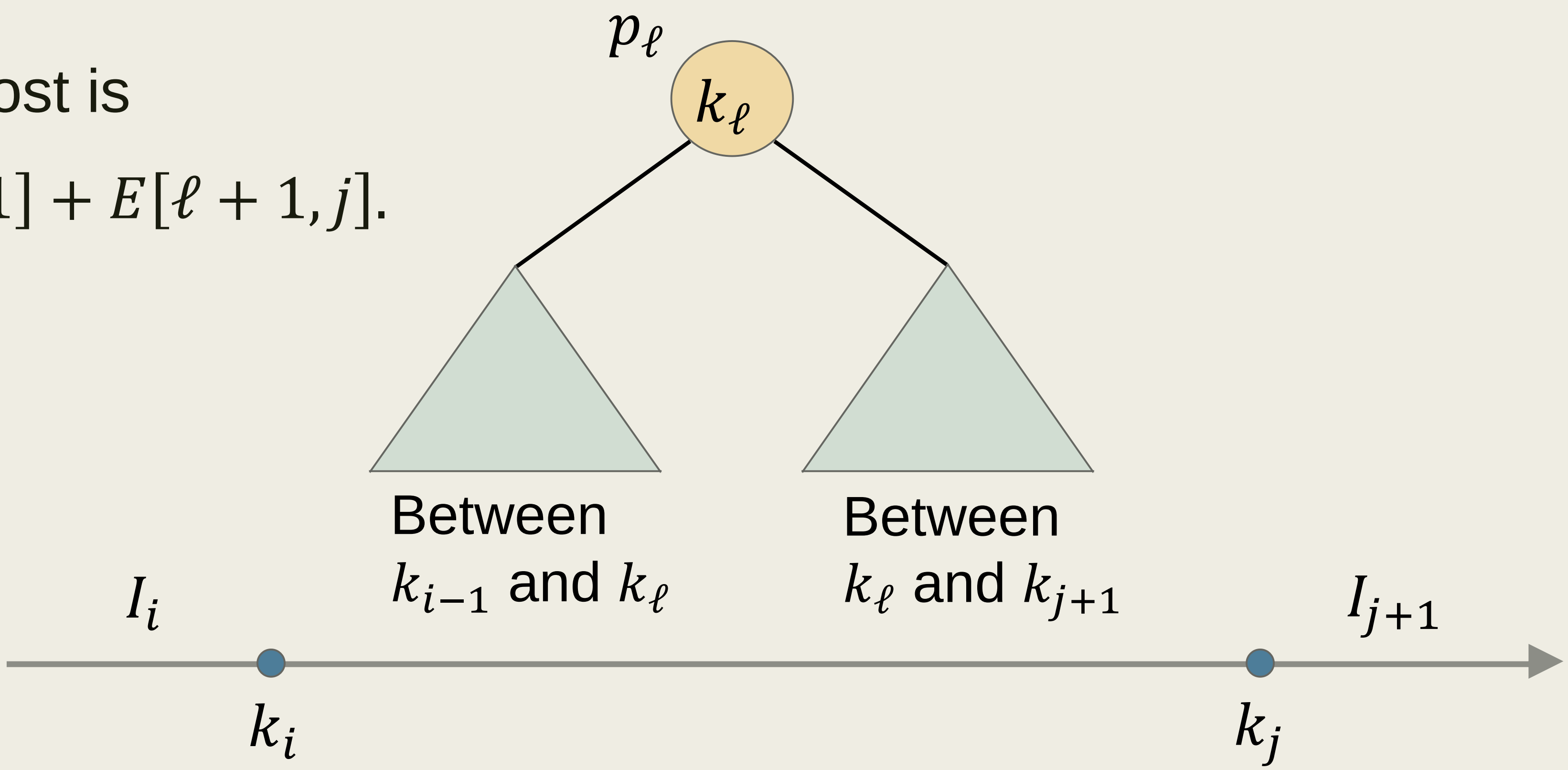
$$p(i,j) \coloneqq \sum_{i \leq \ell \leq j} p_\ell + \sum_{i \leq \ell \leq j+1} q_\ell$$

    be the cumulative probability that a key within (k_{i-1}, k_{j+1}) is to be searched.



- For any i, j with $1 \le i \le j \le n$,
  let $E[i, j]$ be the expected cost of the optimal BST for $k_i, \ldots, k_j$.
  - Some k_ℓ with $i \le \ell \le j$ has to be the root.

- The expected cost is
  $p(i, j) + E[i, \ell - 1] + E[\ell + 1, j]$.



The Recurrence Formula for $E[i,j]$

- We have the following recurrence formula.

$$E[i,j] = \begin{cases} 0, & \text{if } i > j, \\ \min_{i \le \ell \le j} \begin{pmatrix} E[i,\ell-1] + E[\ell+1,j] \\ +p(i,j) \end{pmatrix}, & \text{otherwise.} \end{cases}$$

where $$p(i,j) := \sum_{i \le \ell \le j} p_\ell + \sum_{i \le \ell \le j+1} q_\ell \ .$$

- In time $O(n^3)$, we can compute $E[i,j]$ for all $1 \le i \le j \le n$.