Design and Analysis of Algorithms
Dynamic Programming (III)

Design and Analysis of Algorithms
Dynamic Programming (III)

Return on Investment

Problem. Given m coins, n projects, and function $f_i(x)$ denotes profit of investing x on the i-th project.

- Find the optimal investment scheme that maximizes profit.

Solution: a vector $(x_1, x_2, \ldots, x_n)$, x_i: investment on project i

Optimized function: $\max \sum_{i=1}^{n} f_i(x_i)$

Constraints: $x_1 + x_2 + \cdots + x_n = m$, $x_i \in N$

Table: 5 coins on 4 projects

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

Subproblems and Computation Order

Subproblem: defined by two parameters k and x

- k: invest on the $1, 2, \ldots, k$ projects
- the total investment is less than x

The parameter in matrix multiplication chain is a tuple of index, of the same type

(k, x) are of different types $\rightsquigarrow$ 2-dimension dynamic programming

---

Original problem: $k = n$, $x = m$

Computation oder: $k = 1, 2, \ldots, n$; for any k, $x = 1, 2, \ldots, m$

- can be implemented by two level loop

Iteration Relation of Optimized Function

Optimized function $\mathrm{OPT}_k(x)$: the maximal profit of investing x coins on the first k projects

Iteration relation: Determine $\mathrm{OPT}_k(x)$ from $\mathrm{OPT}_{k-1}(y)$ for all $y \leq x$

$$\mathrm{OPT}_k(x) = \max_{0 \leq x_k \leq x} \{f_k(x_k) + \mathrm{OPT}_{k-1}(x - x_k)\}, k > 1$$

$$\mathrm{OPT}_1(x) = f_1(x), k = 1$$

Demo of $k = 2$

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

$k = 1$ corresponds to the initial values: $\mathsf{OPT}_1(1) = 11$, $\mathsf{OPT}_1(2) = 12$, $\mathsf{OPT}_1(3) = 13$, $\mathsf{OPT}_1(4) = 14$, $\mathsf{OPT}_1(5) = 15$

$\mathsf{OPT}_2(1) = \max\{f_1(1), f_2(1)\} = 11$

$\mathsf{OPT}_2(2) = \max\{f_2(2), \mathsf{OPT}_1(1) + f_2(1), \mathsf{OPT}_1(2)\} = 12$

$\mathsf{OPT}_2(3) = \max\{f_2(3), \mathsf{OPT}_1(1) + f_2(2), \mathsf{OPT}_1(2) + f_2(1), \mathsf{OPT}_1(3)\} = 16$

Similarly, we can compute $\mathsf{OPT}_2(4) = 21$, $\mathsf{OPT}_2(5) = 26$

Memo and Solution

x	$\mathsf{OPT}_1(\cdot)$	$s_1(\cdot)$	$\mathsf{OPT}_2(\cdot)$	$s_2(\cdot)$	$\mathsf{OPT}_3(\cdot)$	$s_3(\cdot)$	$\mathsf{OPT}_4(\cdot)$	$s_4(\cdot)$
1	11	1	11	0	11	0	20	1
2	12	2	12	0	13	1	31	1
3	13	3	16	2	30	3	33	1
4	14	4	21	3	41	3	50	1
5	15	5	26	4	43	4	61	1

- $\mathsf{OPT}_k(x)$ records maximized profit of investing x coins on the first k projects
- $s_k(x)$ records the investment on k-th project

$$s_4(5) = 1 \Rightarrow x_4 = 1, s_3(5-1) = s_3(4)$$
$$s_3(4) = 3 \Rightarrow x_3 = 3, s_2(4-3) = s_2(1)$$
$$s_2(1) = 0 \Rightarrow x_2 = 0, s_1(1-0) = s_1(1)$$
$$s_1(1) = 1 \Rightarrow x_1 = 1$$

Solution: $(x_1 = 1, x_2 = 0, x_3 = 3, x_4 = 1)$, $\mathsf{OPT}_4(5) = 61$

Complexity Analysis

Memo table is a matrix of m rows (total number of coins) and n columns (total number of projects), totally mn items:

$$\mathsf{OPT}_k(x) = \max_{0 \le x_k \le x} \{f_k(x_k) + \mathsf{OPT}_{k-1}(x - x_k)\}, k > 1$$

$$\mathsf{OPT}_1(x) = f_1(x), k = 1 \quad \text{//initial values}$$

The cost of computing $\mathsf{OPT}_k(x)$: there are possible $x + 1$ different choices of $x_k \Rightarrow$ $x + 1$ times add + x times compare

- Total number of add: $\sum_{k=2}^{n} \sum_{x=1}^{m} (x + 1) = \frac{1}{2}(n - 1)m(m + 3)$
- Total number of compare: $\sum_{k=2}^{n} \sum_{x=1}^{m} x = \frac{1}{2}(n - 1)m(m + 1)$

Time complexity $W(n) = O(nm^2)$, space complexity is $O(mn)$

Motivation

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

- His bag (or "knapsack") will hold a total weight if at most W pounds.
- He want to figure out the most valuable combination of items he can fit into his bag, quickly.

There are two version of this problem:

- with repetition: there are unlimited quantities of each item available
- without repetition: there is one of each item (the bugalar has broken into an art gallery)

Neither version is likely to have a polynomial-time algorithm.

Formal Motivation

If the above motivation seems frivolous

- replace "weight" with "CPU time"
- replace "only W pounds can be taken" with "only W units of CPU times are available"

CPU time can also replaced by bandwidth

The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

Outline

Knapsack with Repetition

Problem. Given n items and a knapsack, item i weighs $w_i > 0$ and has value $v_i > 0$, knapsack has capacity of W

Goal. Fill knapsack so as to maximize total value.

Table: knapsack instance, $W = 11$

i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7

- Greedy by value (maximum v_i first): $\{28 \times 1, 6 \times 2\}$ has value 40
- Greedy by weight (minimum w_i first): $\{1 \times 11\}$ has value 11
- Greedy by ratio (maximum ratio v_i/w_i first): $\{5, 2 \times 2\}$ has value 40

Observation. None of the above greedy algorithms is optimal.

Modeling

Solution vector: $x = (x_1, x_2, \ldots, x_n) \in (\mathbb{N})^n$, x_i is the number of item i

Optimized goal: $\max \sum_{i=1}^{n} v_i x_i$

Constraint: $\sum_{i=1}^{n} w_i x_i \leq W$

- linear programming: find min or max of optimized function with linear constraints
  - integer programming: linear programming when x_i are non-negative integers

Looking for Subproblems

As always, the main question in dynamic programming is:

what are subproblems

- It usually takes a little experimentation to figure out exactly what works.

For the knapsack problem with repetition, we can shrink the original problem in three ways:

- smaller knapsack capacities $w \leq W$
- fewer items (for instance, items $1, 2, \ldots, j$ for $j \leq n$)
- combination of the above

Dynamic Programming: Failed Attempt

Initial solution: put restriction on the item number, define $\mathrm{OPT}(j)$ = maximum value achievable with items $1, \ldots, j$ with weight capacity limit W.

Case 1. $\mathrm{OPT}(j)$ does not select item j.

- select best of $\{1, 2, \ldots, j-1\} \rightsquigarrow$ satisfy optimal substructure property (proof via exchange argument)

Case 2. $\mathrm{OPT}(j)$ selects item j

- We don't know the consequence of selecting item j, cause it will change weight limit of subproblems $\rightsquigarrow$ cannot make a decision

We need also consider restriction of capacity to introduce more fine-grained subproblems!

Dynamic programming: Adding a New Variable

Define $\text{OPT}_j(w)$ = max value of choosing from items $\{1, \ldots, j\}$ with weight limit w.

Case 1. $\text{OPT}_j(w)$ does not select item j

- $\text{OPT}_j(w)$ selects best of $\{1, 2, \ldots, j-1\}$ using weight limit w.

Case 2. $\text{OPT}_j(w)$ selects item j (at least 1)

- New weight limit $= w - w_j$.
- $\text{OPT}_j(w)$ selects best of $\{1, 2, \ldots, j\}$ using this new weight limit (cause we allow repetition)

Both cases satisfy optimal substructure property

Wrap it Up

Subproblem: defined by two variables j and w

- j: select from subset of $\{1, 2, \ldots, j\}$
- w: limit on capacity (weight)

$\mathsf{OPT}_j(w)$: maximum value achievable of selecting from the first j items with weight limit w

Computation order: $j = 1 \to n$; for any j, $w = 1 \to W$

$$\begin{cases} \mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\} \\ \mathsf{OPT}_0(w) = 0, 0 \le w \le W, \mathsf{OPT}_j(0) = 0, 0 \le j \le n \\ \text{define } \mathsf{OPT}_j(w) = -\infty, 1 \le j \le n, w < 0 \end{cases}$$

- $\mathsf{OPT}_j(w - w_j) + v_j$: maximum value when selecting at least one j-th item
- setting $\mathsf{OPT}_j(w) = -\infty$ for $w < 0 \rightsquigarrow$ one does not have to explicity check if $w - w_j \ge 0$

Pseudocode of Knapsack

Algorithm 1: Knapsack$(n, W, \{w_i\}_{i\in n}, \{v_i\}_{i\in n})$

1: **for** $w = 0$ *to* W **do** $\mathsf{OPT}_0(w) \leftarrow 0$;
2: **for** $j = 1$ *to* n **do** $\mathsf{OPT}_j(0) \leftarrow 0$;
3: set $\mathsf{OPT}_j(w) \leftarrow -\infty$ for $1 \le j \le n$ $w < 0$;
4: **for** $j = 1$ *to* n **do**
5: **for** $w = 1$ *to* W **do**
6: $\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\}$
7: **end**
8: **end**

- Buttom-up approach to fill the memo table
- Similar trick of reducing the size of memo table in ROI problem also works here but with slight difference on order
  1. reduce the $n \times W$ table to $n \times 2$
  2. further reduce to $n \times 1$ by increasing w

Demo

Table: knapsack instance, $n = 4, W = 10$

i	1	2	3	4
v_i	1	3	5	9
w_i	2	3	4	7

Computation process of $\mathrm{OPT}_j(w)$ (hint: how to fill the matrix)

- left to right, top to down
- top to down, left to right

j \ w	1	2	3	4	5	6	7	8	9	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	12

A Remark

Alternative optimization function: like ROI problem

$$\mathsf{OPT}_j(w) = \max_{0 \le x_j \le \lfloor w/w_j \rfloor} \{\mathsf{OPT}_{j-1}(w - x_j \cdot w_j) + x_j \cdot v_j\}$$

- Pros: more intuitive and easy to understand
- Cons: complexity of computing $\mathsf{OPT}_j(w)$ depends on w, a.k.a. requires $\lfloor w/w_j \rfloor$ comparisons, in contrast to the original optimized function which only requires one comparsion.

Lesson

The design of optimized function is vital

Trace Function

$s_j(w)$: the biggest item number in solution $\mathsf{OPT}_j(w)$

$$s_j(w) = \begin{cases} s_{j-1}(w) & \mathsf{OPT}_{j-1}(w) > \mathsf{OPT}_j(w - w_k) + v_k \\ j & \mathsf{OPT}_{j-1}(w) \leq \mathsf{OPT}_j(w - w_k) + v_k \end{cases}$$

$$s_1(w) = \begin{cases} 0 & w < w_1 \\ 1 & w \geq w_1 \end{cases}$$

Trace function is used to trace solution and output the detailed information

Pseudocode of TraceSolution

Algorithm 2: TraceSolution($s[n, W]$)

Input: table $s_j(w)$, $j \in [n]$, $w \in [W]$
Output: solution vector $x_1, x_2, \ldots, x_n$

1: **for** $i \leftarrow 1$ *to* n **do** $x_i \leftarrow 0$;
2: $w \leftarrow W$, $k \leftarrow n$;
3: **while** $s_k(w) = k$ **do** //continue select k-th item
4: $w \leftarrow w - w_k$;
5: $x_k \leftarrow x_k + 1$;
6: **end**
7: **if** $s_k(w) \neq 0$ **then** $k \leftarrow s_k(w)$, goto 3; //trace next item
8: **else** finishes tracing;

Trace Solution

Table: $s_j(w)$

j \ w	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3	3
4	0	1	2	3	3	3	4	3	4	4

- $s_4(10) = 4 \Rightarrow x_4 \geq 1$
- $s_4(10 - w_4) = s_4(3) = 2 \Rightarrow x_4 = 1, x_3 = 0, x_2 \geq 1$
- $s_2(3 - w_2) = s_2(0) = 0 \Rightarrow x_2 = 1, x_1 = 0$

Solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, $x_4 = 1$, max profit is 12.

Complexity Analysis

The above algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

According to the optimization function

$$\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\}$$

- Memo computation: takes $O(1)$ time per table entry, there are $\Theta(nW)$ table entries
- Trace back: at most $\Theta(n + W)$ steps (think why?)

The total time complexity and space complexity are $O(nW)$

Remarks

- Not polynomial in input size, cause for integer W, binary representation requires $\log W$ bit, thus input size is n and $\log W$ $\leftarrow$ super-polynomial

A Second Thought

Do we really have to use 2-dimension dynamic programming?

A Second Thought

Do we really have to use 2-dimension dynamic programming?

Consider only put restriction on capcitity, define:

$$\text{OPT}(w) = \text{maximum value achievable for a knapsack with capcitity } w$$

A Second Thought

Do we really have to use 2-dimension dynamic programming?

Consider only put restriction on capcitity, define:

$$\text{OPT}(w) = \text{maximum value achievable for a knapsack with capcitity } w$$

How to express this in terms of smaller subproblems?

- If the optimal solution to $\text{OPT}(w)$ includes item i, then removing this item from the knapsack leaves an optimal solution to $\text{OPT}(w - w_i)$ (thanks to with repetition property).
- In other words, $\text{OPT}(w) = \text{OPT}(w - w_i) + v_i$, for some i.
  - We don't know which i, so we need to try all possibilities.
  - If $\text{OPT}(w)$ includes item i and j, it must hold that $\text{OPT}(w - w_i) + v_i = \text{OPT}(w) = \text{OPT}(w - w_j) + v_j$.

Iteration Relation

The algorithm now writes itself $\rightsquigarrow$ incredibly simple and elegant

Algorithm 3: Knapsack$(n, W, \{w_i\}_{i\in n}, \{v_i\}_{i\in n})$

1: $\mathsf{OPT}(0) \leftarrow 0$ // convention that the maximum over an empty knapsack is 0;
2: **for** $w = 1$ *to* W **do**
3: $\quad \mathsf{OPT}(w) = \max_{i:w_i \le w}\{\mathsf{OPT}(w - w_i) + v_i\}$
4: **end**
5: **return** $\mathsf{OPT}(W)$;

The algorithm fills in a one-dimension table of length $W + 1$, in left-to-right order

- space complexity: $O(W)$
- time complexity: each entry can take up to $O(n)$ time to compute $\Rightarrow$ overall running time is $O(nW)$.

Think over It

As always, there is an underlying DAG. Try constructing it, and you will be rewarded with a startling insight

- this particular variant of knapsack boils down to finding the longest path in a DAG

你品 你细品

Outline

Knapsack without Repetition

What if repetitions are not allowed?

The previous ingenious definition of subproblem now become completely useless.

- For instance, knowing the value of the form $\mathsf{OPT}(w - w_j)$ does not help to make further decision, cause we don't know whether or not item j has already got used up in this partial solution.

Knapsack without Repetition

What if repetitions are not allowed?

The previous ingenious definition of subproblem now become completely useless.

- For instance, knowing the value of the form $\mathrm{OPT}(w - w_j)$ does not help to make further decision, cause we don't know whether or not item j has already got used up in this partial solution.

We must refine the subproblem to carry additional information about the items being used as before $\rightsquigarrow$ add another parameter $0 \leq j \leq n$ (same as the first approach to knapsack problem with repetition):

$$\mathrm{OPT}_j(w) = \text{maximum value using items } \{1, \ldots, j\} \text{ and weight limit } w$$

The answer we seek is $\mathrm{OPT}_n(W)$.

Iteration Relation

How to express $\mathsf{OPT}_j(w)$ *in terms of smaller subproblems?*

Quite simple: either item j is needed to achieve the optimal value or it isn't needed.

$$\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w - w_j) + v_j, \mathsf{OPT}_{j-1}(w)\}$$

In other words, we express $\mathsf{OPT}_j(w)$ in terms of subproblems $\mathsf{OPT}_{j-1}(\cdot)$.

Pseudocode

Algorithm 4: Knapsack$(n, W, \{w_i\}_{i\in[n]}, \{v_i\}_{i\in n})$

1: $\mathsf{OPT}_0(w) \leftarrow 0$ for $w \in [0, W]$, $\mathsf{OPT}_j(0) = 0$ for $j \in [0, n]$;
2: set $\mathsf{OPT}_j(w) = -\infty$ for $w < 0$;
3: **for** $j = 1$ *to* n **do**
4: **for** $w = 1$ *to* W **do**
5: $\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w - w_j) + v_j, \mathsf{OPT}_{j-1}(w)\}$
6: **end**
7: **end**
8: **return** $\mathsf{OPT}_n(W)$;

- This algorithm fills out a 2-dimension table, with $W + 1$ rows and $n + 1$ columns. Each table entry takes just constant time. The running time remains the same: $O(nW)$.

Memoization

In dynamic programming, we write out a recursive formula that express large problems in terms of smaller ones and then use it to fill a table of solution values in a bottom-up manner, from smaller subproblem to largest.

The formula also suggests a recursive algorithm.

As we saw earlier that naive recursion can be terribly inefficient, because it solves the same subproblems over and over again.

What about a more intelligent recursive implementation? One that remembers its previous invocations and thereby avoids repeating them?

Memoization

For the knapsack problem with repetitions, the recursive algorithm can use key-value mapping to store $\mathrm{OPT}(\cdot)$ that had already been computed.

- At each recursive call requesting some $\mathrm{OPT}(w)$, the algorithm first checks if the answer was already in the KV mapping and then proceeds to its calculation only if it wasn't.
- This trick is called *memoization*.
- Note: The KV mapping can be realized using array or hash table, depending on the data type and distibution of keys.

Complexity: recursive algorithm never repeats a subproblem $\rightsquigarrow$ running time is $O(nW)$, just like dynamic program.

- However, the constant factor in the big-O notation is substantially larger because of the overhead of recursion.

Disadvantge of Dynamic Programming

In some cases, memoization pays off.

- Dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually needed.

Disadvantge of Dynamic Programming

In some cases, memoization pays off.

- Dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually needed.

For instance, suppose W and all the weights w_i are the multiples of 100. Then a subproblem $\text{OPT}_j(w)$ is useless if 100 does not divide w

- DP will always compute all table entries.
- The memorized recursive algorithm will never look at these extraneous table entries.

Disadvantge of Dynamic Programming

In some cases, memoization pays off.

- Dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually needed.

For instance, suppose W and all the weights w_i are the multiples of 100. Then a subproblem $\text{OPT}_j(w)$ is useless if 100 does not divide w

- DP will always compute all table entries.
- The memorized recursive algorithm will never look at these extraneous table entries.

The worst-case complexity of DP and recursive algorithm are still the same, but the later may perform better on some instances since its programming is instance dependent.

Extension of Knapsack Problem

Decision version of knapsack problem is $\mathcal{NP}$-COMPLETE.

There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum.

Variants of Knapsack

- Knapsack with constraint on item number: maximum number of i-th item is n_i
  - $0-1$ Knapsack: $x_i = 0, 1;\ i \in [n]$
- Multi-Knapsack: m knapsack, the weight limit of knapsack i is W_i, $i \in [m]$.
- 2-dimension Knapsack: each item with weight w_i and volume t_i, $i \in [n]$, the weight limit is W, the volume limit is V

Longest Common Substring

Let $X = (x_1, x_2, \ldots, x_m)$ and $Z = (z_1, z_2, \ldots, z_n)$ be two strings. Z is a substring of X if there exists an index sequence of strict incresasing order $(i_1, \ldots, i_k)$ such that $z_k = x_{i_k}$ for all $k \in [n]$.

Common substring of X and Y: the substring of both X and Y.

Problem. Find the longest string of $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$.

Example

- $X : A\ B\ C\ B\ D\ A\ B$
- $Y : B\ D\ C\ A\ B\ A$

LCS: $B\ C\ B\ A$, length is 4

Brute Force Algorithm

Assume $m \leq n$, $|X| = m$, $|Y| = n$

Brute force algorithm: for each substring of X, check if the substring appears in Y

Complexity analysis

- check whether a candidate substring is a substring of a given string takes $O(n)$
  - think how? hint: sequentially scan two strings using two pointers (after each comparision, at least one point moves forward, thus the maximum number of comparsion is $2n$)
- there are totally 2^m substrings in X

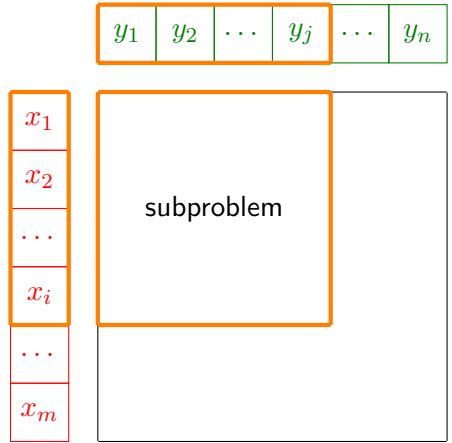
Complexity: $O(n2^m)$

Dynamic Programming: Subproblem

Introduce i and j to define subproblem

X right boundary is i, Y right boundary is j

$X_i = (x_1, x_2, \ldots, x_i)$, $Y_j = (y_1, y_2, \ldots, y_j)$

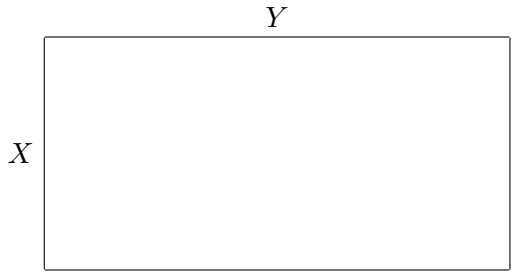


Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:



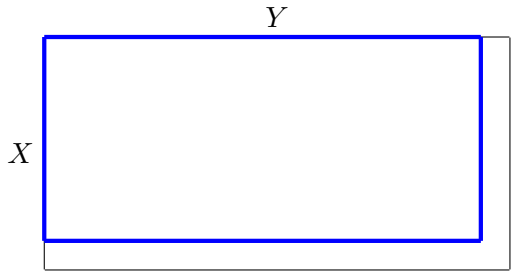
Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:

- $x_m = y_n \Rightarrow z_k = x_m = y_n,\ Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$



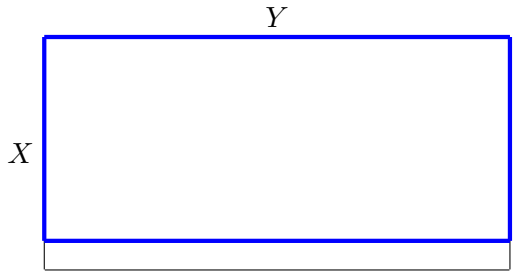
Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:

- $x_m = y_n \Rightarrow z_k = x_m = y_n$, $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$ (one or both of the following cases occur)
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$



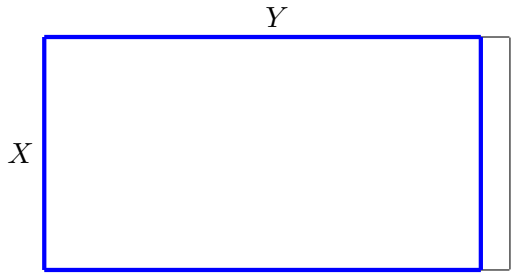
Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:

- $x_m = y_n \Rightarrow z_k = x_m = y_n$, $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$ (one or both of the following cases occur)
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$
  - $z_k \neq y_n \Rightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$



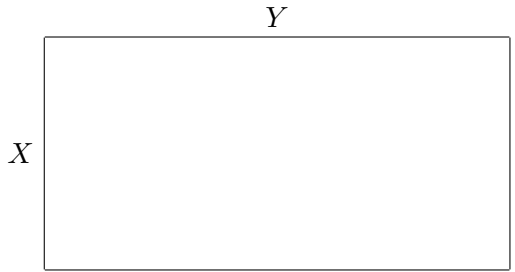
Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:

- $x_m = y_n \Rightarrow z_k = x_m = y_n$, $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$ (one or both of the following cases occur)
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$
  - $z_k \neq y_n \Rightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$



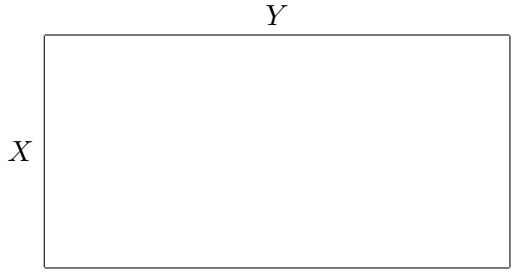
Relations Between Problems and Subproblems

$$X_m = (x_1, x_2, \ldots, x_m), Y_n = (y_1, y_2, \ldots, y_n)$$

$$\text{let } Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$$

Consider the following cases:

- $x_m = y_n \Rightarrow z_k = x_m = y_n$, $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$ (one or both of the following cases occur)
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$
  - $z_k \neq y_n \Rightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$



satisfy optimial sub-structure

Optimized Function and Iteration Relation

Optimized function: $L(i, j)$

- LCS length of $X_i = (x_1, x_2, \ldots, x_i)$ and $Y_j = (y_1, y_2, \ldots, y_j)$

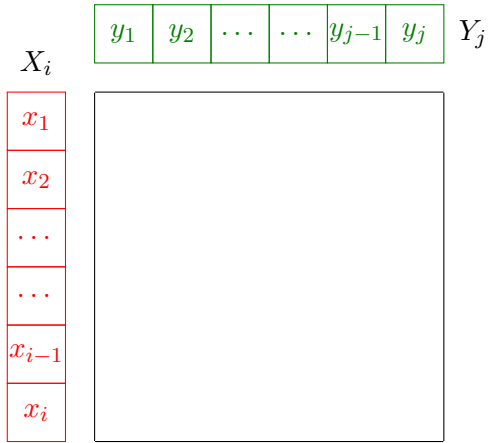
Iteration relation

$$L(i,j) = \begin{cases} 0 & i = 0 \vee j = 0 \\ L(i-1, j-1) + 1 & i, j > 0 \wedge x_i = y_j \\ \max\{L(i, j-1), L(i-1, j)\} & i, j > 0 \wedge x_i \neq y_j \end{cases}$$

Note: We do not know which one occurs in the last case, so we choose the maximial one (if the values are equal, both cases constitute solutions).

Indicator Function

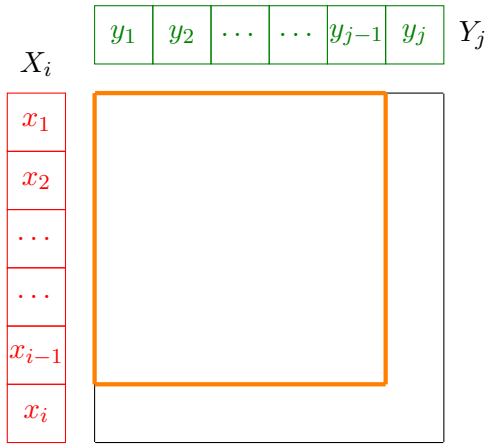
Indicator functuon $s(i, j)$ with values: $\nwarrow$, $\leftarrow$, $\uparrow$



Indicator Function

Indicator functuon $s(i,j)$ with values: $\nwarrow$, $\leftarrow$, $\uparrow$

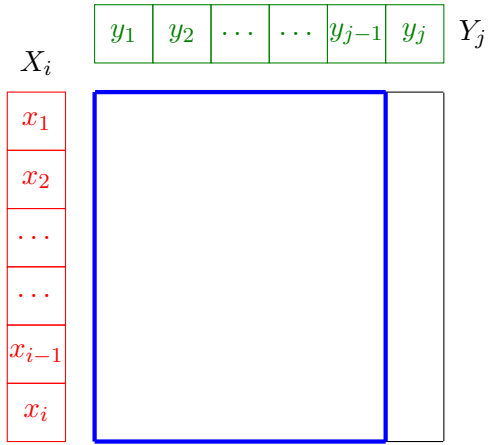
- $L(i,j) = L(i-1,j-1)+1$: $\nwarrow$



Indicator Function

Indicator functuon $s(i,j)$ with values: $\nwarrow$, $\leftarrow$, $\uparrow$

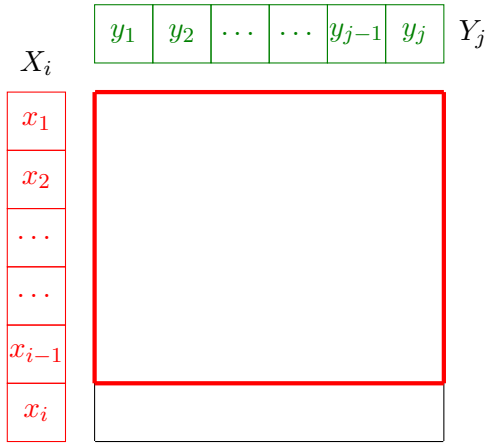
- $L(i,j) = L(i-1,j-1)+1$: $\nwarrow$
- $L(i,j) = L(i,j-1)$: $\leftarrow$



Indicator Function

Indicator functuon $s(i,j)$ with values: $\nwarrow$, $\leftarrow$, $\uparrow$

- $L(i,j) = L(i-1,j-1)+1$: $\nwarrow$
- $L(i,j) = L(i,j-1)$: $\leftarrow$
- $L(i,j) = L(i-1,j)$: $\uparrow$



Pseudocode of LCS

Algorithm 5: LCS$(X[m], Y[n])$

1: $L(i,0) \leftarrow 0$, $i \in [m]$, $L(0,j) \leftarrow 0$, $j \in [n]$;
2: **for** $i \leftarrow 1$ *to* m **do**
3: **for** $j \leftarrow 1$ *to* n **do**
4: **if** $X[i] = Y[j]$ **then**
5: $L(i,j) = L(i-1,j-1) + 1$, $s(i,j) \leftarrow (\nwarrow)$
6: **else if** $L(i-1,j) \geq L(i,j-1)$ **then**
7: $L(i,j) \leftarrow L(i-1,j)$, $s(i,j) \leftarrow (\uparrow)$ **else**
8: $L(i,j) \leftarrow L(i,j-1)$, $s(i,j) \leftarrow (\leftarrow)$
9: **end**
10: **end**
11: **end**

Track Solution

Algorithm 6: TrackLCS(s, m, n)

Output: LCS of X and Y

while $m \neq 0 \wedge n \neq 0$ **do**
  if $s(m, n) = (\nwarrow)$ **then**
    output $X[m]$; $m = m - 1, n = n - 1$, continue;
  end
  if $s(m, n) = (\uparrow)$ **then**
    $m = m - 1$, continue;
  end
  if $s(m, n) = (\leftarrow)$ **then**
    $n = n - 1$, continue;
  end
end

Demo of Indicator Function

$X = (A, B, C, B, D, A, B)$, $Y = (B, D, C, A, B, A)$

	1	2	3	4	5	6
1	$s[1,1] = \uparrow$	$s[1,2] = \uparrow$	$s[1,3] = \uparrow$	$s[1,4] = \nwarrow$	$s[1,5] = \leftarrow$	$s[1,6] = \nwarrow$
2	$s[2,1] = \nwarrow$	$s[2,2] = \leftarrow$	$s[2,3] = \leftarrow$	$s[2,4] = \uparrow$	$s[2,5] = \nwarrow$	$s[2,6] = \leftarrow$
3	$s[3,1] = \uparrow$	$s[3,2] = \uparrow$	$s[3,3] = \nwarrow$	$s[3,4] = \leftarrow$	$s[3,5] = \uparrow$	$s[3,6] = \uparrow$
4	$s[4,1] = \uparrow$	$s[4,2] = \uparrow$	$s[4,3] = \uparrow$	$s[4,4] = \uparrow$	$s[4,5] = \nwarrow$	$s[4,6] = \leftarrow$
5	$s[5,1] = \uparrow$	$s[5,2] = \uparrow$	$s[5,3] = \uparrow$	$s[5,4] = \uparrow$	$s[5,5] = \uparrow$	$s[5,6] = \leftarrow$
6	$s[6,1] = \uparrow$	$s[6,2] = \uparrow$	$s[6,3] = \uparrow$	$s[6,4] = \nwarrow$	$s[6,5] = \uparrow$	$s[6,6] = \nwarrow$
7	$s[7,1] = \uparrow$	$s[7,2] = \uparrow$	$s[7,3] = \uparrow$	$s[7,4] = \uparrow$	$s[7,5] = \uparrow$	$s[7,6] = \uparrow$

Solution: LCS $= (X[2], X[3], X[4], X[6]) = (B, C, B, A)$

Complexity Analysis

Computation of optimized function

- Initialization: $O(m+n)$
- Computation: in each loop, require ≤ 2 times comparision, complexity is $\Theta(mn)$

Computation of indicator function

- Computation: $\Theta(mn)$
- Trace solution: $\Theta(m+n)$ (reduce the size of X or/and Y by 1 in each step)

Overall time complexity: $\Theta(mn)$

Space complexity: $\Theta(mn)$

Further Discussion

Standard LCS problem

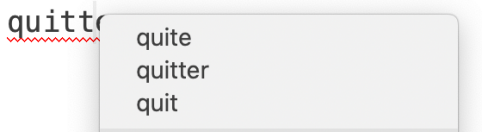
- Dynamic programming: $\Theta(nm)$
- Generalized suffix tree: $\Theta(n+m)$

Generalized LCS problem: find LCS for k strings with length $n_1, \ldots, n_k$

- k-dimension Dynamic programming: $\Theta(n_1 \cdots n_k)$
- Generalized suffix tree: $\Theta(n_1 + \cdots + n_k)$

Motivation of String Similarity

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.



What is the appropriate notion of closeness or similarity for two strings?

Edit Distance

Edit distance. [Levnshtein 1966, Needleman-Wunsch 1970]

Given two strings x and y, after a sequence of operations (replace, insert, delete), change y to x. The minimal number of operations is called the edit distance of between x and y, write as $\Delta(x, y)$.

capture similarity between two strings

Justify the definition: satisfy three rules of distance

- Non-negative: $\Delta(x, y) \geq 0$. $\Delta(x, y) = 0$ iff $x = y$
- Symmetric: $\Delta(x, y) = \Delta(y, x)$ (just reverse the operation)
- Triangle inequality: $\forall x, y, z$, $\Delta(x, z) + \Delta(z, y) \geq \Delta(x, y)$
  - $x \to z \to y$ is one path from x to y

How to Compute Edit Distance

Sequence alignment. A natural measure of edit distance is the extent to which they can be aligned, or matched up.

- an alignment is simply a way of writing the strings one above the other, allow adding $\perp$

S	$\perp$	N	O	W	Y
S	U	N	N	$\perp$	Y

1 mismatches, 2 gap

$\perp$	S	N	O	W	$\perp$	Y
S	U	N	$\perp$	$\perp$	N	Y

1 mismatches, 4 gap

- $\perp$ indicates a "gap": can be placed in either string — interpreting as delete or insert
- Cost of an alignment is the number of columns in which the letters differ.

$$\text{cost} = \underbrace{\sum_{x_i - y_i} \text{diff}(i,j)}_{\text{mismatch}} + \underbrace{\sum_{x_i \text{ unmatched}} \alpha + \sum_{y_j \text{ unmatched}} \beta}_{\text{gap}}$$

Insight of Edit Distance

Edit distance between two strings is the cost of their best alignment.

- Finding the edit distance is equivalent to finding the optimal alignment.

Edit distance is so named because it can also be thought of as the minimum number of *edits* — insertion, deletions, and substitutions — needed to transform the first string to the second.

- The above example: insert 'U', substitute 'O' $\rightarrow$ 'N', and delete 'W'

Insight of Edit Distance

Edit distance between two strings is the cost of their best alignment.

- Finding the edit distance is equivalent to finding the optimal alignment.

Edit distance is so named because it can also be thought of as the minimum number of *edits* — insertion, deletions, and substitutions — needed to transform the first string to the second.

- The above example: insert 'U', substitute 'O' $\rightarrow$ 'N', and delete 'W'

In general, there are so many possible alignments between two strings $\rightsquigarrow$ it would be terribly inefficient to search through all of them for the best one.

A Dynamic Programming Solution

When solving a problem by dynamic programming, the most crucial question is

What are the subproblems?

As long as they are chosen so as to have the optimal substructure, it is easy to write the algorithm: iteratively solve one subproblem after the other, in order of increasing order.

Goal. Finding the edit distance $E(m, n)$ between two strings $x[1 \dots m]$ and $y[1 \dots n]$.

Subproblem. Looking at the edit distance between some *prefix* of $x[1 \dots i]$ and some prefix of $y[1 \dots j]$, call the subproblem $E(i, j)$.

subproblem $E(7, 5)$

Structure of Problem

We need somehow express $E(i, j)$ in terms of smaller subproblems. Analyze the best alignment between $x[1 \dots i]$ and $y[1 \dots j]$: their rightmost column can only be one of three things:

$$\begin{matrix} x_i & \perp & x_i \\ \perp & y_j & y_j \end{matrix}$$

Case 1a. leave x_i unmatched

- gap for x_i + min cost of aligning $x[1 \dots i-1]$ and $y[1 \dots j]$.

Case 1b. leave y_j unmatched

- gap for y_j + min cost of aligning $x[1 \dots i]$ and $y[1 \dots j-1]$.

Case 2. M matches $x_i \sim y_j$.

- (mis)match for $x_i \sim y_j$ + min cost of aligning $x[1 \dots i-1]$ and $y[1 \dots j-1]$.

satisfying optimal substructure property

Iteration Relation for Optimized Function

Optimized function: $E(i, j)$ — edit distance between $x[1, \ldots, i]$ and $y[1, \ldots, j]$

Initial values: $E(i, 0) = i, E(0, j) = j$

Iteration relation. We have expressed $E(i, j)$ in terms of three *smaller* subproblems $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$.

- We have no idea which of them is the right one, so we need to try them all and pick the best

$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \mathsf{diff}(i, j) + E(i-1, j-1)\}$$

$$\mathsf{diff}(i, j) = \begin{cases} 0 & x_i = y_j \\ 1 & x_i \neq y_j \end{cases}$$

Computation Order

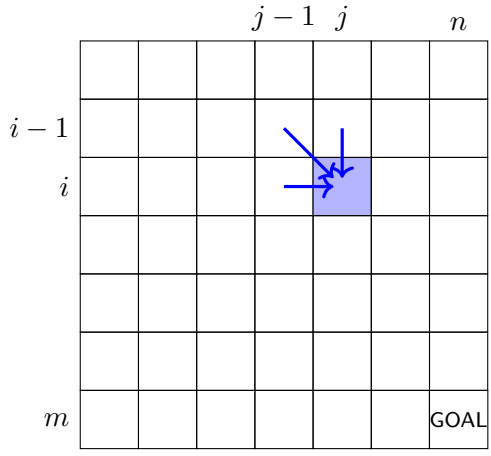
The answers to all the subproblems $E(i, j)$ form a 2-dimensional table.

What order should these subproblems be solved?

Any order is fine, as long as $E(i-1, j)$, $E(i, j-1)$ and $E(i-1, j-1)$ are handled before $E(i, j)$.

1. fill in the table one row at a time, from top row to bottom row, and moving left to right across each row
2. or fill in the table column by column

Both methods ensure that by the time of computing a particular table entry, all the other entries needed are already filled in.


$j-1$
j
n
$i-1$
i
m
GOAL

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$

Y

Y

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$
$$E(4,4) \leftarrow E(4,3) + 1$$

W Y

$\perp$ Y

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$
$$E(4,4) \leftarrow E(4,3) + 1$$
$$E(4,3) \leftarrow E(3,2) + 1$$

O W Y

N $\perp$ Y

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$
$$E(4,4) \leftarrow E(4,3) + 1$$
$$E(4,3) \leftarrow E(3,2) + 1$$
$$E(3,2) \leftarrow E(2,1) + 0$$

N O W Y

N N $\perp$ Y

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$
$$E(4,4) \leftarrow E(4,3) + 1$$
$$E(4,3) \leftarrow E(3,2) + 1$$
$$E(3,2) \leftarrow E(2,1) + 0$$
$$E(2,1) \leftarrow E(1,1) + 1$$

$\perp$ N O W Y

U N N $\perp$ Y

Track Solution

	x	S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0$$
$$E(4,4) \leftarrow E(4,3) + 1$$
$$E(4,3) \leftarrow E(3,2) + 1$$
$$E(3,2) \leftarrow E(2,1) + 0$$
$$E(2,1) \leftarrow E(1,1) + 1$$
$$E(1,1) \leftarrow E(0,0) + 0$$

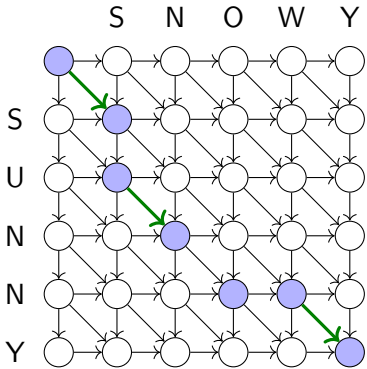
S ⊥ N O W Y

S U N N ⊥ Y

The Underlying DAG

Every dynamic programming has an underlying DAG.

- each node represents a subproblem
- each edge represents a precedence constraint



Set all edge lengths to 1 except the green ones

Final answer is the shortest path from $E(0,0)$ and $E(m,n)$

- move down: delete
- move right: insert
- move diagonal: match or substitution

By altering the weights on the DAG, we can allow generalized forms of edit distance: insertion, deletion, and substitution have different associated costs.

Pseudocode

Algorithm 7: SequenceAlignment($x[m], y[n]$)

1: **for** $i = 0$ *to* m **do** $E(i, 0) = i$;
2: **for** $j = 0$ *to* n **do** $E(0, j) = j$;
3: **for** $i = 1$ *to* m **do**
4: **for** $j = 1$ *to* n **do**
5: $E(i, j) \leftarrow \min\{1 + E(i-1, j), 1 + E(i, j-1), \mathsf{diff}(i, j) + E(i-1, j-1)\}$
6: **end**
7: **end**
8: **return** $E(m, n)$;

There are totally mn subproblems, each subproblem requires constant time $\Rightarrow$ totally $\Theta(mn)$ time and $\Theta(mn)$ space.

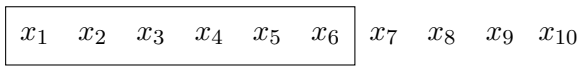
How to Find Subproblems

Finding the right subproblems takes creativity and experimentation.

But there are a few standard choices that seem to arise repeatedly in dynamic programming.

One-Dimension Dynamic Programming

The input is $x_1, x_2, \ldots, x_n$. A subproblem is $x_1, x_2, \ldots, x_i$



The number of subproblems is therefore $O(n)$.

Examples

- shortest path in DAG
- longest increasing subsequence
- max interval sum
- image compression

Two-Dimension Dynamic Programming: Type 1

The input is $x_1, \ldots, x_n$. A subproblem is $x_i, \ldots, x_j$

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5} \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

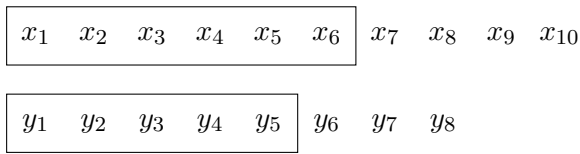
The number of subproblems is therefore $O(n^2)$.

Examples

- matrix multiplication chain
- optimal binary search tree

Two-Dimension Dynamic Programming: Type 2

The input is $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$



The number of subproblems is therefore $O(mn)$.

Examples

- return on investment
- knapsack problem
- longest common substring
- edit distance

Another Important Characterization of DP

The computational complexity of DP algorithm not only depends on the number of subproblems, but also depends on the complexity of the recursive relation, i.e., the extent that a problem relates to its subproblems.

We capture such dependence as *locality*.

Case 1: depend on linear number of subproblems

- shortest path in DAG, longest increasing sequence, maximum interval sum, matrix multiplication chain, optimal binary search tree,

Case 2: depend on constant number of subproblems

- knapsack problem, longest common substring, edit distance

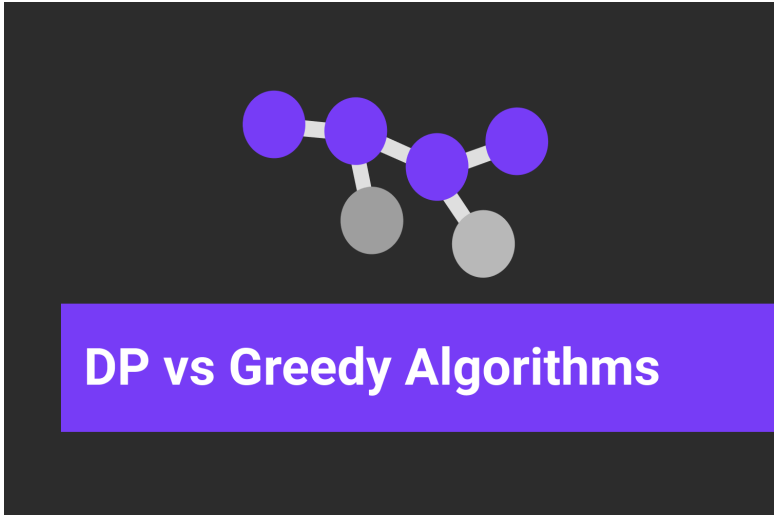
Essence of Dynamic Programming

DP is mainly an optimization over plain recursion.

- Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using DP.
  - simply store the results of subproblems so that we do not have to re-compute them when needed later
- This simple optimization reduces time complexity from exponential to polynomial.
- Example. A simple recursive solution for Fibonacci numbers leads to exponential time complexity. But, if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

We can think of Dynamic Programming as finding a shortest path or computing the path in a DAG (iterative approach) or travel the recursion tree with memo (recursive approach).

Greedy vs. Dynamic Programming



Both Dynamic Programming and Greedy are algorithmic paradigms used to solve optimization problems.

Greedy Paradigm

Theoretical idea: solving the problem step-by-step; on each step, the algorithm makes a choice, based on some heuristic, that achieves the most obvious and beneficial profit.

Applicability: problems satisfying greedy property: choosing the local optimum at each stage will lead to form the global optimum

Optimality: rigorous proof is always needed

Memorization: may need to maintain a data structure to store current states for making greedy choice

Complexity: generally faster

Fashion: computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.

Dynamic Programming Paradigm

Theoretical idea: finding an order between subproblems; solving the current subproblem using solutions to previously solved subproblems

Applicability: problems can be solved via recursive/iterative approach, but one have to visiting the same state multiple times

Optimality: automatically guaranteed since DP actually considers all possible cases and then choose the best

Memorization: requires DP table to store solutions of solved subproblems

Complexity: generally slower

Fashion: computes its solution by synthesizing from smaller optimal sub solutions: bottom up or top down