CS711008Z Algorithm Design and Analysis

Lecture 6. Basic algorithm design technique: Dynamic programming

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

Outline

- The first example: MatrixChainMultiplication
- Elements of dynamic programming technique
- Various ways to describe subproblems: Knapsack, RNA Secondary Structure, Hidden Markov Model, Sequence Alignment, and Shortest Path
- Advanced DP: techniques to reduce space and time consumption, Hirschberg algorithm, banded DP, sparsed DP, and monotonicity in backtracking
- Connection with greedy technique: Interval Scheduling, Shortest Path

Dynamic programming and its connection with divide-and-conquer

- Dynamic programming typically applies to **optimization problems** if:
  1. The original problem can be divided into smaller subproblems, and
  2. The recursion among sub-problems has **optimal-substructure property**, i.e., the optimal solution to the original problem can be calculated through **combining the optimal solutions to subproblems**.
- Unlike the general divide-and-conquer framework, a dynamic programming algorithm usually **enumerates** all possible dividing strategies.
- To identify meaningful recursions, one of the key steps is to define **an appropriate general form of sub-problems**. For this aim, it is helpful to describe the solving process as **a multistage decision process**.

The birth of dynamic programming

August 26, 1920–March 19, 1984

- In 1952, R. Bellman proposed the dynamic programming technique when he worked at RAND corporation. The motivation is to solve multi-stage decision problems, which are difficult for the calculus of variation technique.

Revisiting the Divide and Conquer technique

- To see whether the Divide and Conquer technique applies on a given problem, we need to examine both **input** and **output** of the problem description.
  - Examine the **input** part to determine how to decompose the problem into subproblems of same structure but smaller size: It is relatively easy to decompose a problem into subproblems if the input part is related to the following data structures:
    - An **array** with n elements;
    - A **matrix**;
    - A **set** of n elements;
    - A **tree**;
    - A **directed acyclic graph**;
    - A **general graph**.
  - Examine the **output** part to determine how to construct the solution to the original problem using the solutions to its subproblems.

MatrixChainMultiplication problem: recursion over **sequences**

INPUT:
A sequence of n matrices $A_1, A_2, ..., A_n$; matrix A_i has dimension $p_{i-1} \times p_i$;
OUTPUT:
Fully parenthesizing the product $A_1 A_2 ... A_n$ in a way to minimize the number of scalar multiplications.

Let's start from a simple example

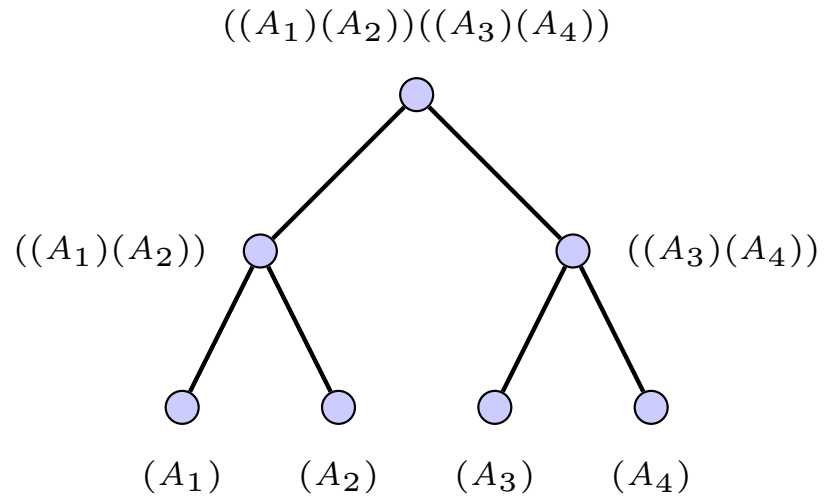
$$A_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Solutions:	$((A_1)(A_2))(A_3)$	$(A_1)((A_2)(A_3))$
#Multiplications:	$1 \times 2 \times 3$	$2 \times 3 \times 4$
	$+1 \times 3 \times 4$	$+1 \times 2 \times 4$
	$= 18$	$= 32$

- Here we assume that the calculation of $A_1 A_2$ needs $1 \times 2 \times 3$ scalar multiplications.
- The objective is to determine a calculation sequence such that the number of multiplications is minimized.

The solution space size

- Intuitively, a calculation sequence can be described as a binary tree, where each node corresponds to a subproblem.



- The total number of possible calculation sequences:

$$\binom{2n-2}{n-1} - \binom{2n-2}{n-2} \text{ (Catalan number)}$$

- Thus, it takes exponential time to enumerate all possible calculation sequences.
- Question: can we design an efficient algorithm?

A dynamic programming algorithm (by S. S. Godbole, 1973)

Defining general form of sub-problems

1. It is not easy to solve the problem directly when n is large. Let's investigate whether it is possible to reduce into smaller sub-problems.
2. Solution: a full parentheses. Let's describe the solving process as a process of **multistage decisions**, where each decision is to add parentheses at a position.
3. Suppose in the optimal solution O, where the first **decision** adds two parentheses as $(A_1...A_k)(A_{k+1}...A_n)$.
4. This decision decomposes the original problem into two **independent sub-problems**: to calculate $A_1...A_k$ and $A_{k+1}...A_n$.
5. Summarizing these two cases, we define the general form of sub-problems as: to calculate $A_i...A_j$ with the minimal number of scalar multiplications.

Optimal substructure property

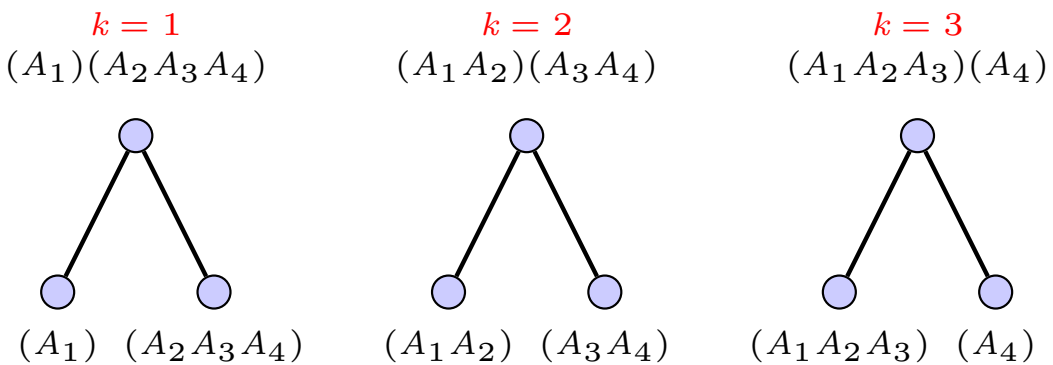
- The general form of sub-problems: to calculate $A_i...A_j$ with the minimal number of scalar multiplications. Let's denote the optimal solution value to the sub-problem as $OPT(i, j)$, thus the original problem can be solved via calculating $OPT(1, n)$.
- The optimal solution to the original problem can be obtained through combining the optimal solutions to sub-problems. This recursion can be stated as the following **optimal substructure property**:
  $OPT(1, n) = OPT(1, k) + OPT(k + 1, n) + p_0 p_k p_n$

Proof of the optimal substructure property

- "Cut-and-paste" proof:
  - Suppose for $A_1 ... A_k$, there is another parentheses $OPT'(1, k)$ better than $OPT(1, k)$. Then the combination of $OPT'(1, k)$ and $OPT(k + 1, n)$ leads to a new solution with lower cost than $OPT(1, n)$: a contradiction.
  - Here, the independence between $A_1 ... A_k$ and $A_{k+1} ... A_n$ guarantees that the substitution of $OPT(1, k)$ with $OPT'(1, k)$ does not affect solution to $A_{k+1} ... A_n$.

A recursive solution

- So far so good! The only difficulty is that we have no idea of the first splitting position k in the optimal solution.
- How to overcome this difficulty? **Enumeration!** We **enumerate all possible options of the first decision**, i.e. for all k, $i \leq k < j$.



- Thus we have the following recursion:

$$OPT(i,j) = \begin{cases} 0 & i = j \\ \mathbf{min_{i \leq k < j}}\{OPT(i,k) + OPT(k+1,j) + p_{i-1}p_kp_j\} & otherwise \end{cases}$$

Implementing the recursion: trial 1

Trial 1: Explore the recursion in the top-down manner

RECURSIVE_MATRIX_CHAIN(i, j)

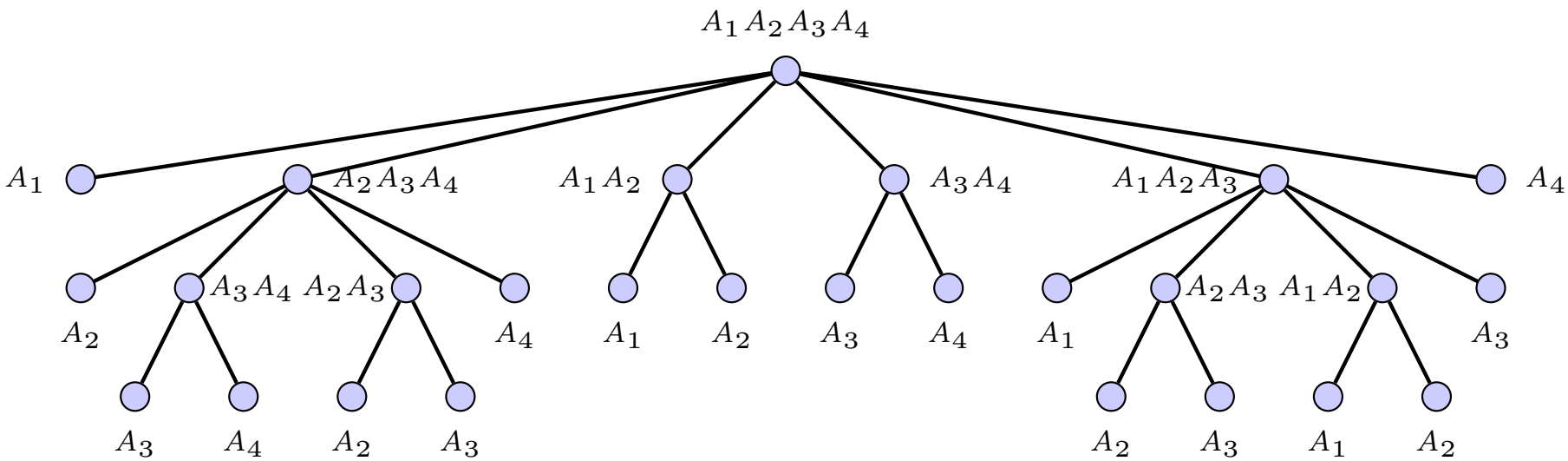
1: **if** $i == j$ **then**
2: return 0;
3: **end if**
4: $OPT(i, j) = +\infty$;
5: **for** $k = i$ to $j - 1$ **do**
6: $q =$ RECURSIVE_MATRIX_CHAIN(i, k)
7: + RECURSIVE_MATRIX_CHAIN$(k + 1, j)$
8: $+p_{i-1}p_kp_j$;
9: **if** $q < OPT(i, j)$ **then**
10: $OPT(i, j) = q$;
11: **end if**
12: **end for**
13: **return** $OPT(i, j)$;

- Note: The optimal solution to the original problem can be obtained through calling RECURSIVE_MATRIX_CHAIN$(1, n)$.

An example

$$A_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \quad A_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$1 \times 2 \qquad 2 \times 3 \qquad 3 \times 4 \qquad 4 \times 5$



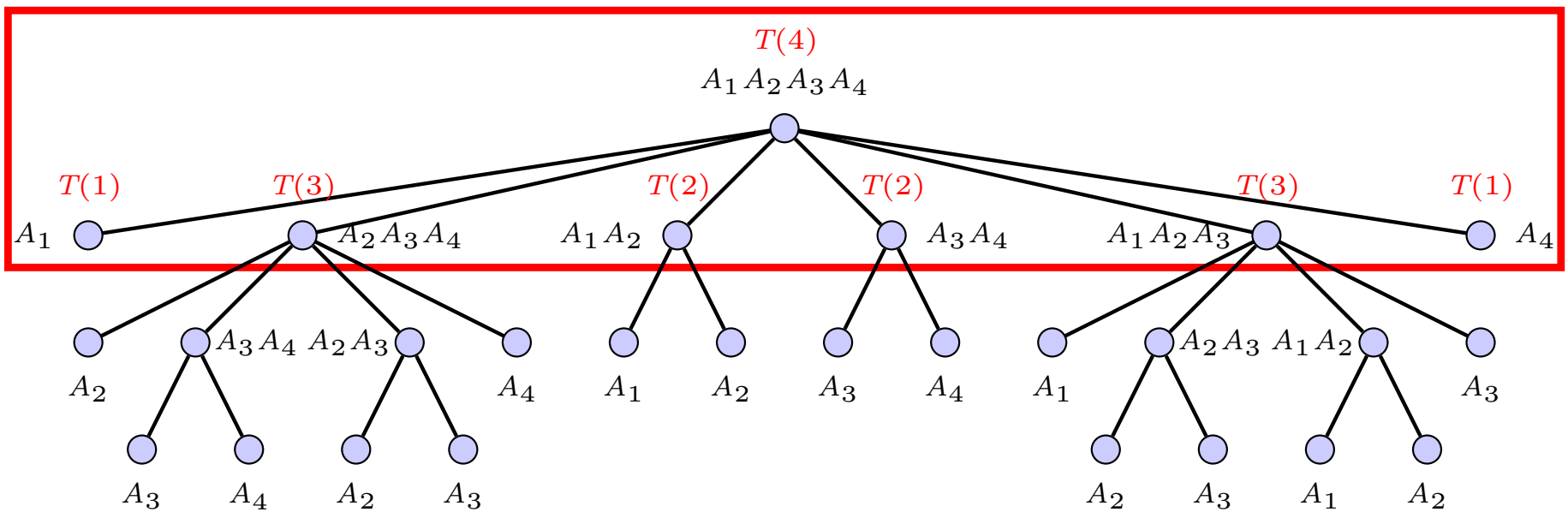
- Note: each node of the recursion tree represents a subproblem.

However, this is not a good implementation

Theorem

Algorithm RECURSIVE-MATRIX-CHAIN *costs exponential time.*

- Let $T(n)$ denote the time used to calculate product of n matrices. Then $T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1)$ for $n > 1$.

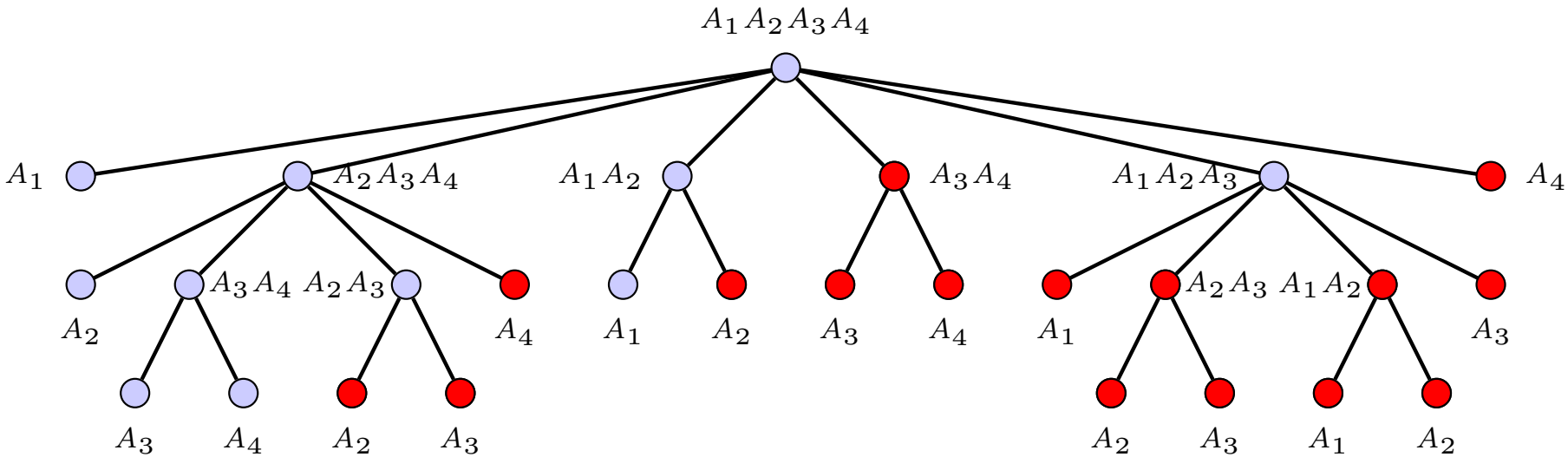


Proof.

- We shall prove $T(n) \geq 2^{n-1}$ using the substitution technique.
  - Basis: $T(1) \geq 1 = 2^{1-1}$.
  - Induction:

$$
\begin{aligned}
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) && (1)\\
&= n + 2\sum_{k=1}^{n-1} T(k) && (2)\\
&\geq n + 2\sum_{k=1}^{n-1} 2^{k-1} && (3)\\
&\geq n + 2(2^{n-1} - 1) && (4)\\
&\geq n + 2^n - 2 && (5)\\
&\geq 2^{n-1} && (6)
\end{aligned}
$$

Why the first trial failed?



- Reason: There are only $O(n^2)$ subproblems. However, some subproblems (in red) were solved repeatedly.
- Solution: **memorize the solutions to subproblems** using an array $OPT[1..n; 1..n]$ for further look-up. The calculation of Fibonacci number is a good example of the power of the "memorizing" technique.

Implementing the recursion: trial 2

The "memorizing" technique

```
MEMORIZE_MATRIX_CHAIN(i, j)
 1: if OPT[i, j] ≠ NULL then
 2:     return OPT(i, j);
 3: end if
 4: if i == j then
 5:     OPT[i, j] = 0;
 6: else
 7:     for k = i to j − 1 do
 8:         q = MEMORIZE_MATRIX_CHAIN(i, k)
 9:               +MEMORIZE_MATRIX_CHAIN(k + 1, j)
10:               +p_{i−1} p_k p_j;
11:         if q < OPT[i, j] then
12:             OPT[i, j] = q;
13:         end if
14:     end for
15: end if
16: return OPT[i, j];
```

The "memorizing" technique cont'd

- The original problem can be solved by calling $\text{MEMORIZE\_MATRIX\_CHAIN}(1, n)$ with all $OPT[i, j]$ initialized as NULL.
- Time complexity: $O(n^3)$ (The calculation of each entry $OPT[i, j]$ makes $O(n)$ recursive calls in line 8.)
- Note that there are exponential ways of fully parenthesizing. The DP algorithm find the optimal solution in only $O(n^3)$ time since it avoids enumerating some redundant fully parenthesizing.

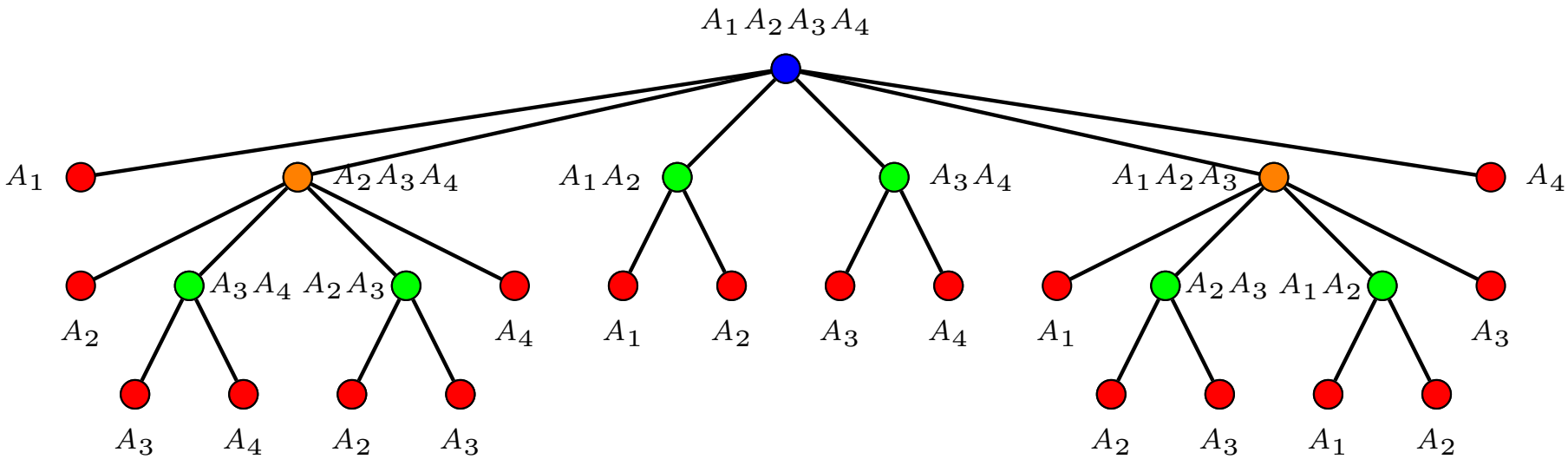
Implementing the recursion faster: trial 3

Trial 3: Faster implementation: unrolling the recursion in the bottom-up manner

MATRIX_CHAIN_MULTIPLICATION$(p_0, p_1, ..., p_n)$

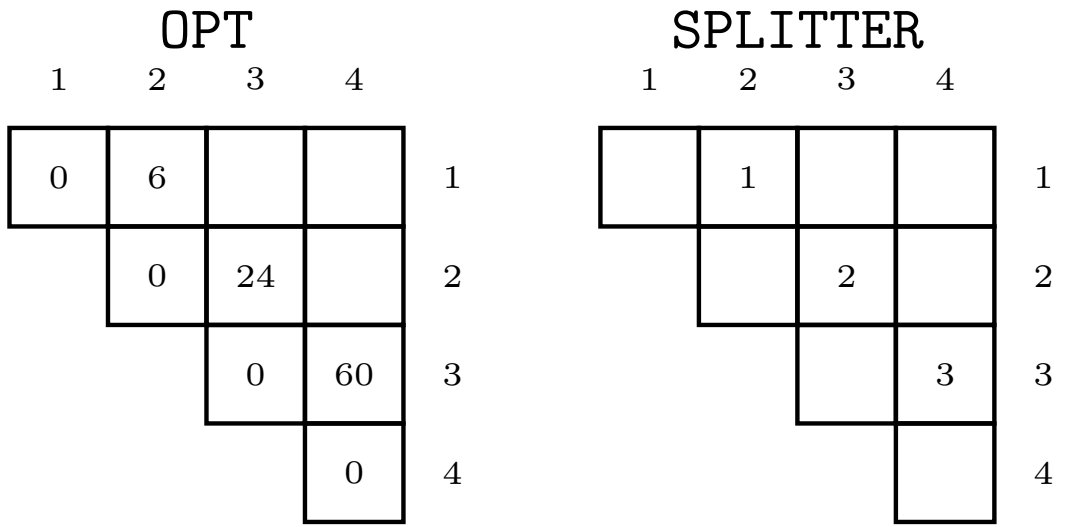
```
1:  for i = 1 to n do
2:      OPT(i, i) = 0;
3:  end for
4:  for l = 2 to n do
5:      for i = 1 to n − l + 1 do
6:          j = i + l − 1;
7:          OPT(i, j) = +∞;
8:          for k = i to j − 1 do
9:              q = OPT(i, k) + OPT(k + 1, j) + p_{i−1} p_k p_j;
10:             if q < OPT(i, j) then
11:                 OPT(i, j) = q;
12:                 S(i, j) = k;
13:             end if
14:         end for
15:     end for
16: end for
17: return OPT(1, n);
```

Recursion tree: an intuitive view of the bottom-up calculation



- Solving sub-problems in a bottom-up manner, i.e.
  1. Solving the sub-problems in red first;
  2. Then solving the sub-problems in green;
  3. Then solving the sub-problems in orange;
  4. Finally we can solve the original problem in blue.

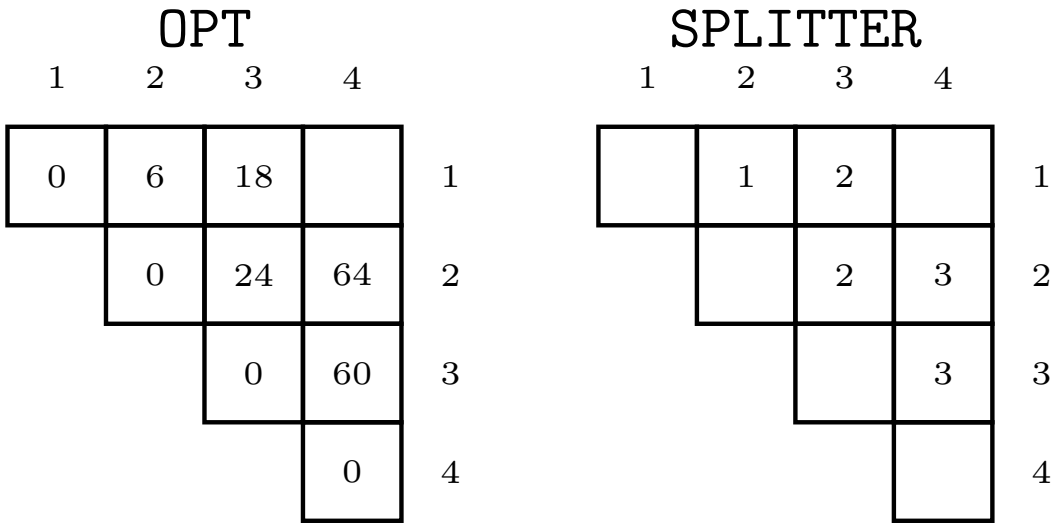
Step 1 of the bottom-up algorithm



Step 1:

$$OPT[1,2] = p_0 \times p_1 \times p_2 = 1 \times 2 \times 3 = 6;$$
$$OPT[2,3] = p_1 \times p_2 \times p_3 = 2 \times 3 \times 4 = 24;$$
$$OPT[3,4] = p_2 \times p_3 \times p_4 = 3 \times 4 \times 5 = 60;$$

Step 2 of the bottom-up algorithm



Step 2:

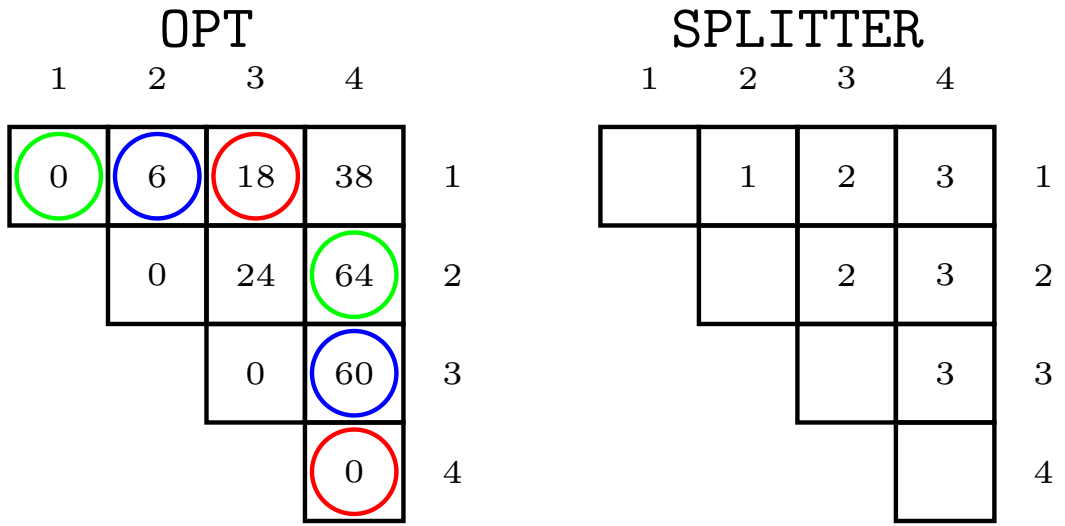
$$OPT[1,3] = \min \begin{cases} OPT[1,2] + OPT[3,3] + p_0 \times p_2 \times p_3 (= 18) \\ OPT[1,1] + OPT[2,3] + p_0 \times p_1 \times p_3 (= 32) \end{cases}$$

Thus, $SPLITTER[1,3] = 2$.

$$OPT[2,4] = \min \begin{cases} OPT[2,2] + OPT[3,4] + p_1 \times p_2 \times p_4 (= 90) \\ OPT[2,3] + OPT[4,4] + p_1 \times p_3 \times p_4 (= 64) \end{cases}$$

Thus, $SPLITTER[2,4] = 3$.

Step 3 of the bottom-up algorithm



Step 3:

$$OPT[1,4] = \min \begin{cases} OPT[1,1] + OPT[2,4] + p_0 \times p_1 \times p_4 (= 74) \\ OPT[1,2] + OPT[3,4] + p_0 \times p_2 \times p_4 (= 81) \\ OPT[1,3] + OPT[4,4] + p_0 \times p_3 \times p_4 (= 38) \end{cases}$$

Thus, $SPLITTER[1,4] = 3$.

Question: We have calculated the optimal **value**, but how to get the optimal **calculation sequence**?

Final step: constructing an optimal solution through "backtracking" the optimal options

- Idea: **backtracking!** Starting from $OPT[1, n]$, we trace back the source of $OPT[1, n]$, i.e. which option we take at each decision stage.
- Specifically, an auxiliary array $S[1..n, 1..n]$ is used.
  - Each entry $S[i, j]$ records the optimal decision, i.e. the value of k such that the optimal parentheses of $A_i...A_j$ occurs between $A_k A_{k+1}$.
  - Thus, the optimal solution to the original problem $A_{1..n}$ is $A_{1..S[1,n]} A_{S[1,n]+1..n}$.
- Note: The optimal option cannot be determined before solving all subproblems.

Backtracking: step 1

OPT

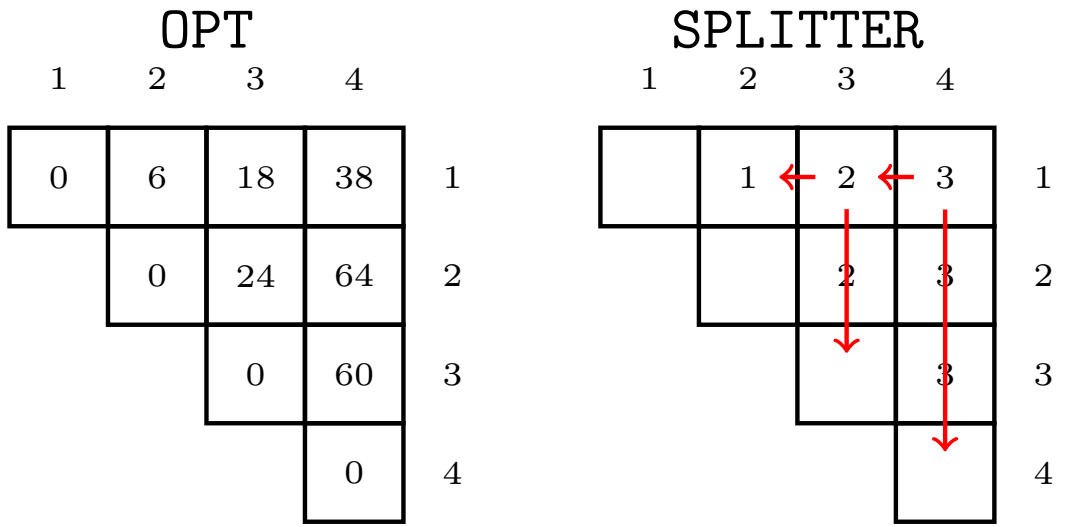
	1	2	3	4
1	0	6	18	38
2		0	24	64
3			0	60
4				0

SPLITTER

	1	2	3	4
1		1	2 ←	3
2			2	3
3				3
4				

Step 1: $($ $A_1 A_2 A_3$ $)$ $($ A_4 $)$

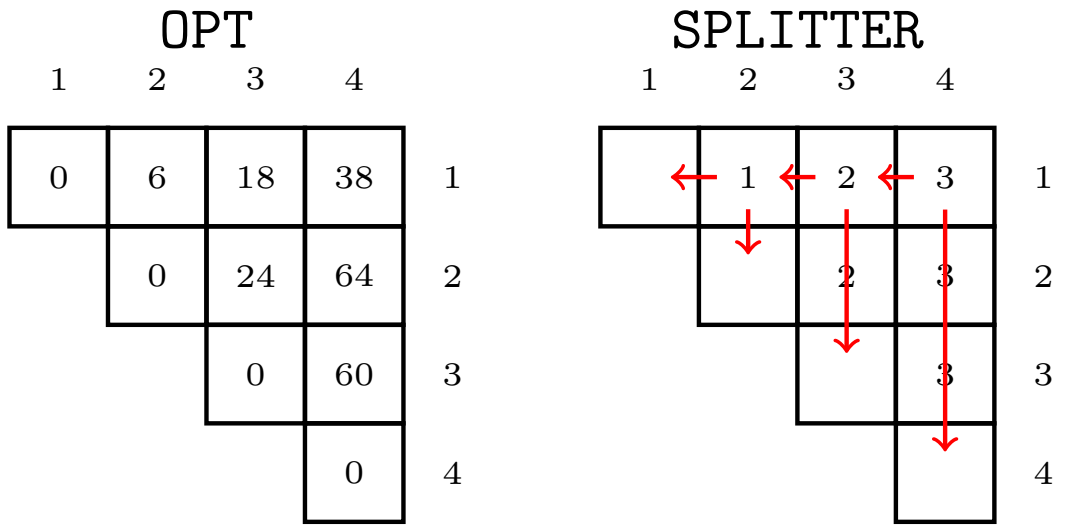
Backtracking: step 2



Step 1: $(A_1 A_2 A_3) (A_4)$

Step 2: $((A_1 A_2) (A_3)) (A_4)$

Backtracking: step 3



Step 1: $(A_1 A_2 A_3) (A_4)$

Step 2: $((A_1 A_2) (A_3)) (A_4)$

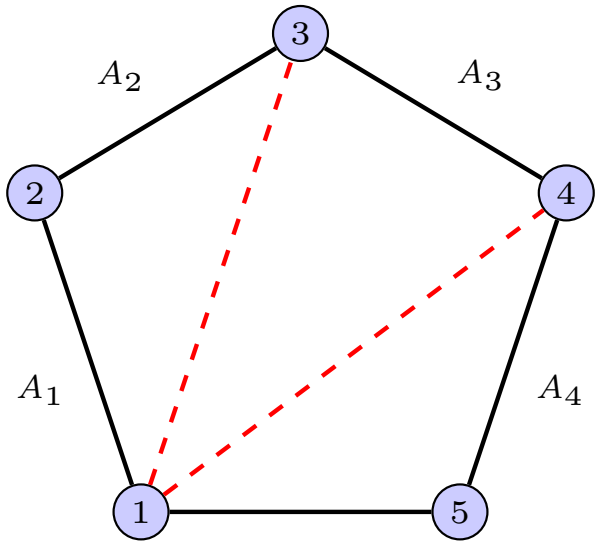
Step 3: $(((A_1) (A_2)) (A_3)) (A_4)$

Summary: elements of dynamics programming

1. It is usually not easy to solve a large problem directly. Let's consider whether the problem can be decomposed into smaller sub-problems. How to define sub-problems?
   - Let's describe the solving process as a process of multistage **decisions** first.
   - Let examine some examples of subproblems first. We consider the **first/final decision (in some order)** in the optimal solution. The **first/final decision** might have several options. We enumerate all possible options for the decision, and examine the generated sub-problems.
   - Next we defined the general form of sub-problems via summarizing all possible forms of sub-problems.
2. Show that the recursion among sub-problems can be stated as the **optimal substructure property**, i.e. the optimal solution to the problem contains within it optimal solutions to subproblems.
3. Programming: if recursive algorithm solves the same subproblem over and over, "tabular" can be used to avoid the repetition of solving same sub-problems.

Question: is $O(n^3)$ the lower bound?

An $O(n \log n)$ algorithm by Hu and Shing 1981



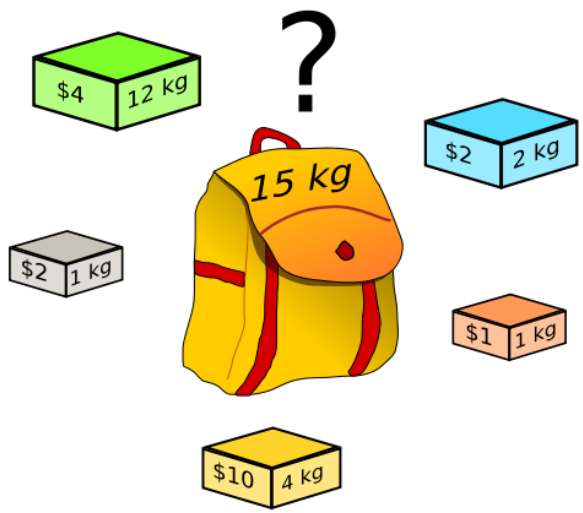
- One-to-one correspondence between parenthesis and partioning a convex polygon into non-intersecting triangles.
  - Each node has a weight w_i, and a triangle corresponds to a product of the weight of its nodes.
  - The decomposition (red, dashed lines) has a weight sum of 38. In fact, it corresponds to the parenthesis (((A_1) (A_2) (A_3))) (A_4).
- The optimal decomposition can be found in $O(n \log n)$ time.

(See Hu and Shing 1981 for details)

0-1 KNAPSACK problem: recursion over **sets**

A Knapsack instance

- Consider a set of items, where each item has a weight and a value. The objective is to select a subset of items such that the total weight is less than a given limit and the total value is as large as possible.



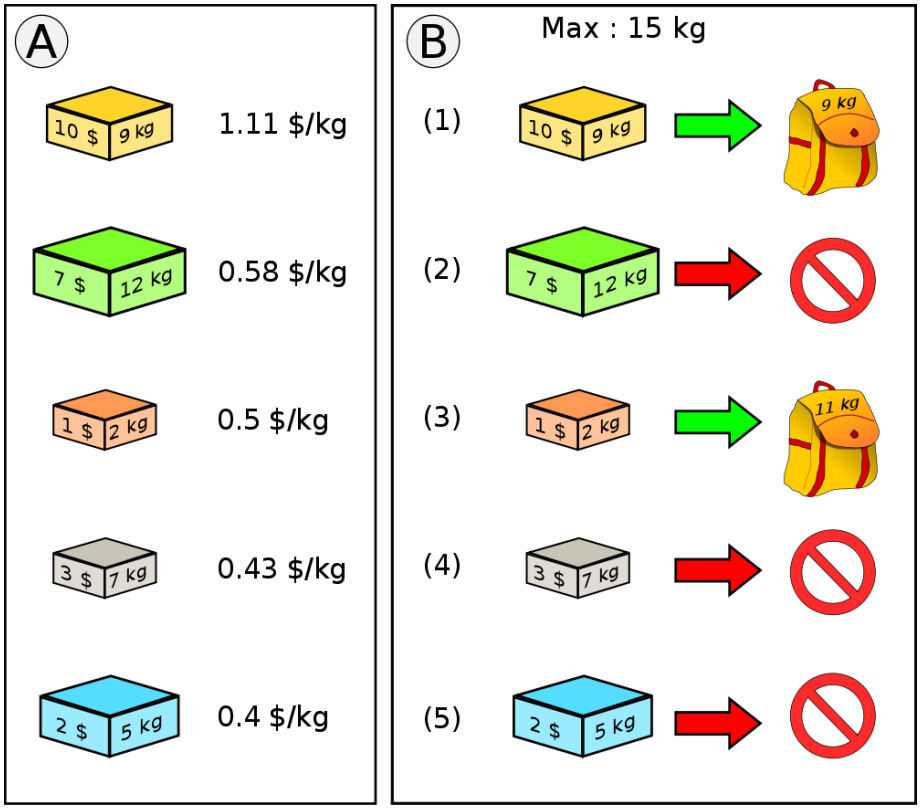
0-1 Knapsack problem

Formalized Definition:

INPUT: A set of items $S = \{1, 2, ..., n\}$. Item i has weight w_i and value v_i. A total weight limit W;
OUTPUT: A subset of items to maximize the total value with total weight below W.

- Here, "$0 - 1$" means that we should select an item (1) or abandon it (0), and we cannot select parts of an item.
- In contrast, Fractional Knapsack problem allow one to select a fractional, say 0.5, of an item.

0-1 Knapsack problem: an intuitive algorithm



- Intuitive method: selecting “expensive” items first.
- But this is not the optimal solution.

Defining the general form of sub-problems

- It is not easy to solve the problem with n items directly. Let's examine whether it is possible to reduce into smaller sub-problems.
- Solution: a subset of items. Let's describe the solving process as a process of **multistage decisions**. At the i-th decision stage, we decide whether item i should be selected.
- Let's consider **the first decision**, i.e. whether the optimal solution contains item n or not (here we assume an order of the items and consider the items from end to beginning). This decision has two options:
  1. SELECT: Then it suffices to select items as "expensive" as possible from $\{1, 2, ..., n-1\}$ with weight limit $W - w_n$.
  2. ABANDON: Otherwise, we should select items as "expensive" as possible from $\{1, 2, ..., n-1\}$ with weight limit W.
- In both cases, the original problem is reduced into smaller sub-problems.

Optimal sub-structure property

- Summarizing these two cases, we can set the general form of sub-problems as: to select items as "expensive" as possible from $\{1, 2, ..., i\}$ with weight limit w. Denote the optimal solution value as $OPT(\{1, 2, ..., i\}, w)$.
- Then we can prove the optimal sub-structure property:

$$OPT(\{1, 2, ..., n\}, W) = \max \begin{cases} OPT(\{1, 2, ..., n-1\}, W) \\ OPT(\{1, 2, ..., n-1\}, W - w_n) + v_n \end{cases}$$

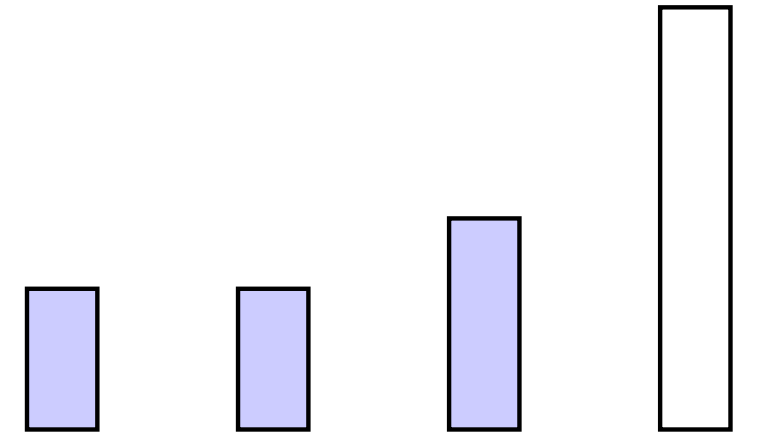
Algorithm

KNAPSACK(n, W)

1: **for** $w = 1$ to W **do**
2: $\quad OPT[0, w] = 0;$
3: **end for**
4: **for** $i = 1$ to n **do**
5: $\quad$ **for** $w = 1$ to W **do**
6: $\quad\quad OPT[i, w] = \max\{OPT[i-1, w], v_i + OPT[i-1, w-w_i]\};$
7: $\quad$ **end for**
8: **end for**
9: **return** $OPT[n, W];$

- Here we use $OPT[i, w]$ to represent $OPT(\{1, 2, ..., i\}, w)$ for simplicity.

An example: Step 1



Initially all $OPT[0, w] = 0$

$w =$	0	1	2	3	4	5	6
$i = 3$							
$i = 2$							
$i = 1$							
$i = 0$	0	0	0	0	0	0	0

Step 2

$$
\begin{aligned}
OPT[1,2] = \max\{ & \\
& OPT[0,2](=0), \\
& OPT[0,0] + V_1(=0+2)\} \\
& = 2
\end{aligned}
$$

$w=$	0	1	2	3	4	5	6
$i=3$							
$i=2$							
$i=1$	0	0	2	2	2	2	2
$i=0$	0	0	0	0	0	0	0

Step 3

$$OPT[2,4] = \max\{ OPT[1,4](=2), OPT[1,2] + V_2(=2+2)\} = 4$$

$w=$	0	1	2	3	4	5	6
$i=3$							
$i=2$	0	0	2	2	4	4	4
$i=1$	0	0	2	2	2	2	2
$i=0$	0	0	0	0	0	0	0

Step 4

$$OPT[3,3] = \max\{$$
$$OPT[2,3](=2),$$
$$OPT[2,0] + V_3(=0+3)\}$$
$$= 3$$

$w=$	0	1	2	3	4	5	6
$i=3$	0	0	2	3	4	5	5
$i=2$	0	0	2	2	4	4	4
$i=1$	0	0	2	2	2	2	2
$i=0$	0	0	0	0	0	0	0

Backtracking

$$OPT[3,6] = \max\{$$
$$\quad OPT[2,6](=4),$$
$$\quad OPT[2,3] + V_3(=2+3)\}$$
$$\quad = 5$$

Decision: Select item 3

$$OPT[2,3] = \max\{$$
$$\quad OPT[1,3](=2),$$
$$\quad OPT[1,1] + V_2(=0+2)\}$$
$$\quad = 2$$

Decision: Select item 2

$w=$	0	1	2	3	4	5	6
$i=3$	0	0	2	3	4	5	5
$i=2$	0	0	2	2	4	4	4
$i=1$	0	0	2	2	2	2	2
$i=0$	0	0	0	0	0	0	0

Time complexity analysis

- Time complexity: $O(nW)$. (Hint: for each entry in the matrix, only a comparison is needed; we have $O(nW)$ entries in the matrix.)
- Notes:
  1. This algorithm is inefficient when W is large, say $W = 1M$.
  2. Remember that a polynomial time algorithm costs time polynomial in the **input length**. However, this algorithm costs time $mW = m2^{\log W} = m2^{\text{input length}}$. Exponential!
  3. Pseudo-polynomial time algorithm: polynominal in the **value** of W rather than the **length** of W ($\log W$).
  4. We will revisit this algorithm in approximation algorithm design.

Why should we consider items from end to beginning?

- Let's examine the following two types of selection of items:
  1. If we consider an arbitrary item i, then the sub-problem becomes into "selecting items as expensive as possible from **a subset** s with weight limit w". We have the following recursion:

$$OPT(\{1,2,...,n\},W) = \max \begin{cases} OPT(\{1,2,...,n\}-\{i\},W) \\ OPT(\{1,2,...,n\}-\{i\},W-w_i)+v_i \end{cases}$$

  2. In contrast, if we assume an order of the items and consider them from end to beginning, then the subproblem can be set as "selecting items as expensive as possible from $\{1,2,...,i\}$ with weight limit w" and we have the following recursion:

$$OPT(\{1,2,...,n\},W) = \max \begin{cases} OPT(\{1,2,...,n-1\},W) \\ OPT(\{1,2,...,n-1\},W-w_n)+v_n \end{cases}$$

- In fact, the first one exploits **recursion over sets**, which leads to an exponential number of subproblems. In contrast, the second one is a **recursion over sequences** and the number of subproblems is only $O(nW)$.

Extension: The first public-key encryption system

- *Cryptosystems based on the knapsack problem were among the first public key systems to be invented, and for a while were considered to be among the most promising. However, essentially all of the knapsack cryptosystems that have been proposed so far have been broken. These notes outline the basic constructions of these cryptosystems and attacks that have been developed on them.*

See *The Rise and Fall of Knapsack Cryptosystems* for details.

Vertex Cover: recursion over **trees**

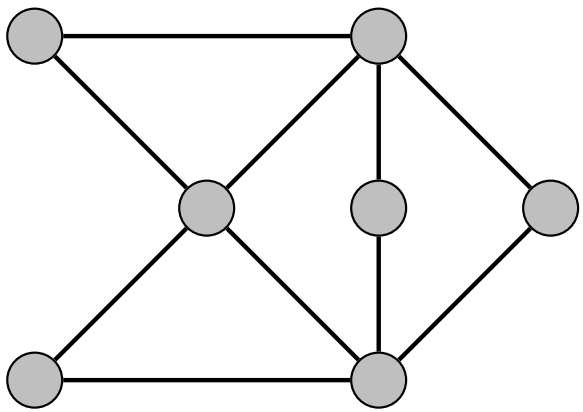
Vertex Cover Problem

- Practical problem:
  Given n sites connected with paths, how many guards (or cameras) should be deployed on sites to surveille **all** *the paths?*

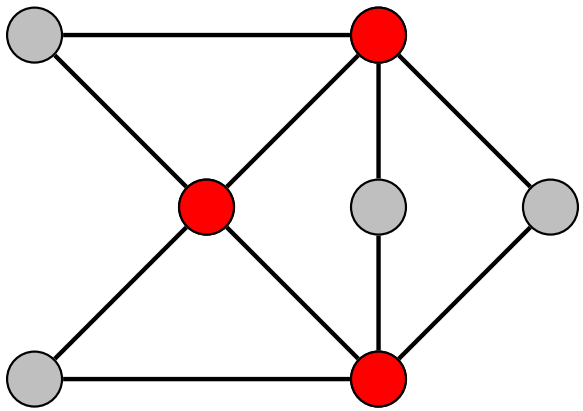
Formalized Definition:

Input: Given a graph $G =< V, E >$
Output: the minimum of nodes $S \subseteq V$, such that each edge has at least one of its endpoints in S

- For example, how many nodes are needed to cover all edges in the following graph?

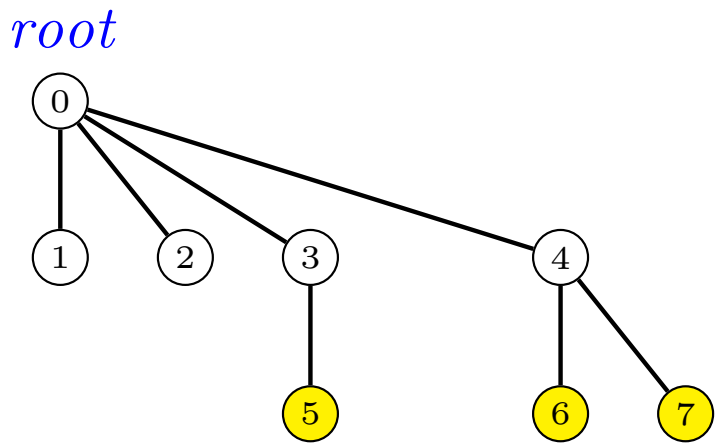


Vertex cover



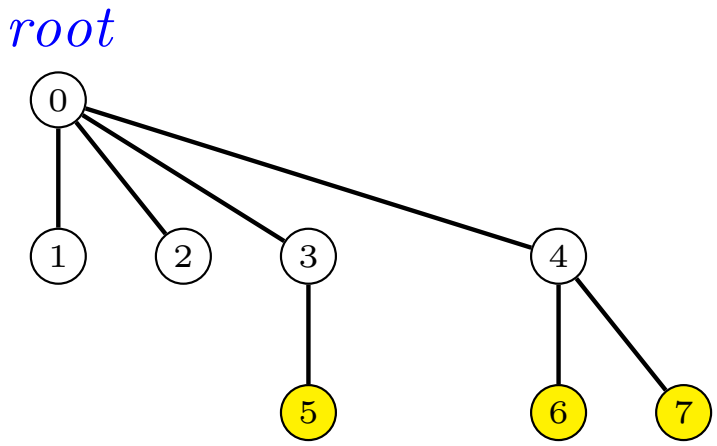
- The nodes in red form a vertex cover.
- Vertex Cover is a hard problem for general graph.
- However, it is easy to find the minimum vertex cover for **trees**.

Optimal sub-structure property



- It is not easy to solve the problem with n nodes. Let's whether it is possible to reduce into smaller sub-problems.
- Solution: selection a subset of nodes. Describe the solving process as as a process of multistage **decisions**. At each decision stage, we decide whether a node should be selected.
- Let's consider the **first** decision in the optimal solution, i.e. whether the optimal solution contains the **root** node or not. The decision has two options:
  1. SELECT: it suffices to consider the sub-trees;
  2. ABANDON: we should select all the children nodes, and then consider all grand-children.

An example



- In both cases, the original problem is reduced into smaller sub-problems.
- General form of sub-problems: find minimum vertex cover on a tree rooted at node v. Let's denote the optimal solution as $OPT(v)$.
- Thus we have the following recursion:

$$OPT(root) = \min \begin{cases} 1 + \sum_c OPT(c) & c \text{ is a child} \\ \#children + \sum_g OPT(g) & g \text{ is a grand-child} \end{cases}$$

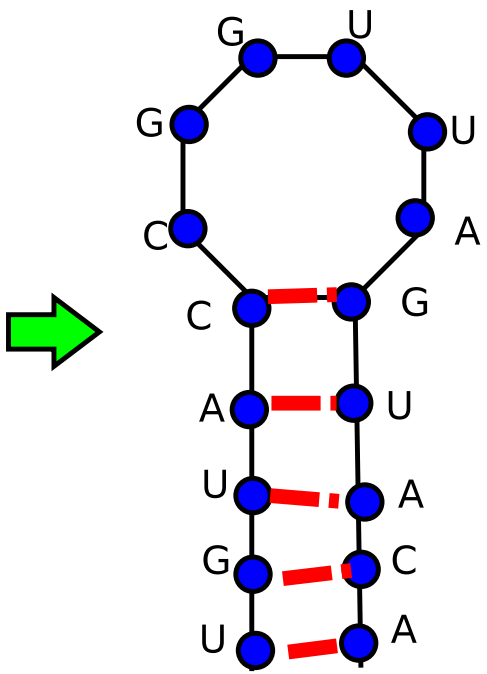
- Time complexity: $O(n)$ (Reason: each node will be visited twice.)

RNA Secondary Structure Prediction: recursion over **sequence**

RNA secondary structure

- RNA is a sequence of nucleic acids. It will automatically form structures in water through the formation of bonds $A - U$ and $C - G$.
- The native structure is the conformation with the lowest energy. Here, we simply use the number of base pairs as the energy function.

A C A U G A U U G G C C A U G U



Formulation

INPUT:
A sequence in alphabet $\Sigma = \{A, U, C, G\}$;
OUTPUT:
A pairing scheme with the maximum pairing number

Requirements of base pairs:

1. Watson-Crick pair: A pairs with U, and C pairs with G;
2. There is no base occurring in more than 1 base pairs;
3. No cross-over (nesting): there is no crossover under the assumption of free pseudo-knots.
4. And two bases i, j ($|i - j| \leq 4$) cannot form a base pair.

Nesting and Pseudo-knot

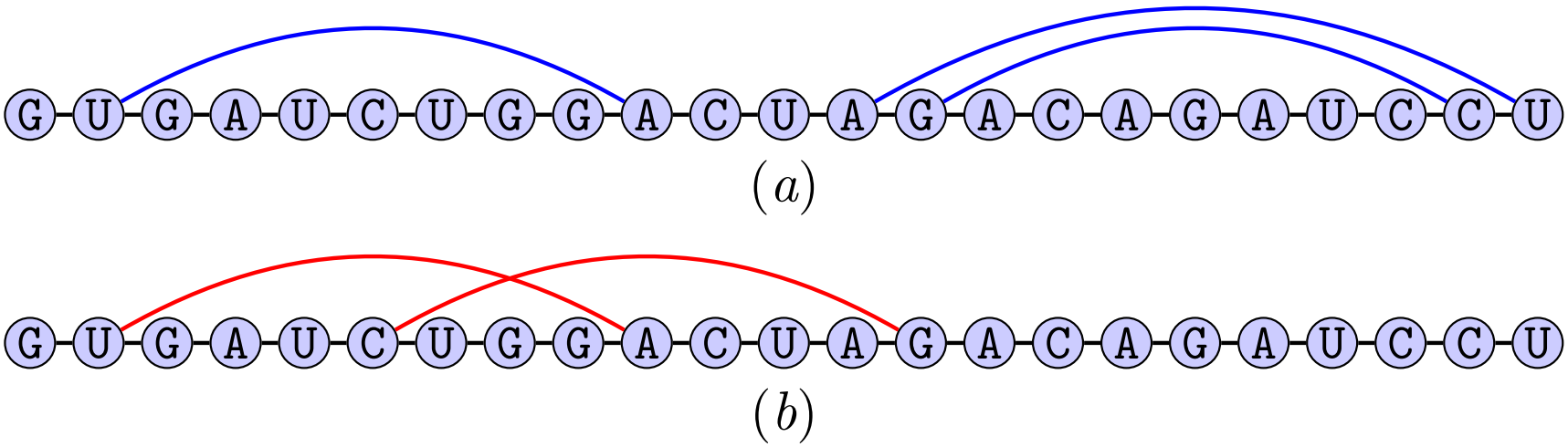
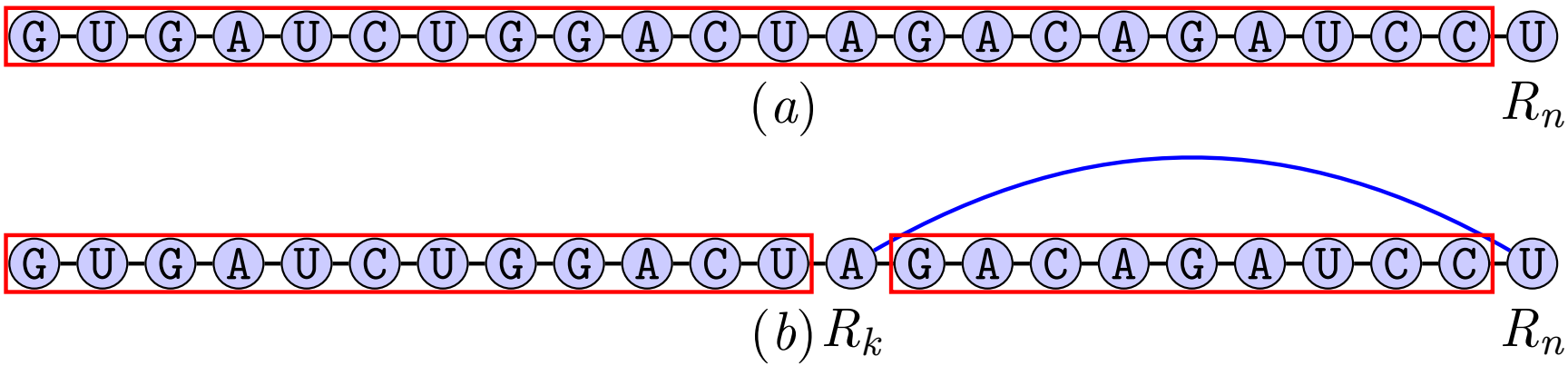


Figure 1: Two base pairs should be independent of nested (a) but cannot form cross-over (b).

Defining general form of sub-problems

- Solution: a set of nested base pairs. Describe the solving process as as a process of multistage **decisions**. At the i-th decision stage, we determine whether base i forms pair or not.
- Let's consider the $first$ decision made for base n in the optimal solution. There are two options:
  1. Base n pairs with a base i: we should calculate optimal pairs for regions $i+1..n-1$ and $1..i-1$. Note that these two sub-problems are independent due to the "nested" property.
  2. Base n doesn't form a pair: we should calculate optimal pairs for regions $1..n-1$.
- Thus we can design the general form of sub-problems as: to calculate the optimal pairs for region $i..j$. (Denote the optimal solution value as: $OPT(i,j)$.)

Optimal sub-structure property



- Optimal substructures property: $OPT(1,n) =$

$$\max \begin{cases} OPT(1,n-1) \\ \max\limits_{\substack{1 \le k < n-4 \\ R_k \text{ pairs with } R_n}} \{1 + OPT(1,k-1) + OPT(k+1,n-1)\} \end{cases} .$$

Algorithm

RNA2D(n)

1: Initialize all $OPT[i,j]$ with 0;
2: **for** $i = 1$ to n **do**
3: **for** $j = i + 5$ to n **do**
4: $OPT[i,j] = \max\{OPT[i,j-1], \max_t\{1 + OPT[i,t-1] + OPT[t+1,j-1]\}\}$;
5: /* t and j can form Watson-Crick base pair. */
6: **end for**
7: **end for**

An example

$j=$	6	7	8	9
$i=4$	0	0	0	0
$i=3$	0	0	1	
$i=2$	0	0		
$i=1$	1			

(a)

6	7	8	9
0	0	0	0
0	0	1	1
0	0	1	
1	1		

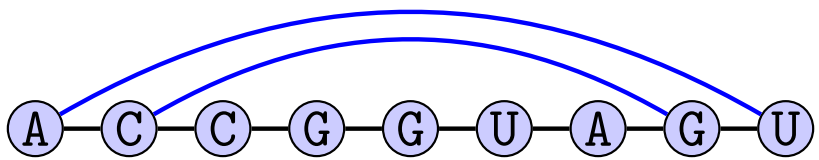
(b)

6	7	8	9
0	0	0	0
0	0	1	1
0	0	1	1
1	1	1	

(c)

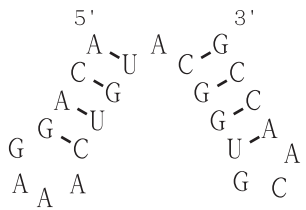
6	7	8
0	0	0
0	0	1
0	0	1
1	1	1

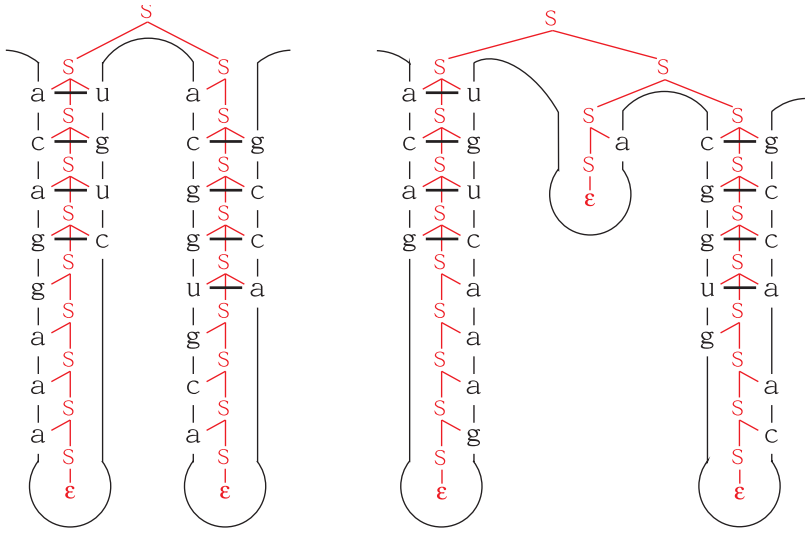
(d)



Time complexity: $O(n^3)$.

Extension: RNA is a good example of SCFG.





(see extra slides)

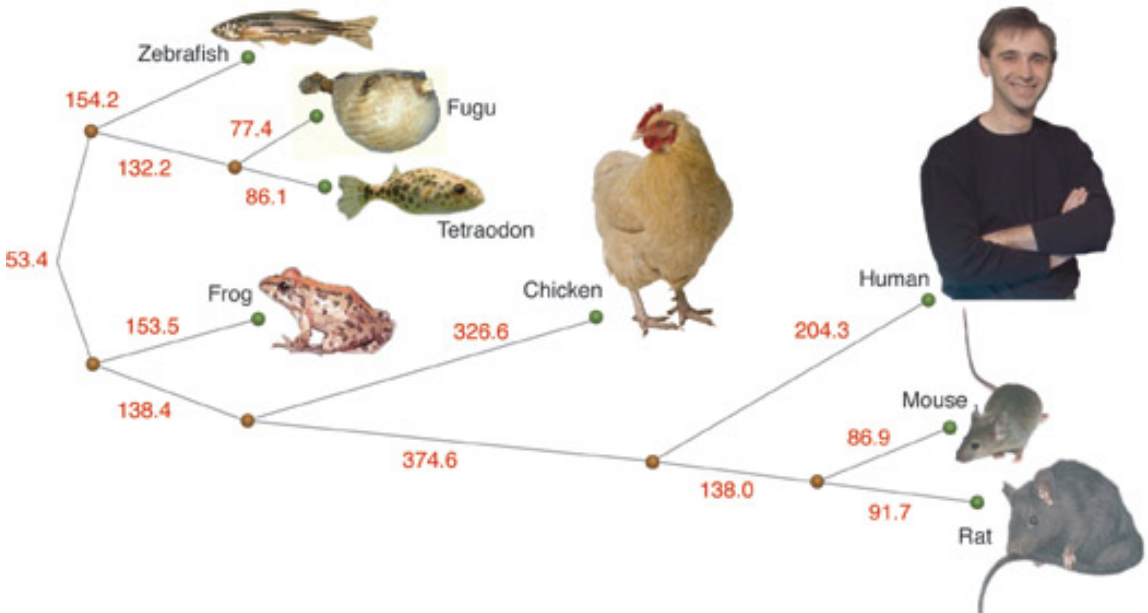
Sequence Alignment problem: recursion over **sequence pairs**

Practical problem: genome similarity

- To identify homology genes of two species, say $Human$ and $Mouse$. Human and Mouse NHPPEs (in KRAS genes) show a high sequence homology (Ref: Cogoi, S., et al. NAR, 2006).

```
GGGCGGTGTG
|||  ||  |  |
GGGAGG-GAG
```

- Having calculating the similarity of genomes of various species, a reasonable phylogeny tree can be estimated (See https://www.llnl.gov/str/June05/Ovcharenko.html)



Practical problem: spell tool to correct typos

- When you type in "OCURRANCE", spell tools might guess what you really want to type through the following alignment:

```
O-CURRANCE
| |||| |||
OCCURRENCE
```

  Here, '|' represent identical characters.

- This alignment suggests that "OCURRANCE" is very similar to "OCCURRENCE", i.e., "OCURRANCE" is generated from "OCCURRENCE" using several INS/DEL/MUTATION operations.

What is an alignment?

- Alignment is introduced to describe the generating process of an erroneous word from the correct word using a series of `Insertion/Deletion/Match/Mutation` operations.
- For this aim, we make the two sequences to have the same length through adding space '-' at appropriate positions, which changes S to S', and changes T to T' with identical length $|S'| = |T'|$.

```
S': O-CURR-ANCE
    | ||||  |||
T': OCCURRE-NCE
```

- The generating process of S from T is described using spaces '-':
  1. $S'[i]$ ='-': $S'[i]$ represents a `deletion` of $T'[i]$.
  2. $T'[i]$ ='-': $S'[i]$ represents an inserted letter.
  3. Otherwise, $S'[i]$ is a copy of $T'[i]$ (with possible `mutation`).

Measuring the possibility of each generating process

- For each generating process, we can measure its possibility using a linear function:

$$s(T, S) = \sum_{i=1}^{|S'|} s(T'[i], S'[i])$$

- Here we assume a simple setting of $s(a, b)$ as follows:
  1. MATCH: $+1$, e.g. s('C', 'C')$= 1$.
  2. MUTATION: -1, e.g. s('E', 'A') $= -1$.
  3. INSERTION/DELETION: -3, e.g. s('C', '-')$= -3$.

1

[1] Ideally, the score function is designed such that $s(T, S)$ is proportional to $\log \Pr[S \text{ is generated from } T]$. See extra slides for the statistical model for sequence alignment, and better similarity definition, say BLOSUM62, PAM250 substitution matrix, etc.

Alignment is useful

- Application 1: Using alignment, we can determine the most likely source of "OCURRANCE".
  1. $T =$ "OCCURRENCE":

```
S': O-CURRANCE
    | |||| |||
T': OCCURRENCE
```

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4$$

  2. $T =$"OCCUPATION":

```
S': OC-URRA---NCE
    || |   |    |
T': OCCU-PATION--
```

$$s(T', S') = 1+1-3+1-3-1+1-3-3-3+1-3-3 = -14$$

- Thus, it is more likely that "OCURRANCE" comes from "OCCURRENCE" relative to "OCCUPATION".

Alignment is useful cont'd

- Application 2: In addition, we can also determine the most likely operations changing "OCCURRENCE" into "OCURRANCE".
  1. Alignment 1:

```
S': O-CURRANCE
    | |||| |||
T': OCCURRENCE
```

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4$$

  2. Alignment 2:

```
S': O-CURR-ANCE
    | |||| |||
T': OCCURRE-NCE
```

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1$$

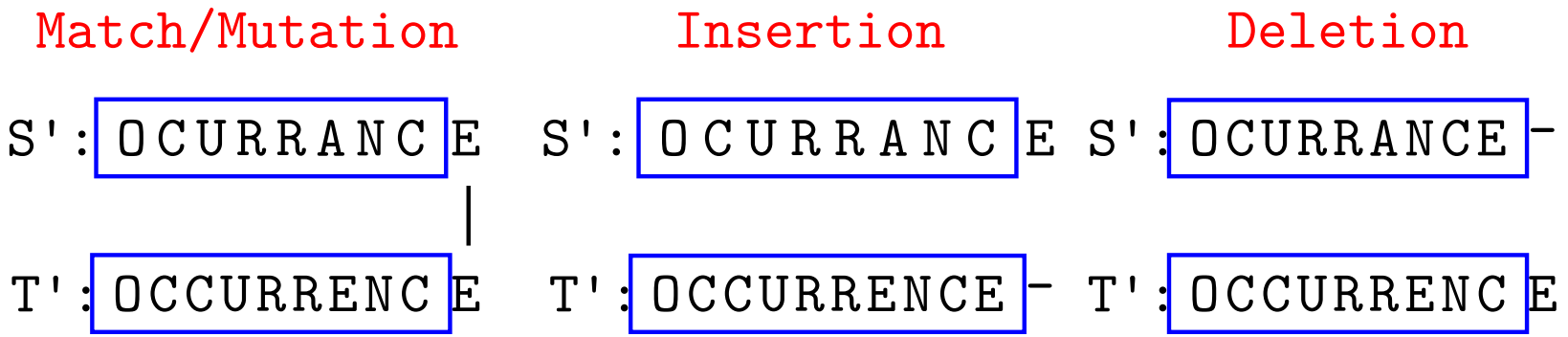
- Thus, the first alignment might describes the real generating process of "OCURRANCE" from "OCCURRENCE".

INPUT:
Two sequences T and S, $|T| = m$, and $|S| = n$;
OUTPUT:
To identify an alignment of T and S that maximizes a pre-defined scoring function.

Note: for the sake of simplicity, the following indexing schema are used: $T = T_1 T_2 ... T_m$ and $S = S_1 S_2 ... S_n$.

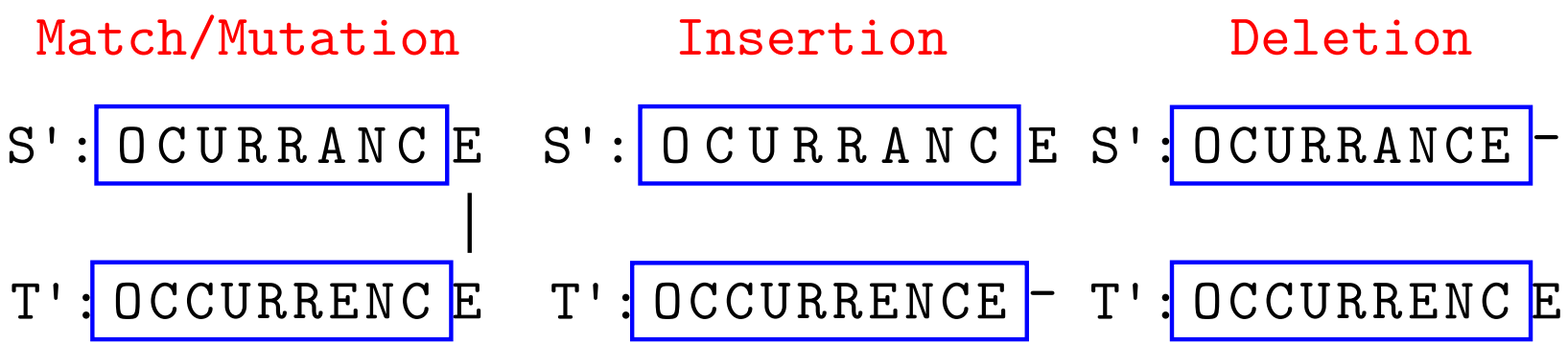
The general form of sub-problems and recursions I

- It is not easy to consider long sequences directly. Let's consider whether it is possible to reduce into smaller subproblem.
- Solution: insert '-' at appropriate positions to represent the generating process of S from T. Let's describe the solving process as a process of **multistage decisions**. At each decision stage, we decide whether T_i aligns with S_j (Match/Mutation), S_j aligns with a '-' (Insertion), or T_i aligns with a '-' (Deletion).



The general form of sub-problems and recursions II

- Let's consider **the first decision made for S_m** in the optimal solution. There are three cases:
  1. S_n comes from T_m: represented as aligning S_n with T_m. Then it suffices to align $T[1..m-1]$ with $S[1..n-1]$.
  2. S_n is an `INSERTION`: represented as aligning S_n with a space '-'. Then it suffices to align $T[1..m]$ and $S[1..n-1]$.
  3. S_n comes from $T[1..m-1]$: represented as aligning T_m with a space '-'. Then it suffices to align $T[1..m-1]$ and $S[1..n]$.



The general form of sub-problems and recursions III

- Summarizing these three examples of sub-problems, we can design the general form of sub-problems as: aligning a **prefix** of T (denoted as $T[1..i]$) and **prefix** of S (denoted as $S[1..j]$). Denote the optimal solution value as $OPT(i, j)$.
- We can prove the following optimal substructure property:

$$OPT(i,j) = \max \begin{cases} s(T_i, S_j) + OPT(i-1, j-1) \\ s(\text{`\_'}, S_j) + OPT(i, j-1) \\ s(T_i, \text{`\_'}) + OPT(i-1, j) \end{cases}$$

Needleman-Wunsch algorithm [1970]

NEEDLEMAN-WUNSCH(T, S)

1: **for** $i = 0$ to m **do**
2: $\quad OPT[i, 0] = -3 * i;$
3: **end for**
4: **for** $j = 0$ to n **do**
5: $\quad OPT[0, j] = -3 * j;$
6: **end for**
7: **for** $j = 1$ to n **do**
8: $\quad$ **for** $i = 1$ to m **do**
9: $\quad\quad OPT[i, j] = \max\{OPT[i-1, j-1] + s(T_i, S_j), OPT[i-1, j] - 3, OPT[i, j-1] - 3\};$
10: $\quad$ **end for**
11: **end for**
12: **return** $OPT[m, n];$

Note: the first column is introduced to describe the alignment of prefixes $T[1..i]$ with an empty sequence ϵ, so does the first row.

The first row/column of the alignment score matrix

T: \ S:	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3									
C	−6									
C	−9									
U	−12									
R	−15									
R	−18									
E	−21									
N	−24									
C	−27									
E	−30									

Score: `OPT("", "OCU") = -9`
Alignment:
```
S'= OCU
T'= ---
```

Score: `OPT("OC", "") = -6`
Alignment:
```
S'= --
T'= OC
```

Why introducing the first row/column?

	S: ''	O	C	U	R	R	A	N	C	E
T:''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: `OPT("O", "OC") =` $\max \begin{cases} \text{OPT(””, ”OC”)} & -3 & (=\text{-9}) \\ \text{OPT(””, ”O”)} & -1 & (=\text{-4}) \\ \text{OPT(”O”,”O”)} & -3 & (=\text{-2}) \end{cases}$

Alignment:
```
S'= OC
T'= O-
```

General cases

	S: ''	O	C	U	R	R	A	N	C	E
T:''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: `OPT("OC", "OCUR")` = $\max$ { `OPT("O","OCUR")` -3 (=-11); `OPT("O","OCU")` -1 (=-6); `OPT("OC","OCU")` -3 (=-4) }

Alignment:
```
S'= OCUR
T'= OC--
```

The final entry

	S: ''	O	C	U	R	R	A	N	C	E
T:''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: `OPT("OCCURRENCE", "OCURRANCE")` $= \max$

`OPT("OCCURRENC", "OCURRANCE")`	-3	(=-3)
`OPT("OCCURRENC", "OCURRANC")`	$+1$	(=4)
`OPT("OCCURRENCE", "OCURRANC")`	-3	(=-3)

Alignment:

```
S'= O-CURRANCE
T'= OCCURRENCE
```

Question: how to find the alignment with the highest score?

Find the optimal alignment via backtracking

T:	S: ''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Optimal Alignment: S'= O-CURRANCE
T'= OCCURRENCE

Optimal alignment versus sub-optimal alignments

- In practice, there are always multiple alignments with nearly the same score as the optimal alignment. Such alignments are called **sub-optimal alignments** and can be divided into the following two categories.
  - Similar sub-optimal alignments: Some sub-optimal alignments differ from the optimal alignment in a few positions. Because variations might occur independently at different positions, the number of sub-optimal alignments grow exponentially with the difference to the optimal alignment. The typical sub-optimal alignments can be obtained through **sampling** over the dynamic programming matrix.
  - Distinct sub-optimal alignments: The other sub-optimal alignments differ completely from the optimal alignment. These sub-optimal alignments frequently appear when one or both sequences have repeats.

Please refer to Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids for details.

Finding sub-optimal alignments

- To find similar sub-optimal alignments, we trace back through the dynamic programming matrix using sampling technique, i.e., instead of taking the highest scoring option at each step, we make probabilistic choices bases on the value of the three options.
- To find the next best alignment sharing no aligned residue pairs with the optimal alignment, we can re-calculate the dynamic programming matrix with cells corresponding to the pairs in the optimal alignment set to zero. The resulting matrix can be used to find the second best alignment. Another approach is extending cells of OPT table to store top alignments.

Advanced DP

- Once have a basic DP algorithm, it is usually possible to make further improvement by saving running time and space requirement.
- Saving space requirement:
  - In 1975, D. S. Hirschberg proposed to apply Divide-and-Conquer technique to calculate optimal sequence alignment using only $O(m+n)$ space. J. Ian Munro improved this algorithm in 1982.
  - In 2018, F. Yang et al. proposed to use neural network to reduce the space requirement of Held-Karp algorithm
- Saving running time:
  - Exploring the **location of optimal solution**: banded DP
  - Exploring the **sparsity of memoization table**: sparse DP
  - Exploring the **monotonicity of backtracking table**

Space efficient algorithm: reducing the space requirement from $O(mn)$ to $O(m+n)$ using DIVIDE-AND-CONQUER (D. S. Hirschberg, 1975)

Technique 1: two arrays are enough if only score is needed

- Key observation 1: it is easy to calculate **the final score** $OPT(T, S)$ **only**, i.e. **the alignment information are not recorded.**

	S: ''	O	C	U	R	R	A	N	C	E
T:''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Technique 1: two arrays are enough if only score is needed

- Why? Only column $j-1$ is needed to calculate column j. Thus, we use two arrays $score[1..m]$ and $newscore[1..m]$ instead of the matrix $OPT[1..m, 1..n]$.

	S:	''	O	C	U	R	R	A	N	C	E
T:	''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
	O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
	C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
	C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
	U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
	R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
	R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
	E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
	N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
	C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
	E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Technique 1: two arrays are enough if only score is needed

- Why? Only column $j-1$ is needed to calculate column i. Thus, we use two arrays $score[1..n]$ and $newscore[1..n]$ instead of the matrix $OPT[1..n, 1..m]$.

	S: ''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Technique 1: two arrays are enough if only score is needed

- Why? Only column $j-1$ is needed to calculate column i. Thus, we use two arrays $score[0..m]$ and $newscore[0..m]$ instead of the matrix $OPT[0..m, 0..n]$.

	S:	''	O	C	U	R	R	A	N	C	E
T:	''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
	O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
	C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
	C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
	U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
	R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
	R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
	E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
	N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
	C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
	E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Algorithm

PREFIX_SPACE_EFFICIENT_ALIGNMENT($T, S, score$)

```
1: for i = 0 to m do
2:     score[i] = −3 ∗ i;
3: end for
4: for j = 1 to n do
5:     newscore[0] = −3 ∗ j;
6:     for i = 1 to m do
7:         newscore[i] = max{score[i − 1] + s(T_i, S_j), score[i] −
           3, newscore[i − 1] − 3};
8:     end for
9:     for i = 1 to m do
10:        score[i] = newscore[i];
11:    end for
12: end for
13: return score[m];
```

Technique 2: aligning suffixes instead of prefixes

- Key observation: Similarly, we can align **suffixes** of T and S instead of **prefixes** and obtain the same score and alignment.

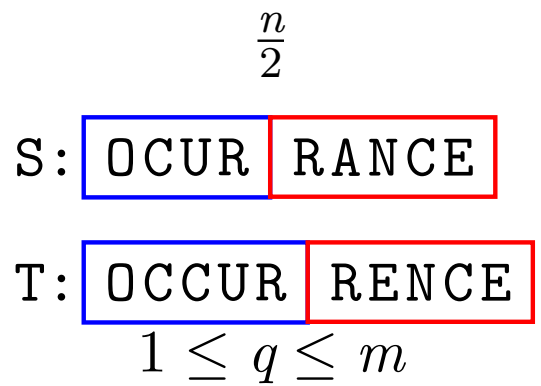
4	0	−4	−10	−12	−16	−18	−22	−26	−30	O
5	3	−1	−7	−9	−13	−15	−19	−23	−27	C
3	6	2	−4	−6	−10	−12	−16	−20	−24	C
−1	2	5	−1	−3	−7	−9	−13	−17	−21	U
−5	−2	1	4	0	−4	−6	−10	−14	−18	R
−9	−6	−3	0	3	−1	−3	−7	−11	−15	R
−13	−10	−7	−4	−1	2	0	−4	−8	−12	E
−15	−12	−9	−6	−3	0	3	−1	−5	−9	N
−19	−16	−13	−10	−7	−4	−1	2	−2	−6	C
−23	−20	−17	−14	−11	−8	−5	−2	1	−3	E
−27	−24	−21	−18	−15	−12	−9	−6	−3	0	''
O	C	U	R	R	A	N	C	E	''	S

Final difficulty: identify optimal alignment besides score

1. However, the optimal alignment cannot be restored via **backtracking** since **only the recent two columns** of the matrix were kept.
2. A clever idea: Suppose we have already obtained the optimal alignment. Let's consider the position **where $S_{[\frac{n}{2}]}$ is aligned to** (denoted as q). This leads to another division of S:

$$OPT(T,S) = OPT(T[1..q], S[1..\tfrac{n}{2}]) + OPT(T[q+1..m], S[\tfrac{n}{2}+1..n])$$

3. The equality holds due to the linearity of $OPT(T,S)$, and $\frac{n}{2}$ is chosen for the sake of time-complexity analysis.



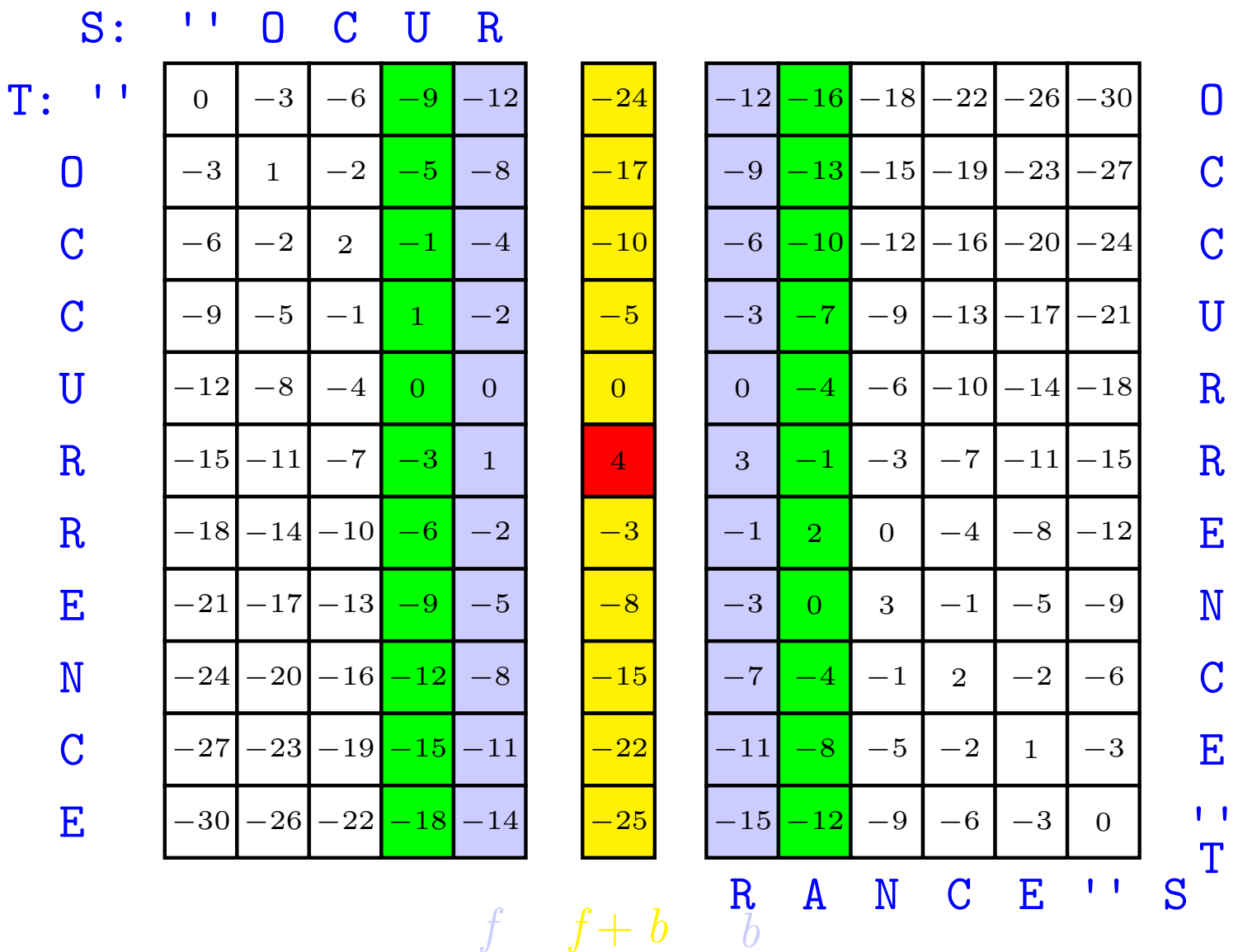
Hirschberg's algorithm for alignment

LINEAR_SPACE_ALIGNMENT(T, S)

1: Allocate two arrays f and b; each array has a size of m .
2: PREFIX_SPACE_EFFICIENT_ALIGNMENT($T, S[1..\frac{n}{2}], f$);
3: SUFFIX_SPACE_EFFICIENT_ALIGNMENT($T, S[\frac{n}{2}+1..n], b$);
4: Let $q = argmax_i(f[i] + b[i])$;
5: Free arrays f and b;
6: Record aligned-position $< \frac{n}{2}, q >$ in an array A;
7: LINEAR_SPACE_ALIGNMENT($T[1..q], S[1..\frac{n}{2}]$);
8: LINEAR_SPACE_ALIGNMENT($T[q+1..m], S[\frac{n}{2}+1..n]$);
9: **return** A;

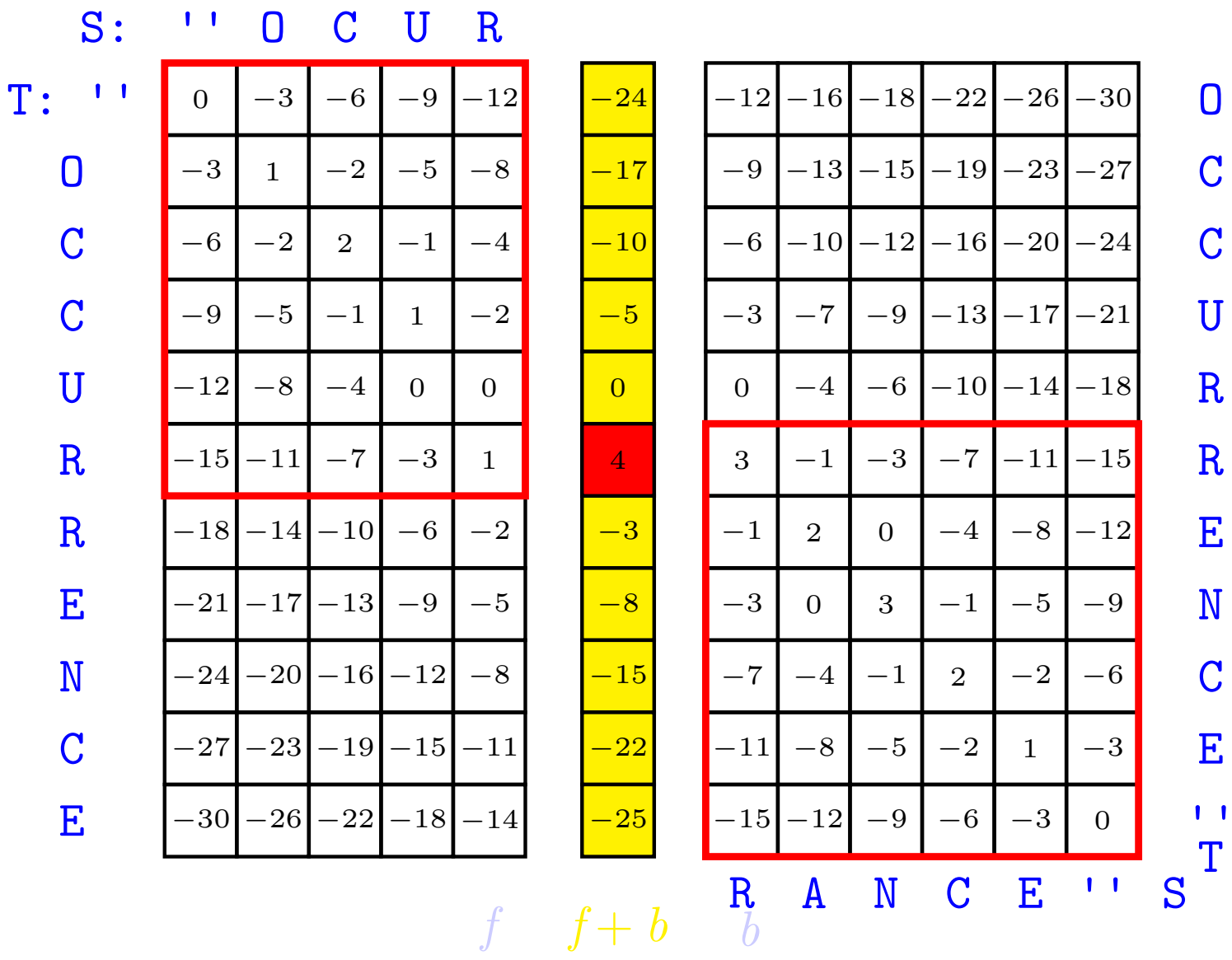
- Key observation: at each iteration step, only $2m$ space is needed.
- How to determine q? **Identifying the largest entry in $f[i] + b[i]$.**

Step 1: Determine the optimal aligned position of $S_{[\frac{n}{2}]}$



The **value** of the largest item is 4, which is actually the optimal score $OPT(S, T)$

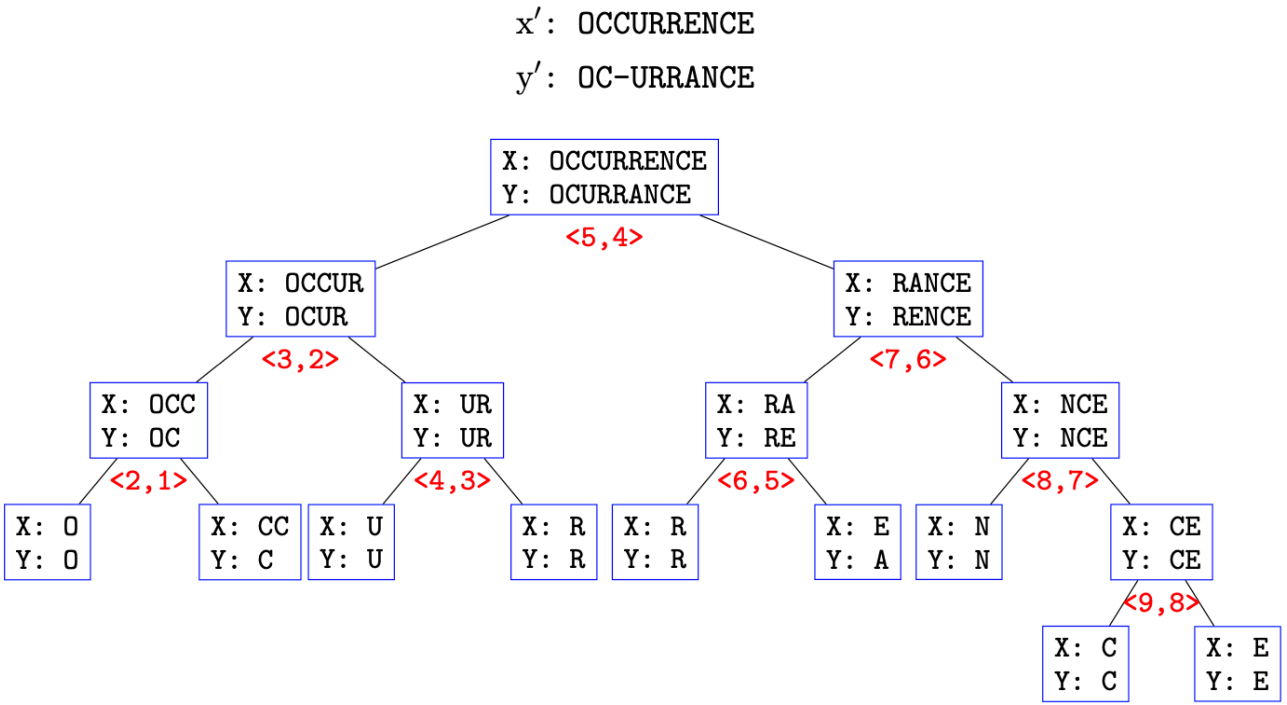
Step 2: Recursively solve sub-problems



We generate two sub-problems according to the position q.

An example

- We obtained A:
  A=[<5, 4>, <3, 2>, <1, 1>, <4,3>, <7,6>, <6,5>, <8,7>, <9,8>



Space complexity analysis

- The total space requirement: $O(m+n)$.
  - PREFIX_SPACE_EFFICIENT_ALIGNMENT$(T, S[1..\frac{n}{2}], f)$ needs only $O(m)$ space;
  - SUFFIX_SPACE_EFFICIENT_ALIGNMENT$(T, S[\frac{n}{2}+1..n], b)$ needs only $O(m)$ space;
  - Line 4 (Record $< \frac{n}{2}, q >$ in array A) needs only $O(n)$ space;

Time complexity analysis

Theorem

Algorithm LINEAR_SPACE_ALIGNMENT(S, T) *still takes* $O(mn)$ *time.*

Proof.

- The algorithm implies the following recursion:

$$T(m,n) = cmn + T(q, \tfrac{n}{2}) + T(m-q, \tfrac{n}{2})$$

- Difficulty: we have no idea of q before algorithm ends; thus, the master theorem cannot apply directly. **Guess and substitution, or unrolling the recursion to find some pattern!**
- Guess: $T(m', n') \leq km'n'$ follows for any $m' < m$ and $n' < n$.
- Verification:

$$\begin{aligned} T(m,n) &= cmn + T(q, \tfrac{n}{2}) + T(m-q, \tfrac{n}{2}) \\ &\leq cmn + kq\tfrac{n}{2} + k(m-q)\tfrac{n}{2} \\ &\leq (c + \frac{k}{2})mn \\ &= kmn \qquad\qquad (set\ k = 2c) \end{aligned}$$

Generalization to other DP algorithms

- It should be pointed out that Hirschberg's DIVIDE-AND-CONQUER technique can apply for almost all DP algorithms.
- In other words, once we have a DP algorithm to calculate optimal solution value, it is usually possible to construct the optimal solution within the same time and space bound.

Further improvement by J. Ian Munro

J. Ian Munro's algorithm [1982]

- Basic idea: Calculate q recursively
- Let's define $R_{i,j}$ the row number at which the optimal alignment of $T[1..i]$ and $S[1..j]$ passes by the column $\frac{n}{2}$. Then we have:

$$R_{i,j} = \begin{cases} i & \text{if } j = \lfloor \frac{n}{2} \rfloor \\ R_{i-1,j} & \text{if } j > \lfloor \frac{n}{2} \rfloor \text{ and } OPT[i,j] \text{ derived from } OPT[i-1,j] \\ R_{i-1,j-1} & \text{if } j > \lfloor \frac{n}{2} \rfloor \text{ and } OPT[i,j] \text{ derived from } OPT[i-1,j-1] \\ R_{i,j-1} & \text{if } j > \lfloor \frac{n}{2} \rfloor \text{ and } OPT[i,j] \text{ derived from } OPT[i,j-1] \end{cases}$$

An example: Step 1

	S: ''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

	S: ''	O	C	U	R	R	A	N	C	E
T:''	–	–	–	–	0	0	0	0	0	0
O	–	–	–	–	1	1	1	1	1	1
C	–	–	–	–	2	2	2	2	2	2
C	–	–	–	–	3	3	3	3	2	2
U	–	–	–	–	4	4	4	4	4	2
R	–	–	–	–	5	5	5	5	5	5
R	–	–	–	–	6	5	5	5	5	5
E	–	–	–	–	7	5	5	5	5	5
N	–	–	–	–	8	5	5	5	5	5
C	–	–	–	–	9	5	5	5	5	5
E	–	–	–	–	10	5	5	5	5	5

- We have $q = 5$.

An example: Step 2 and 3

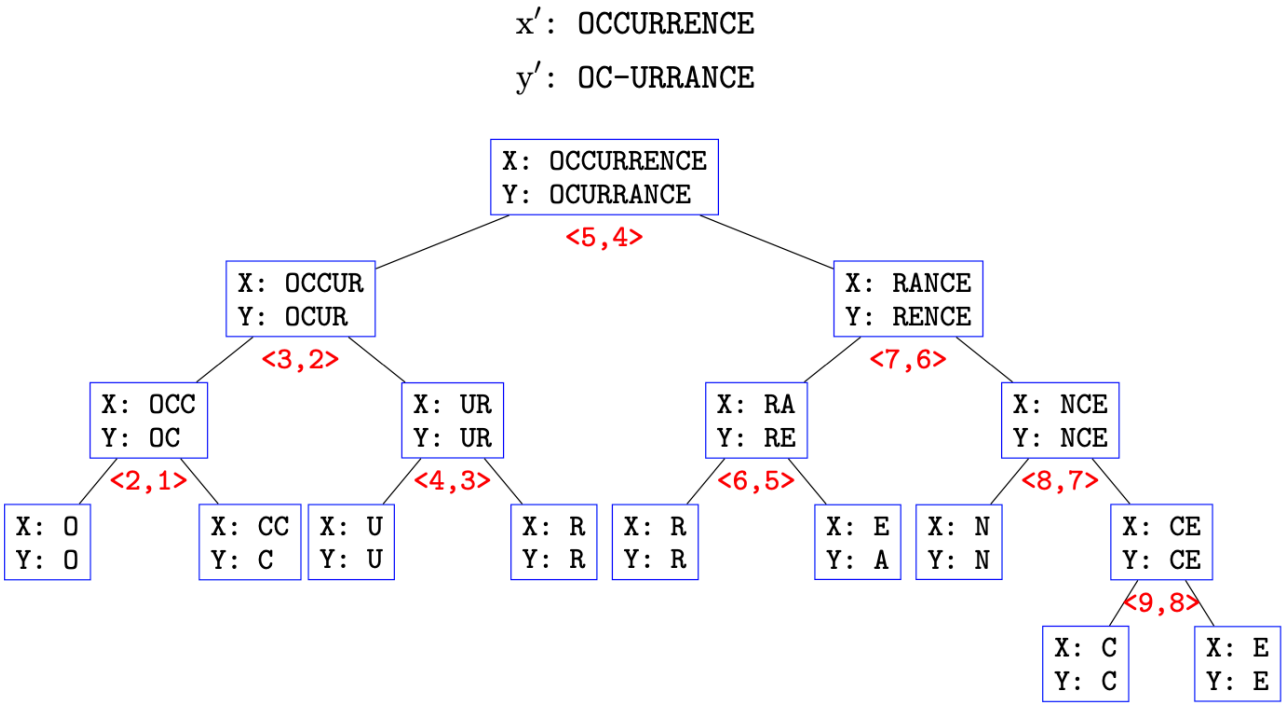
X: \ Y:	''	O	C	U	R
''	–	–	0	0	0
O	–	–	1	1	1
C	–	–	2	2	2
C	–	–	3	2	2
U	–	–	4	3	2, 3
R	–	–	5	3	3

X: \ Y:	''	O	C
''	–	0	0
O	–	1	1
C	–	2	1
C	–	3	1, 2

- We have $q = 3$, and $1, 2$, respectively.

Building sequence alignment

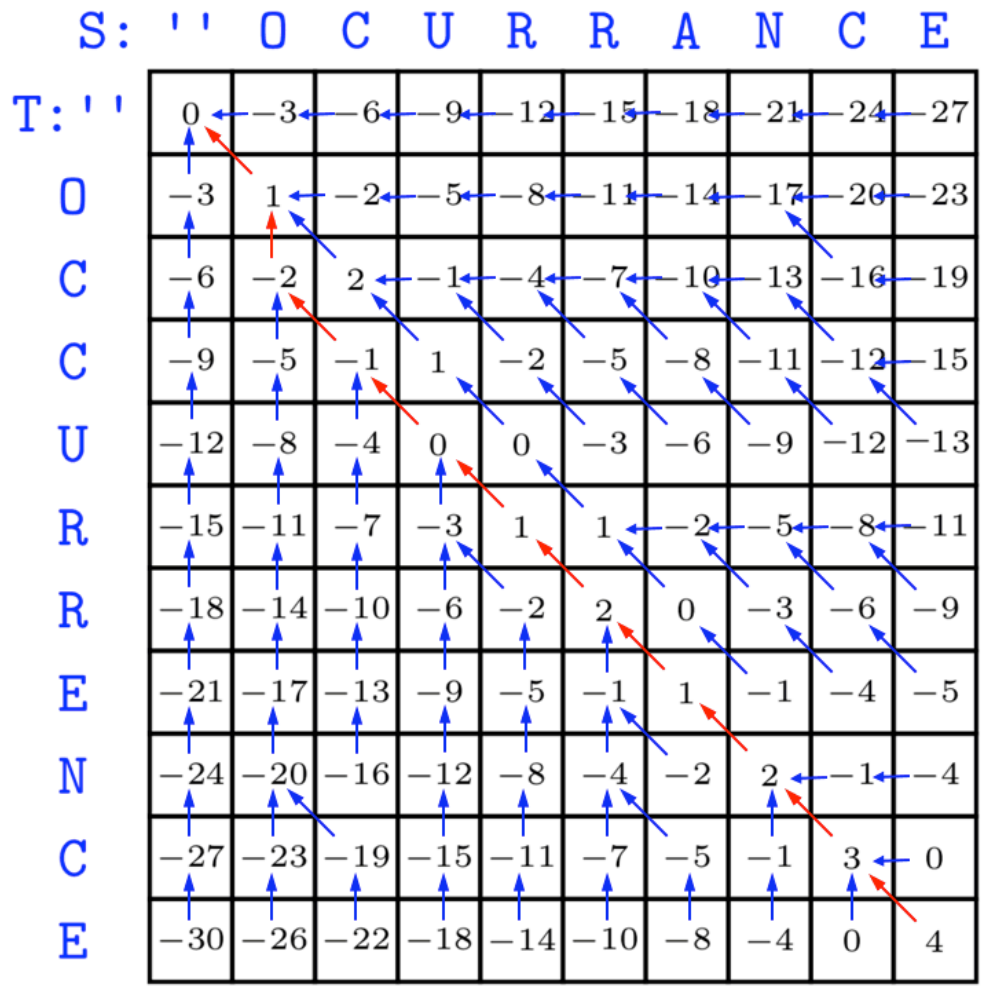
- We obtained A:
  A=[<5, 4>, <3, 2>, <1, 1>, <4,3>, <7,6>, <6,5>, <8,7>, <9,8>



Saving space requirement using neural network [Yang 2018]

Saving running time: Banded DP reduces time complexity from $O(mn)$ to $O(\alpha \max\{m, n\})$ under assumption of small number of `Indels`

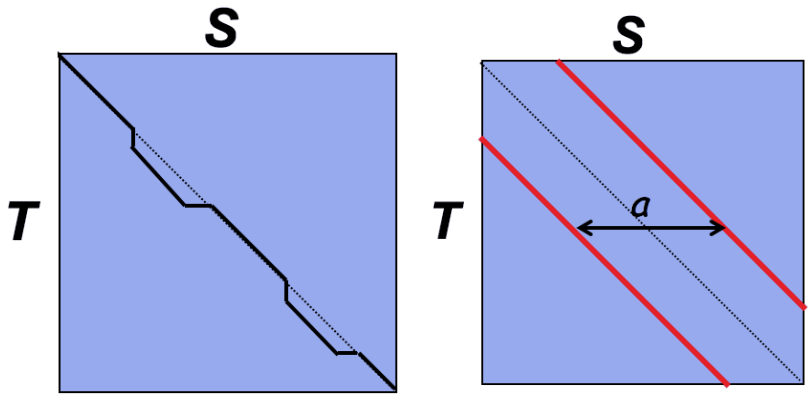
Let's examine an example first



T: \ S:	''	O	C	U	R	R	A	N	C	E
''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	−13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	0	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

- The optimal alignment of "OCURRANCE" and "OCCURRENCE" has only one gap, and most off-diagonal entries are small.

Motivation of banded DP



- Observation: When S and T are sufficiently similar, the optimal alignment of them will have few gaps and thus will be close to the diagonal.
- To find such alignment, it suffices to search within a diagonal region of the matrix. If the band has presumed width α, then the **banded DP will cost only** $O(\alpha \max\{m, n\})$, much faster than $O(mn)$ of the standard DP.

Banded DP [Fickett, 1984]

BANDED-DP(T, S, α)

1: **for** $i = 0$ to m **do**
2: $OPT[i, 0] = -3 * i$;
3: **end for**
4: **for** $j = 0$ to n **do**
5: $OPT[0, j] = -3 * j$;
6: **end for**
7: **for** each entry (i, j) within the α-wide band **do**
8: Set $L = OPT[i-1, j] - 3$ if $(i-1, j)$ within the α-wide band and $L = -\infty$ otherwise;
9: Set $U = OPT[i, j-1] - 3$ if $(i, j-1)$ within the α-wide band and $U = -\infty$ otherwise;
10: $OPT[i, j] = \max\{OPT[i-1, j-1] + s(S_i, T_j), L, U\}$;
11: **end for**
12: **return** $OPT[m, n]$;

An example: set $\alpha = 4$

	S: ''	O	C	U	R	R	A	N	C	E
T:''	0	−3	−6							
O	−3	1	−2	−5						
C	−6	−2	2	−1	−4					
C	−9	−5	−1	1	−2	−5				
U		−8	−4	0	0	−3	−6			
R			−7	−3	1	1	−2	−5		
R				−6	−2	2	0	−3	−6	
E					−5	−1	1	−1	−4	−5
N						−4	−2	2	−1	−4
C							−5	−1	3	0
E								−4	0	4

Saving running time: Sparse DP explores the sparsity in memoization table

Longest Common Subsequence problem

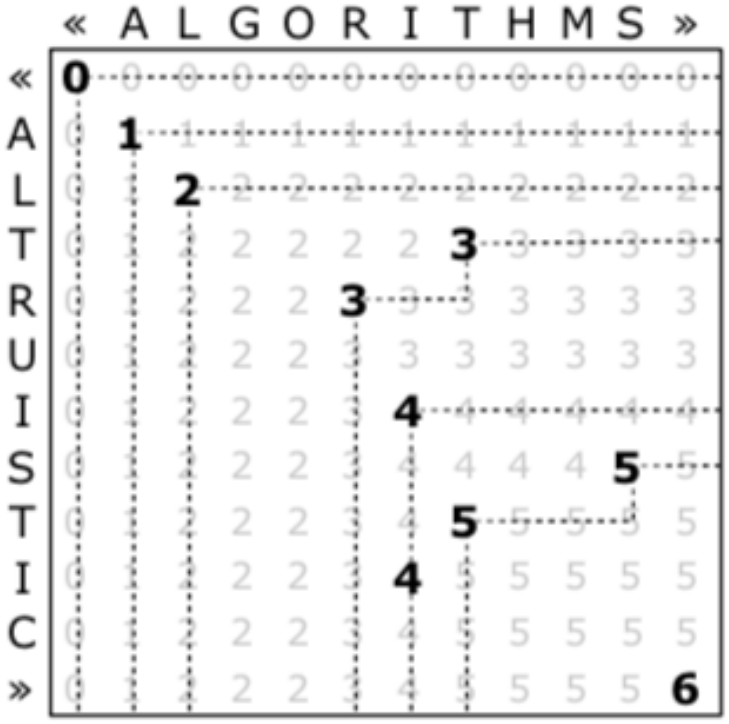
- The alignment of two sequences $S[1..m]$ and $T[1..n]$ reduces to finding the Longest Common Subsequence of them when mutations are not allowed.
- Solution: the longest common subsequence of S and T. Let's describe the solving process as multi-stage decision-making process. At each stage, we decide whether align a letter $S[i]$ with $T[j]$ or not.
- Let's consider the first decision, i.e., whether we align $S[m]$ with $T[n]$ or not. We have two options: :
  - Align $S[m]$ and $T[n]$ (when $S[m] = T[n]$): Then the subproblem is how to find the longest common subsequence of $S[1..m-1]$ and $T[1..n-1]$.
  - Do not align them: Then we have two subproblems: find the longest common subsequence of $S[1..m]$ and $T[1..n-1]$, and find the longest common subsequence of $S[1..m-1]$ and $T[1..n]$.

Basic DP algorithm

- Summarizing these examples, we determine the general form of subproblems as: finding the longest common subsequences of $S[1..i]$ and $T[1..j]$ and denote the optimal value as $OPT(i, j)$.
- Then we have the following recursion:

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ OPT(i-1, j-1) + 1 & \text{if } S[i] = T[j] \\ \max\{OPT(i-1, j), OPT(i, j-1)\} & \text{otherwise} \end{cases}$$

An example



- The memoization table contains many cells with identical value, implying that most subproblems will be solved in the same way.
- It suffices to calculate cells at limited **matching points**. The entire memoization table could be constructed using these cells.

- We simplified the recursion

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ OPT(i-1, j-1) + 1 & \text{if } S[i] = T[j] \\ \max\{OPT(i-1, j), OPT(i, j-1)\} & \text{otherwise} \end{cases}$$

- into

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = \\ \max\{OPT(i', j') | S[i'] = T[j'], i' < i, j' < j\} + 1 & \text{if } S[\\ \max\{OPT(i', j') | S[i'] = T[j'], i' \leq i, j' \leq j\} & \text{othe} \end{cases}$$

Sparse DP: Algorithm

function SPARSELCS($S[1..m]$, $T[1..n]$)

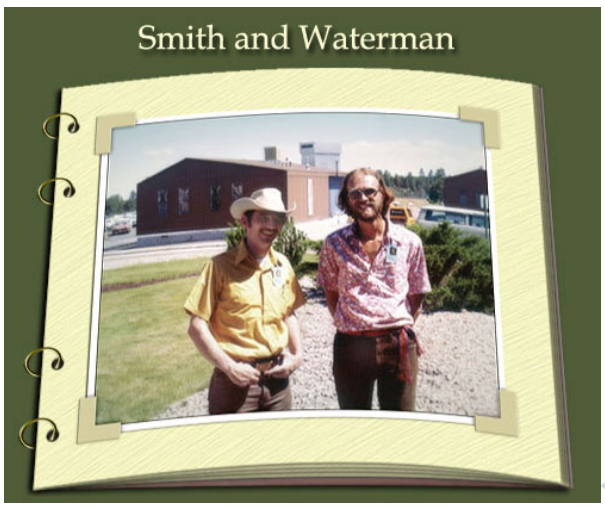
1: $Matches[1..K]$ =FINDMATCHES(S, T);
2: $Matches[K+1] = (m+1, n+1)$;
3: Sort $Matches$ lexicographically;
4: **for** $k = 1$ to K **do**
5: $(i, j) = Matches[k]$;
6: $OPT[k] = 1$;
7: **for** $l = 1$ to $k-1$ **do**
8: $(i', j') = Matches[l]$;
9: **if** $i' < i$ and $j' < j$ **then**
10: $OPT[(k] = \max\{OPT[k], 1 + OPT[l]\}$;
11: **end if**
12: **end for**
13: **end for**
14: **return** $OPT[K+1] - 1$;

- Time-complexity: $O(m \log m + n \log n + K^2)$

Extended Reading 1: From global alignment to local alignment

From global alignment to local alignment: Smith-Waterman algorithm

- Global alignment: to identify similarity between **two entire sequences**.
- Local alignment: It is often that we wish to find **similar segments (sub-sequences)**.
- Needleman-Wunsch **global alignment** algorithm was developed by biologists in 1970s, about twenty years later than Bellman-Ford algorithm was developed. Then Smith-Waterman **local alignment** algorithm was proposed (Please refer to Smith and Waterman1981 for details.).



Local alignment vs. global alignment: two difference

- The objective of local alignment is to identify similar segments of two sequence. The other regions can be treated as independent and thus form "random matches". To distinguish random matches and true matches, the scoring schema was designed to assign random matches with negative expected score, i.e.,

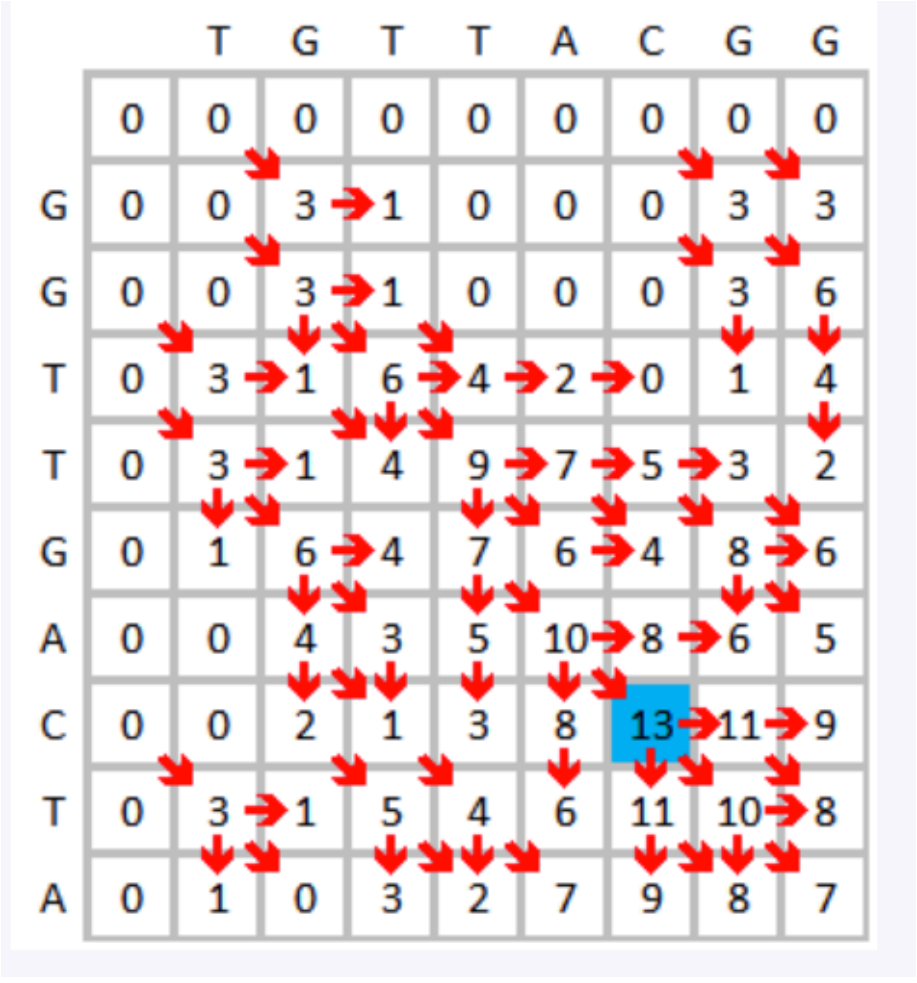
$$\sum_{a,b} q_a q_b s(a,b) < 0$$

- The recursion was changed by adding an extra possibility:

$$OPT(i,j) = \max \begin{cases} \mathbf{0} \\ s(T_i, S_j) + OPT(i-1, j-1) \\ s(T_i, \text{`\_'}, ) + OPT(i-1, j) \\ s(\text{`\_'}, S_j) + OPT(i, j-1) \end{cases}$$

- Taking the option 0 corresponds to starting a new alignment: if we obtain a negative score for $OPT(i,j)$, this means that the subsequences $T[1..i]$ and $S[1..j]$ are independent. Thus, it is better to start a new alignment rather than extend the old one.

An example



		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

- Note that the consequence of the 0 option is that the top row and left column are set as 0.

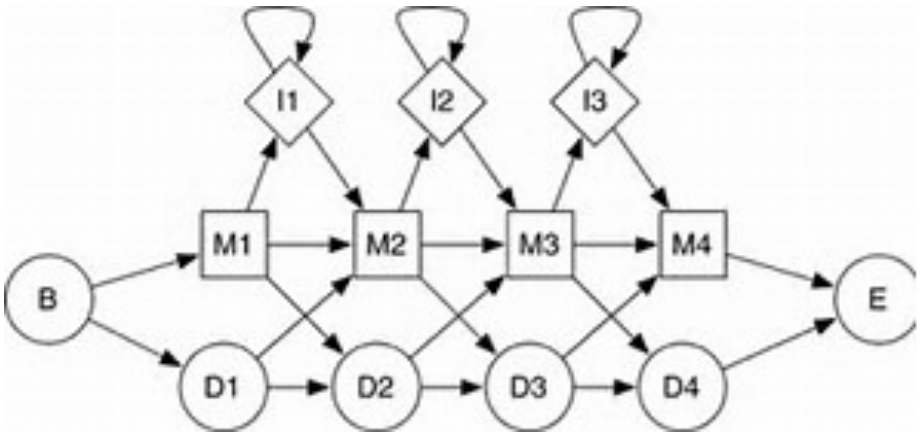
Local alignment vs. global alignment: two difference

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

- As the local alignment aims to represent similar segments, it can end at any cell in the matrix. Thus, instead of taking the bottom-right corner in global alignment, we look for the highest value in the matrix and start backtracking from there.
- The backtrack ends when we meets a cell with value 0, which corresponds to the start of an alignment.

Extended Reading 2: How to derive a reasonable scoring schema?

Profile-HMM: a generative model of multiple-sequence alignment



PAM250: one of the most popular substitution matrices in Bioinformatics

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-1	-1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1

Please refer to "PAM matrix for Blast algorithm" (by C. Alexander, 2002) for the details to calculate PAM matrix.

Extended Reading 3: How to measure the significance of an alignment?

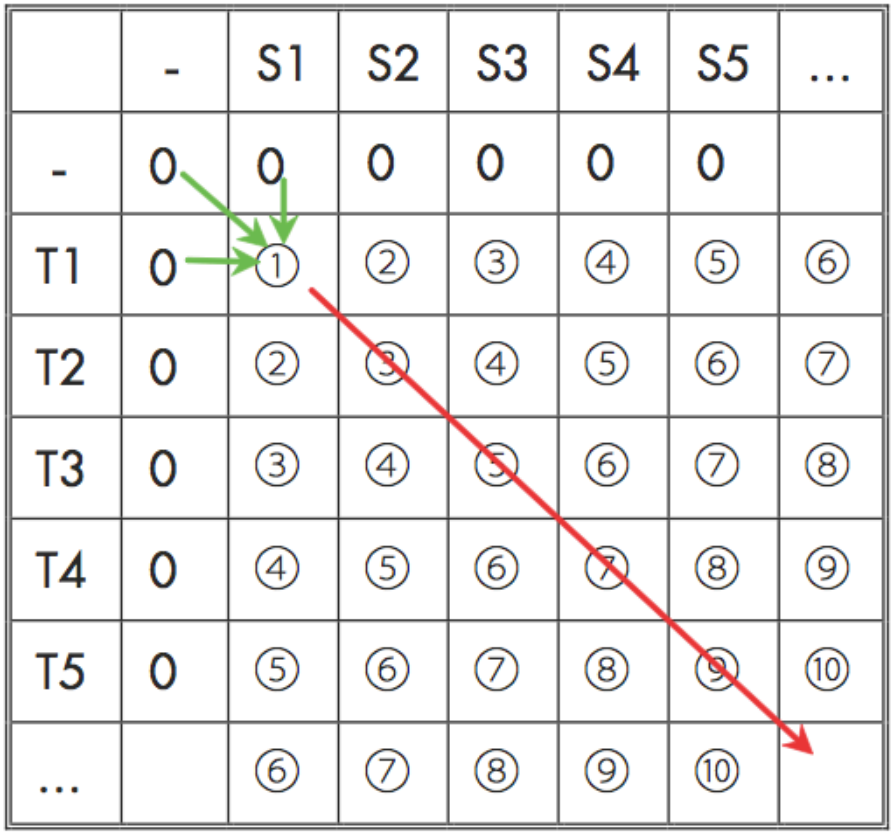
Measure the significance of a segment pair

- When two random sequences of length m and n are compared, the probability of finding a pair of segments with a score greater than or equal to S is $1 - e^{-y}$, where $y = Kmne^{-\lambda S}$.

Please refer to Altschul1990 for details.

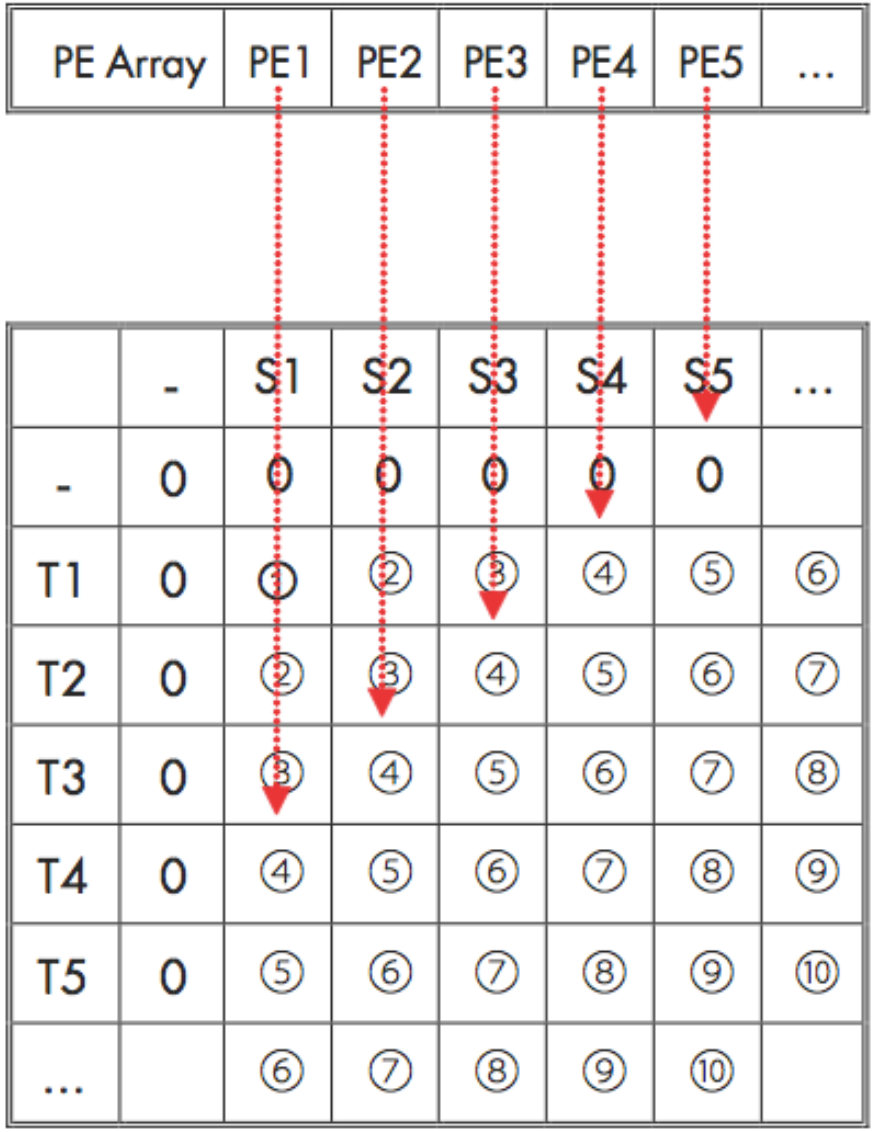
Extended Reading 4: An FPGA implementation of Smith-Waterman algorithm

The potential parallelity of SmithWaterman algorithm



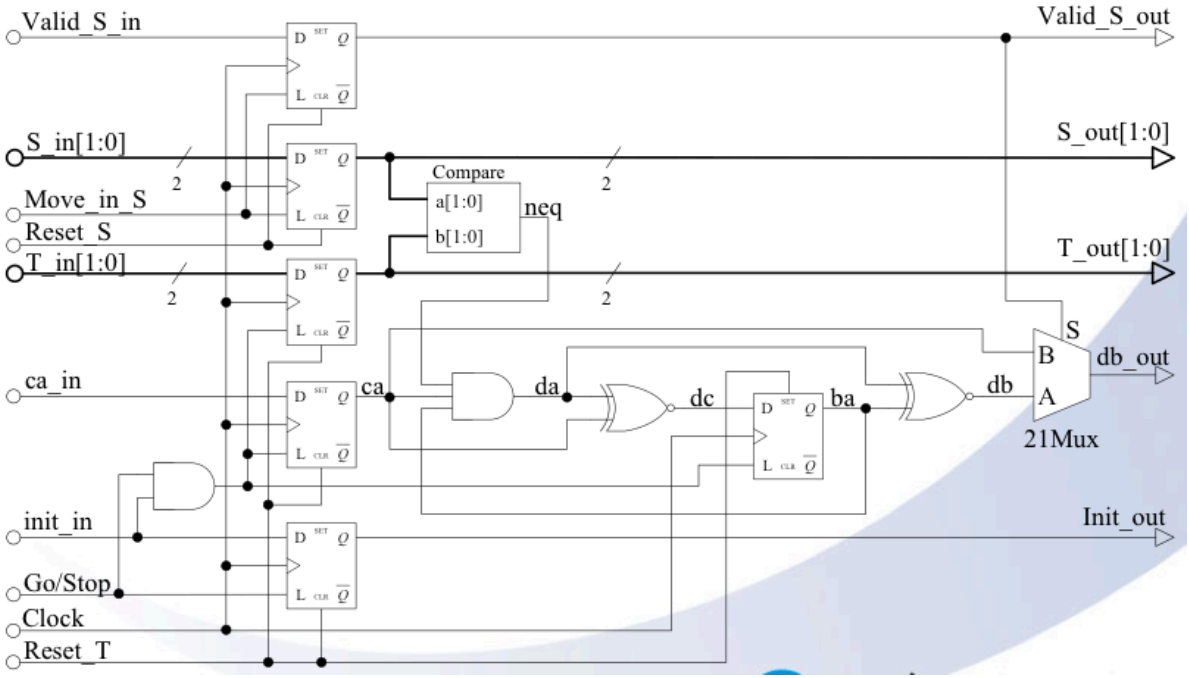
For example, in the first cycle, only one element marked as (1) could be calculated. In the second cycle, two elements marked as (2) could be calculated. In the third cycle, three elements marked as (3) could be calculated, etc., and this feature implies that the algorithm has a very good potential parallelity.

Mapping Smithg-Waterman algorithm on PE



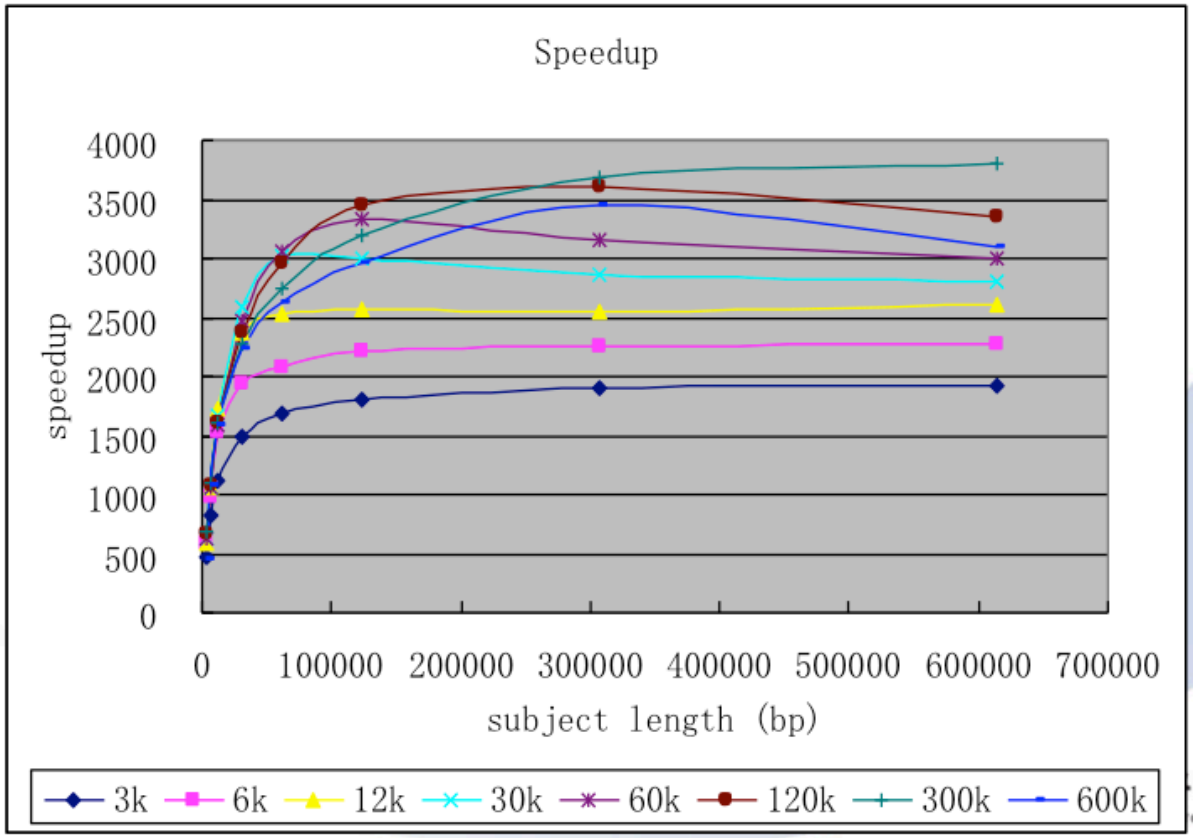
See *Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform* for details.

PE design of a card for Dawning 4000L



Smith-Waterman card for Dawning 4000L

Performance of Dawning 4000L

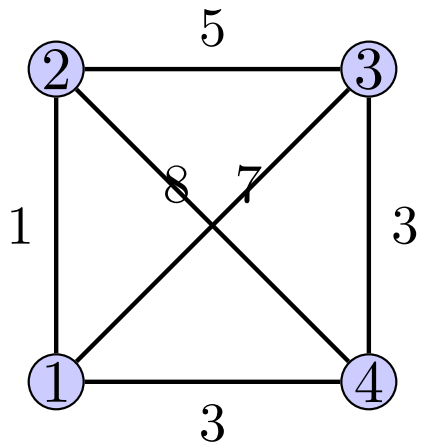


2

[2]Some pictures were excerpted from *Introduction to algorithms*

Bellman-Held-Karp algorithm for TSP problem: recursion over **graphs**

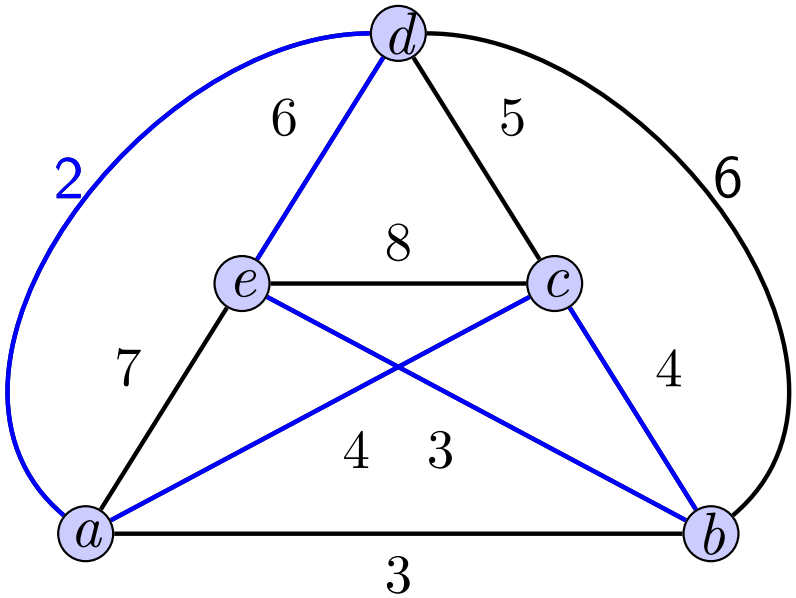
INPUT: a list of n cities (denoted as V), and the distances between each pair of cities d_{ij} $(1 \leq i, j \leq n)$;
OUTPUT: the shortest tour that visits each city exactly once and returns to the origin city

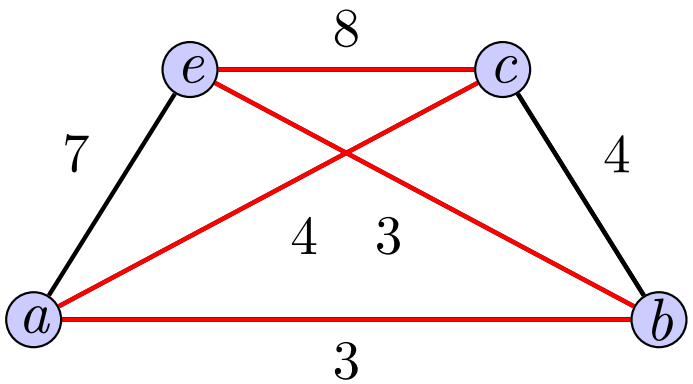


#Tours: 6

- Tour 1: $1 \to 2 \to 3 \to 4 \to 1$ (12)
- Tour 2: $1 \to 2 \to 4 \to 3 \to 1$ (21)
- Tour 3: $1 \to 3 \to 2 \to 4 \to 1$ (23)
- ....

Decompose the original problem into subproblems

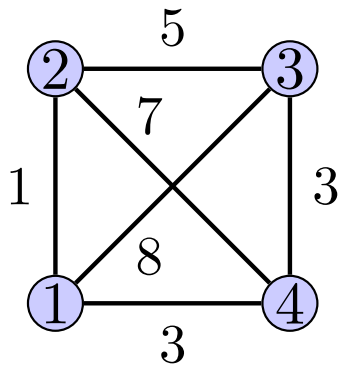




- Note that it is not easy to obtain the optimal solution to the original problem (e.g., tour in blue) through the optimal solution to subproblem (e.g., tour in red).

Consider a tightly-related problem

- Let's consider a tightly-related problem: calculating $M(s, S, e)$, the minimum distance, starting from city s, visiting each city in S once and exactly once, and ending at city e.
- It is easy to decompose this problem into subproblems, and the original problem could be easily solved if $M(s, S, e)$ were determined for all subset $S \subseteq V$ and $e \in V$.

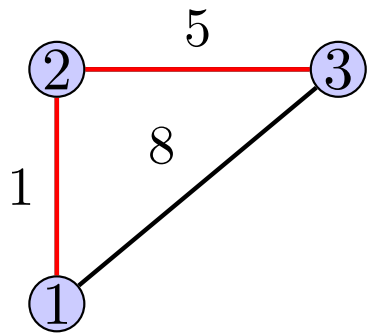


- For example, since there are 3 cases of the city from which we return to 1, the shortest tour can be calculated as:

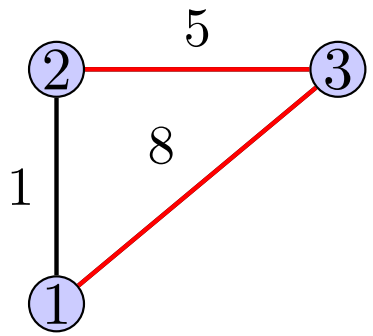
$$\begin{aligned}\min\{ \quad & d_{2,1} + M(1, \{3, 4\}, 2), \\ & d_{3,1} + M(1, \{2, 4\}, 3), \\ & d_{4,1} + M(1, \{2, 3\}, 4)\}\end{aligned}$$

Consider the smallest instance of $M(s, S, e)$

- It is trivial to calculate $M(s, S, e)$ when S consists of only 1 city.



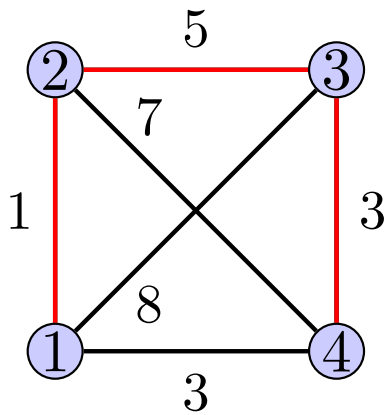
$M(1, \{2\}, 3) = d_{12} + d_{23}$

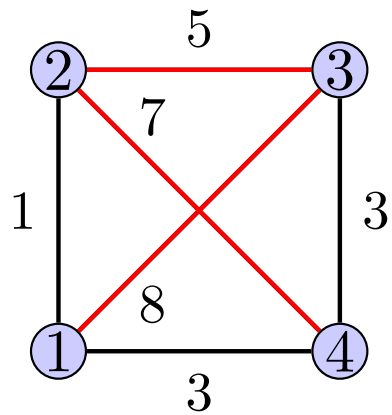


$M(1, \{3\}, 2) = d_{13} + d_{32}$

- But how to solve a larger problem, say $M(1, \{2, 3\}, 4)$?

Divide a large problem into smaller problems





- $M(1, \{2, 3\}, 4) = \min\{ d_{34} + M(1, \{2\}, 3), d_{24} + M(1, \{3\}, 2)\}$

Bellman-Held-Karp algorithm [1962]

function $\mathrm{TSP}(D)$

1: **return** $\min_{e \in V, e \neq s} M(s, V - \{e\}, e) + d_{es}$;

function $\mathrm{M}(s, S, e)$

1: **if** $S = \{v\}$ **then**
2: $\quad \mathrm{M}(s, S, e) = d_{sv} + d_{ve}$;
3: $\quad$ **return** $\mathrm{M}(s, S, e)$;
4: **end if**
5: **return** $\min_{i \in S,\ i \neq e} \mathrm{M}(s, S - \{i\}, i) + d_{ei}$;

An example

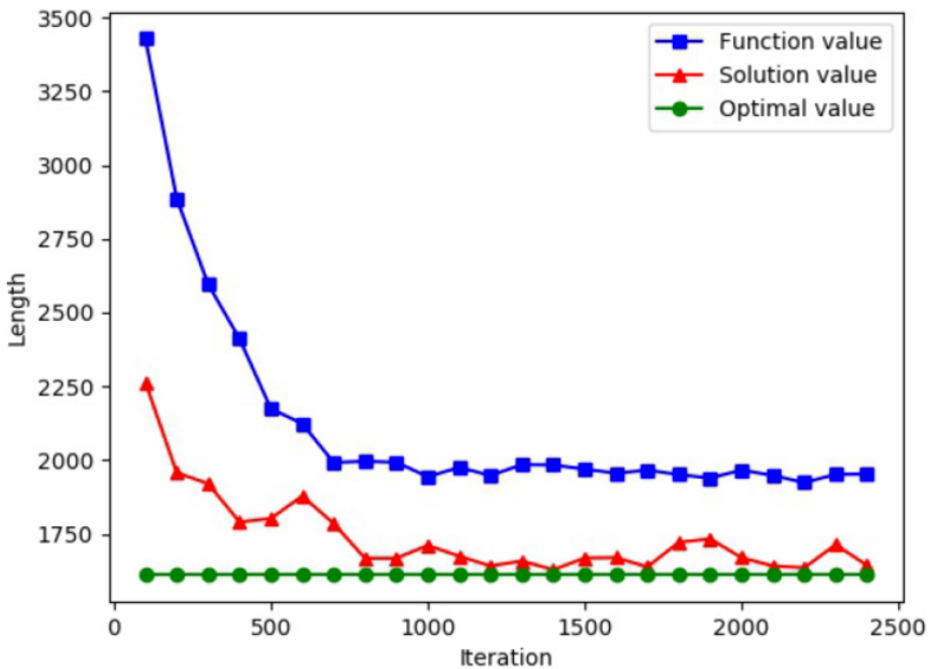
	{b}	{c}	{d}	{e}	{b, c}	{b, d}	{b, e}	{c, d}	{c, e}	{d, e}	{b, c ,d}	{b, c, e}	{b, d, e}	{d, c, e}
b	-	8	8	10	-	-	-	11	15	11	-	-	-	18
c	7	-	7	15	-	12	14	-	-	16	-	-	15	-
d	9	9	-	13	12	-	12	-	18	-	-	17	-	-
e	6	12	8	-	11	11	-	15	-	-	14	-	-	-

- Space complexity: $\sum_{k=2}^{n-1} k\binom{n-1}{k} + n - 1 = (n-1)2^{n-2}$
- Time complexity: $\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k} + n - 1 = O(2^n n^2)$.
- In 2018, Feidiao Yang et al. proposed to use DNN to learn the DP function.

Learn DP function using DNN [Yang, 2019]

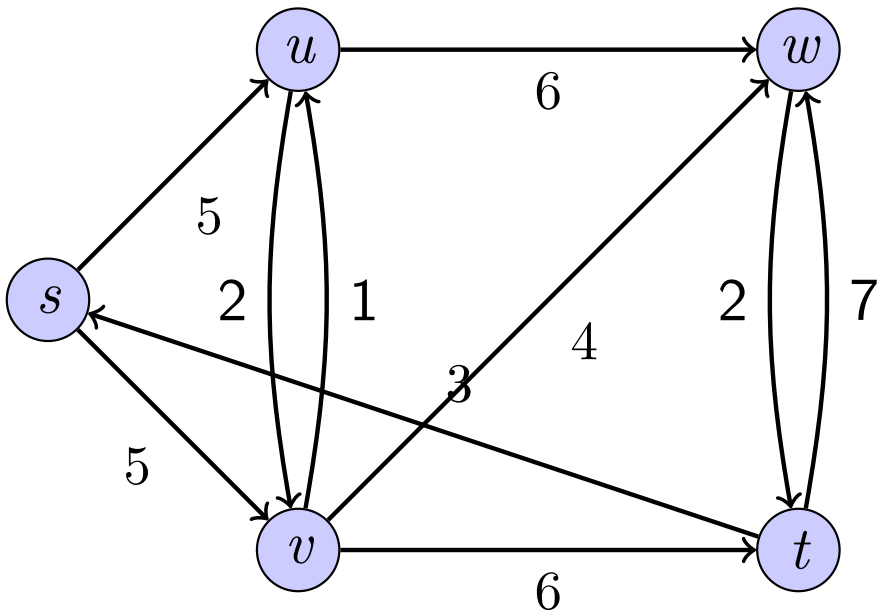
Table: Experimental results in TSPLIB[1]

Data	Opt.	NNDP		Held-Karp		Christofides		Greedy	
Gr17	2085	**2085**	**1**	**2085**	**1**	2287	1.1607	2178	1.0446
Bayg29	1610	**1610**	**1**	NA	NA	1737	1.0789	1935	1.2019
Dantzig42	699	**709**	**1.0143**	NA	NA	966	1.382	863	1.2346
HK48	11461	**11539**	**1.0068**	NA	NA	13182	1.1502	12137	1.059
Att48	10628	**10868**	**1.0226**	NA	NA	15321	1.4416	12012	1.1302
Eil76	538	**585**	**1.0874**	NA	NA	651	1.1128	598	1.1115
Rat99	1211	**1409**	**1.1635**	NA	NA	1665	1.3749	1443	1.1916
Br17	39	**39**	**1**	**39**	**1**	NA	NA	56	1.435
Ftv33	1286	**1324**	**1.0295**	NA	NA	NA	NA	1589	1.2002
Ft53	6905	**7343**	**1.0634**	NA	NA	NA	NA	8584	1.169

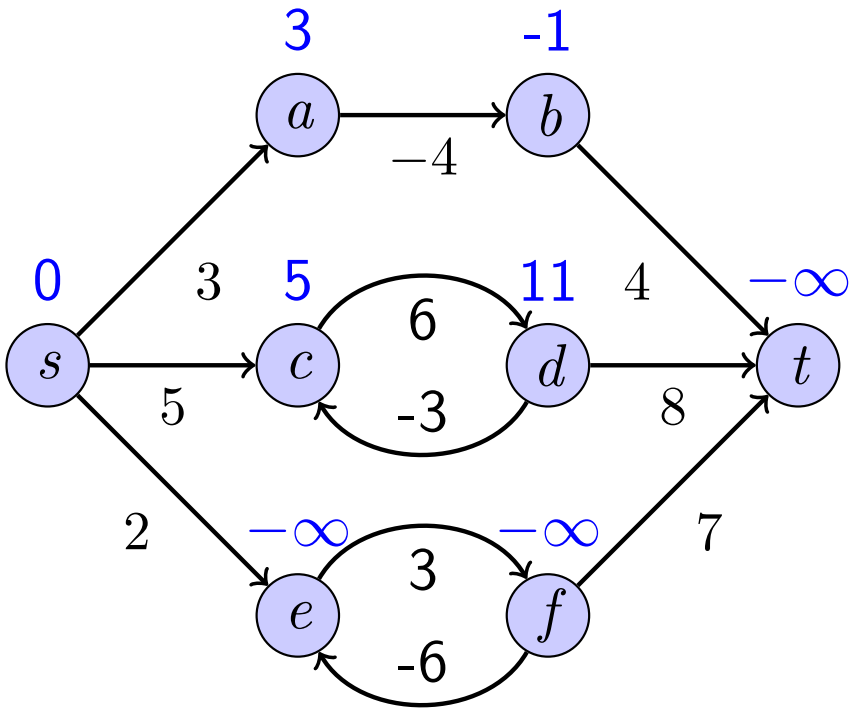


SingleSourceShortestPath problem: recursion over **graphs**

INPUT:
A directed graph $G =< V, E >$. Each edge $e = (u, v)$ has a weight or distance $d(u, v)$. Two special nodes: source s, and destination t;
OUTPUT:
A shortest path from s to t; that is, the sum weight of the edges is minimized.

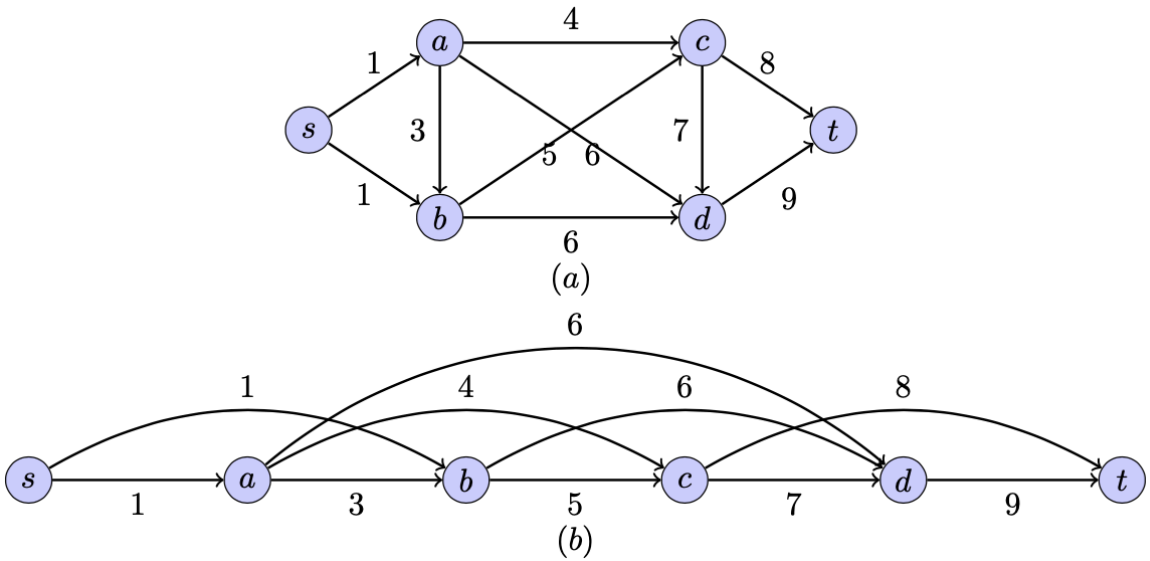


ShortestPath problem: cycles



- Here $d(i, j)$ might be negative; however, there should be **no negative cycle**, i.e. the sum weight of edges in any cycle should be greater than 0.
- In fact, a negative cycle means an $-\infty$ shortest-path weight. Since e and f form a negative-weight cycle reachable from s, they have shortest-path weight of $-\infty$ from s.

Let's start from a simpler case: DAG



- DAG could be linearized to put all nodes into an array.

Multi-stage decision process

- Solution: the shortest path from node s to t is a path. Let's describe the solving process as a process of multistage **decisions**; at each decision stage, we decide **the subsequent node** from current node.
- Let's consider the first decision in the optimal solution O. The feasible options are:
  - All adjacent nodes of s: Suppose we choose an edge (s, v) to node v. Then the left-over is to find the shortest path from v to t.
- Thus the general form of subproblem can be designed as: to find the shortest path from node v to t. Denote the optimal solution value as $OPT(v)$.
- Optimal substructure: $OPT[v] = \min_{(v,w)\in E}\{OPT[w] + d(v, w)\}$

Algorithm

Algorithm 33 Recursive algorithm for calculating shortest-paths in DAG

function DAG-SHORTEST-PATHS(G, s)

1: L=TOPOLOGICAL-SORT(G);
2: Set $d(v) = \infty$ for each node $v \in L$, and $d(s) = 0$;
3: **for** $i = 1$ to $|L|$ **do**
4: $\quad v = L[i]$;
5: $\quad d(v) = \min_{(u,v)\in E}\{d(u) + w(u, v)\}$;
6: $\quad \pi(v) = \operatorname{argmin}_{(u,v)\in E}\{d(u) + w(u, v)\}$;
7: **end for**

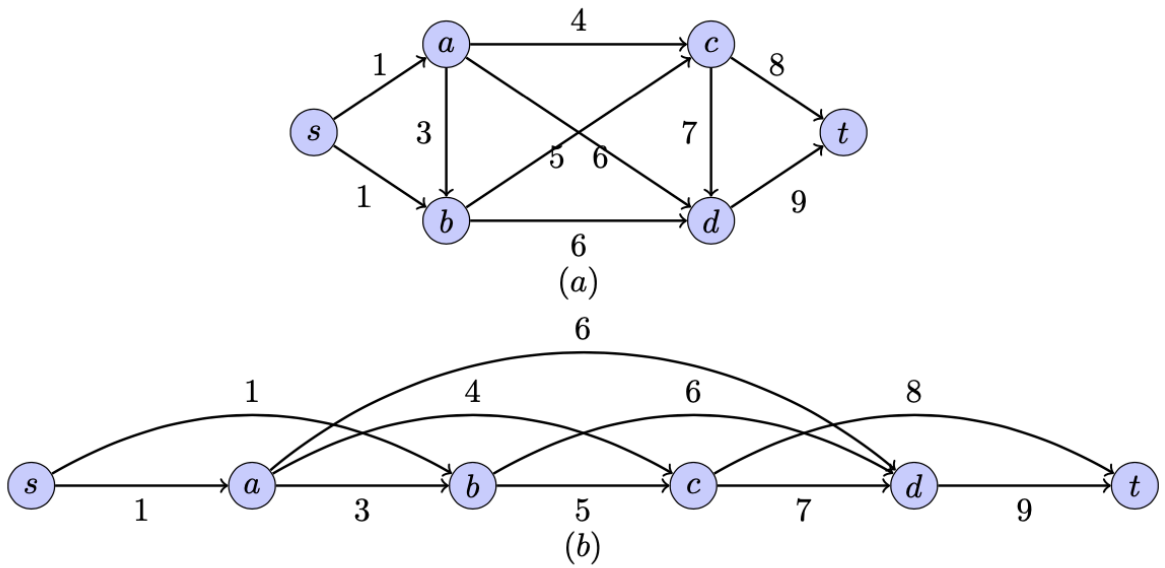
function TOPOLOGICAL-SORT(G)

1: Set L as an empty list, and set $color(u) =$ `WHITE` for each node $u \in V(G)$;
2: **for** each node $u \in V(G)$ **do**
3: $\quad$ Run DFS-VISIT(u) if $color(u) \neq$ `BLACK`;
4: **end for**
5: **return** L;

function DFS-VISIT(u)

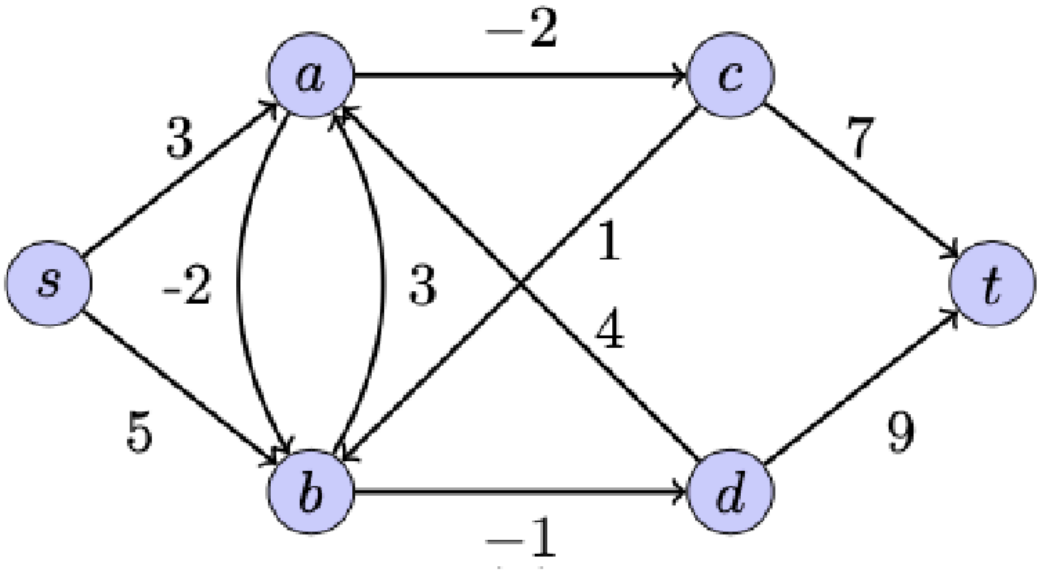
1: Report "Found a cycle" and exit if $color(u) ==$ `GRAY`;
2: **for** each edge $(u, v) \in E(G)$ **do**
3: $\quad$ Run DFS-VISIT(u) if $color(u) \neq$ `BLACK`;
4: **end for**
5: $color(u) =$ `BLACK`;
6: Add u to the head of L;

An example



	s	a	b	c	d	t
$d(v)$	0	1	1	5	7	13
$\pi(v)$	-	s	s	a	b	c

Difficulty when cycles exists



$$d(a) = \min\{3, 3 + d(b)\}$$

$$d(b) = \min\{5, -2 + d(a)\}$$

Define distance by limiting path length

- Solution: the shortest path from node s to t is a path with **at most n nodes** (Why? no negative cycle $\Rightarrow$ removing cycles in a path can shorten the path). Let's describe the solving process as a process of multistage **decisions**; at each decision stage, we decide **the subsequent node** from current node.
- Let's consider the first decision in the optimal solution O. The feasible options are:
  - All adjacent nodes of s: Suppose we choose an edge (s, v) to node v. Then the left-over is to find the shortest path from v to t via at most $n-2$ edges.
- Thus the general form of subproblem can be designed as: to find the shortest path from node v to t with **at most** k edges ($k \leq n-1$). Denote the optimal solution value as $OPT(v, k)$.
- Optimal substructure:

$$OPT[v,k] = \min \begin{cases} OPT[v, k-1], \\ \min_{(v,w)\in E}\{OPT[w, k-1] + d(v, w)\} \end{cases}$$

- Note: the first item $OPT(v, k-1)$ is introduced to express "**at most**".

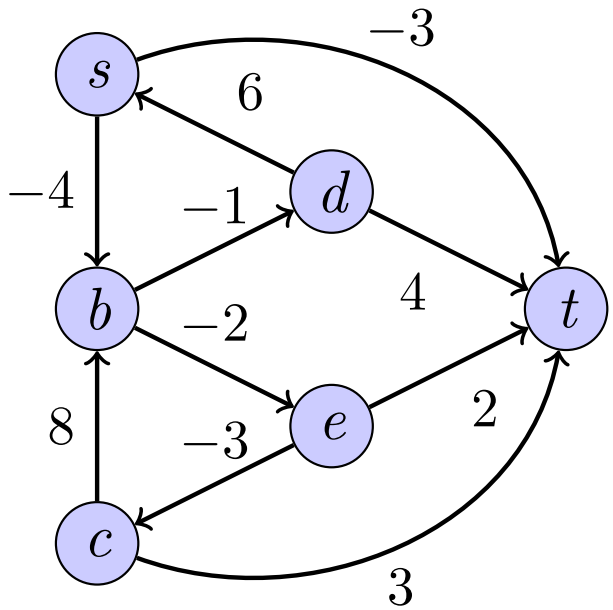
Bellman-Ford algorithm [1956, 1958]

BELLMAN-FORD(G, s, t)

1: **for** each node $v \in V$ **do**
2: $\quad OPT[v, 0] = \infty$;
3: **end for**
4: **for** $k = 0$ to $n - 1$ **do**
5: $\quad OPT[t, k] = 0$;
6: **end for**
7: **for** $k = 1$ to $n - 1$ **do**
8: $\quad$ **for all** node v (in an arbitrary order) **do**
9: $\quad\quad OPT[v, k] = \min \begin{cases} OPT[v, k-1] \\ \min_{(v,w) \in E}\{OPT[w, k-1] + d(v, w)\} \end{cases}$
10: $\quad$ **end for**
11: **end for**
12: **return** $OPT[s, n - 1]$;

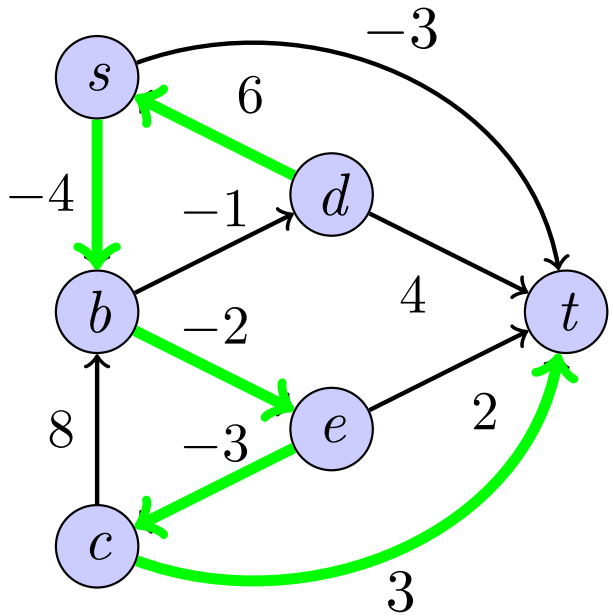
Note that the algorithm actually finds the shortest path from every possible source to t (or from s to every possible destination).

An example



Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
s	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

Shortest path tree



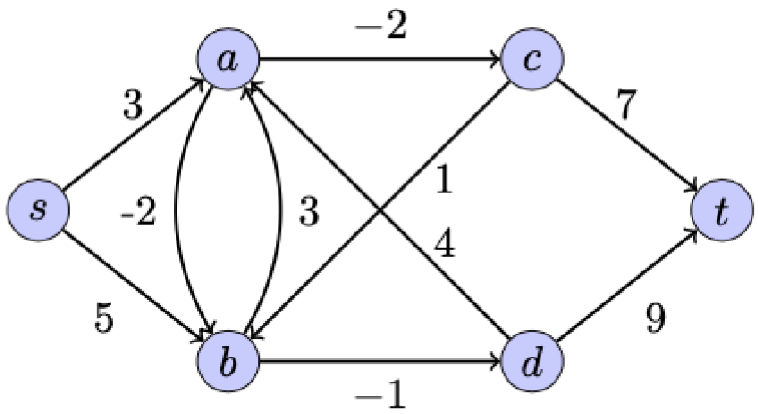
Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
s	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

Note: the shortest paths from all nodes to t form a *shortest path tree*.

Time complexity

1. Cursory analysis: $O(n^3)$. (There are n^2 subproblems, and for each subproblem, we need at most $O(n)$ operations in line 7.
2. Better analysis: $O(mn)$. (Efficient for sparse graph, i.e. $m << n^2$.)
   - For each node v, line 7 need $O(d_v)$ operations, where d_v denotes the degree of node v;
   - Thus **the inner** *for* **loop** (lines 6-8) needs $\sum_v d_v = O(m)$ operations;
   - Thus **the outer** *for* **loop** (lines 5-9) needs $O(nm)$ operations.

Solve the cycling recursion: from $d(v)$ to $d(v,k)$



$$d(a) = \min\{3, 3 + d(b)\}$$

$$d(b) = \min\{5, -2 + d(a)\}$$

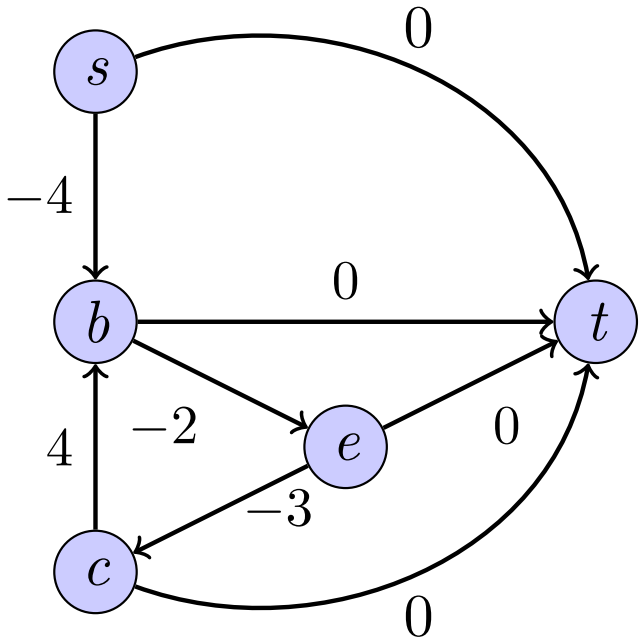
$$d(a,2) = \min\{d(a,1), 3 + d(b,1), 4 + d(e,1)\}$$

$$d(b,2) = \min\{d(b,1), -2 + d(a,1), 1 + d(c,1)\}$$

Extension: detecting negative cycle

Theorem

If t is reachable from node v, and v is contained in a negative cycle, then we have: $\lim_{k\to\infty} OPT(v,k) = -\infty$.



Intuition: a traveling of the negative cycle leads to a shorter length. Say,

$length(b \to t) = 0$

$length(b \to e \to c \to b \to t) = -1$

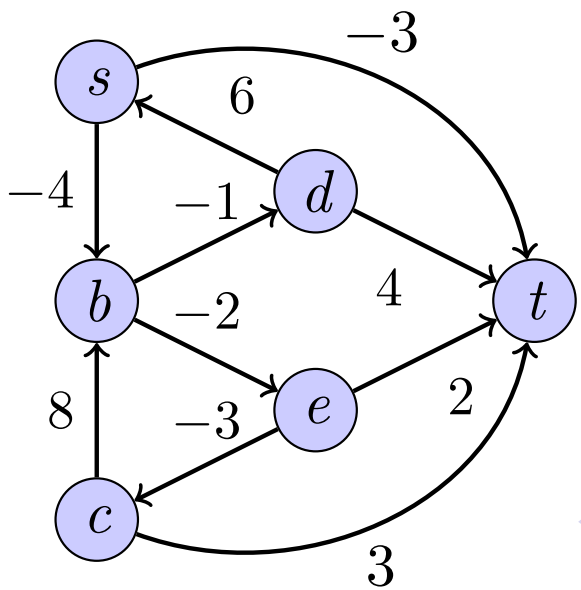
$length(b \to e \to c \to b \to e \to c \to b \to t) = -2$

$\cdots\cdots$

Corollary

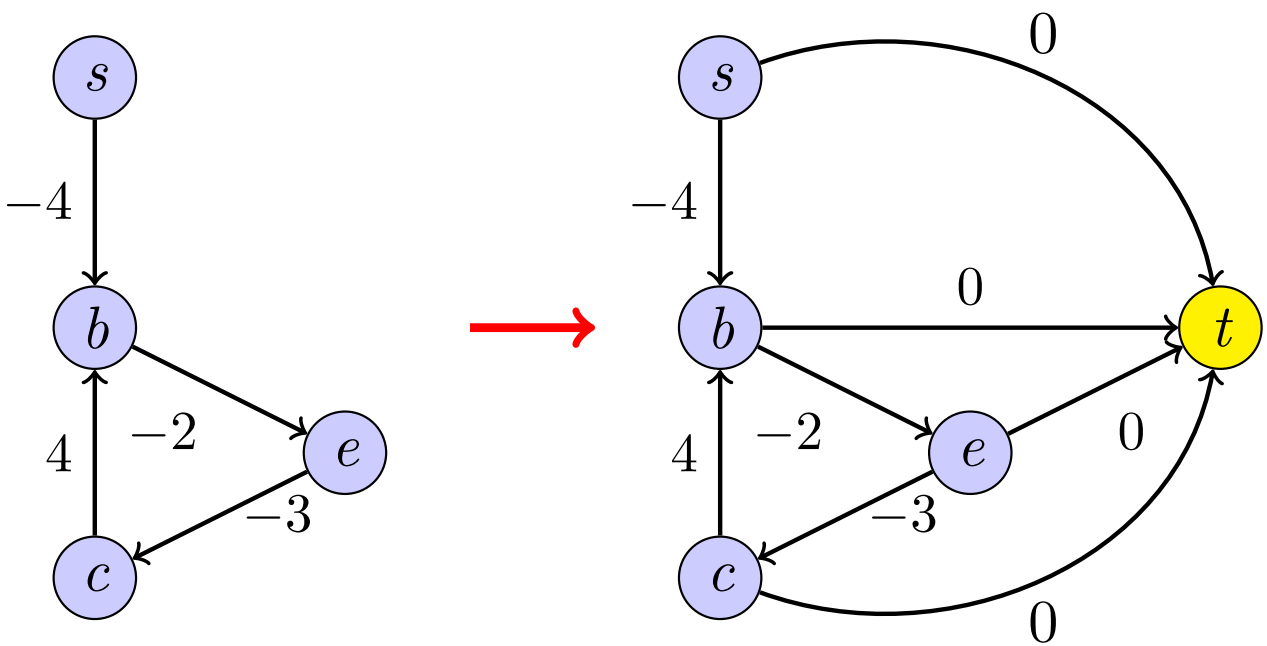
If there is no negative cycle in G, then for all node v, and $k \geq n$, $OPT(v, k) = OPT(v, n)$.

Source node	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9
t	0	0	0	0	0	0	0	0	0	0
s	-	-3	-3	-4	-6	-6	-6	-6	-6	-6
b	-	-	0	-2	-2	-2	-2	-2	-2	-2
c	-	3	3	3	3	3	3	3	3	3
d	-	4	3	3	2	0	0	0	0	0
e	-	2	0	0	0	0	0	0	0	0

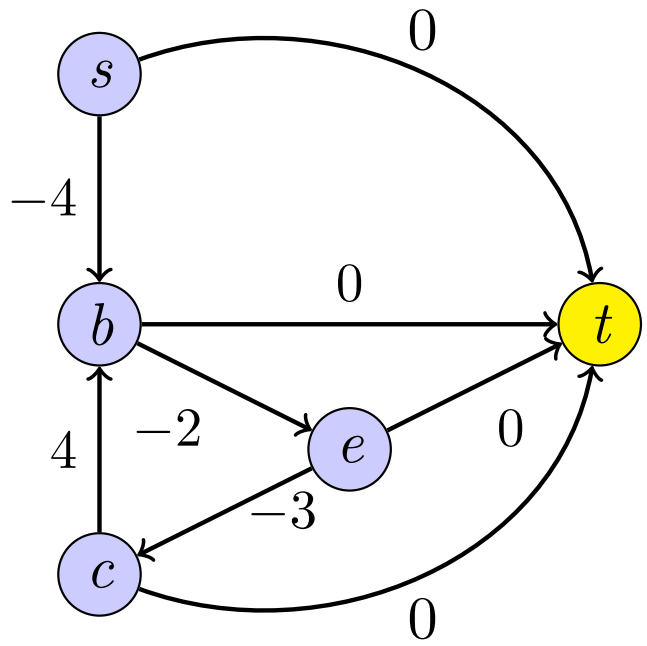


Detecting negative cycle via adding edges and a node t

- Expanding G to G' to guarantee that t is reachable from the negative cycle:
  1. Adding a new node t;
  2. For each node v, adding a new edge (v, t) with $d(v, t) = 0$;
- Property: G has a negative cycle C (say, $b \to e \to c \to b$) $\Rightarrow$ t is reachable from a node in C. Thus, the above theorem applies.



An example of negative cycle



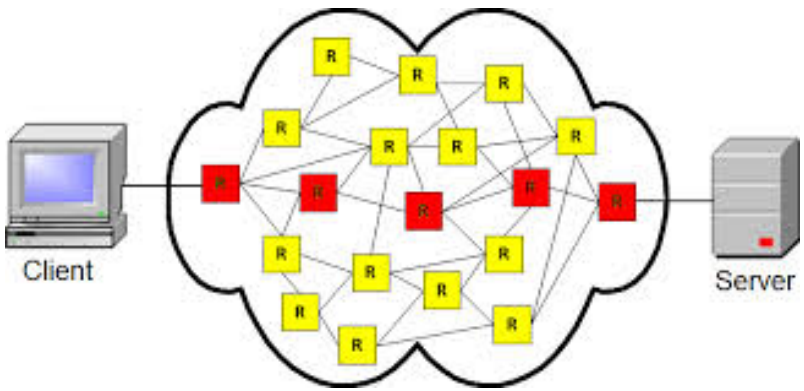
Source node	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	···
t	0	0	0	0	0	0	0	0	0	···
s	-	0	-4	-6	-9	-9	-11	-11	-12	···
b	-	0	-2	-5	-5	-7	-7	-8	-8	···
c	-	0	0	0	-2	-3	-3	-3	-4	···
e	-	0	-3	-3	-5	-5	-6	-6	-6	···

Application of Bellman-Ford algorithm: Internet router protocol

Internet router protocol

Problem statement:

- Each node denotes a route, and the weight denotes the **transmission delay** of the link from router i to j.
- The objective to design a protocol to determine the quickest route when router s wants to send a package to t.



Internet router protocol: Dijkstra's algo vs. Bellman-Ford algo

- Choice: Dijkstra algorithm.
- However, the algorithm needs **global knowledge**, i.e. the knowledge of the whole graph, which is (almost) impossible to obtain.
- In contrast, the Bellman-Ford algorithm **needs only local information**, i.e. the information of its **neighboorhood** rather than **the whole network.**

Application: Internet router protocol

AsynchronousShortestPath(G, t)

1: Initially, set $OPT[t, t] = 0$, and $OPT[v, t] = \infty$;
2: Label node t as "active"; //Here a node v is called "active" if $OPT[v, t]$ has been changed;
3: **while** exists an active node **do**
4: Select an active node w arbitrarily;
5: Remove w's "active" label;
6: **for all** edges (v, w) (in an arbitrary order) **do**
7: $$OPT[v, t] = \min \begin{cases} OPT[v, t] \\ OPT[w, t] + d(v, w) \end{cases}$$
8: **if** $OPT[v, t]$ was updated **then**
9: Label v as "active";
10: **end if**
11: **end for**
12: **end while**

Choice of the name dynamic programming

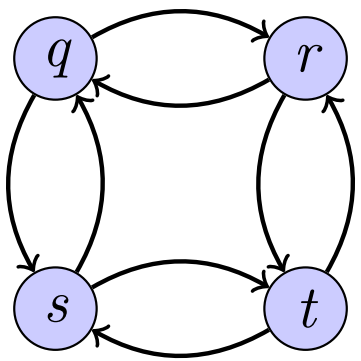
What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various rea- sons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the clas- sical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some com- bination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name.

— *by R. Bellman* See "Richard Bellman on the birth of dynamic programming" (S. Dreyfus, 2002) and "On the routing problem" (R. Bellman, 1958) for details.

A related problem: LongestPath problem

LongestPath problem

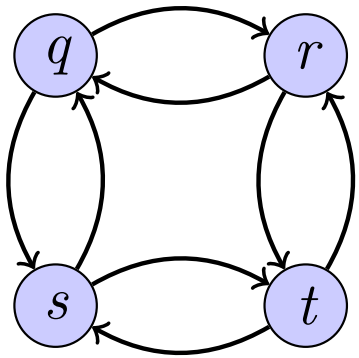
INPUT:
A directed graph $G =< V, E >$. Each edge (u, v) has a distance $d(u, v)$. Two nodes s and t.
OUTPUT:
The longest simple path from s to t.



Hardness: LongestPath problem is NP-hard. (Hint: it is obvious that LongestPath problem contains HamiltonianPath as its special case.)

Subtlety: LongestPath problem I

- Divide: The subproblems are not independent.
- Consider dividing problem to find a path from q to t into two subproblems: to find a path from q to r, and to find a path from r to t.



- Suppose we have already solved the sub-problems. Let's try to combine the solutions to the two sub-problems:
  1. $P(q, r) = q \to s \to t \to r$
  2. $P(r, t) = r \to q \to s \to t$

  We will obtain a path $q \to s \to t \to r \to q \to s \to t$, which is not simple.

Subtlety: LongestPath problem II

- In other words, the use of s in the first subproblem prevents us from using s in the second subproblem. However, we cannot obtain the optimal solution to the second subproblem without using s.

- In contrast, the ShortestPath problem does not have this difficulty.
- Why? The solutions to the subproblems **share no node**. Suppose the shortest paths $P(q, r)$ and $P(r, t)$ share a node $w (w \neq r)$. Then there will be a cycle $w \to \cdots \to r \to \cdots \to w$. Removing this cycle leads to a shorter path (no negative cycle). A contradiction.
- This means that the two subproblems are independent: the solution of one subproblem does not affect the solution to another subproblem.

A greedy algorithm exists when posing a stricter limit, i.e., all edges have a positive weight.

We will talk about this in next lectures.