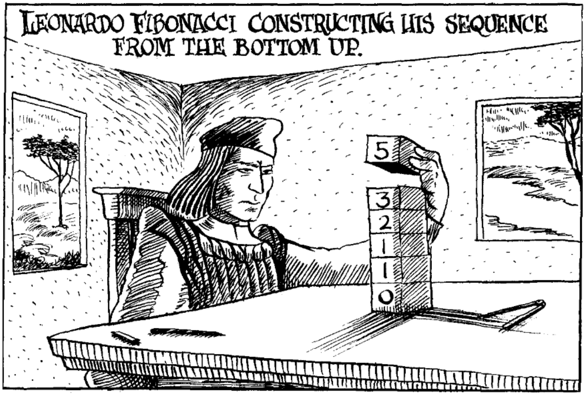
Chapter 3

Dynamic Programming


LEONARDO FIBONACCI CONSTRUCTING HIS SEQUENCE FROM THE BOTTOM UP.
5
3
2
1
1
0

RICHARD BELLMAN (1920–1984) Richard Bellman, born in Brooklyn, where his father was a grocer, spent many hours in the museums and libraries of New York as a child. After graduating high school, he studied mathematics at Brooklyn College and graduated in 1941. He began postgraduate work at Johns Hopkins University, but because of the war, left to teach electronics at the University of Wisconsin. He was able to continue his mathematics studies at Wisconsin, and in 1943 he received his masters degree there. Later, Bellman entered Princeton University, teaching in a special U.S. Army program. In late 1944, he was drafted into the army. He was assigned to the Manhattan Project at Los Alamos where he worked in theoretical physics. After the war, he returned to Princeton and received his Ph.D. in 1946.

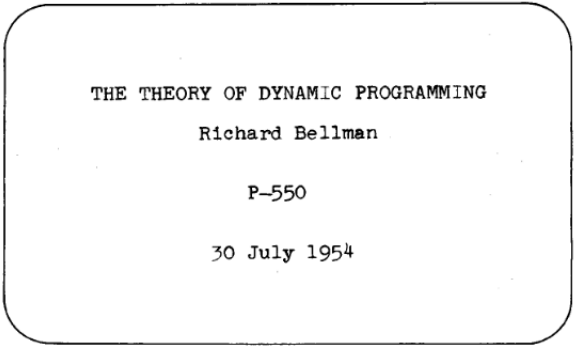
After briefly teaching at Princeton, he moved to Stanford University, where he attained tenure. At Stanford he pursued his fascination with number theory. However, Bellman decided to focus on mathematical questions arising from real-world problems. In 1952, he joined the RAND Corporation, working on multistage decision processes, operations research problems, and applications to the social sciences and medicine. He worked on many military projects while at RAND. In 1965 he left RAND to become professor of mathematics, electrical and biomedical engineering and medicine at the University of Southern California.

In the 1950s Bellman pioneered the use of dynamic programming, a technique invented earlier, in a wide range of settings. He is also known for his work on stochastic control processes, in which he introduced what is now called the Bellman equation. He coined the term *curse of dimensionality* to describe problems caused by the exponential increase in volume associated with adding extra dimensions to a space. He wrote an amazing number of books and research papers with many coauthors, including many on industrial production and economic systems. His work led to the application of computing techniques in a wide variety of areas ranging from the design of guidance systems for space vehicles, to network optimization, and even to pest control.

Tragically, in 1973 Bellman was diagnosed with a brain tumor. Although it was removed successfully, complications left him severely disabled. Fortunately, he managed to continue his research and writing during his remaining ten years of life. Bellman received many prizes and awards, including the first Norbert Wiener Prize in Applied Mathematics and the IEEE Gold Medal of Honor. He was elected to the National Academy of Sciences. He was held in high regard for his achievements, courage, and admirable qualities. Bellman was the father of two children.

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. 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 reasons. 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, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical 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 combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.


THE THEORY OF DYNAMIC PROGRAMMING

Richard Bellman

P-550

30 July 1954

R. Bellman began the systematic study of dynamic programming in 1955. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis.

Dynamic Programming – Building It Piece by Piece

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.)
- However, dynamic programming applies when the subproblems overlap–that is, when subproblems share subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a **table**, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

$\boxed{\text{DP vs. D\&C} \Longleftrightarrow \text{Caching vs. Computation}}$

من جرب المجرب حلت به الندامه

The Binomial Coefficient

$\binom{30}{15}$	155117520
Input	Output

Input description: Positive integers n and k

Problem description: find $\binom{n}{k}$. The direct formula is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n$$

Theorem

For all positive integers k and n, $0 < k < n$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

We can eliminate the need to compute $n!$ or $k!$ by using this recursive property:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

Can you justify this formula using combinatorial discussion?

Algorithm 3.1
Problem: Compute the binomial coefficient.
Inputs: nonnegative integers n and k, where $k \leq n$.
Outputs: bin, the binomial coefficient $\begin{pmatrix} n \\ k \end{pmatrix}$

```
int bin (int n, int k)
{ if(k = = 0 || n ==k)
   return 1;
else
   return bin(n - 1, k - 1) + bin(n - 1, k);
}
```

This algorithm is very inefficient.

► We can show (?) that the divide-and-conquer algorithm (the above algorithm) computes $2\begin{pmatrix} n \\ k \end{pmatrix} - 1$ terms to determine $\begin{pmatrix} n \\ k \end{pmatrix}$.

What is the order of complexity? (hint: search about "central binomial coefficient")

The problem is that the same instances are solved in each recursive call. For example, `bin(n - 1, k - 1)` and `bin(n - 1, k)` both need the result of `bin(n - 2, k - 1)`, and this instance is solved separately in each recursive call.

A dynamic programming approach:

- Establish a recursive property. This has already been done in the above theorem:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ or } j = i \end{cases}$$

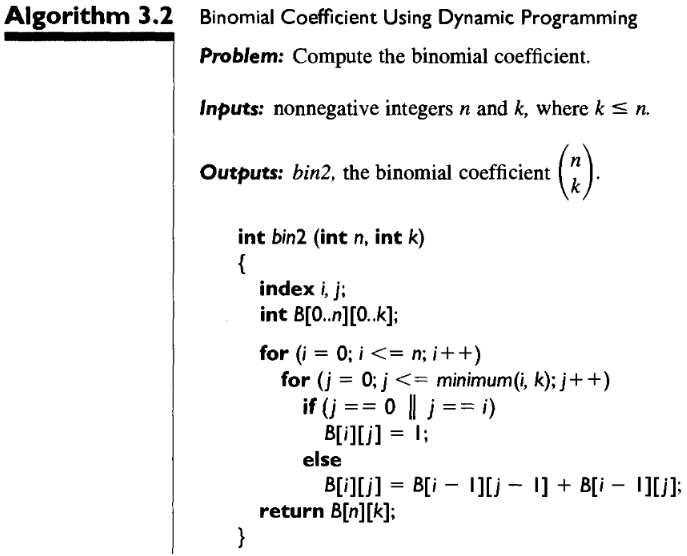
- Solve an instance of the problem in a bottom-up fashion by computing the rows in B in sequence starting with the first row.

► This is the basis for Pascal's triangle:

```
                 1
              1     1
           1     2     1
        1     3     3     1
     1     4     6     4     1
  1     5    10    10     5     1
```

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & & j & k \\
\hline
0 & 1 & & & & & & & \\
1 & 1 & 1 & & & & & & \\
2 & 1 & 2 & 1 & & & & & \\
3 & 1 & 3 & 3 & 1 & & & & \\
4 & 1 & 4 & 6 & 4 & 1 & & & \\
i & & & & & & & & \\
n & & & & & & & &
\end{array}
$$

$$\begin{tikzcd}
B[i-1][j-1] \arrow[dr] & B[i-1][j] \arrow[d] \\
 & B[i][j]
\end{tikzcd}$$



Algorithm 3.2 Binomial Coefficient Using Dynamic Programming

Problem: Compute the binomial coefficient.

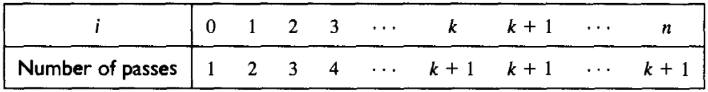
Inputs: nonnegative integers n and k, where $k \le n$.

Outputs: *bin2*, the binomial coefficient $\binom{n}{k}$.

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for (i = 0; i <= n; i++)
    for (j = 0; j <= minimum(i, k); j++)
      if (j == 0 || j == i)
        B[i][j] = 1;
      else
        B[i][j] = B[i - 1][j - 1] + B[i - 1][j];
  return B[n][k];
}
```

The following table shows the number of passes for each value of i:



i	0	1	2	3	$\cdots$	k	$k+1$	$\cdots$	n
Number of passes	1	2	3	4	$\cdots$	$k+1$	$k+1$	$\cdots$	$k+1$

The total number of passes is therefore given by

$$1 + 2 + \cdots + k + \underbrace{(k+1) + \cdots + (k+1)}_{n-k+1\ times}$$

This expression equals

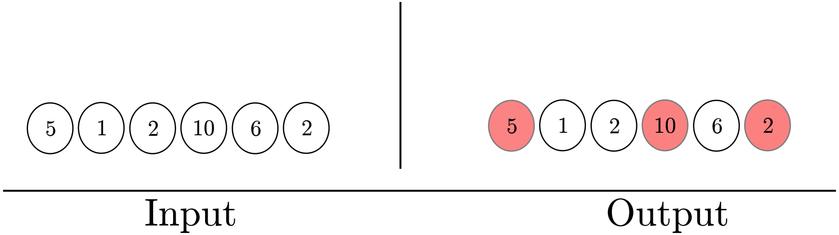
$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$$

By using dynamic programming instead of divide-and-conquer, we have developed a much more efficient algorithm.

We typically apply dynamic programming to **optimization** problems, those with OPTIMAL SUBSTRUCTURE and OVERLAPPING SUBPROBLEMS properties.

optimal substructure: After the first decision, we encounter the same original problem with reduced size!

Coin-row problem



Input description: A row of n coins whose values are some positive integers $c_1, c_2, \cdots, c_n$, not necessarily distinct.
Problem description: Pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

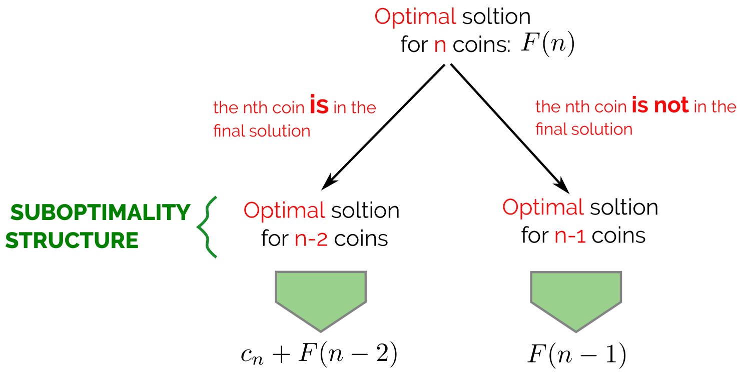
What is the complexity of the brute force algorithm?

- Considering all subsets ... $\rightarrow \Omega(2^n)$
- Considering all legitimate candidates $\rightarrow ?$ (left as an exercise)

$\boxed{Stop\&Think}$

Let $F(n)$ be the maximum amount that can be picked up from the row of n coins. How to derive a recurrence for $F(n)$?

Solution


Optimal soltion for n coins: $F(n)$
the nth coin **is** in the final solution
the nth coin **is not** in the final solution
SUBOPTIMALITY STRUCTURE
Optimal soltion for n-2 coins
Optimal soltion for n-1 coins
$c_n + F(n-2)$
$F(n-1)$

Therefore, we can write

$$F(n) = \max\{c_n + F(n-2), F(n-1)\}, \quad n > 1$$
$$F(0) = 0, \, F(1) = c_1.$$

We can compute $F(n)$ by filling the one-row table left to right

```
**ALGORITHM** *CoinRow*($C[1..n]$)
//Applies recursive formula bottom up to find the maximum
//amount of money that can be picked up from a coin row
// without picking two adjacent coins
//**Input:** Array $C[1..n]$ of positive integers indicating the coin values
//**Output:** The maximum amount of money that can be picked up
$F[0] \leftarrow 0; F[1] \leftarrow C[1]$
**for** $i \leftarrow 2$ **to** $n$ **do**
    $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$
```

The *CoinRow* algorithm takes $\Theta(n)$ time and $\Theta(n)$ space.

Example: The application of the algorithm to the coin row of denominations 5, 1, 2, 10, 6, 2 is shown bellow. It yields the maximum amount of 17

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

► To find the coins with the maximum total value found, we need to **back-trace** the computations to see which of the two possibilities – $c_n + F(n-2)$ or $F(n-1)$ – produced the maxima in formula.

- In the last application of the formula, it was the sum $c_6 + F(4)$, which means that the coin $c_6 = 2$ is a part of an optimal solution.
- Moving to computing $F(4)$, the maximum was produced by the sum $c_4 + F(2)$, which means that the coin $c_4 = 10$ is a part of an optimal solution as well.
- Finally, the maximum in computing $F(2)$ was produced by $F(1)$, implying that the coin c_2 is not the part of an optimal solution and the coin $c_1 = 5$ is.

To avoid repeating the same computations during the backtracing, the information about which of the two terms in the recursive formula was larger can be recorded in an extra array when the values of F are computed.

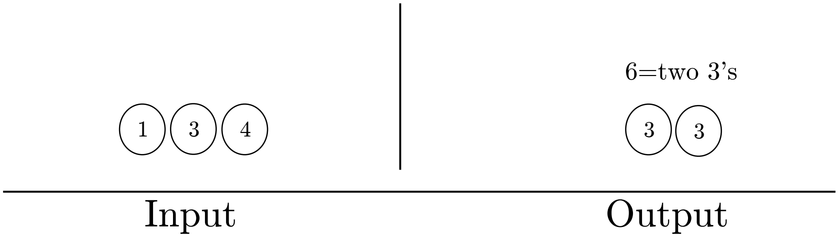
Steps of DP Strategy

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Identify the **subproblem optimality** of the problem.
   - e.g., by considering the backward (or forward) the decision tree.
2. Find a **recursive equation** for the optimal solution **value**.
3. Compute the value of the optimal solution, typically in a bottom-up fashion. (*value version*)
4. **Construct** an optimal solution from computed information. (*construction version*)
   - **back-tracing**

► Steps 1-3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.
► When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Change-making problem



Input description: A set of coins of denominations $d_1 < d_2 < \ldots < d_m$. Assume availability of unlimited quantities of coins for each of the m denominations and $d_1 = 1$.
Problem description: Give change for amount n using the minimum number of coins

What is the complexity of the brute force algorithm?

- hint: search about how to count the number of solutions for a "linear Diophantine equations".

$Stop\&Think$

Let $F(n)$ be the minimum number of coins whose values add up to n. How to derive a recurrence for $F(n)$?

Solution:
The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2, ..., m$ such that $n \geq d_j$. Therefore, we can consider all such denominations and select the one minimizing $F(n - d_j) + 1$. Since 1 is a constant, we can, of course, find the smallest $F(n - d_j)$ first and then add 1 to it. Hence, we have the following recurrence for $F(n)$:

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1, \quad n > 0$$
$$F(0) = 0$$

```
**ALGORITHM** *ChangeMaking*$(D[1..m], n)$
    //Applies dynamic programming to find the minimum number of coins
    //of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
    //given amount $n$
    //Input: Positive integer $n$ and array $D[1..m]$ of increasing positive
    //         integers indicating the coin denominations where $D[1] = 1$
    //Output: The minimum number of coins that add up to $n$
    $F[0] \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $temp \leftarrow \infty$; $j \leftarrow 1$
        while $j \leq m$ and $i \geq D[j]$ do
            $temp \leftarrow \min(F[i - D[j]], temp)$
            $j \leftarrow j + 1$
        $F[i] \leftarrow temp + 1$
    return $F[n]$
```

The *CoinRow* algorithm takes $O(nm)$ time and $\Theta(n)$ space.

Example: The application of the algorithm to amount $n = 6$ and denominations $1, 3, 4$. The answer it yields is two coins.

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

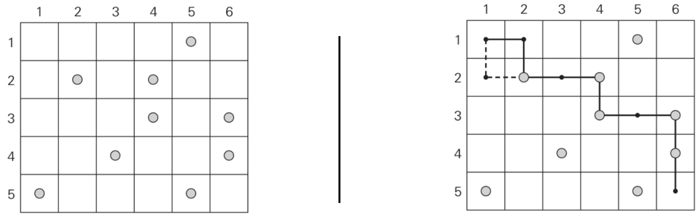
► To find the coins of an optimal solution, we need to backtrace the computations to see which of the denominations produced the minima in the recursive formula.

For the instance considered,

- the last application of the formula (for $n = 6$), the minimum was produced by $d_2 = 3$.
- The second minimum (for $n = 6 - 3$) was also produced for a coin of that denomination.

Thus, the minimum-coin set for $n = 6$ is two 3's.

Coin-collecting problem



Input | Output

Input description: An $n \times m$ board which several coins are placed in its cells (no more than one coin per cell).
Problem description: Compute the largest number of coins a robot can collect by starting at (1, 1) and moving right and down from upper left to down right corner.

Brute-Force: We need to take $n-1$ downward and $m-1$ rightward steps

$$\Rightarrow \frac{(n+m-2)!}{(n-1)!(m-1)!} = \begin{pmatrix} m+n-2 \\ n-1 \end{pmatrix}$$

$\boxed{Stop\&Think}$

Let $F(i,j)$ be the largest number of coins the robot can collect and bring to the cell (i,j) in the i th row and j th column of the board. How to derive a recurrence for $F(i,j)$?

Solution:
It can reach this cell either from **the adjacent cell** $(i-1,j)$ **above** it or from the **adjacent cell** $(i,j-1)$ **to the left** of it. The largest numbers of coins that can be brought to these cells are $F(i-1,j)$ and $F(i,j-1)$, respectively.

- Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of the cells in the first column. For those cells, we assume that $F(i-1,j)$ and $F(i,j-1)$ are equal to 0 for their nonexistent neighbors.

Therefore, we have the following formula for $F(i,j)$:

$$F(i,j) = \max\{F(i-1,j), F(i,j-1)\} + c_{ij}, \quad 1 \le i \le n,\ 1 \le j \le m$$
$$F(0,j) = 0,\ 1 \le j \le m, \quad F(i,0) = 0,\ 1 \le i \le n$$

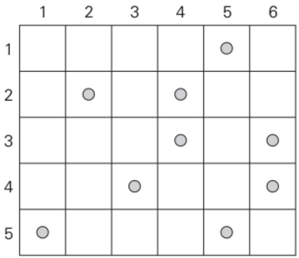
where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise.

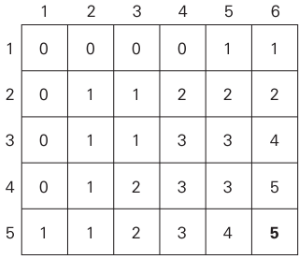
```
ALGORITHM  $RobotCoinCollection(C[1..n, 1..m])$
    //Applies dynamic programming to compute the largest number of
    //coins a robot can collect on an $n \times m$ board by starting at (1, 1)
    //and moving right and down from upper left to down right corner
    //Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
    //for cells with and without a coin, respectively
    //Output: Largest number of coins the robot can bring to cell $(n, m)$
    $F[1, 1] \leftarrow C[1, 1]$;   for $j \leftarrow 2$ to $m$ do $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
    for $i \leftarrow 2$ to $n$ do
        $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
        for $j \leftarrow 2$ to $m$ do
            $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
    return $F[n, m]$
```

The *RobotCoinCollection* algorithm takes $\Theta(nm)$ time and $\Theta(nm)$ space.

- Tracing the computations backward makes it possible to get an optimal path:
  - if $F(i-1,j) > F(i,j-1)$, an optimal path to cell (i,j) must come down from the adjacent cell above it
  - if $F(i-1,j) < F(i,j-1)$, an optimal path to cell (i, j) must come from the adjacent cell on the left
  - and if $F(i-1,j) = F(i,j-1)$, it can reach cell (i,j) from either direction.

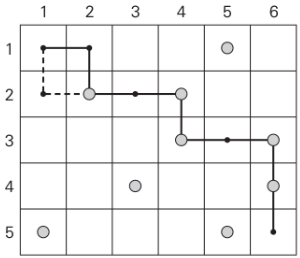
Example:


(a)

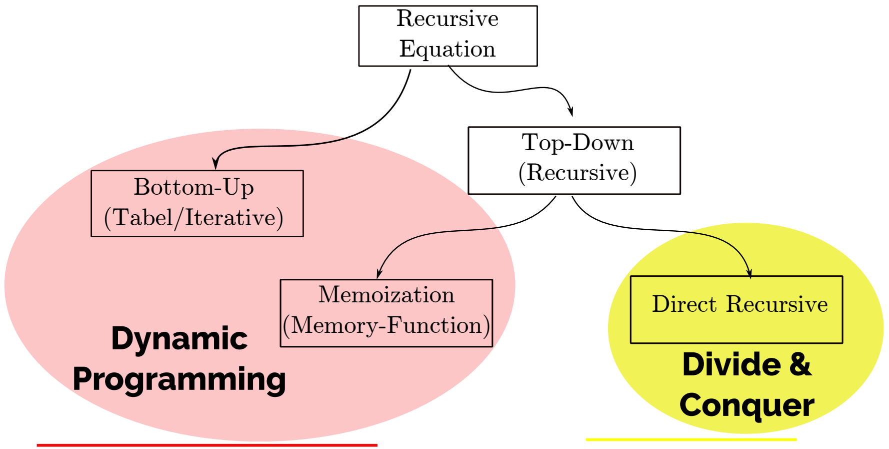


	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

(b)


(c)

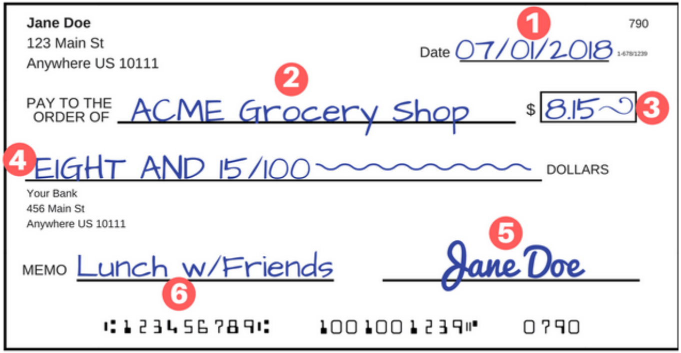
Dynamic Programming: Revisiting Implementation



Efficient for overlapping subproblems

Inefficient for overlapping subproblems

Memoization



Jane Doe
123 Main St
Anywhere US 10111

790

Date 07/01/2018 1-678/1239

PAY TO THE ORDER OF ACME Grocery Shop $ 8.15

EIGHT AND 15/100 DOLLARS

Your Bank
456 Main St
Anywhere US 10111

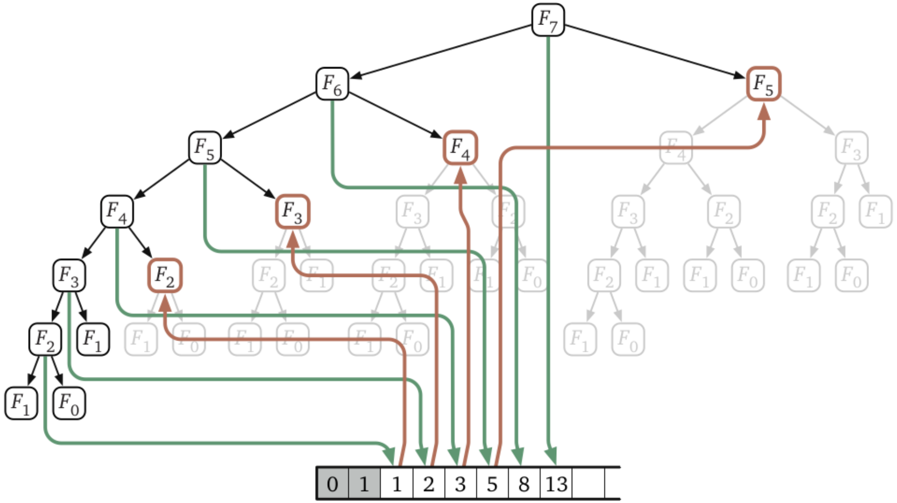
MEMO Lunch w/Friends

Jane Doe

⑆123456789⑆ 1001001239⑈ 0790

► In memoization, we write the procedure recursively in a natural manner, **but modified to save the result of each subproblem** (usually in an array or hash table).

► The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. We say that the recursive procedure has been memoized; it "remembers" what results it has computed previously.



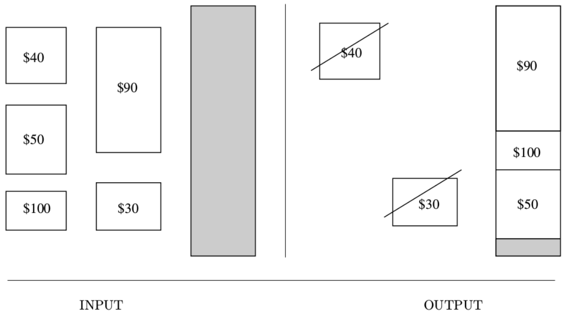
The recursion tree for Fibonacci(7) trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

```
MemFibo($n$):
    if $n = 0$
        return 0
    else if $n = 1$
        return 1
    else
        if $F[n]$ is undefined
            $F[n] \leftarrow$ MemFibo($n-1$) + MemFibo($n-2$)
        return $F[n]$
```

```
IterFibo($n$):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

► These two approaches (top-down & bottom-up) yield algorithms with the same asymptotic running time.
► The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

The 0-1 Knapsack Problem

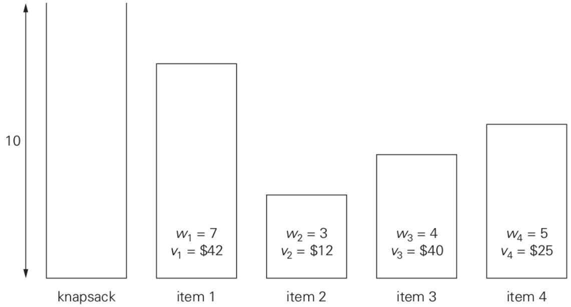


Input description: n items of known weights $w_1, \cdots, w_n$ and values $v_1, \cdots, v_n$ and a knapsack of capacity W
Problem description: find the most valuable subset of the items that fit into the knapsack.

Motivation:
The knapsack problem arises in resource allocation with financial constraints. How do you select what things to buy given a fixed budget? Everything has a cost and value, so we seek the most value for a given cost. The name knapsack problem invokes the image of the backpacker who is constrained by a fixed-size knapsack, and so must fill it only with the most useful and portable items.

Brute-Force

Generating all the subsets of the set of n items given $\Rightarrow \Omega(2^n)$ algorithm.



Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

In Class Question 5

What is the complexity of the brute-force algorithm for the knapsack problem?

This problem is one of the best-known examples of so-called NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important <u>conjecture</u> has never been proven.

A Dynamic Programming Approach to the 0-1 Knapsack Problem

$\boxed{Stop\&Think}$

Let $F(i, j)$ be the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. How to derive a recurrence for $F(i, j)$?

Solution:
Condition on whether the i th item is in the knapsack or not.

$$F(i,j)=\begin{cases}\max\{\overset{\substack{\text{i th item is not in}\\ \text{the knapsack}}}{\overset{\downarrow}{F(i-1,j)}},\ \overbrace{v_i+F(i-1,j-w_i)}^{\text{i th item is in the knapsack}}\}, & if\ j\geq w_i\\ F(i-1,j), & if\ j<w_i\end{cases}$$

Define the initial conditions as follows:

$$F(0,j)=0\quad for\ j\geq 0\quad and\quad F(i,0)=0\quad for\ i\geq 0$$

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1,\ j-w_i)$	$F(i-1,\ j)$	
$w_i,\ v_i$	i	0		$F(i,\ j)$	
	n	0			goal

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

- **Construction step:**

We can find the composition of an optimal subset by backtracing the computations of this entry in the table.

- Since $F(4,5) > F(3,5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5-2=3$ remaining units of the knapsack capacity.
- The value of the latter is $F(3,3)$. Since $F(3,3) = F(2,3)$, item 3 need not be in an optimal subset.
- Since $F(2,3) > F(1,3)$, item 2 is a part of an optimal selection, which leaves element $F(1,3-1)$ to specify its remaining composition.
- Similarly, since $F(1,2) > F(0,2)$, item 1 is the final part of the optimal solution $\{item1, item2, item4\}$.

The time efficiency and space efficiency of this algorithm are both in $\Theta(nW)$. The time needed to find the composition of an optimal solution is in $O(n)$.

NP-hardness!? $\Theta(nW)$!?

- This is linear in n.
- But the other term in that expression is W, with no relationship with n.

$\Rightarrow$ For a given n, we can create instances with arbitrarily large running times by taking arbitrarily large values of W.

Example

if $W = n!$ the complexity is $\Theta(nn!)$, worse than the brute-force algorithm that simply considers all subsets.

```
ALGORITHM  $MFKnapsack(i, j)$
    //Implements the memory function method for the knapsack problem
    //Input: A nonnegative integer $i$ indicating the number of the first
    //          items being considered and a nonnegative integer $j$ indicating
    //          the knapsack capacity
    //Output: The value of an optimal feasible subset of the first $i$ items
    //Note: Uses as global variables input arrays $Weights[1..n]$, $Values[1..n]$,
    //and table $F[0..n, 0..W]$ whose entries are initialized with $-1$'s except for
    //row 0 and column 0 initialized with 0's
    if $F[i, j] < 0$
        if $j < Weights[i]$
            $value \leftarrow MFKnapsack(i-1, j)$
        else
            $value \leftarrow \max(MFKnapsack(i-1, j),$
                                  $Values[i] + MFKnapsack(i-1, j-Weights[i]))$
        $F[i, j] \leftarrow value$
    return $F[i, j]$
```

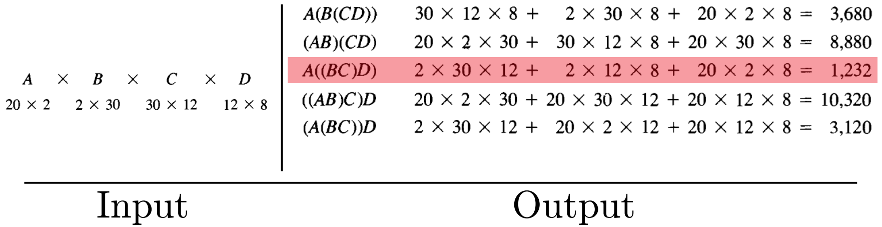
	i	0	1	2	3	4	5
				capacity j			
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

► Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed.

Discussion Question

What is the subset-sum problem and its relationship with the knapsack?

Matrix-chain Multiplication



A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

$A(B(CD))$	$30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 =$	3,680
$(AB)(CD)$	$20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 =$	8,880
$A((BC)D)$	$2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 =$	1,232
$((AB)C)D$	$20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 =$	10,320
$(A(BC))D$	$2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 =$	3,120

Input Output

Input description: the number of matrices n, and an array of integers d, indexed from 0 to n, where $d[i-1] \times d[i]$ is the dimension of the i'th matrix.

Problem description: Determining the minimum number of elementary multiplications needed to multiply n matrices and an order that produces that minimum number.

Suppose we want to multiply a 2×3 matrix times a 3×4 matrix as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

For each entry of the resultant matrix, we need 3 multiplication, and there are 2×4 entries in the resultant matrix $\Rightarrow$ the total number of elementary multiplication is $3 \times 2 \times 4$.

► In general, to multiply an $i \times j$ matrix times a $j \times k$ matrix, using the standard method, it is necessary to do $i \times j \times k$ elementary multiplications.

Brute-Force:
The brute-force algorithm is to consider all possible orders and take the minimum. We will show that this algorithm is at least exponential time.

Proof.

let t_n be the number of different orders in which we can multiply n matrices: $A_1, A_2, \cdots, A_n$.

- A subset of all the orders : A_1 is the last matrix multiplied. This subset has t_{n-1} members (why?).
- Another subset: A_n is the last matrix multiplied. This subset has also t_{n-1} members (why?).

$$\Rightarrow t_n \geq 2t_{n-1}, \quad t_2 = 1$$
$$\Rightarrow t_n \geq 2^{n-2}$$

$\square$

What about the exact value for t_n?

$$(A_1)(A_2 \cdots A_n) \rightarrow t_1 \times t_{n-1}$$
$$(A_1 A_2)(A_3 \cdots A_n) \rightarrow t_2 \times t_{n-2}$$
$$\vdots$$
$$(A_1 \cdots A_{n-1})(A_n) \rightarrow t_{n-1} \times t_1$$
$$\Rightarrow t_n = \sum_{k=1}^{n-1} t_k t_{n-k}$$

Let $C_{n-1} = t_n \rightarrow C_{n-1} = \sum_{k=1}^{n-1} C_{k-1} C_{n-k-1} \rightarrow C_n = \sum_{k=1}^{n} C_{k-1} C_{n-k} = \sum_{k=0}^{n-1} C_k C_{n-k-1}$: The n th Catalan number

The n th Catalan number: $C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1}, \quad C_0 = C_1 = 1$

It can be proved that $C_n = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix} \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$

Therefore,

$$t_n = C_{n-1} = \frac{1}{n}\left(\begin{array}{c} 2(n-1) \\ n-1 \end{array}\right) \approx \frac{4^{n-1}}{(n-1)^{3/2}\sqrt{\pi}}$$

$\boxed{Stop\&Think}$

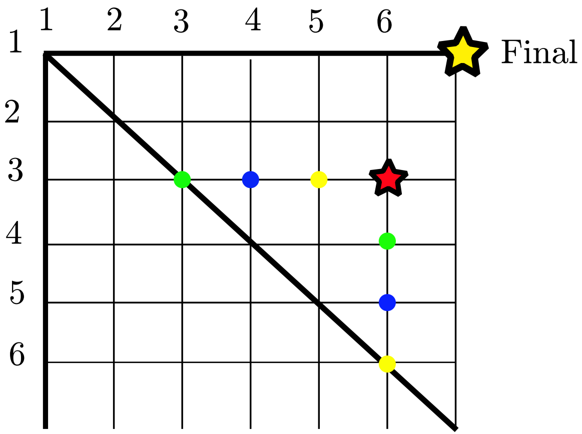
Let $M[i][j]$ be the minimum number of multiplications needed to multiply A_i through A_j ($i \leq j, \quad M[i][i] = 0$). How to derive a recurrence for $M[i][j]$?

Solution:

$$(A_i)(A_{i+1}\cdots A_j) \rightarrow M[i][i] + M[i+1][j] + d_{i-1}d_id_j$$
$$(A_iA_{i+1})(A_{i+2}\cdots A_j) \rightarrow M[i][i+1] + M[i+2][j] + d_{i-1}d_{i+1}d_j$$
$$\vdots$$
$$(A_i\cdots A_{j-1})(A_j) \rightarrow M[i][j-1] + M[j][j] + d_{i-1}d_{j-1}d_j$$

For any $1 \le i \le j \le n$

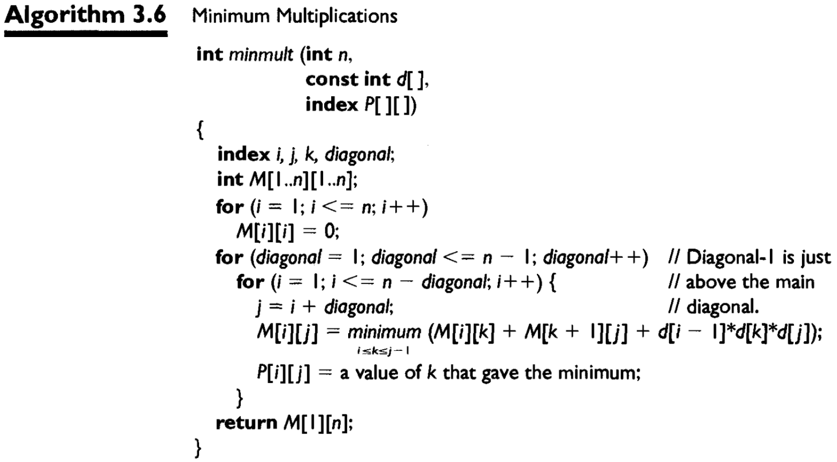
$$M[i][j] = \min_{i\le k\le j-1} \{M[i][k] + M[k+1][j] + d_{i-1}d_kd_j\}$$



$$M[3][6] = \min\{M[3][3] + M[4][6] + d_2 d_3 d_6,$$
$$M[3][4] + M[5][6] + d_2 d_4 d_6,$$
$$M[3][5] + M[6][6] + d_2 d_5 d_6\}$$

$\Rightarrow$ The table should be filled in diagonally.

Inputs: the number of matrices n, and an array of integers d, indexed from 0 to n, where $d[i-1] \times d[i]$ is the dimension of the ith matrix.
Outputs: minmult, the minimum number of elementary multiplications needed to multiply the n matrices; a two-dimensional array P from which the optimal order can be obtained.


Algorithm 3.6 Minimum Multiplications

```
int minmult (int n,
             const int d[ ],
             index P[ ][ ])
{
  index i, j, k, diagonal;
  int M[1..n][1..n];
  for (i = 1; i <= n; i++)
    M[i][i] = 0;
  for (diagonal = 1; diagonal <= n − 1; diagonal++)   // Diagonal-1 is just
    for (i = 1; i <= n − diagonal; i++) {             // above the main
      j = i + diagonal;                               // diagonal.
      M[i][j] = $\underset{i \le k \le j-1}{minimum}$ (M[i][k] + M[k + 1][j] + d[i − 1]*d[k]*d[j]);
      P[i][j] = a value of k that gave the minimum;
    }
  return M[1][n];
}
```

Every-Case Time Complexity:

Basic Operation: We can consider the instructions executed for each value of k to be the basic operation. Included is a comparison to test for the minimum.
Input size: n, the number of matrices to be multiplied.

. . . .

Because $j = i + diagonal$, for given values of $diagonal$ and i, the number of passes through the k loop is

$$j - 1 - i + 1 = i + diagonal - 1 - i + 1 = diagonal.$$

$$\Rightarrow T(n) = \sum_{diagonal=1}^{n-1} ((n - diagonal) \times diagonal)$$

$$= \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Optimal Binary Search Tree
» Background: Binary Search Tree (BST)

Discussion Question: (Comparing Dictionary Implementa

What are the worst-case complexities for `searching`, `deleting`, and `inserting` operations when the data structure (dictionary) is implemented as

- a sorted array
- an unsorted array
- a singly linked list (both sorted and unsorted versions)
- a doubly linked list (both sorted and unsorted versions)

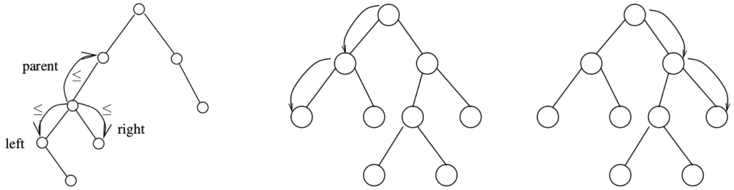
Partial solution: for arrays, we have

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$

We have seen data structures that allow fast search or flexible update, but not fast search and flexible update.

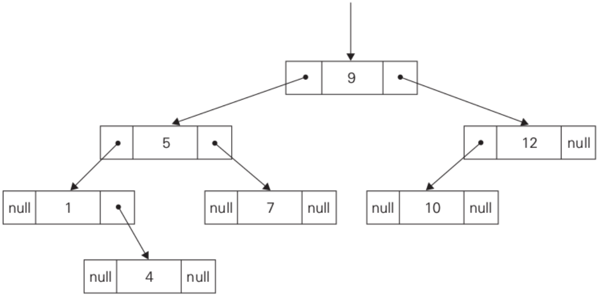
Binary search tree: a data structure which tries to have both features ...

- A binary tree (not more than two children)
- a key assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.

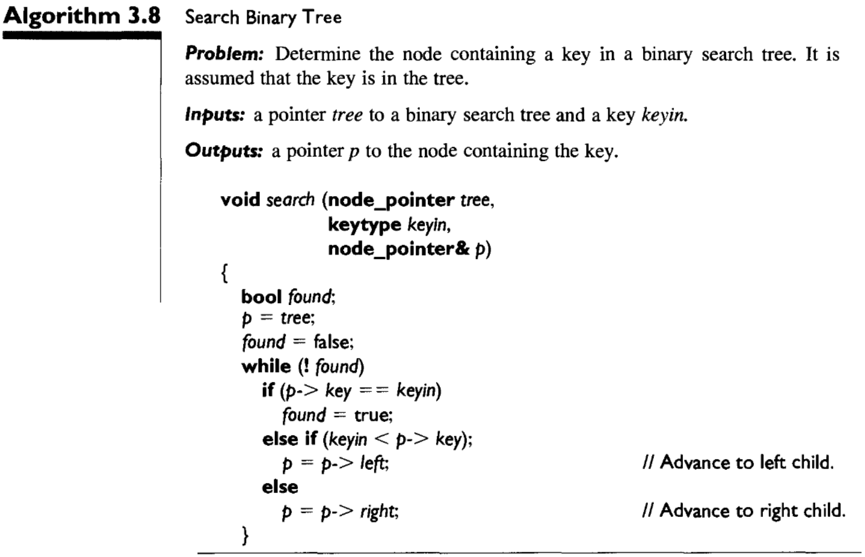


Relationships in a binary search tree (left). Finding the minimum (center) and maximum (right) elements in a binary search tree

- Standard implementation of the binary search tree



- search operation



Algorithm 3.8 Search Binary Tree

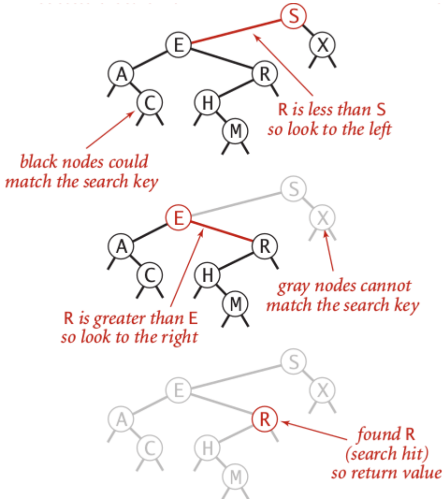
Problem: Determine the node containing a key in a binary search tree. It is assumed that the key is in the tree.

Inputs: a pointer *tree* to a binary search tree and a key *keyin*.

Outputs: a pointer *p* to the node containing the key.

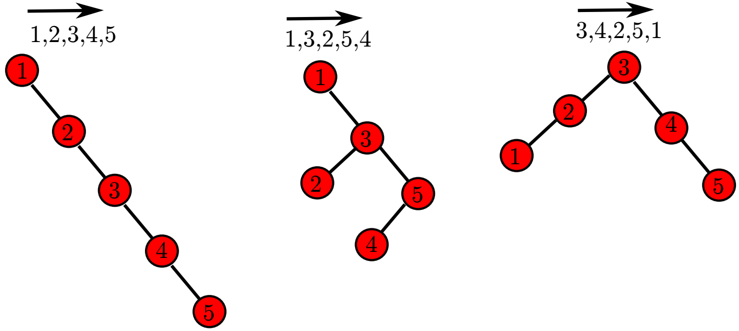
```
void search (node_pointer tree,
                  keytype keyin,
                  node_pointer& p)
{
   bool found;
   p = tree;
   found = false;
   while (! found)
      if (p-> key == keyin)
         found = true;
      else if (keyin < p-> key);
         p = p-> left;                          // Advance to left child.
      else
         p = p-> right;                         // Advance to right child.
   }
```

Example: successful search for R



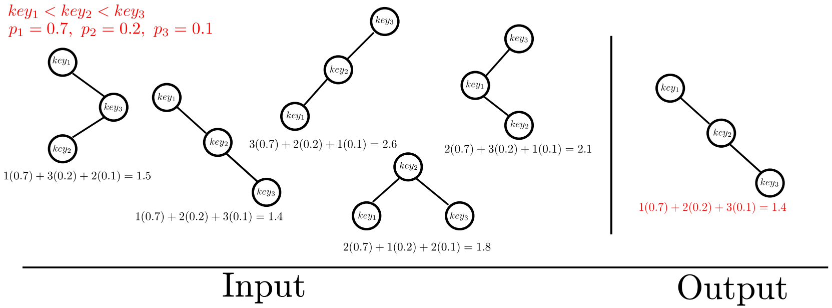
- The number of comparisons done by procedure search to locate a key is called the search time.
- The search time for a given key is $depth(key) + 1$

- insertion operation: search the key and insert a leaf where it should be!
  - Tree's height: $\lceil \log n \rceil \leq h \leq n-1$ (balanced/unbalanced)
    - h depends on the order of inserting keys into tree



- BST is a way of implementing dictionaries (find, insert, delete records)
  - Each operation in $O(h)$ ($\rightarrow O(\log n)$ if it is well organized)
  - Many versions: Random, AVL, red-black, splay, ...

Optimal Binary Search Tree



Input description: n, the number of keys, and an array of real numbers p indexed from 1 to n, where $p[i]$ is the probability of searching for the ith key.
Problem description: Determine an optimal binary search tree

Brute-Force:
The brute-force algorithm is to consider all possible binary trees of $key_1 < key_2 < \cdots < key_n$ with $p_1, p_2, \cdots, p_n$ as the probabilities of occurrence and take the one with minimum average search time $\sum_{i=1}^{n} p_i c_i$ where $c_i = depth(key_i) + 1$.

We will show that this algorithm is at least exponential time.

Proof is left as an exercise.

Proof skeleton: let t_n be the number of different binary search trees in which we can build for $key_1, key_2, \cdots, key_n$.

- A subset of all binary trees: Key_1 is the root ($t_{n,key_1\ rooted}$)
- Another subset: key_n is the root ($t_{n,key_n\ rooted}$)

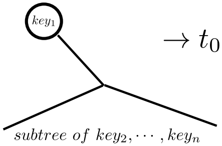
$$t_n \geq t_{n,key_1\ rooted} + t_{n,key_n\ rooted}$$

....

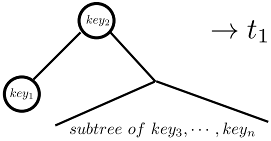
Discussion Question

An alternative way: show that if we just consider all binary search trees with a depth of $n-1$, we have an exponential number of trees.

What about the exact value for t_n?

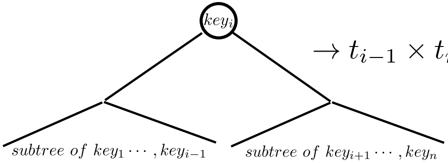


$\rightarrow t_0 \times t_{n-1}$



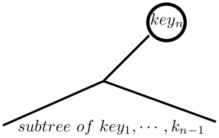
$\rightarrow t_1 \times t_{n-2}$

$\vdots$


key_i
$subtree\ of\ key_1 \cdots, key_{i-1}$
$subtree\ of\ key_{i+1} \cdots, key_n$

$\rightarrow t_{i-1} \times t_{n-i}$

$\vdots$



$\rightarrow t_{n-1} \times t_0$

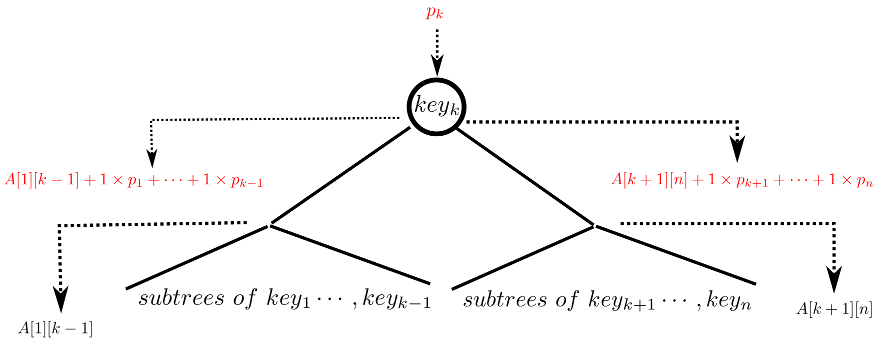
$\Rightarrow t_n = \sum_{k=1}^{n} t_{k-1} t_{n-k}$

Again the Catalan number $\rightarrow t_n = C_n = \frac{1}{n+1} \binom{2n}{n}$

$\boxed{Stop\&Think}$

Let $A[i][j]$ be the optimal search time for key_i through key_j ($i \leq j, \quad A[i][i] = p_i$). How to derive a recurrence for $A[i][j]$?

Solution:
In the final optimal tree, one of the keys is in the root. Consider the case in which key_k is in the root:

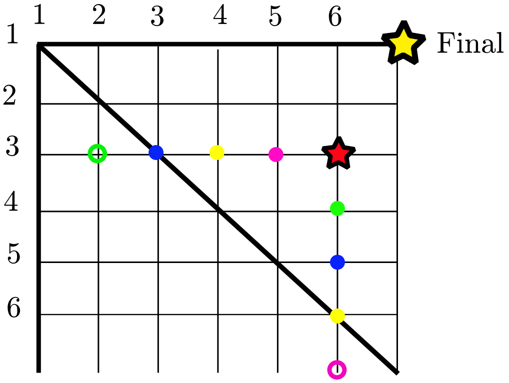

p_k
key_k
$A[1][k-1] + 1 \times p_1 + \cdots + 1 \times p_{k-1}$
$A[k+1][n] + 1 \times p_{k+1} + \cdots + 1 \times p_n$
$subtrees\ of\ key_1 \cdots, key_{k-1}$
$subtrees\ of\ key_{k+1} \cdots, key_n$
$A[1][k-1]$
$A[k+1][n]$

Therefore,

$$A[1][n] = \min_k \{A[1][k-1] + A[k+1][n]\} + \sum_{m=1}^{n} p_m$$

Generally, we can write

$$\begin{cases} A[i][j] = \min_{i \leq k \leq j} \{A[i][k-1] + A[k+1][j]\} + \sum_{m=i}^{j} p_m & i < j \\ A[i][i] = p_i \\ A[i][i-1] = 0, A[j+1][j] = 0 \end{cases}$$



Algorithm 3.9 Optimal Binary Search Tree

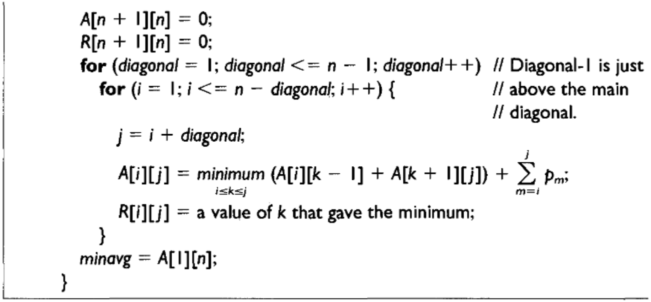
Problem: Determine an optimal binary search tree for a set of keys, each with a given probability of being the search key.

Inputs: *n*, the number of keys, and an array of real numbers *p* indexed from 1 to *n*, where *p*[*i*] is the probability of searching for the *i*th key.

Outputs: a variable *minavg*, whose value is the average search time for an optimal binary search tree; and a two-dimensional array *R* from which an optimal tree can be constructed. *R* has its rows indexed from 1 to $n + 1$ and its columns indexed from 0 to *n*. *R*[*i*][*j*] is the index of the key in the root of an optimal tree containing the *i*th through the *j*th keys.

```
void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
  index i, j, k, diagonal;
  float A[1..n + 1][0..n];

  for (i = 1; i <= n; i++) {
    A[i][i - 1] = 0;
    A[i][i] = p[i];
    R[i][i] = i;
    R[i][i - 1] = 0;
  }
```



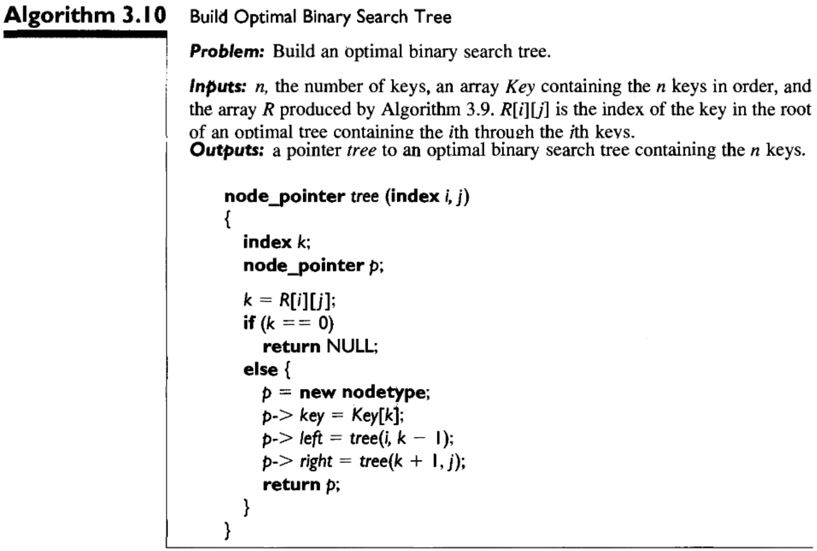
```
    A[n + 1][n] = 0;
    R[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n − 1; diagonal++)  // Diagonal-1 is just
       for (i = 1; i <= n − diagonal; i++) {           // above the main
                                                       // diagonal.
          j = i + diagonal;
          A[i][j] = $\underset{i \le k \le j}{minimum}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m$;
          R[i][j] = a value of k that gave the minimum;
       }
    minavg = A[1][n];
 }
```

Every-Case Time Complexity Analysis:

$$T(n) = \sum_{diagonal=1}^{n-1} (n - diagonal)(diagonal + 1)$$

$$= \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

Build Optimal Binary Search Tree



Algorithm 3.10 Build Optimal Binary Search Tree

Problem: Build an optimal binary search tree.

Inputs: *n*, the number of keys, an array *Key* containing the *n* keys in order, and the array *R* produced by Algorithm 3.9. *R*[*i*][*j*] is the index of the key in the root of an optimal tree containing the *i*th through the *j*th keys.

Outputs: a pointer *tree* to an optimal binary search tree containing the *n* keys.

```
node_pointer tree (index i, j)
{
   index k;
   node_pointer p;

   k = R[i][j];
   if (k == 0)
      return NULL;
   else {
      p = new nodetype;
      p-> key = Key[k];
      p-> left = tree(i, k − 1);
      p-> right = tree(k + 1, j);
      return p;
   }
}
```

- Following our convention for recursive algorithms, the parameters n, Key, and R are not inputs to function tree. If the algorithm were implemented by defining n, Key, and R globally, a pointer root to the root of an optimal binary search tree is obtained by calling tree as follows: $root = tree(1, n)$;

Example:

Don	Isabelle	Ralph	Wally
Key[1]	*Key*[2]	*Key*[3]	*Key*[4]

and

$$p_1 = \frac{3}{8} \qquad p_2 = \frac{3}{8} \qquad p_3 = \frac{1}{8} \qquad p_4 = \frac{1}{8}.$$

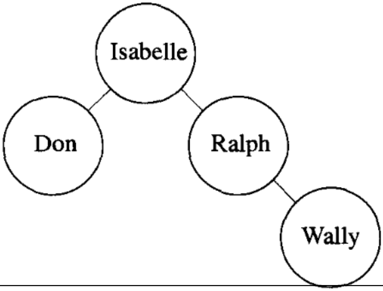
. . . .

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

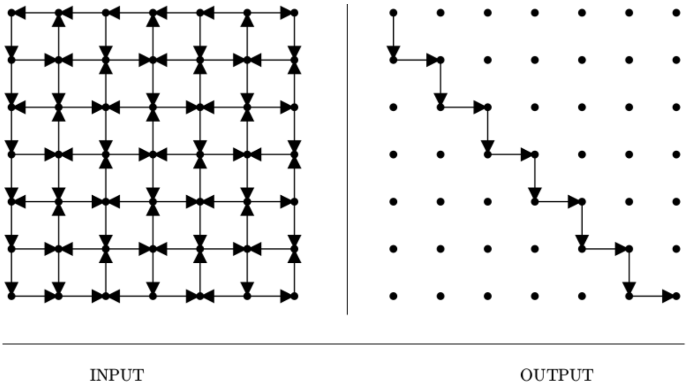
A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R



Shortest Path



Input description: An edge-weighted graph G
Problem description: finding the shortest paths from each vertex to all other vertices

Brute-force:
For each vertex, determine the lengths of all the paths from that vertex to each other vertex, and compute the minimum of these lengths. This algorithm is worse than exponential-time.

. . . .

► For example„ suppose there is an edge from every vertex to every other vertex. Then a subset of all the paths from one vertex to another vertex is the set of all those paths that start at the first vertex, end at the other vertex, and pass through all the other vertices. The total number of such paths is

$$(n-2)(n-3)\cdots 1 = (n-2)!$$

which is worse than exponential.

Assume that the vertices of G are $V = \{1, 2, \cdots, n\}$ and the graph's adjacency matrix is $W = (w_{i,j})$.

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight on edge}(i, j) & \text{if there is an edge from } v_i \text{ to } v_j \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

Define $d_{i,j}^{(k)}$ as the weight of the shortest path among any pair of vertices $i, j \in V$ when the intermediate vertices can be any vertex belongs to the subset $\{1, \cdots, k\}$ of vertices for some k.

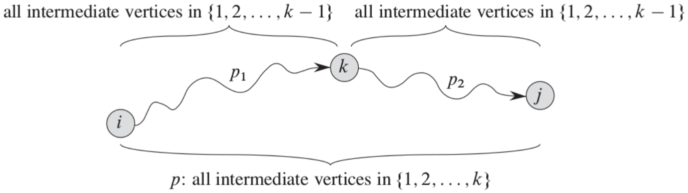
$\boxed{Stop\&Think}$

How to derive a recurrence for $d_{i,j}^{(k)}$?

Solution:

Based on on whether or not k is an intermediate vertex of path p:

- If k is not an intermediate vertex of path p, then $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$
- If k is an intermediate vertex of path p, then $d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$


all intermediate vertices in $\{1,2,\ldots,k-1\}$
all intermediate vertices in $\{1,2,\ldots,k-1\}$
p_1
k
p_2
i
j
p: all intermediate vertices in $\{1,2,\ldots,k\}$

Therefore, we have the following recursion:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0 \\ \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

```
FLOYD-WARSHALL(W)
1  $n = W.rows$
2  $D^{(0)} = W$
3  for $k = 1$ to $n$
4      let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
5      for $i = 1$ to $n$
6          for $j = 1$ to $n$
7              $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8  return $D^{(n)}$
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3–7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$.

Constructing a shortest path

We need to have a matrix $\Pi = (\pi_{i,j})$ called the predecessor matrix in which $\pi_{i,j}$ gives the predecessor of vertex j on a shortest path from vertex i. Accordingly, the following algorithm outputs the shortest path from i to j:

```
PRINT-ALL-PAIRS-SHORTEST-PATH($\Pi, i, j$)
1  if $i == j$
2      print $i$
3  elseif $\pi_{ij}$ == NIL
4      print "no path from" $i$ "to" $j$ "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH($\Pi, i, \pi_{ij}$)
6      print $j$
```

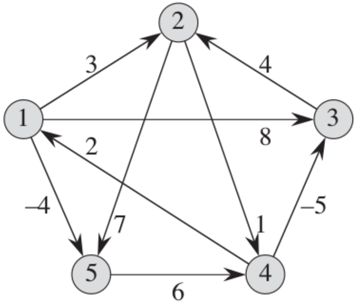
we can compute the predecessor matrix Π while the algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \cdots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$, and we define $\pi_{i,j}^{(k)}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \cdots, k\}$.
We can give a recursive formulation of $\pi_{i,j}^{(k)}$.

$$\pi_{i,j}^{(0)} = \begin{cases} NIL & \text{if } i = j \text{ or } \ w_{i,j} = 0 \\ i & \text{if } i \neq j \text{ and } w_{i,j} < \infty \end{cases}$$

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{if } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)} & \text{if } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \end{cases}$$

Exercise: incorporate the computation the $\Pi(k)$ matrix into the FLOYD- WARSHALL procedure.

Example:



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Transitive Closure

Definition

Given a directed graph $G = (V, E)$ with vertex set$V = \{1, 2, \cdots, n\}$, we define the transitive closure of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in G}\}$$

- One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

- Another (but similar) way that can save time and space in practice:
  For $i, j, k = 1, 2, \cdots, n$, define $t_{i,j}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \cdots, k\}$, and 0 otherwise. Then, we can construct G^* by connecting v_i to v_j if $t_{i,j}^{(n)} = 1$

$$t_{i,j}^{(0)} = \begin{cases} 1 & \text{if } i = j \text{ or there is an edge from } v_i \text{ to } v_j \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

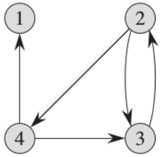
$$t_{i,j}^{(k)} = t_{i,j}^{(k-1)} \vee (t_{i,k}^{(k-1)} \wedge t_{k,j}^{(k-1)}), \quad k \geq 1$$

```
TRANSITIVE-CLOSURE($G$)
1   $n = |G.V|$
2   let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
3   for $i = 1$ to $n$
4       for $j = 1$ to $n$
5           if $i == j$ or $(i, j) \in G.E$
6               $t_{ij}^{(0)} = 1$
7           else $t_{ij}^{(0)} = 0$
8   for $k = 1$ to $n$
9       let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
10      for $i = 1$ to $n$
11          for $j = 1$ to $n$
12              $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
13  return $T^{(n)}$
```

The TRANSITIVE-CLOSURE procedure, like the Floyd-Warshall algorithm, runs in $\Theta(n^3)$ time. But

- On some computers, though, logical operations on single-bit values execute faster than arithmetic operations on integer words of data.
- because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less.

Example

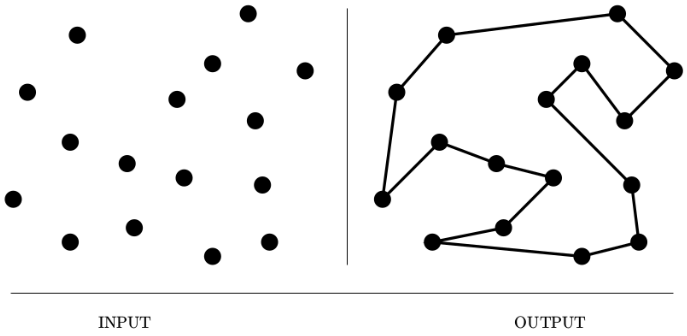


$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

Traveling Salesperson (TSP)



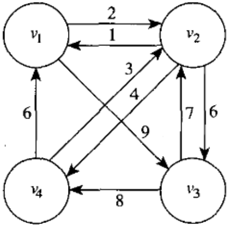
Input description: A weighted graph G
Problem description: Find the cycle of minimum cost, visiting each vertex of G exactly once.

Why is it called Traveling Salesperson Problem?

Imagine a traveling salesman planning a car trip to visit a set of cities. What is the shortest route that will enable him to do so and return home, thus minimizing his total driving?

Brute-force:
In general, there can be an edge from every vertex to every other vertex. If we consider all possible tours, the total number of tours is $(n-1)(n-2)\cdots 1 = (n-1)!$ which is worse than exponential.

Denote by V the set of all vertices, and by A a subset of V. Define $D[v_i][A]$ as the length of the shortest path from v_i to v_1 passing through each vertex in A exactly once.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 2 & 9 & \infty \\ 2 & 1 & 0 & 6 & 4 \\ 3 & \infty & 7 & 0 & 8 \\ 4 & 6 & 3 & \infty & 0 \end{array}$$

If $A = \{v_3\}$, then

$$D[v_2][A] = length[v_2, v_3, v_1] = \infty.$$

If $A = \{v_3, v_4\}$, then

$$D[v_2][A] = minimum(length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1]) = minimum(20, \infty) = 20.$$

Stop&Think

How to derive a recurrence for $D[v_i][A]$?

Solution:

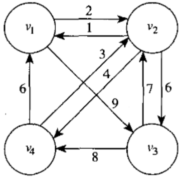
$$D[v_i][A] = \min_{v_j \in A} \left(w_{i,j} + D[v_j][A - \{v_j\}] \right) \quad if\ A \neq \Phi$$

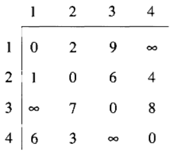
$$D[v_i][\Phi] = w_{i,1}$$

- Do the bottom-up procedure: solve the above recursion for all $A \in V$ starting from Φ to subsets of higher cardinalities $(|A| = 1, 2, \cdots, (n-2))$
- TSP (minimum cost tour):

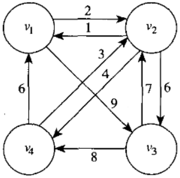
$$D[v_1][V - \{v_1\}] = \min_{v_j : 2 \leq j \leq n} \left(w_{1,j} + D[v_j][V - \{v_1, v_j\}] \right)$$

Example: Determine an optimal tour for the following graph:





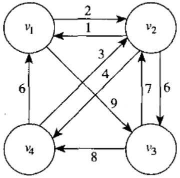
	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 2 & 9 & \infty \\ 2 & 1 & 0 & 6 & 4 \\ 3 & \infty & 7 & 0 & 8 \\ 4 & 6 & 3 & \infty & 0 \end{array}$$

First consider the empty set:

$$D[v_2][\emptyset] = 1$$
$$D[v_3][\emptyset] = \infty$$
$$D[v_4][\emptyset] = 6$$



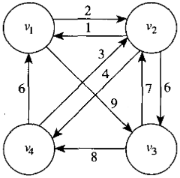
	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Next consider all sets containing one element:

$$D[v_3][\{v_2\}] = \underset{v_j \in \{v_2\}}{minimum}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_2][\emptyset] = 7 + 1 = 8$$

Similarly,

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$



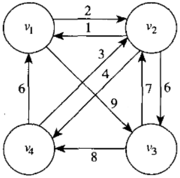
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 2 & 9 & \infty \\ 2 & 1 & 0 & 6 & 4 \\ 3 & \infty & 7 & 0 & 8 \\ 4 & 6 & 3 & \infty & 0 \end{array}$$

Next consider all sets containing two elements:

$$\begin{aligned} D[v_4][\{v_2, v_3\}] &= \underset{v_j \in \{v_2, v_3\}}{minimum}(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}]) \\ &= minimum(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}]) \\ &= minimum(3 + \infty, \infty + 8) = \infty \end{aligned}$$

Similarly,

$$D[v_3][\{v_2, v_4\}] = minimum(7 + 10, 8 + 4) = 12$$
$$D[v_2][\{v_3, v_4\}] = minimum(6 + 14, 4 + \infty) = 20$$



$$\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 2 & 9 & \infty \\
2 & 1 & 0 & 6 & 4 \\
3 & \infty & 7 & 0 & 8 \\
4 & 6 & 3 & \infty & 0
\end{array}$$

Finally, compute the length of an optimal tour:

$$\begin{aligned}
D[v_1][\{v_2, v_3, v_4\}] &= \underset{v_j \in \{v_2, v_3, v_4\}}{minimum}(W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}]) \\
&= minimum(W[1][2] + D[v_2][\{v_3, v_4\}], \\
&\qquad\qquad W[1][3] + D[v_3][\{v_2, v_4\}], \\
&\qquad\qquad W[1][4] + D[v_4][\{v_2, v_3\}]) \\
&= minimum(2 + 20, 9 + 12, \infty + \infty) = 21.
\end{aligned}$$

Algorithm 3.11 The Dynamic Programming Algorithm for the Traveling Salesperson Problem

Problem: Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.

Inputs: a weighted, directed graph, and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from ith vertex to the jth vertex.

Outputs: a variable *minlength*, whose value is the length of an optimal tour, and a two-dimensional array P from which an optimal tour can be constructed. P has its rows indexed from 1 to n and its columns indexed by all subsets of $V - \{v_1\}$. $P[i][A]$ is the index of the first vertex after v_i on a shortest path from v_i to v_1 that passes through all the vertices in A exactly once.

```
void travel (int n,
             const number W[][],
             index P[][],
             number& minlength)
{
  index i, j, k;
  number D[1..n][subset of $V - \{v_1\}$];

  for (i = 2; i <= n; i++)
    D[i][∅] = W[i][1];
  for (k = 1; k <= n - 2; k++)
    for (all subsets $A \subseteq V - \{v_1\}$ containing k vertices)
      for (i such that $i \neq 1$ and $v_i$ is not in A) {
        D[i][A] = $\underset{v_j \in A}{minimum}$ (W[i][j] + D[$v_j$][$A - \{v_j\}$]);
        P[i][A] = value of j that gave the minimum;
      }
  D[1][$V - \{v_1\}$] = $\underset{2 \le j \le n}{minimum}$ (W[1][j] + D[$v_j$][$V - \{v_1\}$]);
  P[1][$V - \{v_1\}$] = value of j that gave the minimum;
  minlength = D[1][$V - \{v_1\}$];
}
```

Every-case Time Complexity

Basic operation: the instructions executed for each value of v_j in innermost loop.
Input size: n, the number of vertices in the graph.

$$\Rightarrow T(n) = \sum_{k=1}^{n-2} (n-1-k)k \binom{n-1}{k}$$

We can show that $T(n) \in \Theta(n^2 2^n)$

Proof of $T(n) = \sum_{k=1}^{n-2}(n-1-k)k\begin{pmatrix} n-1 \\ k \end{pmatrix} \in \Theta(n^2 2^n)$

$$(n-1-k)\begin{pmatrix} n-1 \\ k \end{pmatrix} = (n-1)\begin{pmatrix} n-2 \\ k \end{pmatrix} \rightarrow T(n) = (n-1)\times$$
$$\sum_{k=1}^{n-2} k\begin{pmatrix} n-2 \\ k \end{pmatrix}$$

On the other hand

$$k\begin{pmatrix} n-2 \\ k \end{pmatrix} = (n-2)\begin{pmatrix} n-3 \\ k-1 \end{pmatrix}$$
$$\rightarrow T(n) = (n-1)(n-2)\sum_{k=1}^{n-2}\begin{pmatrix} n-3 \\ k-1 \end{pmatrix}$$
$$= (n-1)(n-2)\sum_{k=0}^{n-3}\begin{pmatrix} n-3 \\ k \end{pmatrix} = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$

Longest Common Subsequence (LCS)

X=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

Y=GTCGTTCGGAATGCCGTTGCTCTGTAAA

S=GTCGTCGGAAGCCGGCCGAA

X=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

Y=GTCGTTCGGAATGCCGTTGCTCTGTAAA

Input Output

Input description: two character strings, X of size n and Y of size m, over some alphabet

Problem description: find a longest string S that is a subsequence of both X and Y

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called **bases**, where the possible bases are **adenine**, **guanine**, **cytosine**, and **thymine**. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{A, C, G, T\}$.

► For example, the DNA of one organism may be

$$X = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$$

, and the DNA of another organism may be

$$Y = GTCGTTCGGAATGCCGTTGCTCTGTAAA.$$

► One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

We can, and do, define similarity in many different ways:

- **longest Common Sub-String:** Two DNA strands are similar if one is a substring of the other.
- **Minimum Edit Distance:** Two strands are similar if the number of changes needed to turn one into the other is small.
- **Longest Common Subsequence:** finding a third strand S in which the bases in S appear in each of X and Y; these bases must appear in the same order, but not necessarily consecutively.

Problem Definition:
Given a string X of size n, a subsequence of X is any string that is of the form

$$X[i_1]X[i_2]\cdots X[i_k], \quad i_j < i_{j+1} \quad \text{for } j = 1, \cdots, k-1$$

that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X.

The specific text similarity problem we address here is the longest common subsequence (LCS) problem. In this problem, we are given two character strings, X of size n and Y of size m, over some alphabet and are asked to find a longest string S that is a subsequence of both X and Y.

Brute-force:
Enumerate all subsequences of X and take the largest one that is also a subsequence of Y.
Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires $O(m)$ time to determine whether it is a subsequence of Y.
Thus, the brute-force approach yields an exponential algorithm that runs in $O(2^n m)$ time, which is very inefficient.

Dynamic-Programming:
Define $L[i, j]$ as the the length of the longest common subsequence of $X[0..i]$ and $Y[0..j]$

Stop&Think

How to derive a recurrence for $L[i, j]$?

Solution:
We consider the following two cases:

- $X[i] = Y[j]$. Let $c = X[i] = Y[j]$. We claim that a longest common subsequence of$X[0..i]$ and$Y[0..j]$ ends with c (why?). Therefore, we have

$$L[i,j] = L[i-1,j-1]+1, \quad \text{if } X[i] = Y[j].$$

- $X[i] \neq Y[j]$. In this case, we cannot have a common subsequence that includes both $X[i]$ and $Y[j]$. That is, a common subsequence can end with $X[i]$, $Y[j]$, or neither, but not both. Therefore, we set

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \quad \text{if } X[i] \neq Y[j]$$

$$L[-1,.] = 0, L[.,-1] = 0$$

```
Algorithm LCS($X, Y$):
  Input: Strings $X$ and $Y$ with $n$ and $m$ elements, respectively
  Output: For $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$, the length $L[i, j]$ of a longest
    common subsequence of $X[0..i]$ and $Y[0..j]$
  for $i \leftarrow -1$ to $n-1$ do
      $L[i, -1] \leftarrow 0$
  for $j \leftarrow 0$ to $m-1$ do
      $L[-1, j] \leftarrow 0$
  for $i \leftarrow 0$ to $n-1$ do
      for $j \leftarrow 0$ to $m-1$ do
          if $X[i] = Y[j]$ then
              $L[i, j] \leftarrow L[i-1, j-1] + 1$
          else
              $L[i, j] \leftarrow \max\{L[i-1, j], L[i, j-1]\}$
  return array $L$
```

Every Case Time Complexity:

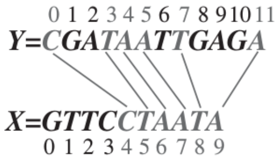
The running time of the Algorithm is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Construction Step:
Given the table of L[i, j] values, constructing a longest common subsequence is straightforward. start from $L[n-1, m-1]$ and work back through the table, reconstructing a longest common subsequence from back to front. At any position $L[i, j]$,

- If $X[i] = Y[j]$, then we take $X[i]$ as the next character of the subsequence (noting that X[i] is before the previous character we found, if any), moving next to $L[i-1, j-1]$.
- If $X[i] \neq Y[j]$, then we move to the larger of$L[i, j-1]$ and $L[i-1, j]$.
- We stop when we reach a boundary entry (with $i = -1$ or $j = -1$).

This method constructs a longest common subsequence in $O(n+m)$ additional time since it decrements at least one of i and j in each step.

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6


0 1 2 3 4 5 6 7 8 9 10 11
Y=CGATAATTGAGA
X=GTTCCTAATA
0 1 2 3 4 5 6 7 8 9