

CS332

Dynamic Programming

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Dynamic Programming algorithm

- *Dynamic Programming* is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “**planning**”
- Main idea:
 - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table

Introduction

- Dynamic Programming(DP) applies to optimization problems in which a set of choices must be made in order to arrive at an optimal solution.
- As choices are made, subproblems of the same form arise.
- DP is effective when a given problem may arise from more than one partial set of choices.
- The *key technique* is to store the solution to each subproblem in case it should appear

Introduction (cont.)

- Divide and Conquer algorithms partition the problem into independent subproblems.
- Dynamic Programming is applicable when the subproblems are not independent.(In this case DP algorithm does more work than necessary)
- Dynamic Programming algorithm solves every subproblem just once and then saves its answer in a table.

Dynamic Programming (DP) vs. Divide-and-Conquer

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Differences between divide-and-conquer and DP:
 - **Independent** sub-problems, solve sub-problems **independently** and **recursively**, (so same sub(sub)problems solved **repeatedly**)
 - Sub-problems are **dependent**, i.e., sub-problems **share** sub-sub-problems, every sub(sub)problem solved **just once**, solutions to sub(sub)problems are **stored in a table** and used for solving higher level sub-problems.

Application domain of DP

- Optimization problem: find a solution with optimal (maximum or minimum) value.
- *An* optimal solution, not *the* optimal solution, since may more than one optimal solution, any one is OK.

Typical steps of DP

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Compute an optimal solution from computed/stored information.

The steps 4 may be omitted if only the value of an optimal solution is required

Dynamic Programming Applications

- **Areas.**
 - Bioinformatics.
 - Control theory.
 - Information theory.
 - Operations research.
 - Computer science: theory, graphics, AI, systems,
- **Some famous dynamic programming algorithms.**
 - **Viterbi** for hidden Markov models.
 - Unix *diff* for comparing two files.
 - Smith-Waterman for **sequence alignment**.
 - Bellman-Ford for **shortest path routing** in networks.
 - Cocke-Kasami-Younger for **parsing** context free grammars.

Recursive routine for n^{th} Fibonacci numbers

Recursive routine for n^{th} Fibonacci numbers

- Recall definition of Fibonacci numbers:

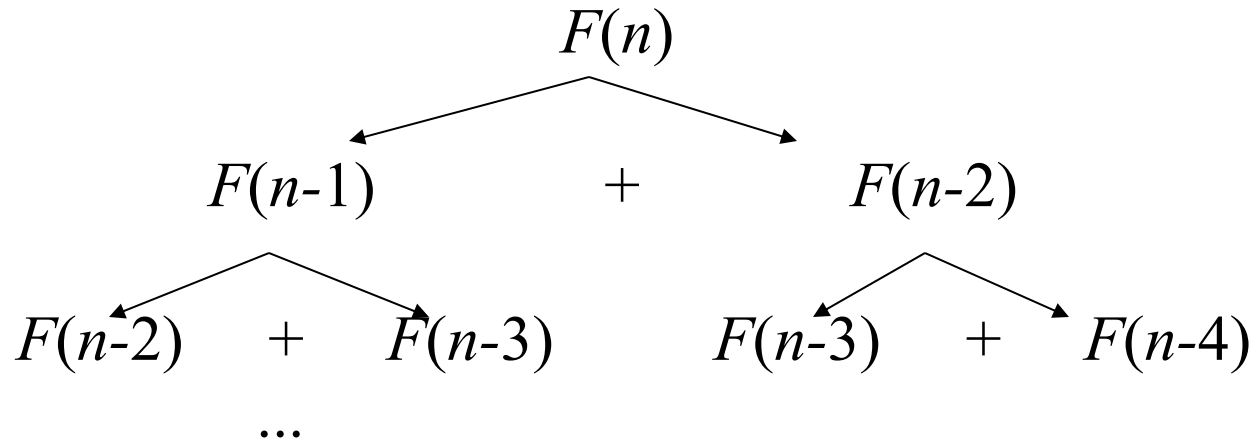
$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

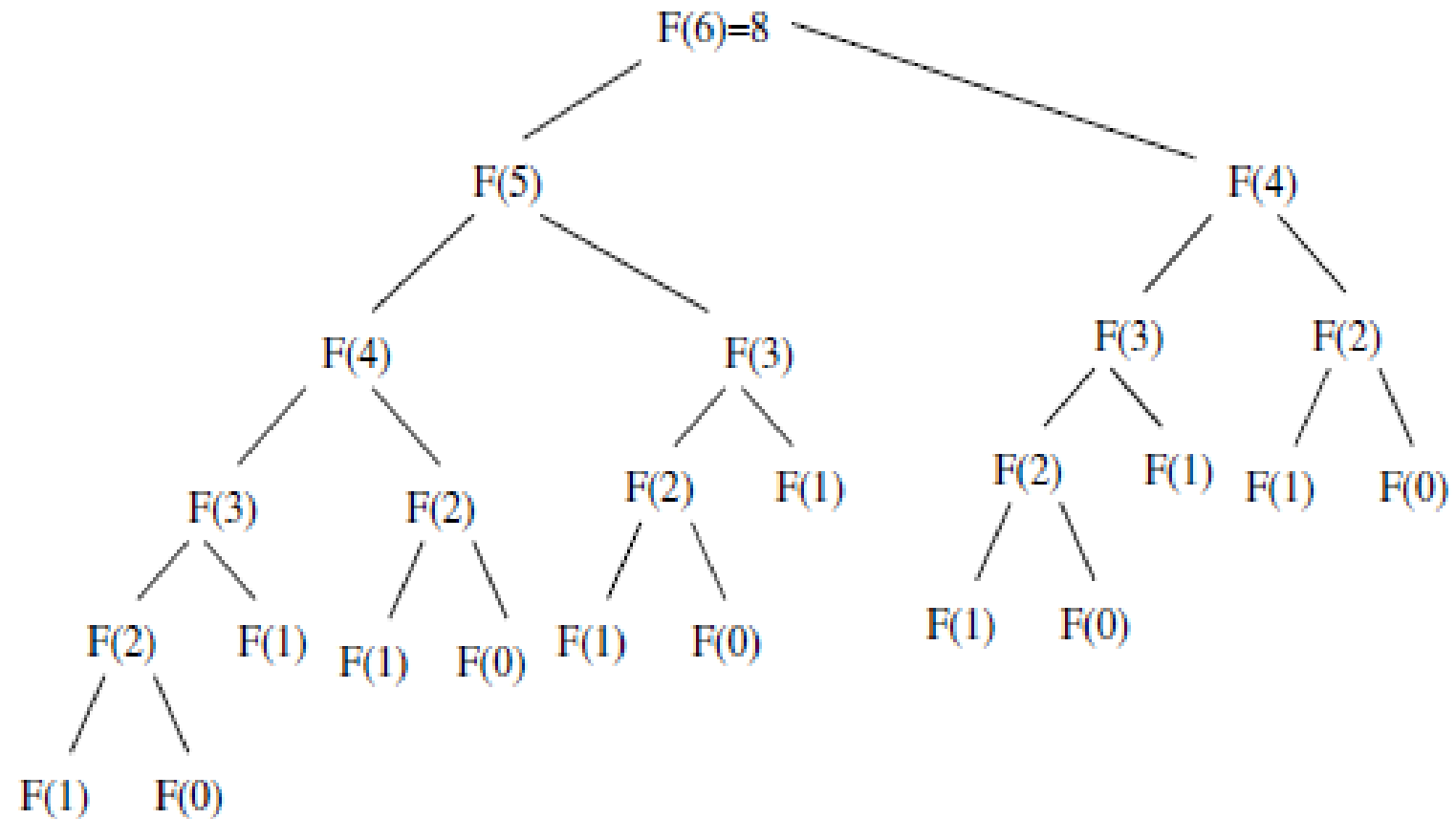
$$F(1) = 1$$

```
int Fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return Fib(n - 1) + Fib(n - 2);
}
```

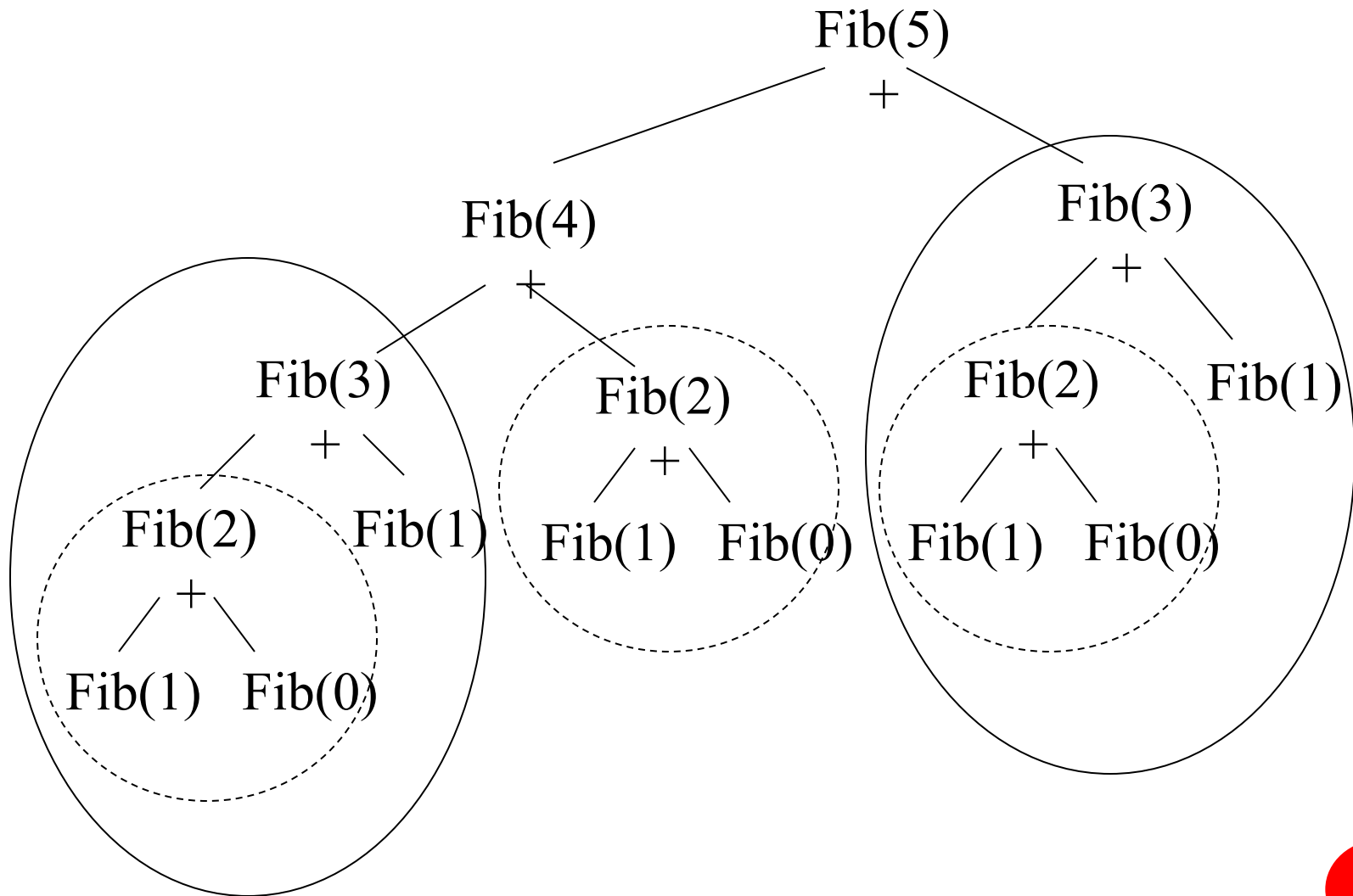
- Computing the n^{th} Fibonacci number recursively (top-down):



The computation tree for computing Fibonacci numbers recursively



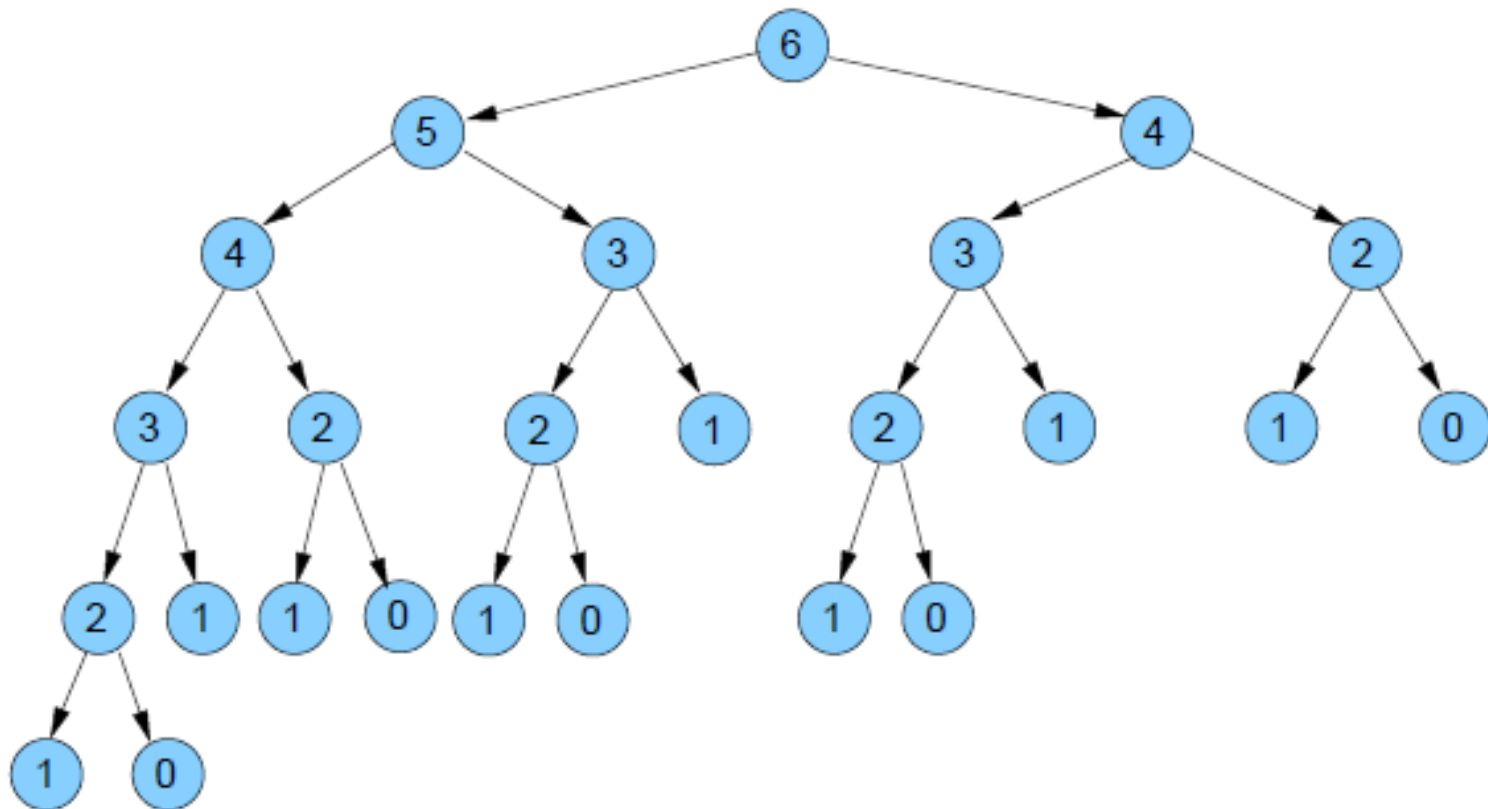
The computation tree for computing Fibonacci numbers recursively



Complexity of Recursive routine for n^{th} Fibonacci numbers

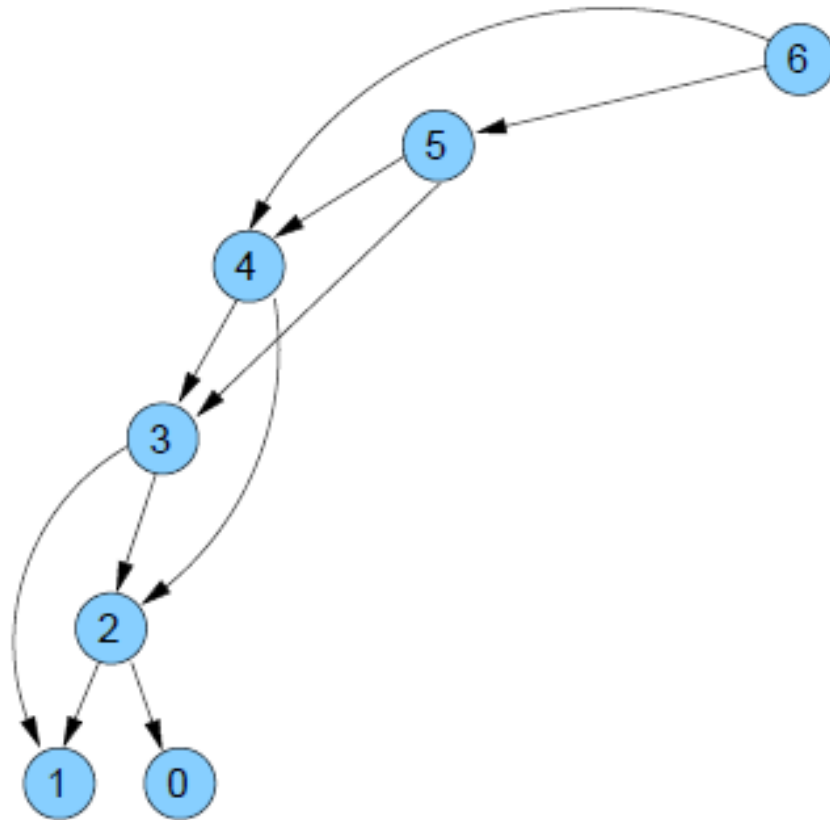
- What is the Recurrence relationship?
 - $T(n) = T(n-1) + T(n-2) + 1$
- What is the solution to this?
 - Clearly it is $O(2^n)$, but this is not tight.
 - A lower bound is $\Omega(2^{n/2})$.
 - You should notice that $T(n)$ grows very similarly to $F(n)$, so in fact $T(n) = \Theta(F(n))$.
- Obviously not very good, but we know that there is a better way to solve it!

Subproblem tree for Fibonacci function



vertex labels are the parameters of the recursive calls

Subproblem graph for Fibonacci function



note that this is a dependency graph

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1+0 = 1, \quad \dots, \quad F(n-2) = **, \quad F(n-1) = **$$

$$F(n) = F(n-1) + F(n-2)$$

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$
---	---	---	-------	----------	----------	--------

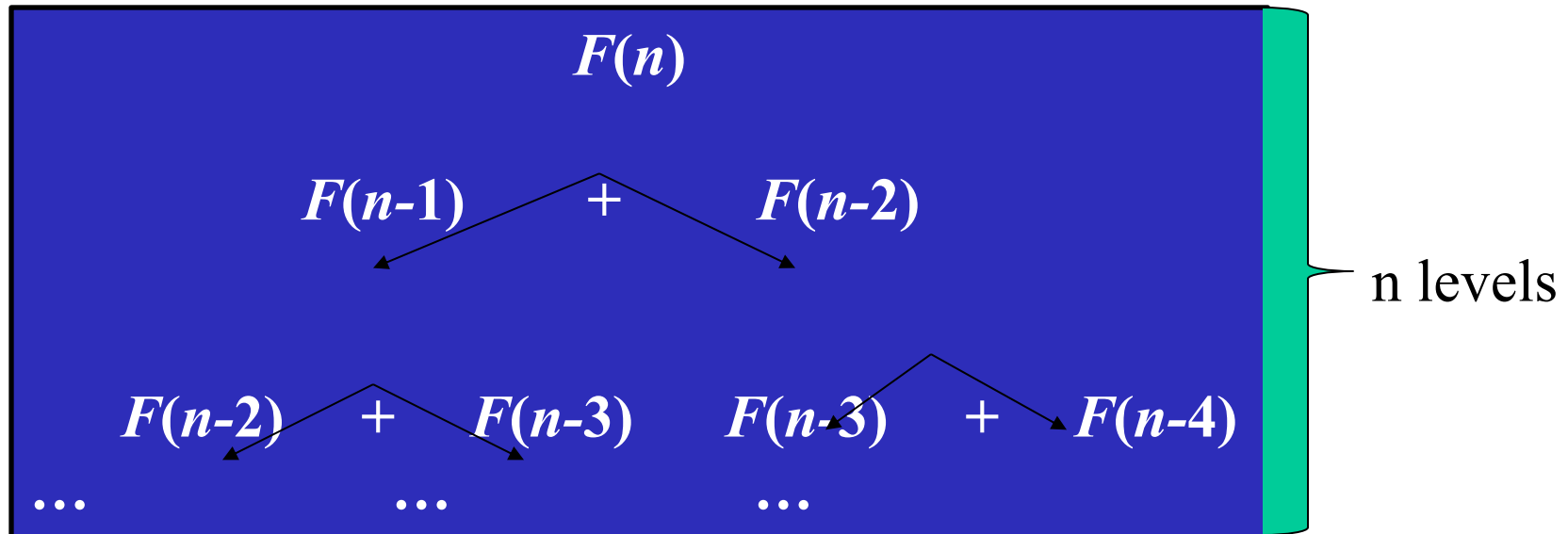
Efficiency:

Time – $O(n)$

Space – $O(n)$

Example: Fibonacci numbers (cont.)

- The bottom-up approach is only $\Theta(n)$.
- Why is the top-down so inefficient?
 - Recomputes many sub-problems.
 - How many times is $F(n-5)$ computed?



Excercise

- Write three different algorithm to compute nth Fibonacci number and comment on its time and space complexity

Problem solving

In dynamic programming

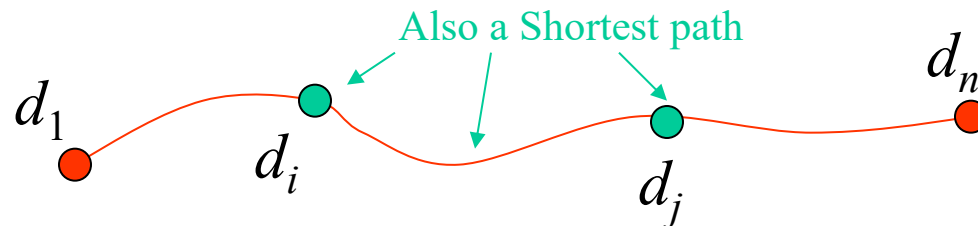
Dynamic Programming

- It uses a “bottom-up” approach in that the subproblems are arranged and solved in a systematic fashion, which leads to a solution to the original problem.
- This bottom-up approach implementation is more efficient than a “top-down” counterpart mainly because duplicated computation of the same problems is eliminated.
- This technique is typically applied to solving optimization problems, although it is not limited to only optimization problems.
- Dynamic programming typically involves two steps:
 - (1) **develop a recursive strategy for solving the problem**
 - (2) **develop a “bottom-up” implementation without recursion.**

The Principle of Optimality:

In solving optimization problems which require making a sequence of decisions, such as the change making problem, we often apply the following principle in setting up a recursive algorithm: Suppose an optimal solution made decisions d_1, d_2 , and ..., d_n . The subproblem starting after decision point d_i and ending at decision point d_j , also has been solved with an optimal solution made up of the decisions d_i through d_j . That is, any subsequence of an optimal solution constitutes an optimal sequence of decisions for the corresponding subproblem. This is known as the principle of optimality which can be illustrated by the shortest paths in weighted graphs as follows:

A shortest path from d_1 to d_n

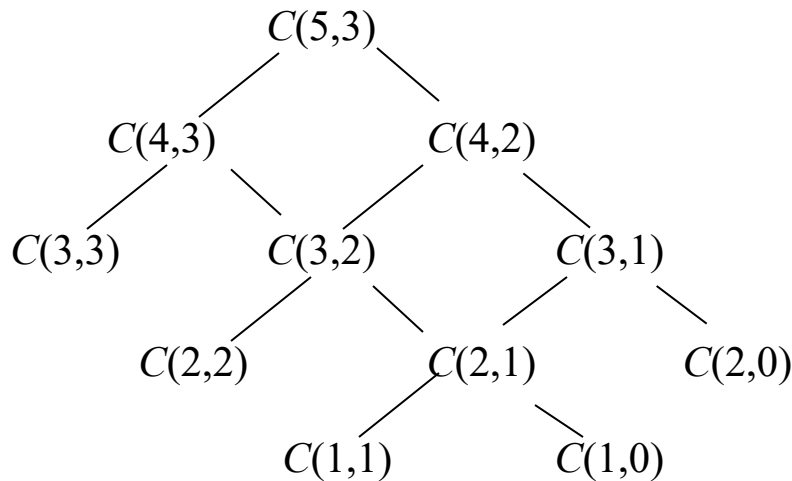


Binomial coefficients $C(n, k)$

Example: Compute the **binomial coefficients** $C(n, k)$ defined by the following recursive formula:

$$C(n, k) = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n; \\ C(n-1, k) + C(n-1, k-1), & \text{if } 0 < k < n; \\ 0, & \text{otherwise.} \end{cases}$$

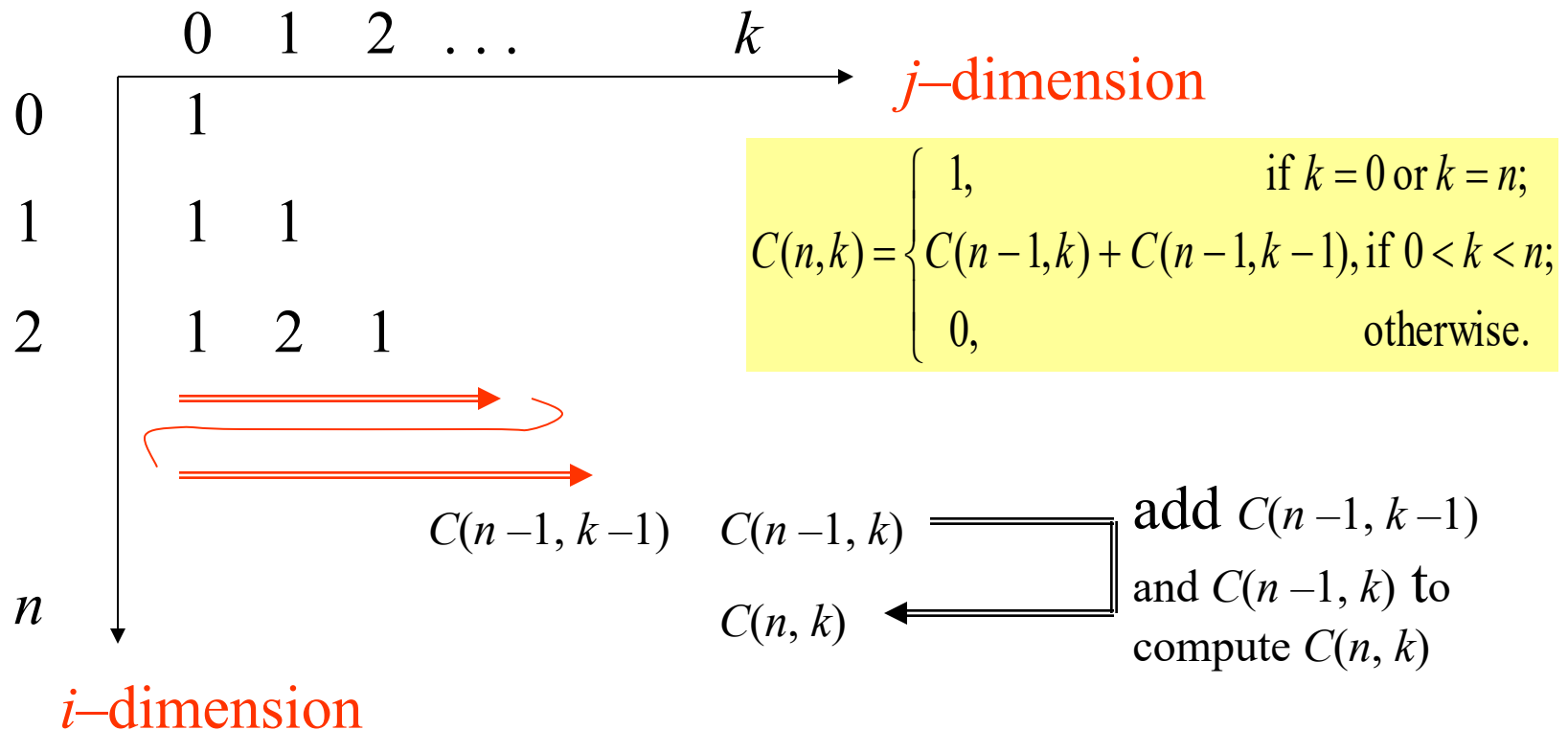
The following “**call tree**” demonstrates repeated (duplicated) computations in a straightforward recursive implementation:



Notice repeated calls to $C(3, 2)$ and to $C(2, 1)$.

In general, the number of calls for computing $C(n, k)$ is $2C(n, k) - 1$, which can be exponentially large.

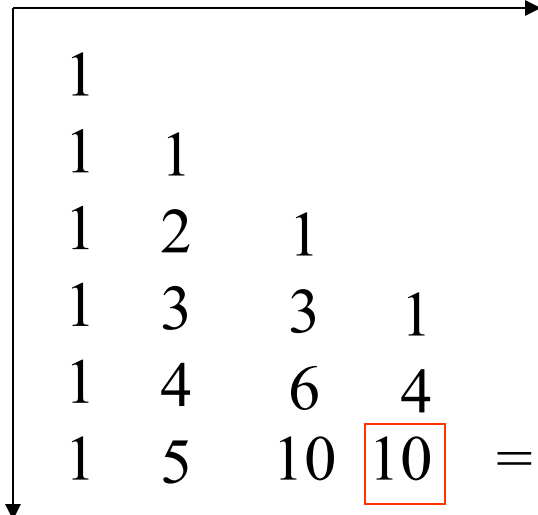
A more efficient way to compute $C(n, k)$ is to organize the computation steps of $C(i, j)$, $0 \leq i \leq n$ and $0 \leq j \leq k$, in a tabular format and compute the values by rows (the i -dimension) and within the same row by columns (the j -dimension):



It can be seen that the number of steps (add operation) is $O(nk)$ in computing $C(n, k)$, using $O(nk)$ amount of space (I.e., the table).

In fact, since only the previous row of values are needed in computing the next row of the table, space for only two rows is needed reducing the space complexity to $O(k)$.

The following table demonstrates the computation steps for calculating $C(5,3)$:

	0	1	2	3	
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	
5	1	5	10	10	
					$= C(5,3)$

Note that this table shows **Pascal's triangle** in computing the binomial coefficients.

Computing $C(n, k)$: pseudo code

ALGORITHM *Binomial*(n, k)

//Computes $C(n, k)$ by the dynamic programming algorithm

//Input: A pair of nonnegative integers $n \geq k \geq 0$

//Output: The value of $C(n, k)$

for $i \leftarrow 0$ **to** n **do**

for $j \leftarrow 0$ **to** $\min(i, k)$ **do**

if $j = 0$ **or** $j = i$

$C[i, j] \leftarrow 1$

else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

return $C[n, k]$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Make-change problem

Example: Solve the **make-change problem** using dynamic programming. Suppose there are n types of coin denominations, d_1, d_2, \dots , and d_n . (We may assume one of them is penny.) There are an infinite supply of coins of each type. To make change for an arbitrary **amount** j using the minimum number of coins, we first apply the following recursive idea:

If there are only pennies, the problem is simple: simply use j pennies to make change for the total amount j . More generally, if there are coin types 1 through i , let $C[i, j]$ stands for the minimum number of coins for making change of amount j .

By considering coin denomination i , there are two cases: either we use at least one coin denomination i , or we don't use coin type i at all.

In the **first case**, the total number of coins must be $1 + C[i, j - d_i]$ because the total amount is reduced to $j - d_i$ after using one coin of amount d_i , the rest of coin selection from the solution of $C[i, j]$ must be an optimal solution to the reduced problem with reduced amount, still using coin types 1 through i .

In the **second case**, i.e., suppose no coins of denomination i will be used in an optimal solution. Thus, the best solution is identical to solving the same problem with the total amount j but using coin types 1 through $i - 1$, i.e. $C[i - 1, j]$. Therefore, the overall best solution must be the better of the two alternatives, resulting in the following recurrence:

$$C[i, j] = \min (1 + C[i, j - d_i], C[i - 1, j])$$

The boundary conditions are when $i \leq 0$ or when $j < 0$ (in which case let $C[i, j] = \infty$), and when $j = 0$ (let $C[i, j] = 0$).

Example: There are 3 coin denominations $d_1 = 1$, $d_2 = 4$, and $d_3 = 6$, and the total amount to make change for is $K = 8$. The following table shows how to compute $C[3,8]$, using the recurrence as a basis but arranging the computation steps in a tabular form (by rows and within the row by columns):

$$C[i, j] = \min (1 + C[i, j - d_i], C[i - 1, j])$$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Boundary condition for amount $j = 0$

$C[3, 8 - 6]$ $C[3, 8] = \min(1 + C[3, 8 - 6], C[2, 8])$

$C[2, 8]$

Note the time complexity for computing $C[n, K]$ is $O(nK)$, using space $O(K)$ by maintaining last two rows.

The 0–1 Knapsack Problem

The 0–1 Knapsack Problem:

Given n objects 1 through n , each object i has an integer weight w_i and a real number value v_i , for $1 \leq i \leq n$. There is a knapsack with a total integer capacity W . The 0–1 knapsack problem attempts to fill the sack with these objects within the weight capacity W while **maximizing** the total value of the objects included in the sack, where an object is totally included in the sack or no portion of it is in at all. That is, solve the following optimization problem with $x_i = 0$ or 1, for $1 \leq i \leq n$:

$$\text{Maximize } \sum_{i=1}^n x_i v_i \text{ subject to } \sum_{i=1}^n x_i w_i \leq W.$$

To solve the problem using dynamic programming, we first define a notation (expression) and derive a recurrence for it.

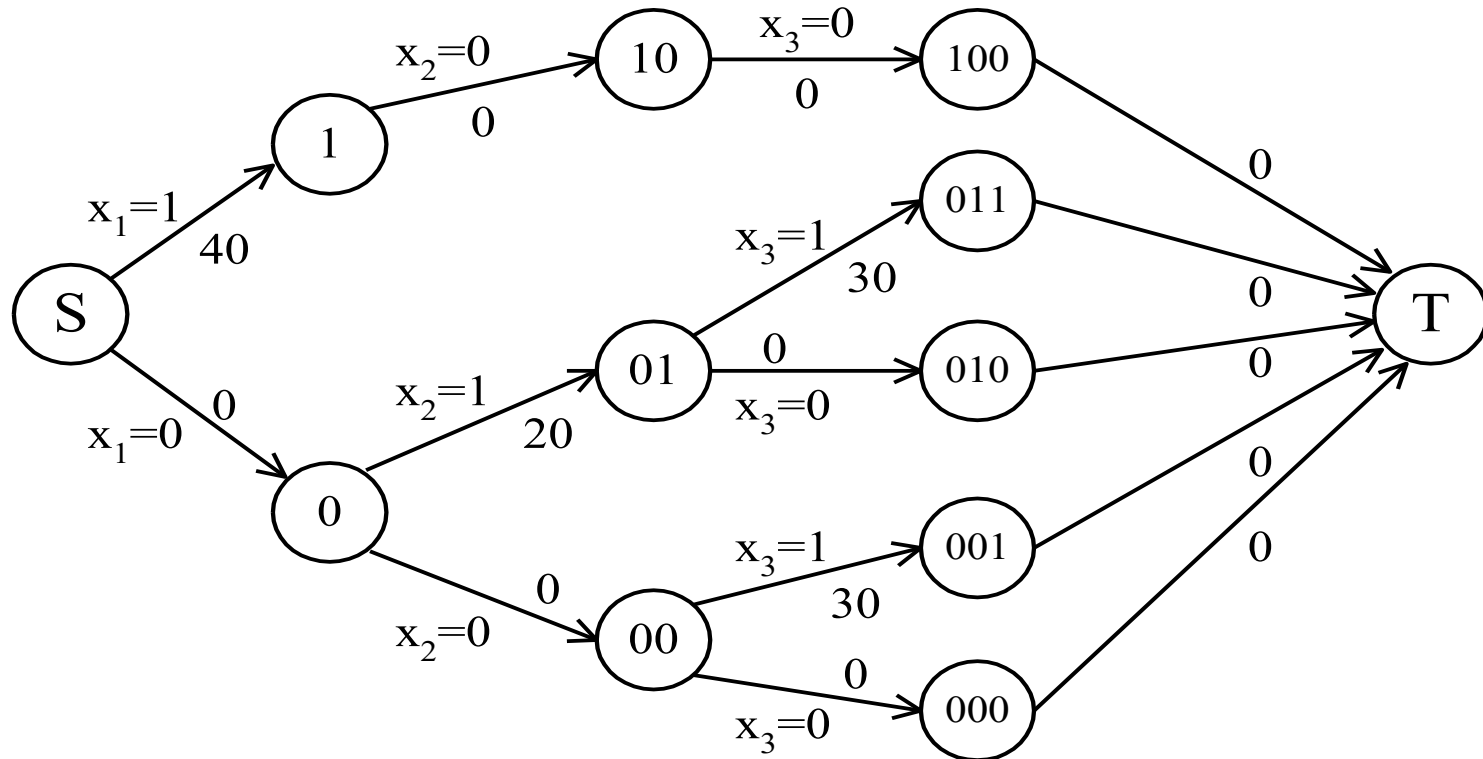
Knapsack problem

□ *Knapsack problem* : Suppose we have n integers a_1, a_2, \dots, a_n and a constant W . We want to find a subset of integers so that their sum is less than or equal to but is as close to W as possible. There are 2^n subsets. ($n = 100$, $2^n = 1.26765 \times 10^{30}$, it takes 4.01969×10^{12} years if our computer can examine 10^{10} subsets per second.)

□ A problem is considered *tractable* (computationally easy, $O(n^k)$) if it can be solved by an efficient algorithm and *intractable* (computationally difficult, the lower bound grows fast than n^k) if there is no efficient algorithm for solving it.

□ The class of *NP-complete* problems: There is a class of problems, including TSP and Knapsack, for which no efficient algorithm is currently known.

Multistage Graph representation of 0/1 Knapsack problem



i	W_i	P_i
1	10	40
2	3	20
3	5	30

M=10

Let $V[i, j]$ denote the maximum value of the objects that fit in the knapsack, selecting objects from 1 through i with the sack's weight capacity equal to j .

To find $V[i, j]$ we have two choices concerning the decisions made on object i (in the optimal solution for $V[i, j]$): We can either ignore object i or we can include object i .

In the **former case**, the optimal solution of $V[i, j]$ is identical to the optimal solution to using objects 1 through $i - 1$ with sack's capacity equal to W (by the definition of the V notation).

In the **latter case**, the parts of the optimal solution of $V[i, j]$ concerning the choices made to objects 1 through $i - 1$, must be an optimal solution to $V[i - 1, j - w_i]$, an application of the principle of optimality. Thus, we have derived the following recurrence for $V[i, j]$:

$$V[i, j] = \max(V[i - 1, j], v_i + V[i - 1, j - w_i])$$

The boundary conditions are $V[0, j] = 0$ if $j \geq 0$, and $V[i, j] = -\infty$ when $j < 0$.

The problem can be solved using dynamic programming (i.e., a bottom-up approach to carrying out the computation steps) based on a tabular form when the weights are integers.

Example: There are $n = 5$ objects with integer weights $w[1..5] = \{1, 2, 5, 6, 7\}$, and values $v[1..5] = \{1, 6, 18, 22, 28\}$. The following table shows the computations leading to $V[5, 11]$ (i.e., assuming a knapsack capacity of 11).

Sack's capacity		0	1	2	3	4	5	6	7	8	9	10	11
w_i	v_i												
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	6	7	7	18	19	24	25	25	25	25
6	22	0	1	6	7	7	18	22	24	28	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	40

Time: $O(nW)$
 space: $O(W)$

$V[3, 8]$ (pointing to 25)

$V[4, 8] = \max(V[3, 8], 22 + V[3, 2])$ (pointing to 28)

$V[3, 8 - w_4] = V[3, 2]$ (pointing to 6)

Knapsack Problem by DP (pseudocode)

1. Algorithm DPKnapsack($w[1..n]$, $v[1..n]$, W)
2. var $V[0..n, 0..W]$, $P[1..n, 1..W]$: int
3. for $j := 0$ to W do
4. $V[0, j] := 0$
5. for $i := 0$ to n do
6. $V[i, 0] := 0$
7. for $i := 1$ to n do
8. for $j := 1$ to W do
9. if $w[i] \leq j$ and $v[i] + V[i-1, j-w[i]] > V[i-1, j]$ then
10. $V[i, j] := v[i] + V[i-1, j-w[i]]$; $P[i, j] := j-w[i]$
11. else
12. $V[i, j] := V[i-1, j]$; $P[i, j] := j$
13. return $V[n, W]$ and the optimal subset by backtracing

Suggested questions

- Write an dynamic programming algorithm to find optimal solution to 0-1 knapsack problem for n items with time complexity $O(nW)$; for the knapsack capacity W .
- Write an dynamic programming algorithm to find a solution to 0-1 knapsack problem that yields maximum profit P for n items with the knapsack capacity W .

0/1 knapsack problem

E Horowitz, S. Sahni, S. Rajasekaran

Dynamic Programming algorithm for 0/1 Knapsack problem
with space complexity $O(w)$

Algorithm *01Knapsack*(S, W):

Input: set S of n items with benefit p_i and weight w_i ;
maximum weight W

Output: benefit of best subset of S with weight at most W

1. let A and B be arrays of length $W + 1$
2. for $w \leftarrow 0$ to W do
3. $B[w] \leftarrow 0$
4. for $k \leftarrow 1$ to n do
5. copy array B into array A
6. for $w \leftarrow w_k$ to W do
7. if $A[w - w_k] + p_k > A[w]$ then
8. $B[w] \leftarrow A[w - w_k] + p_k$
9. return $B[W]$

The 0/1/2 Knapsack Problem:

Given n objects 1 through n , each object i has an integer weight w_i and a real number value v_i , for $1 \leq i \leq n$. There is a knapsack with a total integer capacity W . The 0–1–2 knapsack problem attempts to fill the sack with these objects within the weight capacity W while **maximizing** the total value of the objects included in the sack, where an object selection is based upon three values. That is, solve the following optimization problem with $x_i = 0$ or 1 or 2 , for $1 \leq i \leq n$:

$$\text{Maximize } \sum_{i=1}^n x_i v_i \text{ subject to } \sum_{i=1}^n x_i w_i \leq W.$$

Write an algorithm to solve the problem using dynamic programming.

Two dimensional Knapsack Problem:

Given n objects 1 through n , each object i has an integer weight w_i and two real number value v_i and u_i for $1 \leq i \leq n$. There is a knapsack with a total integer capacity W . That is to solve the following optimization problem with $x_i = 0$ or 1 , for $1 \leq i \leq n$:

$$\text{Maximize } \sum_{i=1}^n x_i v_i \text{ subject to } \sum_{i=1}^n x_i w_i \leq W.$$

$$\text{and } \sum_{i=1}^n x_i u_i \leq C.$$

Write an algorithm to solve the problem using dynamic programming.

The Partition Problem

The Partition Problem:

Given a set of positive integers, $A = \{a_1, a_2, \dots, a_n\}$.

The question is to select a subset B of A such that the sum of the numbers in B equals the sum of the numbers not in B , i.e.,

$$\sum_{a_i \in B} a_i = \sum_{a_j \in A-B} a_j$$

We may assume that the sum of all numbers in A is $2K$, an even number. We now propose a dynamic programming solution. For $1 \leq i \leq n$ and $0 \leq j \leq K$,

define $P[i, j]$ = True if there exists a subset of the first i numbers a_1 through a_i whose sum equals j ;
False otherwise.

Thus, $P[i, j]$ = True if either $j = 0$ or if $(i = 1 \text{ and } j = a_1)$.

The Partition Problem:

- $P[i, j] = \text{True}$ if there exists a subset of the first i numbers a_1 through a_i whose sum equals j ; False otherwise.
- Thus, $P[i, j] = \text{True}$ if either $j = 0$ or if $(i = 1 \text{ and } j = a_1)$.

When $i > 1$, we have the following recurrence:

$$P[i, j] = P[i - 1, j] \text{ or } (P[i - 1, j - a_i] \text{ if } j - a_i \geq 0)$$

- That is, in order for $P[i, j]$ to be true, either there exists a subset of the first $i - 1$ numbers whose sum equals j , or whose sum equals $j - a_i$ (this latter case would use the solution of $P[i - 1, j - a_i]$ and add the number a_i).
- The value $P[n, K]$ is the answer.

Example: Suppose $A = \{2, 1, 1, 3, 5\}$ contains 5 positive integers. The sum of these number is $2+1+1+3+5=12$, an even number. The partition problem computes the truth value of $P[5, 6]$ using a tabular approach as follows:

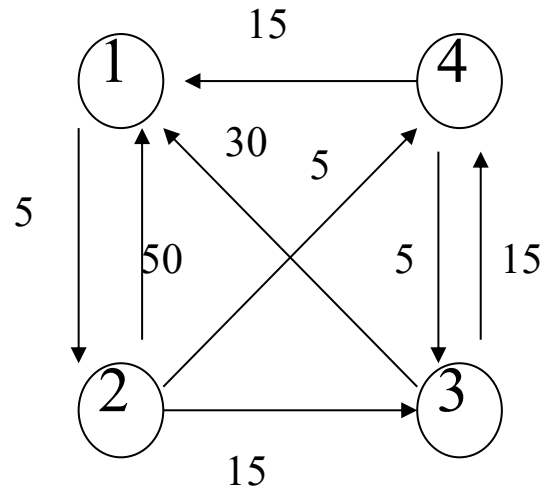
$i \backslash j$	0	1	2	3	4	5	6
$a_1 = 2$	T	F	T	F	F	F	F
$a_2 = 1$	T	T	T	T	F	F	F
$a_3 = 1$	T	T	T	T	T	F	F
$a_4 = 3$	T	T	T	T	T	T	T
$a_5 = 5$	T	T	T	T	T	T	T

Because $j \neq a_1$

Always true

$P[4,5] = (P[3,5]$
or $P[3, 5 - a_4]$)

The time complexity is $O(nK)$; the space complexity $O(K)$.



All-Pairs Shortest Paths Problem:

All-Pairs Shortest Paths Problem:

- Given a weighted, directed graph represented in its weight matrix form $W[1..n][1..n]$, where n = the number of nodes, and $W[i][j]$ = the edge weight of edge (i, j) .
- The problem is find a shortest path between every pair of the nodes.
- Find the “shortest path” from a to b (where the length of the path is the sum of the edge weights on the path).
- Perhaps we should call this the **minimum weight path**!

All Pairs Shortest Paths (APSP)

- **given** : directed graph $G = (V, E)$,
weight function $\omega : E \rightarrow \mathbb{R}$, $|V| = n$
- **goal** : create an $n \times n$ matrix $D = (d_{ij})$ of shortest path distances
i.e., $d_{ij} = \delta(v_i, v_j)$
- **trivial solution** : run a SSSP algorithm n times, one for each vertex as the source.

All Pairs Shortest Paths (APSP)

- ▶ **all edge weights are nonnegative : use **Dijkstra's algorithm****
 - PQ = linear array : $O(V^3 + VE) = O(V^3)$
 - PQ = binary heap : $O(V^2 \lg V + EV \log V) = O(V^3 \log V)$
for dense graphs
 - better only for sparse graphs
 - PQ = fibonacci heap : $O(V^2 \log V + EV) = O(V^3)$
for dense graphs
 - better only for sparse graphs
- ▶ **negative edge weights : use **Bellman-Ford algorithm****
 - $O(V^2 E) = O(V^4)$ on dense graphs

Dijkstra's algorithm

- **Dijkstra's algorithm**, conceived by computer scientist **Edsger Dijkstra** in 1956 and published in 1959, is a **graph search algorithm** that solves the **single-source shortest path problem** for a graph with non-negative edge path costs, producing a shortest path tree.

Bellman–Ford algorithm

- The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.
- It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are **negative numbers**.
- The algorithm is usually named after two of its developers, **Richard Bellman** and **Lester Ford, Jr.**, who published it in 1958 and 1956, respectively.
- **Edward F. Moore** also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**.¹

Adjacency Matrix Representation of Graphs

► $n \times n$ matrix $\mathbf{W} = (\omega_{ij})$ of edge weights :

$$\omega_{ij} = \begin{cases} \omega(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{if } (v_i, v_j) \notin E \end{cases}$$

► assume $\omega_{ii} = 0$ for all $v_i \in V$, because

- no neg-weight cycle

\Rightarrow shortest path to itself has no edge,

i.e., $\delta(v_i, v_i) = 0$

Why not Greedy algorithm?

- Start at **a**, and greedily construct a path that goes to **w** by adding vertices that are closest to the current endpoint, until you reach **b**.
- Problem: it doesn't work correctly! Sometimes you can't reach **b** at all, and would have to backtrack... and sometimes you get a path that doesn't have the minimum weight.

Designing a DP solution

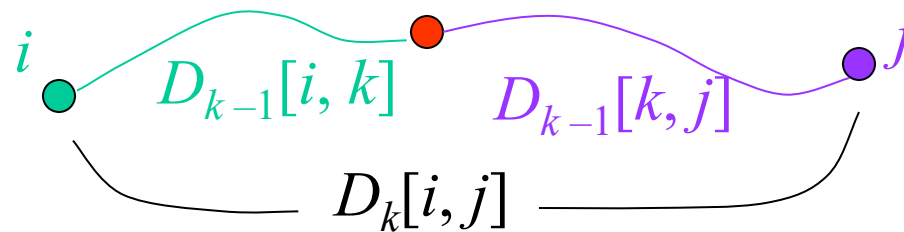
- How are the sub-problems defined?
- Where is the answer stored?
- How are the boundary values computed?
- How do we compute each entry from other entries?
- What is the order in which we fill in the matrix?

Floyd's algorithm

- In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, **Roy–Warshall algorithm**, **Roy–Floyd algorithm**, or the **WFI algorithm**)
- This is a graph analysis algorithm for finding shortest paths in a weighted graph with **positive** or **negative** edge weights (but with no negative cycles).
- This algorithm can also be used for finding transitive closure of a relation .
- A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves.

All-Pairs Shortest Paths Problem:

- We first note that the principle of optimality applies:
- If node k is on a shortest path from node i to node j , then the subpath from i to k , and the subpath from k to j , are also shortest paths for the corresponding end nodes.



- Therefore, the problem of finding shortest paths for all pairs of nodes becomes developing a strategy to compute these shortest paths in a systematic fashion.

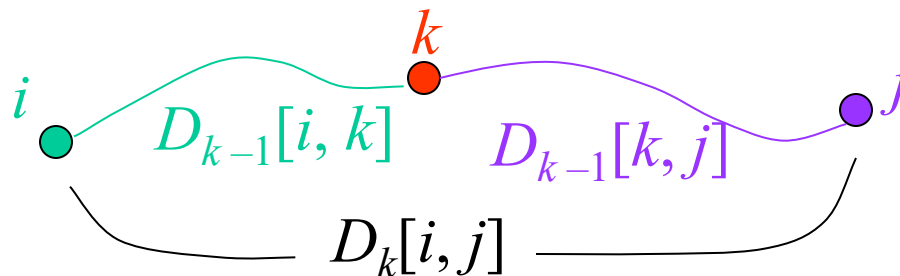
Floyd's Algorithm: Define the notation $D_k[i, j]$, $1 \leq i, j \leq n$, and $0 \leq k \leq n$, that stands for the shortest distance (via a shortest path) from node i to node j , passing through nodes whose number (label) is $\leq k$. Thus, when $k = 0$, we have

$$D_0[i, j] = W[i][j] = \text{the edge weight from node } i \text{ to node } j$$

This is because no nodes are numbered ≤ 0 (the nodes are numbered 1 through n). In general, when $k \geq 1$,

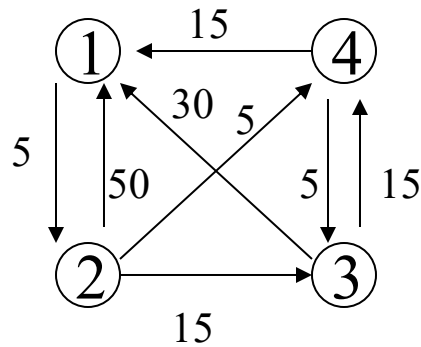
$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

The reason for this recurrence is that when computing $D_k[i, j]$, this shortest path either doesn't go through node k , or it passes through node k exactly once. The former case yields the value $D_{k-1}[i, j]$; the latter case can be illustrated as follows:



Example: We demonstrate Floyd's algorithm for computing $D_k[i, j]$ for $k = 0$ through $k = 4$, for the following weighted directed graph:

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$



$$D_0 = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

reduced from ∞ because the path (3,1,2) going thru node 1 is possible in D_1

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Implementation of Floyd's Algorithm:

Input: The weight matrix $W[1..n][1..n]$ for a weighted directed graph, nodes are labeled 1 through n .

Output: The shortest distances between all pairs of the nodes, expressed in an $n \times n$ matrix.

Algorithm:

Create a matrix D and initialize it to W .

for $k = 1$ to n do

 for $i = 1$ to n do

 for $j = 1$ to n do

$D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$

Note that one single matrix D is used to store D_{k-1} and D_k , i.e., updating from D_{k-1} to D_k is done immediately. This causes no problems because in the k th iteration, the value of $D_k[i, k]$ should be the same as it was in $D_{k-1}[i, k]$; similarly for the value of $D_k[k, j]$. The time complexity of the above algorithm is $O(n^3)$ because of the triple-nested loop; the space complexity is $O(n^2)$ because only one matrix is used.

Function to compute lengths of shortest paths:

Algorithm *Allpaths*(W, A, n)

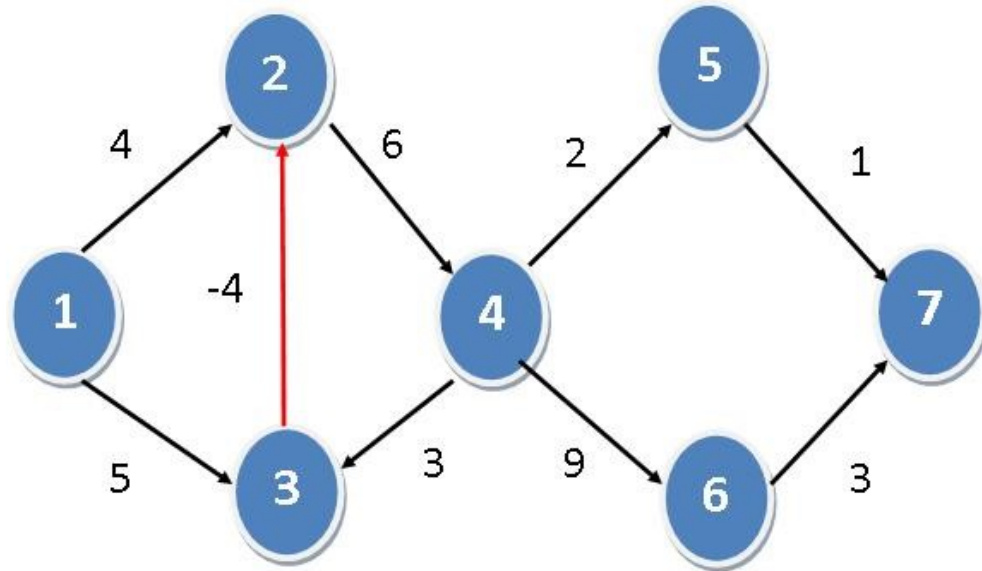
Input: The weight matrix $W[1..n][1..n]$ for a weighted directed graph, nodes are labeled 1 through n .

Output: The matrix $A[1..n][1..n]$ shortest distances between all pairs of the nodes, expressed in an $n \times n$ matrix.

1. for $i = 1$ to n do
2. for $j = 1$ to n do
3. $A[i][j] = W[i][j]$; // Copy cost to A
4. for $k = 1$ to n do
5. for $i = 1$ to n do
6. for $j = 1$ to n do
7. $A[i][j] = \min(A[i][j], D[i][k] + A[k][j]);$
8. }

Exercise

- Write an algorithm that output a shortest path for each pair of vertices (i, j) in directed graph $G(V, E)$ in represented as adjacency matrix?
- Is it possible to obtain all pair shortest path for the following graph



- Can the Floyd's Algorithm applicable for the graph with cycles of negative length.

Elements of Dynamic Programming



Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting **space of subproblems**.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
 - 1. *How many subproblems* are used in an optimal solution.
 - 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) \times (# of choices).
- How many subproblems and choices do the examples considered contain?
- Dynamic programming uses optimal substructure **bottom up**.
 - *First* find optimal solutions to subproblems.
 - *Then* choose which to use in optimal solution to the problem.

Optimal Substructure

- Does optimal substructure apply to all optimization problems? No.
- Applies to determining the **shortest path** but **NOT** the **longest simple path** of an unweighted directed graph.
- Why?
 - **Shortest path has independent subproblems.**
 - Solution to one subproblem does not affect solution to another subproblem of the same problem.
 - **Subproblems are not independent in longest simple path.**
 - Solution to one subproblem affects the solutions to other subproblems.
 - **Example:**

Overlapping Subproblems

- The space of subproblems must be “small”.
- The total number of distinct subproblems is a polynomial in the input size.
 - A recursive algorithm is exponential because it solves the same problems repeatedly.
 - If divide-and-conquer is applicable, then each problem solved will be brand new.

Dynamic programming recipe

1. Tackle the problem “top-down”; this yields a recursive algorithm
2. Define an appropriate dictionary; transform the recursive solution into a DP algorithm
3. Complexity of DP-algorithm is complexity DFS on subproblem graph
4. Choose an appropriate data structure for implementing the dictionary
5. If possible, analyse the subproblem graph and find a reverse **topological** order; simplify your DP-algorithm accordingly
6. Decide how to get the solution to the problem from the data in the dictionary

Dynamic Programming Topics

- The General Method
- Matrix chain multiplication
- Multistage Graph
- All-pairs Shortest Path
- Single Source Shortest path
- 0/1 knapsack problem
- Longest Common Sub sequence
- The Travelling Salesperson Problem
- Flow Shop scheduling

Comparison with divide-and-conquer

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem
 - Example: Quicksort
 - Example: Mergesort
 - Example: Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms
- In contrast, a dynamic programming algorithm proceeds by solving small problems, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results.

Exercise

1. What is the principle of optimality that is used in dynamic programming paradigm? Explain with an example how the *use of table* is found to be efficient instead of using *recursion*.
2. Write an algorithm to find the minimum cost path from source to sink in a multistage graph using backward approach. What is the time complexity of the algorithm? How the time complexity of the algorithm is effected with their storage representation
3. One goal of global telecommunications is to enable data packets to reach their final intended destination as fast as possible. Describe an algorithm that could be used to generate the quickest route for packets over the network. The route should be traceable so it does not have to be re-calculated each time. Illustrate your answer with a simple worked example. Give diagrams were appropriate. Find the time complexity for you proposed solution

Exercise

4. Write a dynamic programming algorithm to compute a binomial coefficient $C(n,k)$? Prove that time complexity of the algorithm is $O(nk)$. What is the space complexity of this algorithm? Explain how the space efficiency of this algorithm can be improved.
5. What are the general characteristics of dynamic programming algorithm? What are the four basic steps to find optimal solution for the problems using dynamic programming? Compare divide-and-conquer and dynamic programming?
6. Write a dynamic programming algorithm to find a solution to 0-1 knapsack problem that yields maximum profit P for n items with the knapsack capacity W .

Exercise

7. Explain the principle of ordering Matrix multiplication in the light of dynamic programming. Show that the number of ways that n matrices can be multiplied is of $O(4^n/n^{3/2})$.

The most popular dynamic programming problems

1. Given a matrix consisting of 0's and 1's, find the maximum size sub-matrix consisting of only 1's.
2. Given an array containing both positive and negative integers, find the contiguous array with the maximum sum.
3. Longest Increasing Subsequence - Find the length of the longest subsequence of a given sequence such that all the elements are sorted in increasing/non-decreasing order. There are many problems which reduce to this problem such as box stacking and the building bridges. These days the interviewers expect an $(n \log n)$ solution.

The most popular dynamic programming problems

4. Edit Distance - Given two strings and a set of operations Change (C), insert (I) and delete (D) , find minimum number of edits (operations) required to transform one string into another.
5. 0/1 Knapsack Problem - A thief robbing a store and can carry a maximal weight of W into their knapsack. There are n items and i^{th} item weigh w_i and is worth v_i dollars. What items should thief take?
6. Balanced Partition - You have a set of n integers each in the range $0 \dots K$. Partition these integers into two subsets such that you minimize $|S1 - S2|$, where $S1$ and $S2$ denote the sums of the elements in each of the two subsets.

The most popular dynamic programming problems

7. Coin Change - Given a value N , if we want to make change for N cents, and we have infinite supply of each of $S = \{ S_1, S_2, \dots, S_m \}$ valued coins, how many ways can we make the change?
8. Longest Common Subsequence - Find the longest common subsequence of two strings A and B where the elements are letters from the two strings and they should be in the same order.
9. Longest Palindromic Subsequence - The question is same as above but the subsequence should be palindromic as well.
10. Minimum Number of Jumps - Given an array of integers where each element represents the maximum number of steps that can be made forward from that element, find the minimum number of jumps to reach the end of the array (starting from the first element).