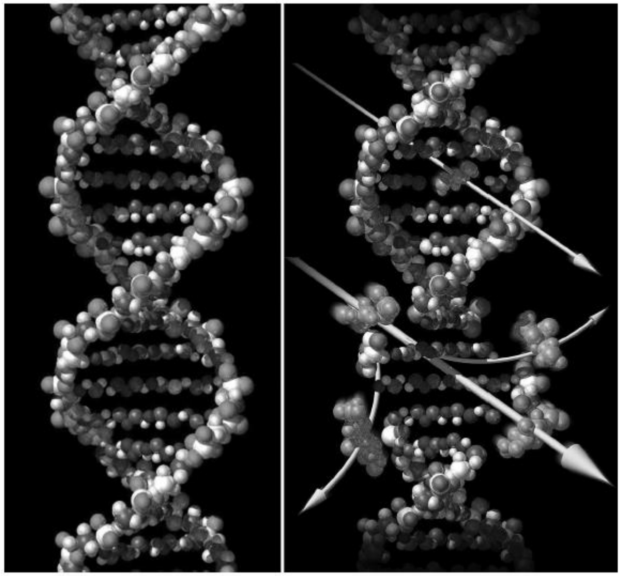
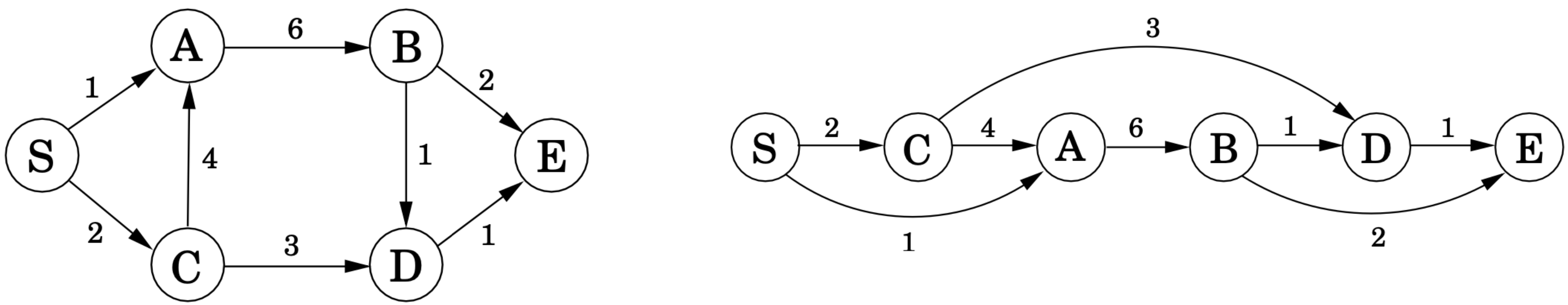
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Dynamic Programming



Effects of radiation on DNA's double helix, 2003. U.S. government image. NASA-MSFC.

Figure 6.1 A dag and its linearization (topological ordering).



Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of **A**, **C**, **G**, and **T** characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.

Application: DNA Sequence Alignment

- Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other.

```
X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
      | ||  || || | ||||||||||||
      G TC  GT CG G AAGCCGGCCGAA
     GTCGT CGGAA GCCG  GC C G AA
     ||||| ||||| ||||  || | | ||
Y:   GTCGTTCGGAATGCCGTTGCTCTGTAA
```

Figure 12.1: Two DNA sequences, X and Y, and their alignment in terms of a longest subsequence, GTCGTCGGAAGCCGGCCGAA, that is common to these two strings.

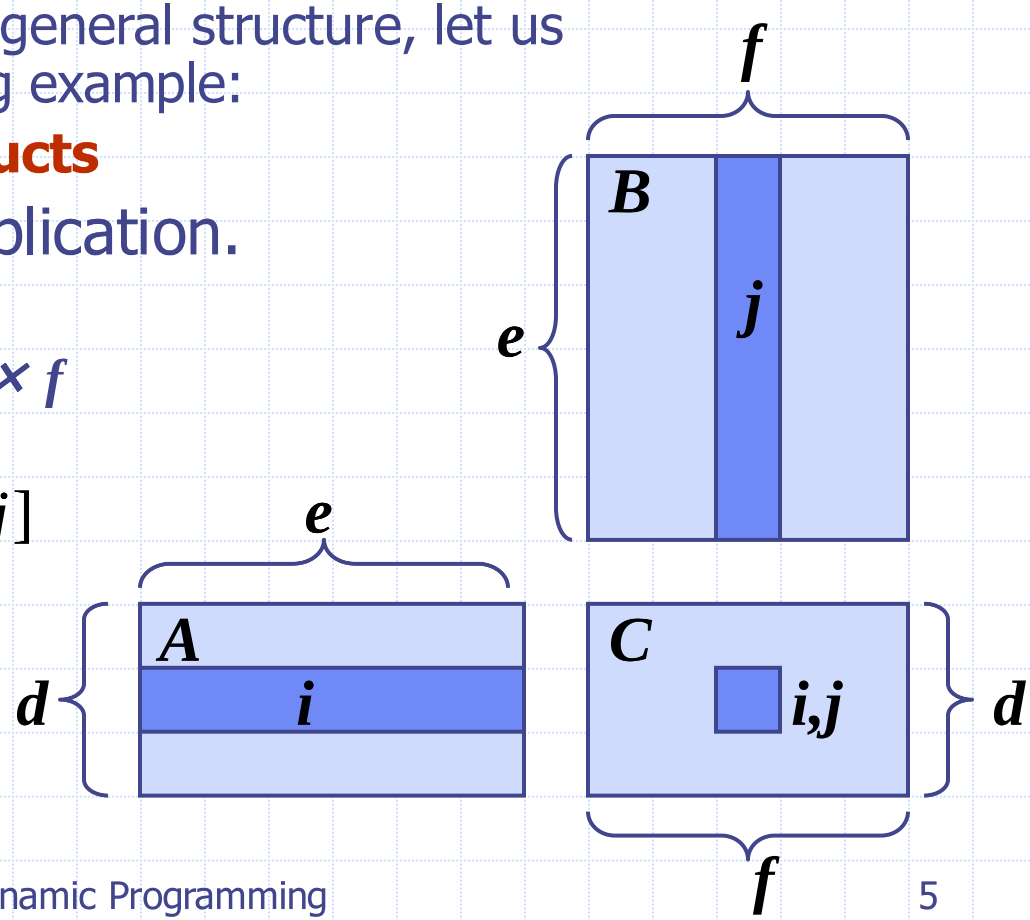
- A brute-force search would take exponential time (in fact $O(n2^{2n})$), but we can do much better using **dynamic programming**.

Warm-up: Matrix Chain-Products

- **Dynamic Programming** is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**
- Review: Matrix Multiplication.
  - $C = A * B$
  - A is $d \times e$ and B is $e \times f$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

  - $O(def)$ time



Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - $(B*C)*D$ takes $1500 + 75 = 1575$ ops
  - $B*(C*D)$ takes $1500 + 2500 = 4000$ ops

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of paranethesizations is equal to the number of binary trees with n nodes (why?)
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

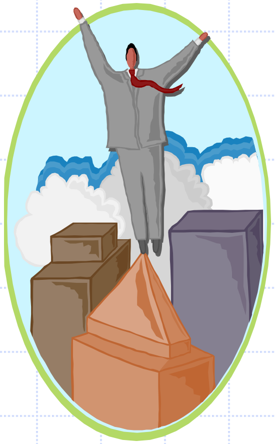
A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Another Greedy Approach

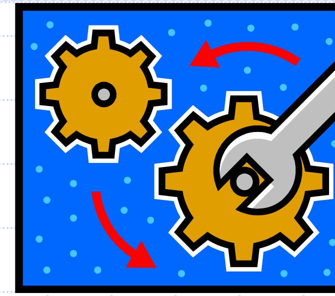
- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

A "Recursive" Approach



- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

A Characterizing Equation

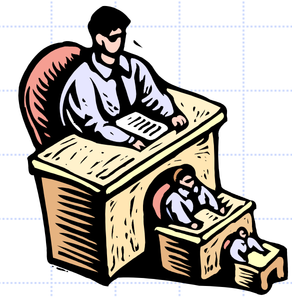


- The global optimal has to be defined in terms of optimal subproblems, depending on the location of the final multiply.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note that subproblems are not independent--the **subproblems overlap**.

A Dynamic Programming Algorithm



- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- The running time is $O(n^3)$

Algorithm ***matrixChain***(***S***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal paranethization of ***S***

for *i* ← 1 **to** *n-1* **do**

$\quad N_{i,i} \leftarrow 0$

for *b* ← 1 **to** *n-1* **do**

$\quad$ **for** *i* ← 0 **to** *n-b-1* **do**

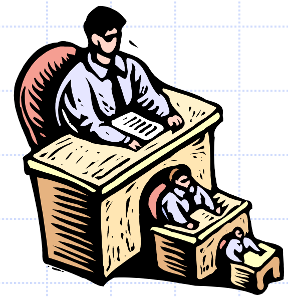
$\quad\quad j \leftarrow i+b$

$\quad\quad N_{i,j} \leftarrow$ +**infinity**

$\quad\quad$ **for** *k* ← i **to** *j-1* **do**

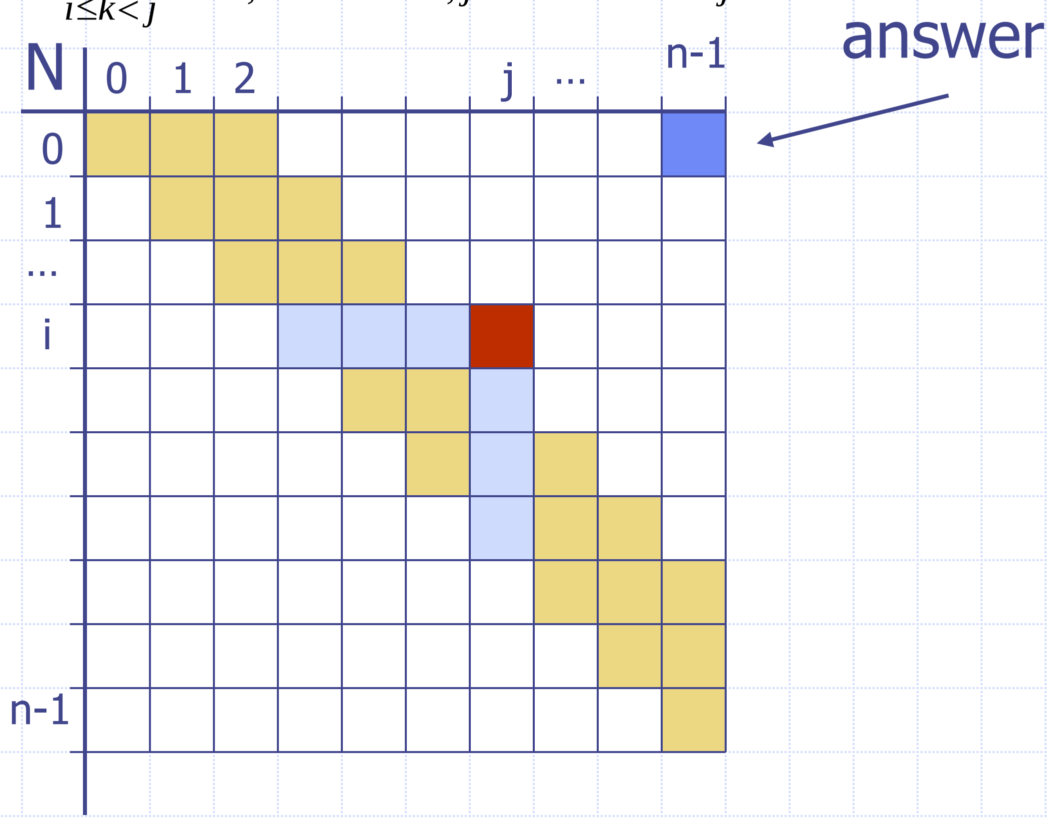
$$N_{i,j} \leftarrow \min\{N_{i,j} ,\ N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$$

A Dynamic Programming Algorithm Visualization



$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry



Determining the Run Time

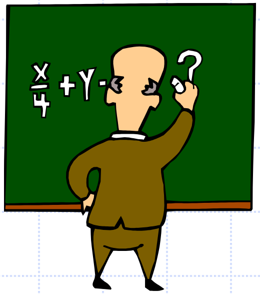
- It's not easy to see that filling in each square takes O(n) time. So, lets look at this another way

$$(n-1) + 2(n-2) + 3(n-3) + 4(n-4) + \cdots + (n-1)(n-(n-1))$$

$$\begin{aligned}
&= \sum_{i=1}^{n-1} i(n-i) = n\sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 \\
&= n\frac{(n-1)n}{2} - \frac{(n-1)n(2(n-1)+1)}{6} \\
&= n(n-1)\left[\frac{n}{2} - \frac{2n-1}{6}\right] = n(n-1)\frac{n+1}{6}
\end{aligned}$$

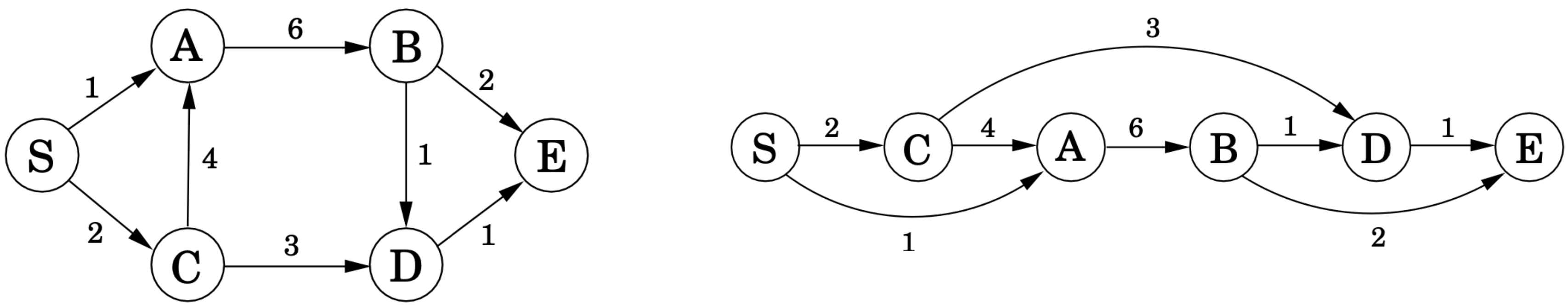
$O(n^3)$

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Figure 6.1 A dag and its linearization (topological ordering).

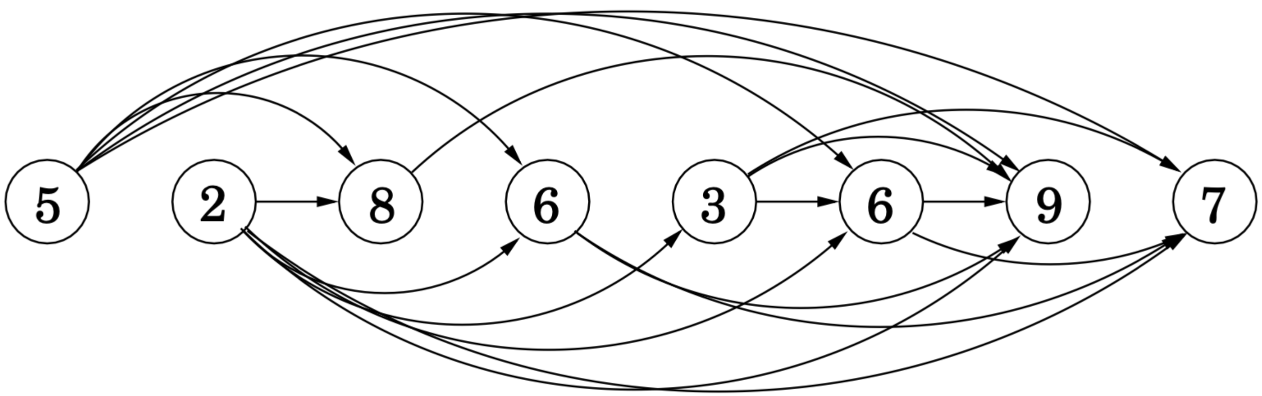


Let's try some of these

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$:

$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

Figure 6.2 The dag of increasing subsequences.



```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

$$\texttt{for } j = 1, 2, \ldots, n\texttt{:}$$
$$\quad L(j) = 1 + \max\{L(i) : (i,j) \in E\}$$
$$\texttt{return } \max_j L(j)$$

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems $\{L(j) : 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> (*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

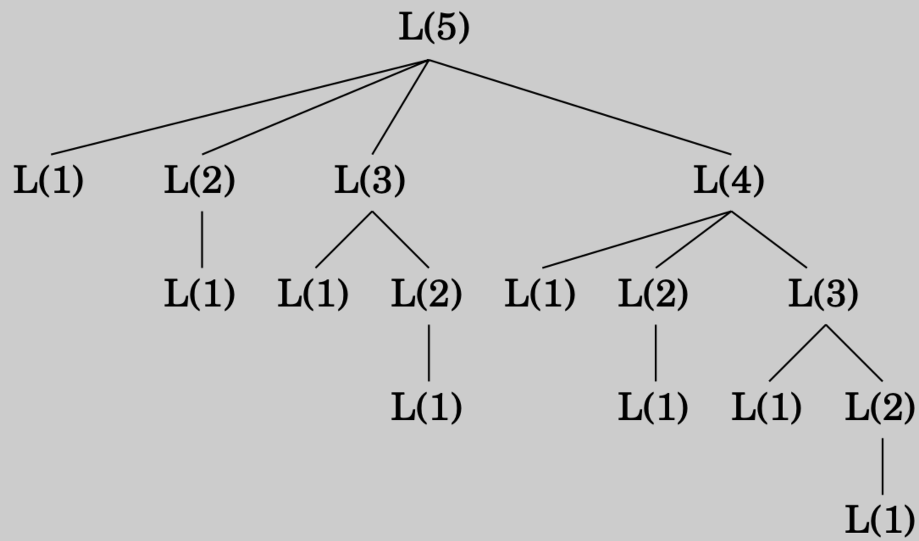
In our case, each subproblem is solved using the relation

$$L(j) \;=\; 1 + \max\{L(i) : (i,j) \in E\},$$

Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges (i, j) for *all* $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

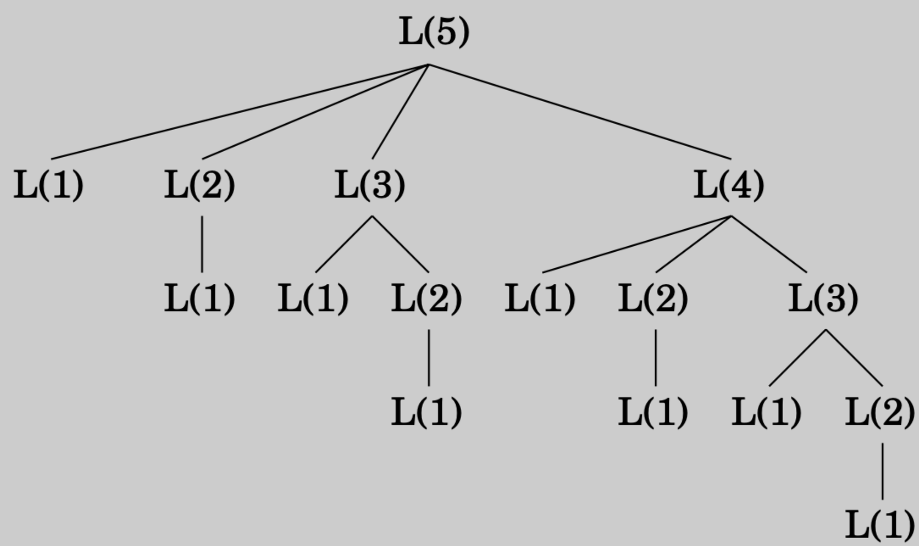
The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!



Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges (i, j) for *all* $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!



If recursion is so bad, why does it work for divide and conquer?

Another problem: Edit Distance

S	—	N	O	W	Y		—	S	N	O	W	—	Y
S	U	N	N	—	Y		S	U	N	—	—	N	Y

Cost: 3

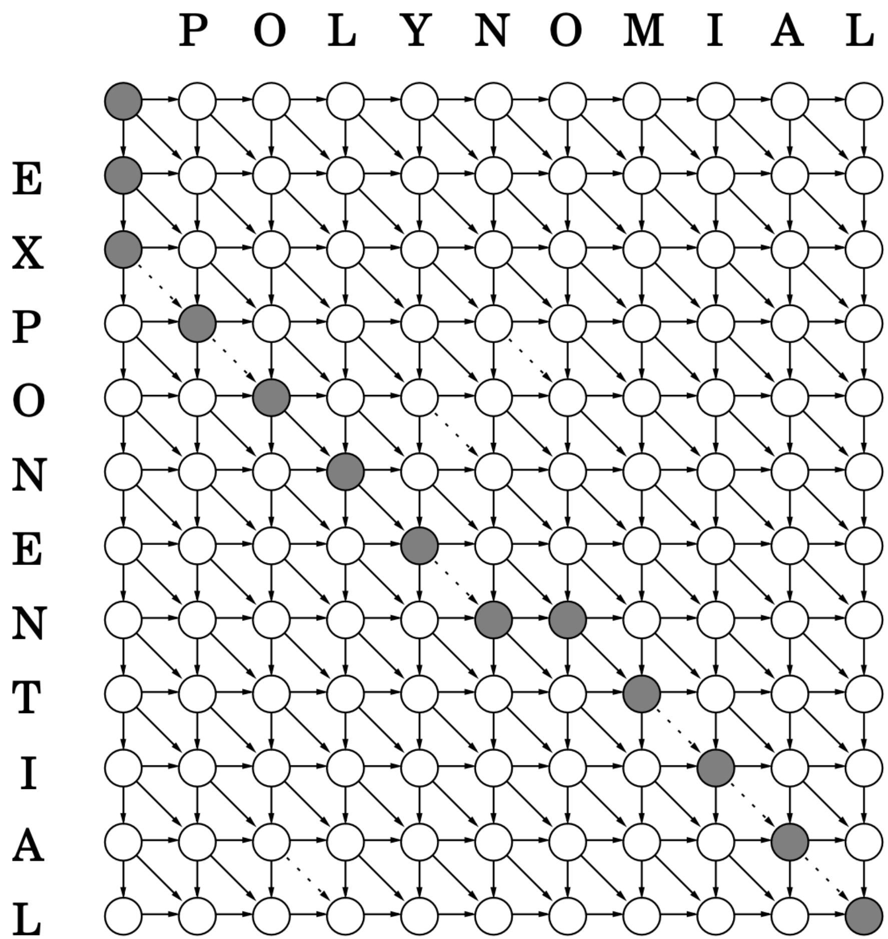
Cost: 5

Another problem: Edit Distance

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

for $i = 0, 1, 2, \ldots, m$**:**
    $E(i, 0) = i$
for $j = 1, 2, \ldots, n$**:**
    $E(0, j) = j$
for $i = 1, 2, \ldots, m$**:**
    for $j = 1, 2, \ldots, n$**:**
        $E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+\texttt{diff}(i,j)\}$
return $E(m, n)$

Figure 6.5 The underlying dag, and a path of length 6.



Another problem: Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

Two variants:
With repitition s (unlimited amount of each item)
Without repetition (only one of each item)

Knapsack with repitition

- What are the subproblems?
- What is the relation between the subproblems?
- What is the final algorithm?

Knapsack with Repitition

$K(w) =$ maximum value achievable with a knapsack of capacity w.

$$K(w) = \max_{i:w_i \le w} \{K(w - w_i) + v_i\},$$

$K(0) = 0$
for $w = 1$ to W:
  $K(w) = \max\{K(w - w_i) + v_i : w_i \le w\}$
return $K(W)$

Knapsack with Repitition

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

$K(0) = 0$
for $w = 1$ **to** W:
    $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
return $K(W)$

Runtime?

Knapsack with Repitition

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

$$
\begin{aligned}
&K(0) = 0 \\
&\texttt{for } w = 1 \texttt{ to } W\texttt{:} \\
&\qquad K(w) = \max\{K(w - w_i) + v_i : w_i \le w\} \\
&\texttt{return } K(W)
\end{aligned}
$$

Is O(nW) polynomial?

Knapsack without repitition

- What are the subproblems?
- What is the relation between the subproblems?
- What is the final algorithm?

Knapsack without Repitition

$K(w, j)$ = **maximum value achievable using a knapsack of capacity** w **and items** $1, \ldots, j$.

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
    for w = 1 to W:
        if w_j > w:   K(w,j) = K(w,j-1)
        else:   K(w,j) = max{K(w,j-1), K(w-w_j,j-1) + v_j}
return K(W,n)
```

Knapsack without Repitition

$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$.

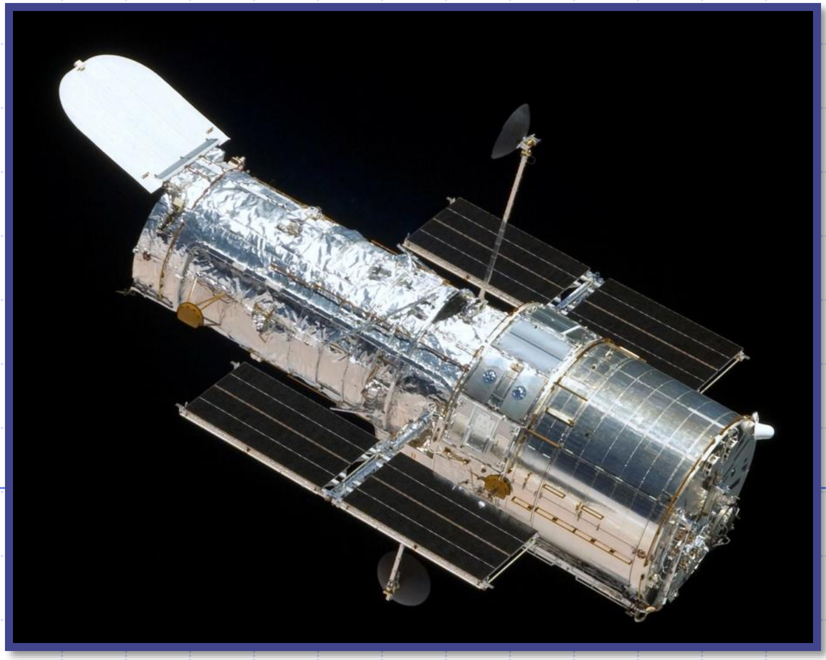
$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
    for w = 1 to W:
        if w_j > w:   K(w,j) = K(w,j-1)
        else:   K(w,j) = max{K(w,j-1), K(w-w_j,j-1) + v_j}
return K(W,n)
```

Runtime?

Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Dynamic Programming: Telescope Scheduling



Hubble Space Telescope. Public domain image, NASA, 2009.

Motivation

- Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them.
- This high demand for observation times is especially true, for instance, for a space telescope, which could receive thousands of observation requests per month.

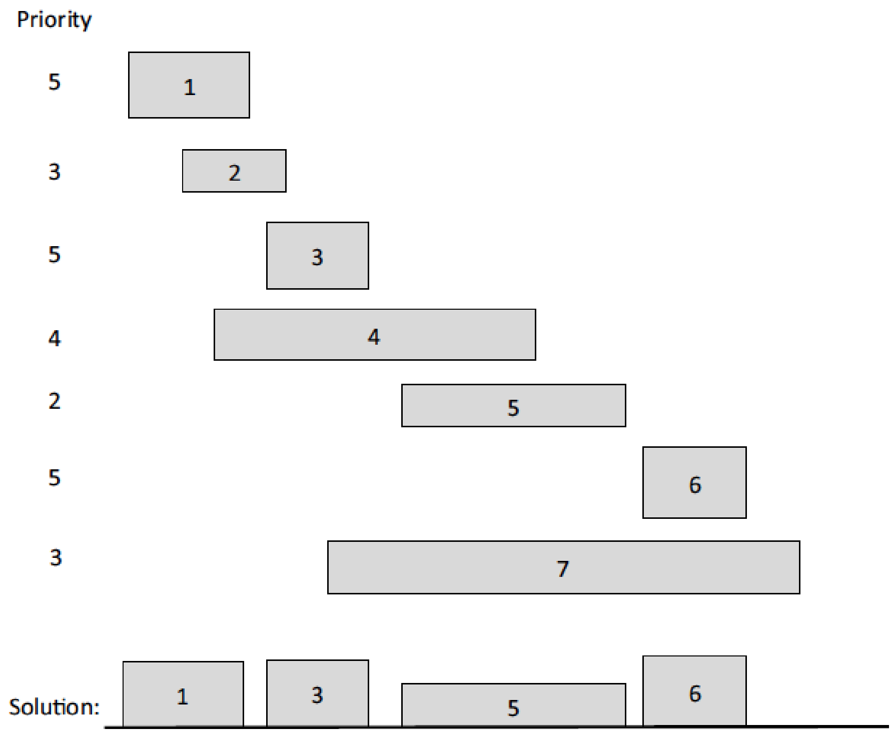
Telescope Scheduling Problem

- The input to the telescope scheduling problem is a list, **L**, of observation requests, where each request, **i**, consists of the following elements:
  - a **requested start time**, s_i, which is the moment when a requested observation should begin
  - a **finish time**, f_i, which is the moment when the observation should finish (assuming it begins at its start time)
  - a positive numerical **benefit**, b_i, which is an indicator of the scientific gain to be had by performing this observation.
- The start and finish times for an observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee.

Telescope Scheduling Problem

- To get the benefit, $\mathbf{b_i}$, for an observation request, **i**, that observation must be performed by the telescope for the entire time period from the start time, $\mathbf{s_i}$, to the finish time, $\mathbf{f_i}$.
- Thus, two requests, **i** and **j**, **conflict** if the time interval $\mathbf{[s_i, f_i]}$, intersects the time interval, $\mathbf{[s_j, f_j]}$.
- Given the list, **L**, of observation requests, the optimization problem is to schedule observation requests in a nonconflicting way so as to maximize the total benefit of the observations that are included in the schedule.

Example

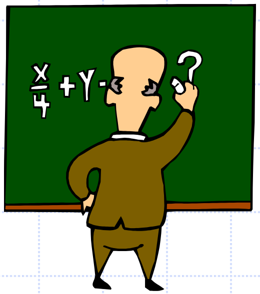


The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request's benefit (Priority) on the left. The optimal solution has total benefit 17=5+5+2+5.

False Start 1: Brute Force

- There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of L and choose the one that has the highest total benefit without causing any scheduling conflicts.
- Implementing this brute-force algorithm would take $O(n2^n)$ time, where n is the number of observation requests.
- We can do much better than this, however, by using the dynamic programming technique.

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Defining Simple Subproblems

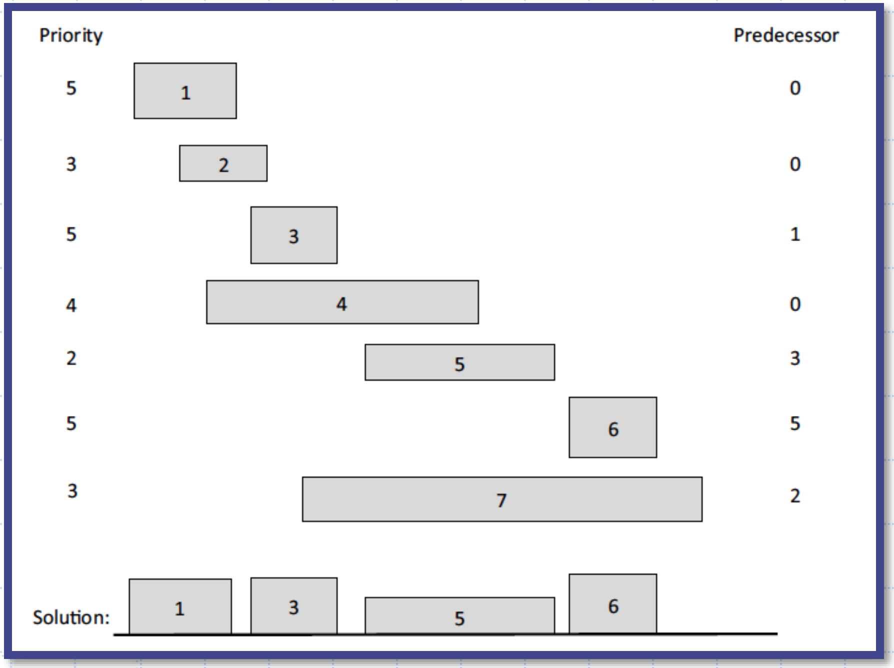
- A natural way to define subproblems is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits.
  - We already saw that ordering by benefits is a false start.
  - Start times and finish times are essentially symmetric, so let us order observations by finish times.

$B_i =$ the maximum benefit that can be achieved with the first i requests in L.

So, as a boundary condition, we get that $B_0 = 0$.

Predecessors

- For any request **i**, the set of other requests that conflict with **i** form a contiguous interval of requests in **L**.
- Define the **predecessor**, pred(**i**), for each request, **i**, then, to be the largest index, **j < i**, such that requests **i** and **j** don't conflict. If there is no such index, then define the predecessor of **i** to be 0.



Subproblem Optimality

- A schedule that achieves the optimal value, **B**$_i$, either includes observation **i** or not.

- If the optimal schedule achieving the benefit B_i includes observation i, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most $\text{pred}(i)$.
- On the other hand, if the optimal schedule achieving the benefit B_i does not include observation i, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves B_{i-1}.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1},\ B_{\text{pred}(i)} + b_i\}.$$

Subproblem Overlap

- The above definition has subproblem overlap.
- Thus, it is most efficient for us to use memoization when computing **B_i** values, by storing them in an array, **B**, which is indexed from 0 to **n**.
- Given the ordering of requests by finish times and an array, **P**, so that **P**[**i**] = pred(**i**), then we can fill in the array, **B**, using the following simple algorithm:

$B[0] \leftarrow 0$
for $i = 1$ to n **do**
    $B[i] \leftarrow \max\{B[i-1],\ B[P[i]] + b_i\}$

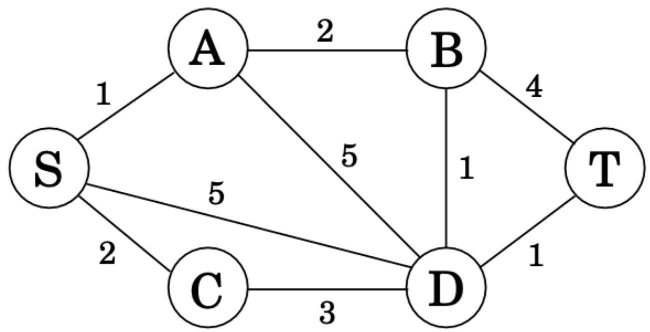
After this algorithm completes, the benefit of the optimal solution will be $B[n]$

Analysis of the Algorithm

- It is easy to see that the running time of this algorithm is **O(n)**, assuming the list **L** is ordered by finish times and we are given the predecessor for each request **i**.
- Of course, we can easily sort **L** by finish times if it is not given to us already sorted according to this ordering.
- To compute the predecessor of each request, note that it is sufficient that we also have the requests in **L** sorted by start times.
  - In particular, given a listing of **L** ordered by finish times and another listing, **L′**, ordered by start times, then a merging of these two lists, as in the merge-sort algorithm (Section 8.1), gives us what we want.
  - The predecessor of request **i** is literally the index of the predecessor in **L** of the value, $\mathbf{s_i}$, in **L′**.

Shortest Reliable Path

Figure 6.8 We want a path from s to t that is both short *and* has few edges.



Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t *that uses at most k edges*.

Shortest Reliable Path

Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t *that uses at most k edges*.

- You know the drill...

Shortest Reliable Path

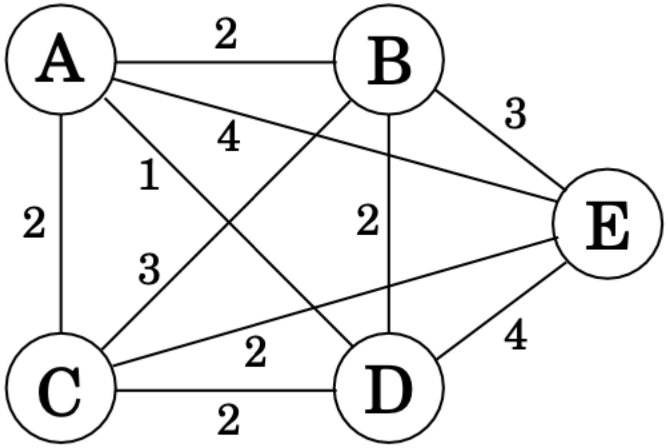
Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t *that uses at most k edges*.

In dynamic programming, the trick is to choose subproblems so that all vital information is remembered and carried forward. In this case, let us define, for each vertex v and each integer $i \leq k$, $\texttt{dist}(v, i)$ to be *the length of the shortest path from s to v that uses i edges*. The starting values $\texttt{dist}(v, 0)$ are ∞ for all vertices except s, for which it is 0. And the general update equation is, naturally enough,

$$\texttt{dist}(v, i) = \min_{(u,v) \in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}.$$

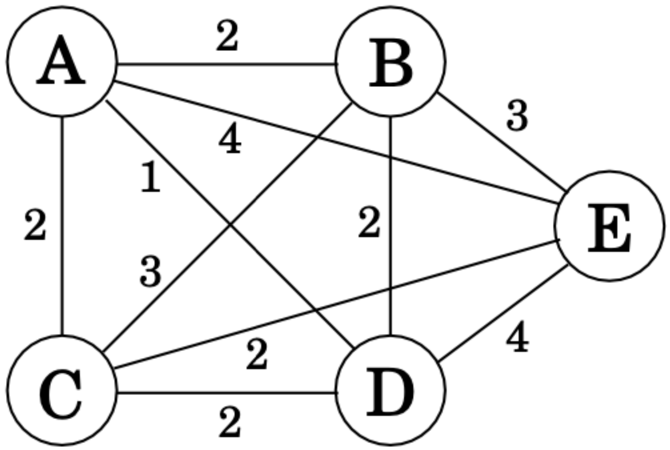
Traveling Salesperson Problem

Figure 6.9 The optimal traveling salesman tour has length 10.



Traveling Salesperson Problem

Figure 6.9 The optimal traveling salesman tour has length 10.



Have at it…

Traveling Salesperson Problem

For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1.

Traveling Salesperson Problem

For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1.

$$C(S, j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}.$$

Traveling Salesperson Problem

The subproblems are ordered by $|S|$. Here's the code.

```
C({1},1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
        C(S,1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S,j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1,...,n}, j) + d_j1
```

Runtime?

Traveling Salesperson Problem

The subproblems are ordered by $|S|$. Here's the code.

$C(\{1\}, 1) = 0$
for $s = 2$ **to** n:
  for all subsets $S \subseteq \{1, 2, \ldots, n\}$ **of size** s **and containing** 1:
    $C(S, 1) = \infty$
    for all $j \in S, j \neq 1$:
      $C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\}$
return $\min_j C(\{1, \ldots, n\}, j) + d_{j1}$

Runtime: $O(n^2 2^n)$ – $n2^n$ subproblems, each taking linear time to solve

DP: Time and memory

- Almost always the total number of edges in the corresponding DAG
  - Since we are effectively just checking each
- Memory is more complicated
  - Almost always can be done with amount of memory proportional to the number of vertices
    - But we can often do better. How?