Recap Divide-and-Conquer (D&C)

- **Divide and conquer:**
  - (Divide) Break down a problem into two or more sub-problems of the same (or related) type
  - (Conquer) Recursively solve each sub-problems and solve them directly if simple enough
  - (Combine) Combine these solutions to the sub-problems to give a solution to the original problem
- **Correctness:** proved by mathematical induction
- **Complexity:** determined by solving recurrence relations

Dynamic Programming (DP)

- **Dynamic “programming” came from the term “mathematical programming”**
  - Typically on optimization problems (a problem with an objective)
  - Inventor: Richard E. Bellman, 1953
- **Basic idea:** One implicitly explores the space of all possible solutions by
  - Carefully decomposing things into a series of subproblems
  - Building up correct solutions to larger and larger subproblems
- **Can you smell the D&C flavor? However, DP is another story!**
  - DP does not exam all possible solutions explicitly
  - Be aware of the condition to apply DP!!

Outline

- **Content:**
  - Weighted interval scheduling: a recursive procedure
  - Principles of dynamic programming (DP)
    - Memoization or iteration over subproblems
  - Example: maze routing
  - Example: Fibonacci sequence
  - Subset sums and Knapsacks: adding a variable
  - Shortest paths in a graph
  - Example: traveling salesman problem
- **Reading:**
  - Chapter 6
Weighted Interval Scheduling

Thinking in an inductive way

Given: A set of \( n \) intervals with start/finish times, weights (values)

Find: A subset \( S \) of mutually compatible intervals with maximum total values

Greedy?

The greedy algorithm of unit-weighted \((v_i = 1, 1 \leq i \leq n)\) intervals no longer works!
- Sort intervals in ascending order of finish times
- Pick up if compatible; otherwise, discard it

Q: What if variable values?

Designing a Recursive Algorithm (1/3)

In the induction perspective, a recursive algorithm tries to compose the overall solution using the solutions of sub-problems (problems of smaller sizes)
- First attempt: Induction on time?
  - Granularity?

Maximum weighted compatible set \( \{26, 16\} \)
Designing a Recursive Algorithm (2/3)

- Second attempt: Induction on interval index
  - First of all, sort intervals in ascending order of finish times
  - In fact, this is also a trick for DP
  - \( p(j) \) is the largest index \( i < j \) s.t. intervals \( i \) and \( j \) are disjoint
  - \( p(j) = 0 \) if no request \( i < j \) is disjoint from \( j \)

- \( p \)

\begin{align*}
p(1) &= 0 \\
p(2) &= 0 \\
p(3) &= 1 \\
p(4) &= 0 \\
p(5) &= 3 \\
p(6) &= 3
\end{align*}

Designing a Recursive Algorithm (3/3)

- \( O_j \) = the optimal solution for intervals 1, ..., \( j \)
- \( \text{OPT}(j) \) = the value of the optimal solution for intervals 1, ..., \( j \)
  - e.g., \( O_6 = ? \) Include interval 6 or not?
    - \( \Rightarrow O_6 = \{O_3 \} \text{ or } O_5 \)
    - \( \text{OPT}(6) = \max\{\{v_6 + \text{OPT}(3)\}, \text{OPT}(5)\} \)
- \( \text{OPT}(j) = \max\{\{v_j + \text{OPT}(p(j))\}, \text{OPT}(j-1)\} \)

Direct Implementation

- Preprocessing:
  - 1. Sort intervals by finish times: \( f_1 \leq f_2 \leq ... \leq f_n \)
  - 2. Compute \( p(1), p(2), ..., p(n) \)
- Compute-Opt(\( j \))
  1. if (\( j = 0 \)) then return 0
  2. else return \( \max\{\{v_j + \text{Compute-Opt}(p(j))\}, \text{Compute-Opt}(j-1)\} \)

Memoization: Top-Down

- The tree of calls widens very quickly due to recursive branching!
- e.g., exponential running time when \( p(j) = j - 2 \) for all \( j \)
- Q: How to eliminate this redundancy?
- A: Store the value for future! (memoization)

M-Compute-Opt(\( j \))
  1. if (\( j = 0 \)) then return 0
  2. else if (\( M[j] \) is not empty) then return \( M[j] \)
  3. else return \( M[j] = \max\{\{v_j + \text{M-Compute-Opt}(p(j))\}, \text{M-Compute-Opt}(j-1)\} \)

Running time: \( O(n) \)
Iteration: Bottom-Up

We can also compute the array $M[j]$ by an iterative algorithm.

1. Compute-Opt
2. for $j = 1, 2, \ldots, n$ do
3. $M[j] = \max\{v[M[p(j)]], M[j-1]\}$

Running time: $O(n)$

Summary: Memoization vs. Iteration

Memoization
- Top-down
  - An recursive algorithm
    - Compute only what we need

Iteration
- Bottom-up
  - An iterative algorithm
    - Construct solutions from the smallest subproblem to the largest one
    - Compute every small piece

The running time and memory requirement highly depend on the table size

Keys for Dynamic Programming

- Dynamic programming can be used if the problem satisfies the following properties:
  - There are only a polynomial number of subproblems
  - The solution to the original problem can be easily computed from the solutions to the subproblems
  - There is a natural ordering on subproblems from "smallest" to "largest," together with an easy-to-compute recurrence

Keys for Dynamic Programming

- DP typically is applied to optimization problems.
- DP works best on objects that are linearly ordered and cannot be rearranged
- Elements of DP
  - Optimal substructure: an optimal solution contains within its optimal solutions to subproblems.
  - Overlapping subproblem: a recursive algorithm revisits the same problem over and over again; typically, the total number of distinct subproblems is a polynomial in the input size.

In optimization problems, we are interested in finding a thing which maximizes or minimizes some function.

Keys for Dynamic Programming

- **Standard operation procedure for DP:**
  1. Formulate the answer as a recurrence relation or recursive algorithm. *(Start with divide-and-conquer)*
  2. Show that the number of different instances of your recurrence is bounded by a polynomial.
  3. Specify an order of evaluation for the recurrence so you always have what you need.

Algorithmic Paradigms

- **Brute-force** *(Exhaustive)*: Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- **Greedy**: Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide-and-conquer**: Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- **Dynamic programming**: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Fibonacci Sequence

- **Recurrence relation**: $F_n = F_{n-1} + F_{n-2}$, $F_0=0$, $F_1=1$
  - e.g., 0, 1, 1, 2, 3, 5, 8, …
- **Direct implementation**:
  - Recursion!
    
    ```
    fib(n)
    1. if $n \leq 1$ return $n$
    2. return $fib(n-1) + fib(n-2)$
    ```
What's Wrong?

- What if we call fib(5)?
  - fib(5)
  - fib(4) + fib(3)
  - (fib(3) + fib(2)) + (fib(2) + fib(1))
  - ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
  - (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

  A call tree that calls the function on the same value many different times
  - fib(2) was calculated three times from scratch
  - Impractical for large n

Too Many Redundant Calls!

- How to remove redundancy?
  - Prevent repeated calculation

Dynamic Programming -- Memoization

- Store the values in a table
  - Check the table before a recursive call
  - Top-down!
    - The control flow is almost the same as the original one

  fib(n)
  1. Initialize \( f[0..n] \) with -1 // -1: unfilled
  2. \( f[0] = 0; f[1] = 1 \)
  3. fibonacci(n, f)

  fibonacci(n, f)
  1. If \( f[n] == -1 \) then
  2. \( f[n] = \text{fibonacci}(n - 1, f) + \text{fibonacci}(n - 2, f) \)
  3. return \( f[n] \) // if \( f[n] \) already exists, directly return

Dynamic Programming -- Bottom-up?

- Store the values in a table
  - Bottom-up
    - Compute the values for small problems first
    - Much like induction

  fib(n)
  1. initialize \( f[1..n] \) with -1 // -1: unfilled
  2. \( f[0] = 0; f[1] = 1 \)
  3. for \( i=2 \) to \( n \) do
  4. \( f[i] = f[i-1] + f[i-2] \)
  5. return \( f[n] \)
Appendix: Maze Routing

Maze Routing Problem

- Restrictions: Two-pin nets on single-layer rectilinear routing
- Given:
  - A planar rectangular grid graph
  - Two points $S$ and $T$ on the graph
  - Obstacles modeled as blocked vertices
- Find:
  - The shortest path connecting $S$ and $T$
- Applications: Routing in IC design

Lee’s Algorithm (1/2)

- Idea:
  - Bottom up dynamic programming: Induction on path length
- Procedure:
  1. Wave propagation
  2. Retrace

Lee’s Algorithm (2/2)

- Strengths
  - Guarantee to find connection between 2 terminals if it exists
  - Guarantee minimum path
- Weaknesses
  - Large memory for dense layout
  - Slow
- Running time
  - $O(MN)$ for $M \times N$ grid

Adding a variable

Subset Sums & Knapsacks

29

Adding a variable

Dynamic programming

Subset Sum

30

Given
- A set of $n$ items and a knapsack
  - Item $i$ weighs $w_i > 0$.
  - The knapsack has capacity of $W$.

Goal:
- Fill the knapsack so as to maximize total weight.
  - maximize $\Sigma_{i \in S} w_i$

Greedy ≠ optimal
- Largest $w_i$ first: $7 + 2 + 1 = 10$
- Optimal: $5 + 6 = 11$

$W = 11$

Dynamic programming

Dynamic Programming: False Start

31

- Optimization problem formulation
  - $\max \Sigma_{i \in S} w_i$
  - $\sum_{i \in S} w_i < W$, $S \subseteq \{1, \ldots, n\}$

- $OPT(i)$ = the total weight of the optimal solution for items $1, \ldots, i$
  - $OPT(i) = \max_S \sum_{j \in S} w_j$, $S \subseteq \{1, \ldots, i\}$

- Consider $OPT(n)$, i.e., the total weight of the final solution $O$
  - Case 1: $n \notin O$ ($OPT(n)$ does not count $w_n$)
    - $OPT(n) = OPT(n-1)$ (Optimal solution of $\{1, 2, \ldots, n-1\}$)
  - Case 2: $n \in O$ ($OPT(n)$ counts $w_n$)
    - $OPT(n) = w_n + OPT(n-1)$

Q: What's wrong?
A: Accept item $n$ ⇒ For items $\{1, 2, \ldots, n-1\}$, we have less available weight, $W - w_n$.

Dynamic programming

Adding a New Variable

32

- Optimization problem formulation
  - $\max \Sigma_{i \in S} w_i$
  - $\sum_{i \in S} w_i < W$, $S \subseteq \{1, \ldots, n\}$

- $OPT(i)$ depends not only on items $\{1, \ldots, i\}$ but also on $W$
  - $OPT(i-1, w)$ if $w > \max\{OPT(i-1, w), w + OPT(i-1)\}$ otherwise

- Consider $OPT(n)$, i.e., the total weight of the final solution $O$
  - Case 1: $n \notin O$ ($OPT(n)$ does not count $w_n$)
    - $OPT(n) = OPT(n-1)$
  - Case 2: $n \in O$ ($OPT(n)$ counts $w_n$)
    - $OPT(n) = w_n + OPT(n-1)$

- Recurrence relation:

Dynamic programming

Karp’s 21 NP-complete problems:
R. M. Karp, “Reducibility among combinatorial problems”.
**Dynamic programming**

**Example**

Subset-sum($n, w_1, \ldots, w_n, W$)

1. for $w = 1, 2, \ldots, W$ do
2. $M(0, w) = 0$
3. for $i = 1, 2, \ldots, n$ do
4. $M(i, 0) = 0$
5. for $i = 1, 2, \ldots, n$ do
6. for $w = 1, 2, \ldots, W$ do
7. if ($w_i > w$) then
8. $M(i, w) = M(i-1, w)$
9. else
10. $M(i, w) = \max \{M(i-1, w), w_i + M(i-1, w-w_i)\}$

**The Knapsack Problem**

Given
- A set of $n$ items and a knapsack
- Item $i$ weighs $w_i > 0$ and has value $v_i > 0$
- The knapsack has capacity of $W$

Goal:
- Fill the knapsack so as to maximize total value.
  - Maximize $\sum_{i=1}^{n} v_i$

Optimization problem formulation
- $\max \sum_{i=1}^{n} v_i$
  - s.t. $\sum_{i=1}^{n} w_i < W, S \subseteq \{1, \ldots, n\}$

Greedy ≠ optimal
- Largest $v_i$ first: $28 + 6 + 1 = 35$
- Optimal: $18 + 22 = 40$
Recurrence Relation

- We know the recurrence relation for the subset sum problem:
  \[ \text{OPT}(i, w) = \begin{cases} 
    0 & \text{if } i, w = 0 \\
    \text{OPT}(i-1, w) & \text{if } w_i > w \\
    \max\{\text{OPT}(i-1, w), \text{OPT}(i-1, w-w_i)\} & \text{otherwise} 
  \end{cases} \]

- Q: How about the Knapsack problem?
  
  **A:**

  \[ \text{OPT}(i, w) = \begin{cases} 
    0 & \text{if } i, w = 0 \\
    \text{OPT}(i-1, w) & \text{if } w_i > w \\
    \max\{\text{OPT}(i-1, w), \text{OPT}(i-1, w-w_i)\} & \text{otherwise} 
  \end{cases} \]

Recap: Dijkstra’s Algorithm

- The shortest path problem:
  - Given:
    - Directed graph \( G = (V, E) \), source \( s \) and destination \( t \)
      - cost \( c_{uv} = \text{length of edge } (u, v) \in E \)
  - Goal:
    - Find the shortest path from \( s \) to \( t \)
    - Length of path \( P: c(P) = \sum_{(u, v) \in P} c_{uv} \)

  **Dijkstra(G,c)**
  
  //: the set of explored nodes
  // \( d(u) \): shortest path distance from \( s \) to \( u \)
  1. initialize \( S = \{s\}, d(s) = 0 \)
  2. while \( S \neq V \) do
  3.   select node \( v \notin S \) with at least one edge from \( S \)
  4.   \( d'(v) = \min_{u \in S, d(u) + c_{uv}} d(u) + c_{uv} \)
  5.   add \( v \) to \( S \) and define \( d(v) = d'(v) \)

  - Q: What if negative edge costs?

Modifying Dijkstra’s Algorithm?

- Observation: A path that starts on a cheap edge may cost more than a path that starts on an expensive edge, but then compensates with subsequent edges of negative cost.
- Reweighting: Increase the costs of all the edges by the same amount so that all costs become nonnegative.

Q: What’s wrong with s-a-b-t path?

Q: What’s wrong?!
Bellman-Ford Algorithm (1/2)

- Induction either on nodes or on edges works!
  - If $G$ has no negative cycles, then there is a shortest path from $s$ to $t$ that is **simple** (i.e., does not repeat nodes), and hence has at most $n-1$ edges.
  - Pf:
    - Suppose the shortest path $P$ from $s$ to $t$ repeat a node $v$.
    - Since every cycle has nonnegative cost, we could remove the portion of $P$ between consecutive visits to $v$ resulting in a simple path $Q$ of no greater cost and fewer edges.
      - $c(Q) = c(P) - c(C) \leq c(P)$

Bellman-Ford Algorithm (2/2)

- Induction on edges
  - $\text{OPT}(i, v) = \text{length of shortest } v\to t \text{ path } P \text{ using at most } i \text{ edges.}$
  - $\text{OPT}(n-1, s) = \text{length of shortest } s\to t \text{ path.}$
  - Case 1: $P$ uses at most $i-1$ edges.
    - $\text{OPT}(i, v) = \text{OPT}(i-1, v)$
  - Case 2: $P$ uses exactly $i$ edges.
    - $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$
    - If $(v, w)$ is the first edge, then $P$ uses $(v, w)$ and then selects the shortest $w\to t$ path using at most $i-1$ edges

Bellman-Ford Algorithm (1/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (2/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry

Bellman-Ford Algorithm (2/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (1/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry

Bellman-Ford Algorithm (2/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (1/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry

Bellman-Ford Algorithm (2/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (1/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry

Bellman-Ford Algorithm (2/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (1/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry

Bellman-Ford Algorithm (2/2)

- **Initialization:**
  - $M[0, v] = 0$ for $v \in V$, $M[i, v] = \infty$ for $i > 0, v \in V$.
  - $M[0, t] = 0$.

Bellman-Ford Algorithm (1/2)

- **Main Loop:**
  - For $i = 1$ to $n-1$,
    - For each vertex $v \in V$,
      - Update $M[i, v] = \min\{M[i-1, v], M[i-1, u] + c_{uv}\}$

**Example**

**Running time:**
1. naive: $O(nm)$
2. detailed: $O(nm)$

**Space:** $O(n^2)$

Q: How to find the shortest path?
A: Record "successor" for each entry
### Running Time

- **Lines 5-6:**
  - **Naïve:** for each $v$, check $v$ and others: $O(n^2)$
  - **Detailed:** for each $v$, check $v$ and its neighbors (out-going edges): $\sum_{v \in V}(\deg_{\text{out}}(v)+1) = O(m)$

- **Lines 4-6:**
  - **Naïve:** $O(n^3)$
  - **Detailed:** $O(nm)$

### Space Improvement

- Maintain a 1D array instead:
  - $M[v]$ = shortest $v-t$ path length that we have found so far.
  - Iterator $i$ is simply a counter
  - No need to check edges of the form $(v, w)$ unless $M[w]$ changed in previous iteration.
  - In each iteration, for each node $v$, $M[v]=\min\{M[v], \min_{w \in V} (c_{vw} + M[w])\}$

- **Observation:** Throughout the algorithm, $M[v]$ is the length of some $v-t$ path, and after $i$ rounds of updates, the value $M[v]$ is no larger than the length of shortest $v-t$ path using at most $i$ edges.

---

### Negative Cycles?

- If a $s-t$ path in a general graph $G$ passes through node $v$, and $v$ belongs to a negative cycle $C$, Bellman-Ford algorithm fails to find the shortest $s-t$ path.
- Reduce cost over and over again using the negative cycle

### Application: Currency Conversion (1/2)

**Q:** Given $n$ currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

- **The currency graph:**
  - Node: currency; edge cost: exchange rate $r_{uv}$; $r_{uv} \cdot r_{vu} < 1$
  - **Arbitrage:** a cycle on which product of edge costs $> 1$

**E.g.,** $\$1 \Rightarrow 1.3941$ Francs $\Rightarrow 0.9308$ Euros $\Rightarrow \$1.00084$
Application: Currency Conversion (2/2)

Arbitrage
- Product of edge costs on a cycle $C = v_1, v_2, \ldots, v_1$
  - Arbitrage: $r_{v_1v_2}r_{v_2v_3}\cdots r_{v_nv_1} > 1$
- Sum of edge costs on a cycle $C = v_1, v_2, \ldots, v_1$
  - Arbitrage: $c_{v_1v_2} + c_{v_2v_3} + \cdots + c_{v_nv_1} = -\log r_{v_1v_2}$

Detecting Negative Cycles by Bellman-Ford
- Augmented graph $G'$ of $G$
  1. Add new node $t$
  2. Connect all nodes to $t$ with 0-cost edge
- $G$ has a negative cycle iff $G'$ has a negative cycle reaching $t$
- Check if $\text{OPT}(n, v) = \text{OPT}(n-1, v)$:
  - If yes, no negative cycles
  - If no, then extract cycle from shortest path from $v$ to $t$
- Procedure:
  - Build the augmented graph $G'$ for $G$
  - Run Bellman-Ford on $G'$ for $n$ iterations (instead of $n-1$).
  - Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.

Dynamic programming

Negative Cycle Detection
- If $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all $v$, then no negative cycles.
  - Bellman-Ford: $\text{OPT}(i, v) = \text{OPT}(n-1, v)$ for all $v$ and $i \geq n$.
- If $\text{OPT}(n, v) < \text{OPT}(n-1, v)$ for some $v$, then shortest path contains a negative cycle
  - Pf: by contradiction
    - Since $\text{OPT}(n, v) < \text{OPT}(n-1, v)$, $P$ has exactly $n$ edges.
    - Every path using at most $n-1$ edges costs more than $P$.
    - (By pigeonhole principle.) $P$ must contain a cycle $C$.
    - If $C$ were not a negative cycle, deleting $C$ yields a $v$-$t$ path with $n$ edges and no greater cost.

Traveling Salesman Problem

Richard E. Bellman, 1962
Travelling Salesman Problem

- **TSP:** A salesman is required to visit once and only once each of \( n \) different cities starting from a base city, and returning to this city. What path minimizes the total distance travelled by the salesman?
- The distance between each pair of cities is given.

- **TSP contest**
  - http://www.tsp.gatech.edu

- **Brute-Force**
  - Try all permutations: \( O(n!) \)

Dynamic Programming

- For each subset \( S \) of the cities with \( |S| \geq 2 \) and each \( u, v \in S \), \( OPT(S, u, v) \) is the length of the shortest path that starts at \( u \) and ends at \( v \), visits all cities in \( S \).
- **Recurrence**
  - Case 1: \( S = \{u, v\} \)
    - \( OPT(S, u, v) = d(u, v) \)
  - Case 2: \( |S| > 2 \)
    - Assume \( w \in S - \{u, v\} \) is visited first:
      - \( OPT(S, u, v) = d(u, w) + OPT(S - u, w, v) \)
    - \( OPT(S, u, v) = \min_{w \in S - \{u, v\}}\{d(u, w) + OPT(S - u, w, v)\} \)
- **Efficiency**
  - Space: \( O(2^n n^2) \)
  - Running time: \( O(2^n n^3) \)
  - Although much better than \( O(n!) \), DP is suitable when the number of subproblems is polynomial.

Summary: Dynamic Programming

- **Smart recursion:** In a nutshell, dynamic programming is recursion without repetition.
  - Dynamic programming is **NOT** about filling in tables; it’s about smart recursion.
  - Dynamic programming algorithms store the solutions of intermediate subproblems often but not always in some kind of array or table.
  - A common mistake: focusing on the table (because tables are easy and familiar) instead of the much more important (and difficult) task of finding a correct recurrence.
  - If the recurrence is wrong, or if we try to build up answers in the wrong order, the algorithm will **NOT** work!

Summary: Algorithmic Paradigms

- **Brute-force (Exhaustive):** Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- **Greedy:** Build up a solution incrementally, myopically optimizing some local criterion.
  - Optimization problems that can be solved correctly by a greedy algorithm are **very** rare.
- **Divide-and-conquer:** Break up a problem into two subproblems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- **Dynamic programming:** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.