Dynamic Programming and Greedy Algorithm

Coin Problem

For a value V in Australian Dollars, find the minimum number of notes and coins required to get V.

Denominations: {5c, 10c, 20c, 50c, $1, $2, $5, $10, $20, $50, $100}

(1) V = $188.35
(2) V = $351.30
(3) V = $8749.95
(4) V = $11

Coin Problem

For a value V in Australian Dollars, find the minimum number of notes and coins required to get V.

Denominations: {5c, 10c, 20c, 50c, $1, $2, $5, $10, $20, $50, $100}

(1) V = $188.35 = {100, 50, 20, 10, 5, 2, 1, 20c, 10c, 5c}
(2) V = $351.30 = {100, 100, 100, 50, 1, 20c, 10c}
(3) V = $8749.95 = {100 x87, 20, 20, 10, 5, 2, 2, 50c, 20c, 20c, 5c}
(4) V = $11 = {10, 1}

(Greedy Workshop)

Coin Problem 2

For a value V, find the minimum number of coins required to get V.

Denominations: {$9, $6, $5, $1}

(1) V = $10
(2) V = $15
(3) V = $11

Coin Problem 2

For a value V, find the minimum number of coins required to get V.

Denominations: {$9, $6, $5, $1}

(1) V = $10 = $9 + $1
(2) V = $15 = $9 + $6
(3) V = $11 = $9 + $1 + $1

Coin Problem 2

For a value V, find the minimum number of coins required to get V.

Denominations: {$9, $6, $5, $1}

(1) V = $10 = $9 + $1
(2) V = $15 = $9 + $6
(3) V = $11 = $9 + $1 + $1 = $6 + $5

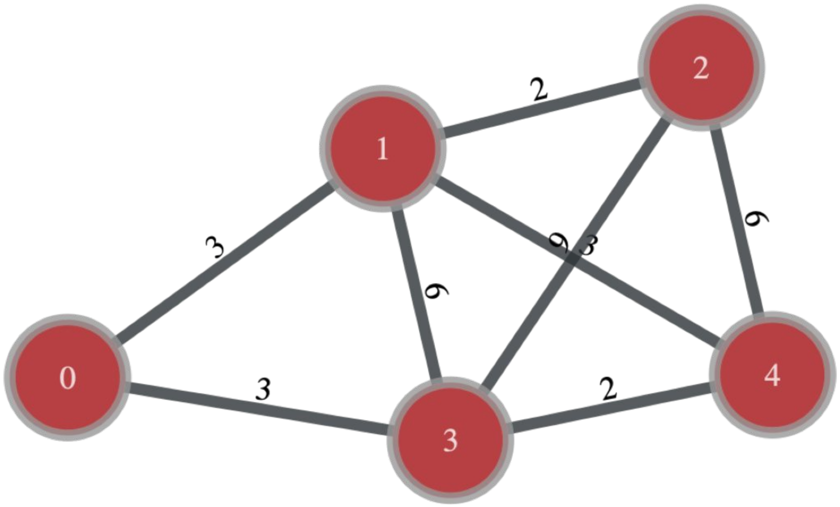
Example — Travelling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Example — Travelling Salesman Problem

Nearest Neighbour Algorithm

1. Initialize all vertices as unvisited.
2. Start with vertex **u**. Mark **u** as visited.
3. Find out the shortest edge connecting the current vertex **u** and an unvisited vertex **v**.
4. Set **v** as the current vertex **u**. Mark **v** as visited.
5. If all the vertices in the domain are visited, then terminate. Else, go to step 3.

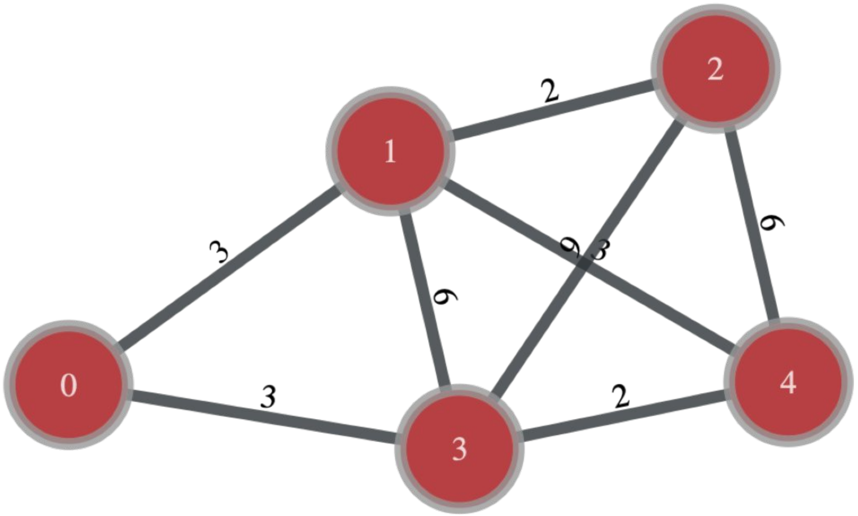


0 > 1 > 2 > 4 > 3 > 0

Example — Travelling Salesman Problem

Nearest Neighbour Algorithm

- Almost never optimal
- But it's fast — Worst case $O(n)$
- Optimal
  - NP-hard
  - Brute-force — $O(n!)$
  - Dynamic Programming — $O(n^2 2^n)$



When is greedy a good idea?

Submodular

Discrete Optimization

- If we have a continuous function
  - Just take derivative = 0
- Not so obvious for a discrete analogy
- Let X be a finite set, function f is submodular if $\forall S \subset T \subset X$ and all $x \in X \backslash T$, we have:

$$\underbrace{\boxed{f(S \cup \{x\}) - f(S)}}_{\text{Gain from including } x \text{ in } S} \geq \underbrace{\boxed{f(T \cup \{x\}) - f(T)}}_{\text{Gain from including } x \text{ in } T}$$

Submodular

In human language

- Diminishing Returns
  - A doughnuts worth less to me when I have 100 doughnuts than when I have 0 doughnut
- f need to be monotonic
  - Giving me more doughnuts —> total worth of my doughnut increase/stay the same

Submodular

Where do submodular function appear?

- Economics
- Graph theory — vertex covering
- Maximizing influence in a social network (Kempe et.al. 2015)
- Machine Learning — objective functions of machine learning tasks such as sensor placement (Guestrin, Krause, Gupta, Golovin, Bilmes,... 2005-now)

Theorem — Nemhauser, Wolsey, Fisher 1978

If f is monotone submodular, Greedy finds a solution of value at least $(1 - 1/e) \times$ optimum for the problem

Greedy Algorithm

When is it used

- Dijkstra algorithm — Finding shortest path in positively weight graph
- Graph Coloring
- Job Sequencing
- Huffman Encoding
- Fining eigenvalues (Hernandez et.al. 2021)
- Solving PDEs (Schaback 2019)

Feel like more greedy?

- **Competitive Programmer's Handbook,** Antti Laaksonen, Chapter 6
- **LeetCode** — Greedy Algorithm
- **General questions**: https://brilliant.org/practice/greedy-algorithm/
- **Scheduling** - (Job sequencing with deadlines, Maximum number of events attendable) Discussed today
- **Data Compression** - (Huffman Coding): https://www.hackerrank.com/challenges/tree-huffman-decoding/problem
- **Single-Source Shortest Path** - (Dijkstra's Algorithm): https://www.hackerrank.com/challenges/dijkstrashortreach/problem
- **Minimum Spanning Tree** - (Kruskal's / Prim's): https://www.hackerrank.com/challenges/kruskalmstrsub/problem

Dynamic Programming

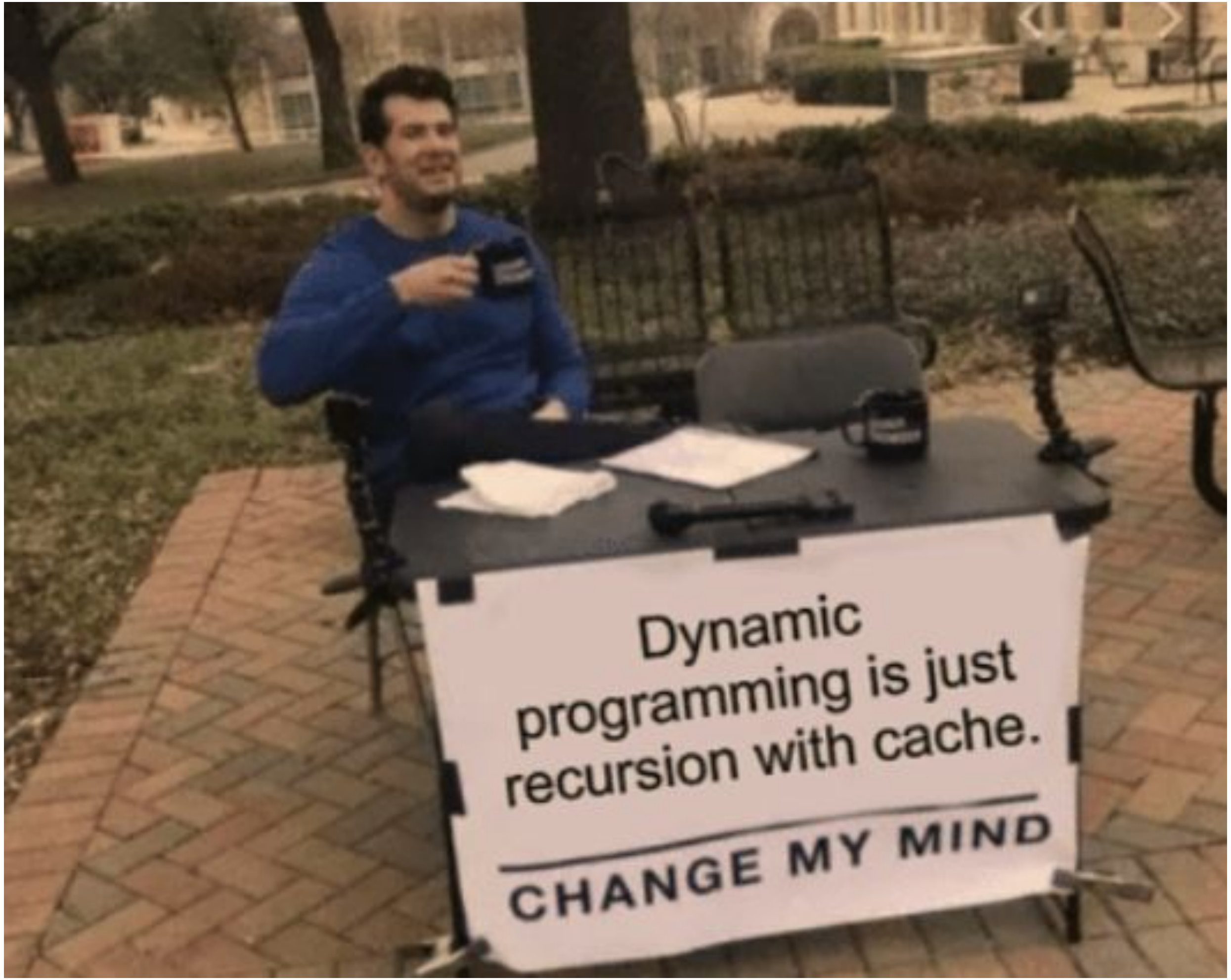
“Simplify a complicated problem, by breaking it down into simpler, overlapping problems in a recursive manner.” - Wikipedia

Dynamic Programming - Requirements

1. **Optimal Substructure**
    - An optimal solution for a problem can be constructed by optimal solutions of its subproblems.

2. **Overlapping Subproblems**
    - Problem can be broken down into subproblems, which are reused several times

No overlapping subproblems? Use greedy.


Dynamic
programming is just
recursion with cache.
CHANGE MY MIND

Dynamic Programming - **Uses**

Problems to solve:

1. Find an optimal solution
    - One that’s as large (max) or as small (min) as possible.

2. Count the number of solutions
    - Calculate the total number of possible solutions

Problems where we use DP

- Longest increasing subsequence
- Paths on a grid
- Knapsack
- Edit distance
- Counting tilings
- TSP
- Interval scheduling
- Subset Sum (psuedo-polynomial time)
- Bellman-Ford O(|V||E|) - find shortest distance in a graph,
  - Slower than Dijkstra (greedy), but works for negative weights.
- Floyd-Warshall O(|V|^3) - All-pairs shortest path, compute transitive closure
- Kadane's algorithm O(n) - optimal maximum subarray - S1W1: Complexity)

Dynamic Programming - Steps

1. **Optimal Substructure**
   a) Formulate a recursive definition of the problem
   b) Find optimal solutions to subproblems.

2. **Overlapping Subproblems**
   a) **Top-down** (Memoization)
   b) **Bottom-up** (Tabulation)

Fibonacci

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
- We wish to write a function to calculate the *n*th fibonacci value.

Fibonacci

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

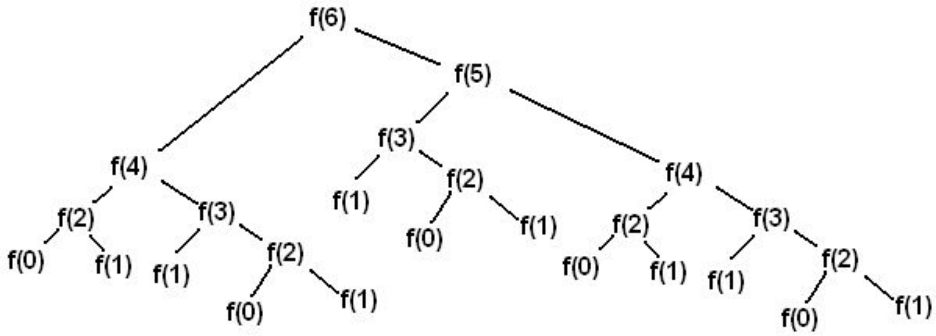
```
def fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

Algorithmic complexity?

Fibonacci

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

```
def fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```



Algorithmic complexity?

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

proof: https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming#Fibonacci_Numbers

Fibonacci - DP?

Could we use DP? It needs to have ...

1. **Optimal Substructure**

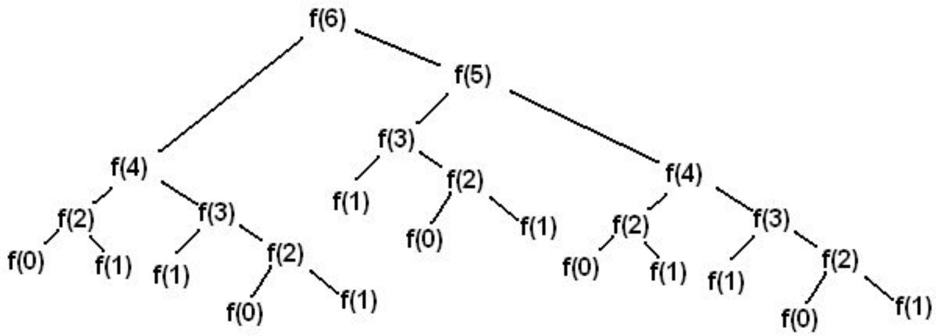
2. **Overlapping Subproblems**

Fibonacci - DP?

Could we use DP?

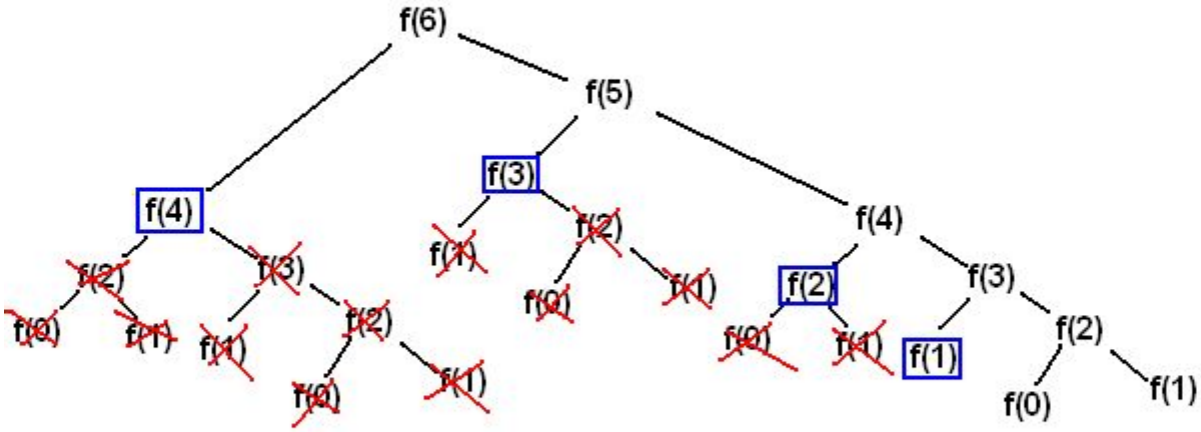
1. **Optimal Substructure**
   - fib(n) is constructed by fib(n-1) and fib(n-2)
   - To calculate the solution to the problem, we need to calculate the solution to its subproblems.

2. **Overlapping Subproblems**
   - see the tree



DP Top-down: Memoisation

- Identify overlapping subproblems
- Save the solutions to these subproblems
- When we come across these subproblems again, no need to recalculate!



Fibonacci DP Top-Down

```
def fib(n): # assume n ⩾ 0
    if n ⩽ 1: return n
    M = [-1] * (n+1)          # Memoisation array
    M[0], M[1] = 0, 1         # base case fib(0) = 0, fib(1) = 1
    def fib_mem(n):
        # look up the value; calculate it if we haven't already
        if M[n] == -1:
            M[n] = fib_mem(n-1) + fib_mem(n-2)
        return M[n]
    return fib_mem(n)
```

Algorithmic complexity? **Time: O(n) Space: O(n), uses recursion**

DP Bottom-Up

`fib(n) = fib(n-1) + fib(n-2)`

- Start with base cases: fib(0) and fib(1)

	0	1				

DP Bottom-Up

`fib(n) = fib(n-1) + fib(n-2)`

- Calculate and store fib(2) = fib(1) + fib(0)

	0	1	1			

DP Bottom-Up

`fib(n) = fib(n-1) + fib(n-2)`

- Calculate and store fib(3) = fib(2) + fib(1)

	0	1	1	2		

DP Bottom-Up

`fib(n) = fib(n-1) + fib(n-2)`

- Calculate and store fib(4) = fib(3) + fib(2)

	0	1	1	2	3	

DP Bottom-Up

`fib(n) = fib(n-1) + fib(n-2)`

- Calculate and store fib(5) = fib(4) + fib(3)

	0	1	1	2	3	5

- Only need to store fib(n-1) and fib(n-2) -- we never use the earlier calculated subproblems.
- Let f1 = fib(n-1), f2 = fib(n-2)

Fibonacci DP Bottom-up

```
def fib(n):
  if n <= 1: return n                # base case: f(0)=0 and f(1)=1
  f2, f1 = 0, 1                      # buffers for f(n-2), f(n-1)
  for i in range(2, n+1):            # loop over [2,n]
    temp = f1 + f2
    f2 = f1
    f1 = temp
  return f1
```

`Complexity?` Time: **O**(n) Space: **O**(1)

Fibonacci Bonus

A closed form solution. Time: **O**(1) Space: **O**(1)

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

Dynamic Programming Methods

- **Top Down**: Memoization
    - Start at topmost state: solve(n)
    - Cache any calculated values [solve(i)] in an array.

- **Bottom Up**: Tabulation
    - Start at bottommost state: solve(0) {or 1}
    - Iterate over all states in order*, putting the result of solve(i) in a table.   (* hard part: find the order)

Coin Problem 2, DP Top down, Py

```
INF = float('inf')
def solve(n, denominations=[1,5,6,9]):
  M = [INF]*(n+1)           # 'Memoisation' array
  M[0] = 0                  # base case
  def dp(x):
    if x < 0: return INF    # invalid n
    if M[x] == INF:         # not in Mem array? ⇒ calc subproblem
      M[x] = min(dp(x-c) + 1 for c in denominations)
    return M[x]
  return dp(n)
solve(23)
```

O(nk)

```
26.3 µs ± 221 ns per loop (mean ± std. dev. of 7 runs, 10000
loops each)
```

DP perf

```
solve(23)
Naive 5.29 ms ± 75.9 µs per loop (mean ± std. dev. of 7
runs, 100 loops each)
DP    26.3 µs ± 221 ns per loop (mean ± std. dev. of 7 runs,
10000 loops each)
```

```
solve(230)
Naive  good luck.
DP    332 µs ± 5.09 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

Coin Problem 2, DP Bottom up, Py

```python
# Python
def solve(n, denominations=[1,5,6,9]):
  M = [0]*(n+1)               # Tabulation array
  for x in range(1, n+1):     # [1,n]
    M[x] = float('INF')
    for c in denominations:
      if x - c >= 0:
        M[x] = min(M[x], M[x-c]+1)
  return M[n]
```

Time: **O**(nk)   Space: **O**(n)

Coin Problem 2, DP Bottom up, C++

```cpp
// C++
int solve(int x) {
    int value[N]; value[0] = 0;
    for (int x = 1; x <= n; x++) {
        value[x] = INF;
        for (auto c : coins) {
            if (x-c >= 0) value[x] = min(value[x], value[x-c]+1);
        }
    }
    return x;
}
```

Time: **O**(nk)   Space: **O**(n)

Knapsack

- Many variants!
- Generally:
    - Given a set of objects, need to find subset with some properties.
- Brute force is exponential
- DP solution can give answer in psuedo-polynomial time.
- Used for many optimisation problems

Knapsack

- Got a knapsack to fill with valuable items.
- Only holds a certain amount of weight.
- Need to carry the most valuable items, each with their own weights.

Knapsack

- Got a knapsack to fill with valuable items.
- Only holds a certain amount of weight.
- Need to carry the most valuable items, each with their own weights.

Knapsack

- A set of items, with values v_i and weights w_i (value, weight)
    - i_1 = (42, 7), i_2 = (12, 3), i_3 = (40, 4), i_4 = (25, 5)

- Capacity of knapsack = 8

Knapsack

- A set of items, with values v_i and weights w_i (value, weight)
  - i_1 = (42, 7), i_2 = (12, 3), i_3 = (40, 4), i_4 = (25, 5)

- Capacity of knapsack = 8

best(i, w) = v

the value of the best choice of items out of the first i items, using capacity w.

Knapsack

- 2 choices for a given item i:
    1. Select it
        - $best(i, w) = best(i - 1, W - w_i) + v_i$
    2. Don't select it
        - $best(i, w) = best(i - 1, W)$
- Why?

Knapsack

- 2 choices for a given item i:
  1. Select it
     - $\text{best}(i, w) = \text{best}(i - 1, w - w_i) + v_i$
  2. Don't select it
     - $\text{best}(i, w) = \text{best}(i - 1, w)$
- Base case: K(i, w) = 0 if i=0 or w = 0
- Why?

Knapsack

-

$$K(i, w) = \begin{cases} max(K(i-1, w), K(i-1, w-w_i) + v_i) & \text{if } w \geq w_i \\ K(i-1, w) & \text{if } w < w_i \end{cases}$$

Knapsack solution

```
int K[N][W]; // n items, w weights
for (int i = 1; i <= N; i++) {
    for (int w = 1; w <= W; w++) {
        if (w < item_weight[i]){
            K[i][w] = K[i-1][w]
        } else{
            K[i][w] = max(K[i-1][w], K[i-1][w-item_weight[i]]
                        + item_value[i])
        } sum[i][w] = max(sum[y][x-1], sum[y-1][x])
                        + value[y][x];
      }
   }  // result: K[n, W]
```

Knapsack solution

	$i \backslash j$	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
$v_1 = 42, w_1 = 7$	1	0	0	0	0	0	0	0	42	42
$v_2 = 12, w_2 = 3$	2	0	0	0	12	12	12	12	42	42
$v_3 = 40, w_3 = 4$	3	0	0	0	12	40	40	40	52	52
$v_4 = 25, w_4 = 5$	4	0	0	0	12	40	40	40	52	52

Knapsack solution

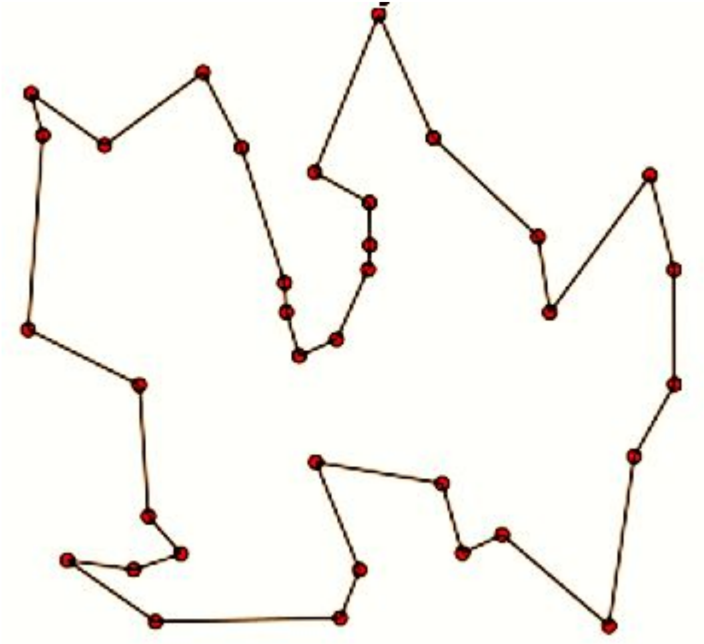
Time complexity: **O**(nW)

Space complexity: **O**(nW)

Where W = the maximum weight allowance of the knapsack

Travelling Salesman Problem (TSP)

Given a list of cities and distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

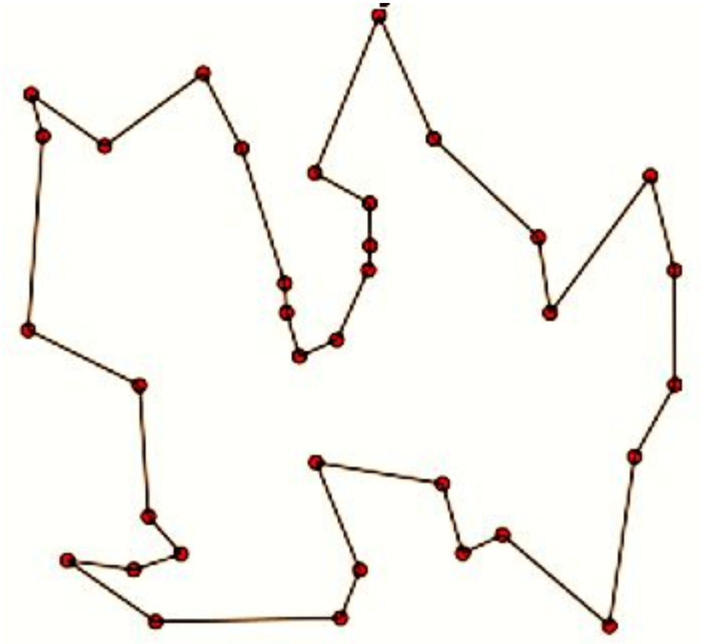


Brute Force:

DP:

Travelling Salesman Problem (TSP)

Given a list of cities and distances between each pair of cities,
what is the shortest possible route that visits each city and returns to the origin city?



Brute Force: **O**(n!)

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

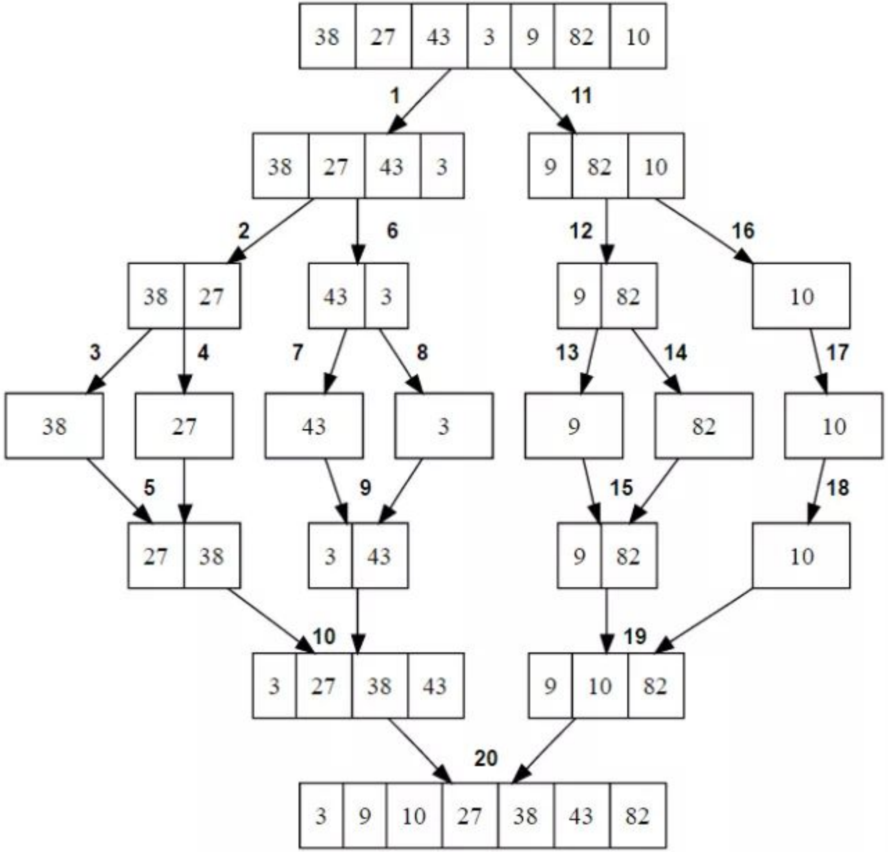
Held-Karp algorithm. (NP-Complete)

Divide and Conquer

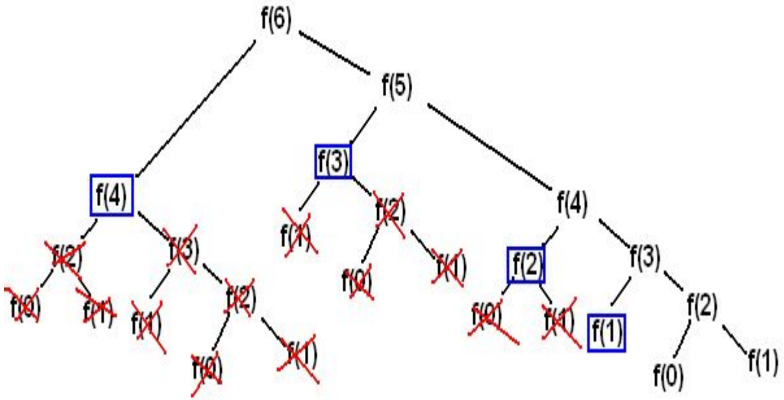
- Recursively break down problem into independent subproblems of the same type (**divide**)

- Solve the simpler subproblems directly (**conquer**)

- Combine the solutions of the subproblems to form a solution for the original problem



Dynamic Programming or Divide and Conquer?



DP: Overlapping subproblems.