# Dynamic Programming I

## Top-Down, Bottom-Up and Classical Problems

beOI Training
(many thanks to François Aubry)

OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

April 15, 2016

# Table of Contents

# Let's start with an example

**Partition Problem**:

Given a set of $n \leq 50$ goodies each with value $v[i] \leq 10$, is it possible to divide them between 3 persons evenly?

# Strategy

Put yourself in the shoes of the one who divides the goodies.

For each goodie, what **choices** can you make?

# Strategy

Put yourself in the shoes of the one who divides the goodies.

For each goodie, what **choices** can you make?

# Brute-Force?

Suppose we make a brute force algorithm that tries all choices.

What should we keep track of?

# Brute-Force?

Suppose we make a brute force algorithm that tries all choices.

What should we keep track of?

1. The items we considered so far.

   Does the order in which we consider the items matter?

# Brute-Force?

Suppose we make a brute force algorithm that tries all choices.

What should we keep track of?

1. The items we considered so far.

   Does the order in which we consider the items matter?
   **No**.

   $\Rightarrow$ Just keep an integer $i$ such that we have made choices for items $< i$.

# Brute-Force?

Suppose we make a brute force algorithm that tries all choices.

What should we keep track of?

1. The items we considered so far.

   Does the order in which we consider the items matter?
   **No**.

   $\Rightarrow$ Just keep an integer $i$ such that we have made choices for items $< i$.

2. At the end we need to know if the division is fair. Do we need to know the items given to each person?

# Brute-Force?

Suppose we make a brute force algorithm that tries all choices.

What should we keep track of?

1. The items we considered so far.

   Does the order in which we consider the items matter? **No**.

   $\Rightarrow$ Just keep an integer $i$ such that we have made choices for items $< i$.

2. At the end we need to know if the division is fair. Do we need to know the items given to each person? **No**.

   $\Rightarrow$ Just keep track of how much was given to each person.

# Brute-Force solution

```cpp
bool complete_search(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;
    else {
        bool giveTo1 = complete_search(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = complete_search(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = complete_search(i+1, given1, given2, given3 + v[i]);
        return giveTo1 || giveTo2 || giveTo3;
    }
}
```

# Brute-Force solution

```
bool complete_search(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;
    else {
        bool giveTo1 = complete_search(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = complete_search(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = complete_search(i+1, given1, given2, given3 + v[i]);
        return giveTo1 || giveTo2 || giveTo3;
    }
}
```

Complexity?

# Brute-Force solution

```
bool complete_search(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;
    else {
        bool giveTo1 = complete_search(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = complete_search(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = complete_search(i+1, given1, given2, given3 + v[i]);
        return giveTo1 || giveTo2 || giveTo3;
    }
}
```

Complexity? $\mathcal{O}(3^n)$ TLE

We can do better...

# State space and state graph

We call one tuple $(i, given1, given2, given3)$ a **state**.

We can now define the **state graph**: its vertices are the states, and there is an edge from $s_1$ to $s_2$ if $s_1$ calls $s_2$ recursively.

How many nodes does the state graph of the previous algorithm have?

$\mathcal{O}\left(n \cdot S^3\right)$ where $S$ is the sum of the goodie values.

But the algorithm is $\mathcal{O}\left(3^n\right)$. What's going on?

# State space and state graph

We call one tuple $(i, given1, given2, given3)$ a **state**.

We can now define the **state graph**: its vertices are the states, and there is an edge from $s_1$ to $s_2$ if $s_1$ calls $s_2$ recursively.

How many nodes does the state graph of the previous algorithm have?

$\mathcal{O}(n \cdot S^3)$ where $S$ is the sum of the goodie values.

But the algorithm is $\mathcal{O}(3^n)$. What's going on?

**Each state is visited several times $\Rightarrow$ waste of time!**

What we want to do is to traverse the state graph (DFS like).

How can we achieve that?

What we want to do is to traverse the state graph (DFS like).

How can we achieve that?

```
int canDo[n][S][S][S];
// undefined: -1
// we cannot divide evenly from that state: 0
// we can    divide evenly from that state: 1

bool solve(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;

    if(canDo[i][given1][given2][given3] != -1) {
        // the state (i, given1, given2, given3) has already been visited
        return canDo[i][given1][given2][given3];
    } else {
        bool giveTo1 = solve(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = solve(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = solve(i+1, given1, given2, given3 + v[i]);
        // save the value for later
        canDo[i][given1][given2][given3] = giveTo1 || giveTo2 || giveTo3;
        return canDo[i][given1][given2][given3];
    }
}
```

What we want to do is to traverse the state graph (DFS like).

How can we achieve that?

```
int canDo[n][S][S][S];
// undefined: -1
// we cannot divide evenly from that state: 0
// we can    divide evenly from that state: 1

bool solve(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;

    if(canDo[i][given1][given2][given3] != -1) {
        // the state (i, given1, given2, given3) has already been visited
        return canDo[i][given1][given2][given3];
    } else {
        bool giveTo1 = solve(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = solve(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = solve(i+1, given1, given2, given3 + v[i]);
        // save the value for later
        canDo[i][given1][given2][given3] = giveTo1 || giveTo2 || giveTo3;
        return canDo[i][given1][given2][given3];
    }
}
```

Is this enough to get AC?

What we want to do is to traverse the state graph (DFS like).

How can we achieve that?

```
int canDo[n][S][S][S];
// undefined: -1
// we cannot divide evenly from that state: 0
// we can    divide evenly from that state: 1

bool solve(int i, int given1, int given2, int given3) {
    if(i == n) // all goodies have been considered
        return given1 == given2 && given2 == given3;

    if(canDo[i][given1][given2][given3] != -1) {
        // the state (i, given1, given2, given3) has already been visited
        return canDo[i][given1][given2][given3];
    } else {
        bool giveTo1 = solve(i+1, given1 + v[i], given2, given3);
        bool giveTo2 = solve(i+1, given1, given2 + v[i], given3);
        bool giveTo3 = solve(i+1, given1, given2, given3 + v[i]);
        // save the value for later
        canDo[i][given1][given2][given3] = giveTo1 || giveTo2 || giveTo3;
        return canDo[i][given1][given2][given3];
    }
}
```

Is this enough to get AC? **No**.
The graph has $\approx n \cdot S^3 = 50 \cdot 500^3 = 6250000000$ nodes $\Rightarrow$ MLE.


Or can we make it work?

# State space reduction

Observe that at the end $given3 = S - given1 - given2$.

Thus we can drop one parameter and reduce the state space to $n \cdot S^2$.

```
int canDo[n][S][S];
// undefined: -1
// we cannot divide evenly from that state: 0
// we can    divide evenly from that state: 1
int S; // sum of the v[i]

bool solve(int i, int given1, int given2) {
    if(i == n) { // all goodies have been considered
        int given3 = S - given1 - given2;
        return given1 == given2 && given2 == given3;
    }
    if(canDo[i][given1][given2] != -1) {
        // the state (i, given1, given2) has already been visited
        return canDo[i][given1][given2];
    } else {
        bool giveTo1 = solve(i+1, given1 + v[i], given2);
        bool giveTo2 = solve(i+1, given1, given2 + v[i]);
        bool giveTo3 = solve(i+1, given1, given2);
        // save the value for later
        canDo[i][given1][given2] = giveTo1 || giveTo2 || giveTo3;
        return canDo[i][given1][given2];
    }
}
```

# What we learned so far

1. View the problem as a **sequence of choices**.
2. Represent the problem with the smallest state space possible.
3. Perform a DFS on the state graph (remembering visited states).

# What we learned so far

1. View the problem as a **sequence of choices**.
2. Represent the problem with the smallest state space possible.
3. Perform a DFS on the state graph (remembering visited states).

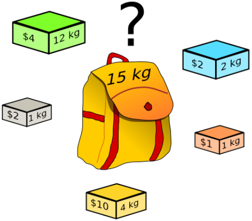
Let's see another example!

# Table of Contents

# Knapsack Problem

Given a set of $n$ objects each with value $v[i]$ and weight $w[i]$, and a knapsack that can hold a total capacity of $C$.

Choose a subset of objects that fits into the knapsack and has maximum total value.



In what way is this problem similar to the previous one?

In what way is this problem similar to the previous one?

- Succession of choices: for each item, take it or leave it.
- Order does not matter.

State space?

In what way is this problem similar to the previous one?

- Succession of choices: for each item, take it or leave it.

- Order does not matter.

State space? $(i, wtaken)$

- $i = item\ we\ are\ considering$
- $wtaken = total\ weight\ of\ the\ items\ we\ selected\ so\ far$

Size of the state space?

In what way is this problem similar to the previous one?

- Succession of choices: for each item, take it or leave it.

- Order does not matter.

State space? $(i, wtaken)$

- $i$ = *item we are considering*
- $wtaken$ = *total weight of the items we selected so far*

Size of the state space? $\mathcal{O}(n \cdot C)$

Successors of state $(i, wtaken)$?

Successors of state $(i, wtaken)$?

$$
\begin{tikzcd}
 & (i+1, wtaken) \\
(i, wtaken) \arrow[ur, "\text{do not take item } i"] \arrow[dr, "\text{take item } i", "\text{if } wtaken + w[i] \leq C"'] & \\
 & (i+1, wtaken + w[i])
\end{tikzcd}
$$

Recurrence relation?

Recurrence relation?

$$f(i, wtaken) = \max\Big(f(i{+}1, wtaken), v[i]{+}f(i{+}1, wtaken{+}w[i])\Big)$$

Beware of the knapsack constraint! If $wtaken > C$, the knapsack has no value.

$$wtaken > C \Rightarrow f(i, wtaken) = -\infty$$

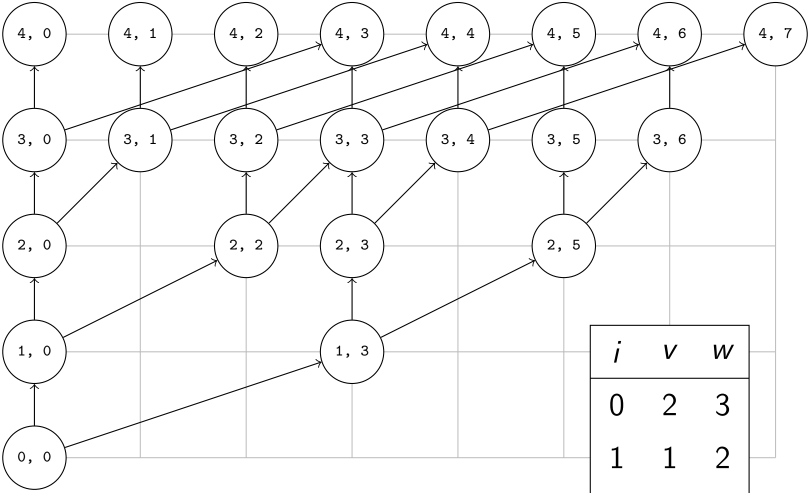
# Knapsack solution

```
int C, n;
int w[n], v[n];
int memo[n][C];

int solve(int i, int wtaken) {
    if(i == n) {
        // we cannot take any object anymore
        return 0;
    } else if(wtaken > C) {
        // knapsack is invalid
        return -INF;
    }

    if(memo[i][wtaken] != -1)
        return memo[i][wtaken];
    else {
        memo[i][wtaken] = max(solve(i+1, wtaken), v[i] + solve(i+1, wtaken - w[i]));
        return memo[i][wtaken];
    }
}
```

# Example of a Knapsack state space.




| $i$ | $v$ | $w$ |
|---|---|---|
| 0 | 2 | 3 |
| 1 | 1 | 2 |
| 2 | 2 | 1 |
| 3 | 3 | 3 |

# Memory optimization

Observe that we don't need all the entries from the `memo` table.

Sometimes the table is too big and causes MLE.

In this situation an alternative is to use a `HashMap` (or `unordered_map`).

This way we only use the memory we need.

# Another approach

This is one view of Dynamic Programming. Usually refered to as **memoization**.

Another view is to decompose the problem into **sub-problems**.

Define an order on the sub-problems: The bigger the harder.

Express the solution of large sub-problems as a function of smaller sub-problems.

Solve from smaller to larger using the function.

# Let's redo the Knapsack this way

Decomposition into sub-problems:

$$\begin{aligned} best[i][c] = &\text{ best way to take objects } 0, 1, \cdots, i \\ &\text{ on a knapsack of capacity } c \end{aligned}$$

Observe that the real problem we want to solve is $best[n-1][C]$.

The idea is that maybe $best[i][c]$ relates to $best[i-1][c']$.

# Case 1: item $i$ **does not** belong to Knapsack

Suppose we know ***magically*** that item $i$ **does not belong** to the optimal solution of $best[i][c]$.

Then we **forget** about $i$ and take items $0, 1, \cdots, i-1$ in the best possible way in the same knapsack.

That is, $best[i][c] = best[i-1][c]$.

# Case 2: item $i$ **does** belong to Knapsack

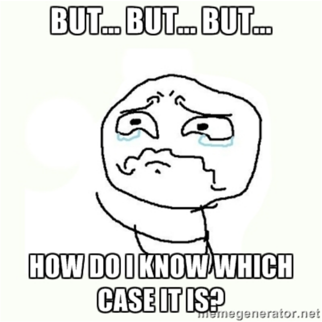
But what if item $i$ belongs to the optimal solution of $best[i][c]$?

In this case we simply start by putting item $i$ into the knapsack.

Then we put items $0, 1, \cdots, i-1$ in the best possible way in a new knapsack of capacity $c - w[i]$.

Thus, $best[i][c] = v[i] + best[i-1][c - w[i]]$.


BUT... BUT... BUT...
HOW DO I KNOW WHICH CASE IT IS?
memegenerator.net




Well... who cares? We know that item $i$ either is or isn't in the knapsack.

So... just take the best of the two possibilities!

$$best[i][c] = \max\left(best[i-1][c], v[i] + best[i-1][c-w[i]]\right)$$

Note that $best[i-1][c-w[i]]$ might not be defined if $c < w[i]$.

It remains to solve the easiest sub-problems, when $i = 0$.

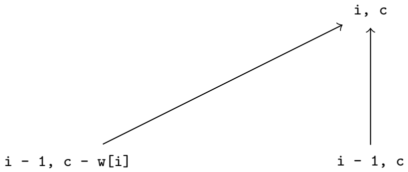
It remains to solve the easiest sub-problems, when $i = 0$.

$$
best[0][c] = \begin{array}{ll} v[0] & \text{if } c \geq w[0] \\ 0 & \text{otherwise} \end{array}
$$

# It all comes down to graphs

We can also think about sub-problems as nodes in a graph.

The edges link sub-problems as nodes in a graph.




We need to solve the sob-problems in a **topological order** of this graph.

$best[i-1][c]$ and $best[i-1][c-w[i]]$ need to be solved before $best[i][c]$.

# Implementation

```
int n, C;
int w[n], v[n];

int knapsack() {
    int best[n][C+1];

    // solve the base case (easier sub-problems)
    for(int c = 0; c <= C; ++c)
        if(c < w[0])
            best[0][c] = 0;
        else
            best[0][c] = v[0];

    // iterate in the right order and solve all sub-problems
    for(int i = 1; i < n; ++i)
        for(int c = 0; c <= C; ++c)
            if(c < w[i])
                best[i][c] = best[i-1][c];
            else
                best[i][c] = max(best[i-1][c], v[i] + best[i-1][c - w[i]]);

    // return biggest problem
    return best[n-1][C];
}
```

For most DP problems, a topological order can be achieved simply with the proper sequencing of some (nested) loops.

# Table of Contents

# Two approaches

The first approach starts from the hardest sub-problem (the pair $(0, 0)$) and breaks it down into easier sub-problems.

We call that a **Top-Down DP**.

The second approach starts from the easy sub-problems and builds up harder sub-problems upon it.

We call that a **Bottom-Up DP**

Generally, Top-Down is implemented **recursively** and Bottom-Up **iteratively**.

How do you know which one you should use?

# Top-Down DP vs Bottom-Up DP

Top-Down:

+ Only computes relevant states.

+ Easier to come up with.

Bottom-Up:

+ Usually possible to reduce the memory.

✎Implement Knapsack with $\mathcal{O}(C)$ memory.

– Solves all sub-problems.

# Table of Contents

## Let's solve a problem together

# UVa 562: Dividing coins

# Table of Contents

# Shortest path problem

Given a directed, weighted graph $G$ and a vertex $s$, compute the length of the shortest path from $s$ to all other vertices.

Let's solve this problem using **DP**.

What could be our sub-problems?

Let's try

$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

How does $sp[v]$ relate with other problems?

Let's try

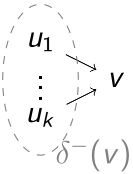
$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

How does $sp[v]$ relate with other problems?

$$sp[s] = 0$$

$$sp[v] = \min_{u \in \delta^-(v)} sp[u] + w(u, v)$$

where $\delta^-(u)$ is the set of in-neighbours of $v$.




Does this work? **No!**

Let's try

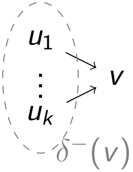
$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

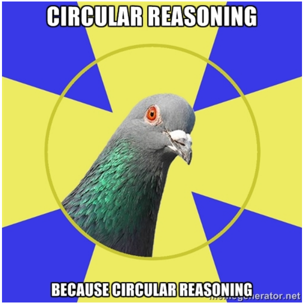
How does $sp[v]$ relate with other problems?

$$sp[s] = 0$$

$$sp[v] = \min_{u \in \delta^-(v)} sp[u] + w(u, v)$$

where $\delta^-(u)$ is the set of in-neighbours of $v$.




Does this work? **No!**

If $v$ belongs to a **cycle**, $sp[v]$ depends on $sp[v]$...

For **DP** to work, we need the underlying sub-problem graph to be **acyclic**.

Observe that the **sub-problem** graph is actually...the input graph $G$.

The above recurrence works for acyclic graphs.

# Shortest path for acyclic graphs

$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

$$sp[s] = 0$$

$$sp[v] = \min_{u \in \delta^-(v)} sp[u] + w(u, v) \quad \forall v \in V \setminus \{s\}$$

In what order should we compute the problems?

# Shortest path for acyclic graphs

$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

$$sp[s] = 0$$

$$sp[v] = \min_{u \in \delta^-(v)} sp[u] + w(u,v) \quad \forall v \in V \setminus \{s\}$$

In what order should we compute the problems?

We must have computed $sp[u]$ for all $u \in \delta^-(v)$ before we compute $sp[v]$.

$\Rightarrow$ we need to do it in the **topological order** of $G$.

# Shortest path for acyclic graphs

$$sp[v] = \text{length of the shortest path from } s \text{ to } v$$

$$sp[s] = 0$$

$$sp[v] = \min_{u \in \delta^-(v)} sp[u] + w(u,v) \quad \forall v \in V \setminus \{s\}$$

In what order should we compute the problems?

We must have computed $sp[u]$ for all $u \in \delta^-(v)$ before we compute $sp[v]$.

$\Rightarrow$ we need to do it in the **topological order** of $G$.

That is not surprising, we said we **always** evaluate DP states in the topological order of the sub-problem graph.
And in this case it is $G$.

# Sub-problem graph must be acyclic

This just to say that

**DP only works if your state space is acyclic!**

Be careful defining your state space.

# Table of Contents

# Exercises

- UVa 10130
- UVa 562
- UVa 624

- UVa 990
- UVa 10261

- UVa 10032