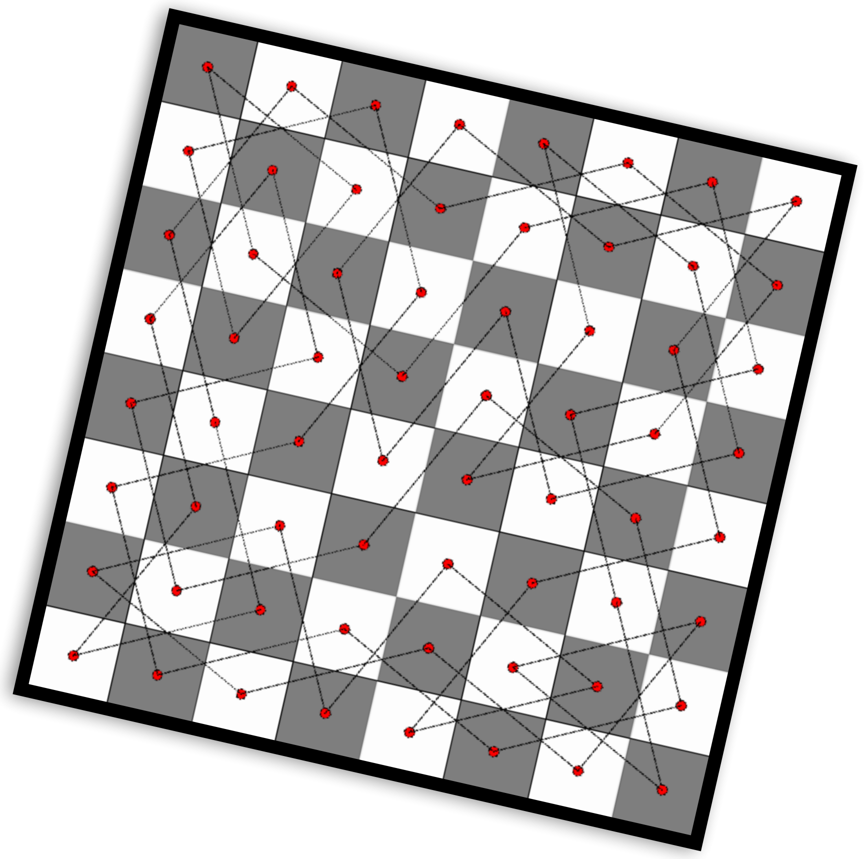
Dynamic Programming

LINFO2266

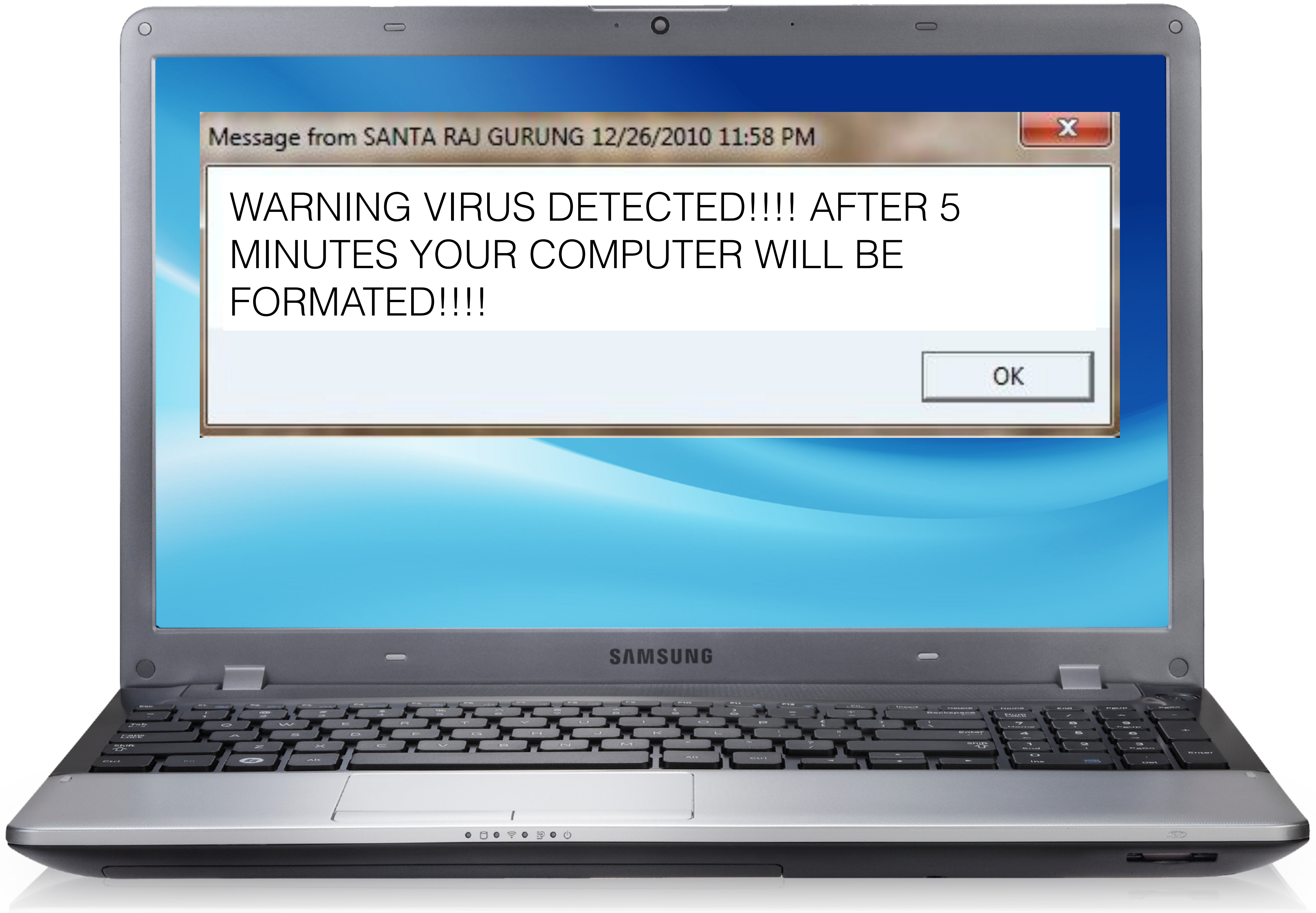
https://github.com/pschaus/linfo2266

Pierre Schaus



Knapsack Problem
Brute force recursive approach

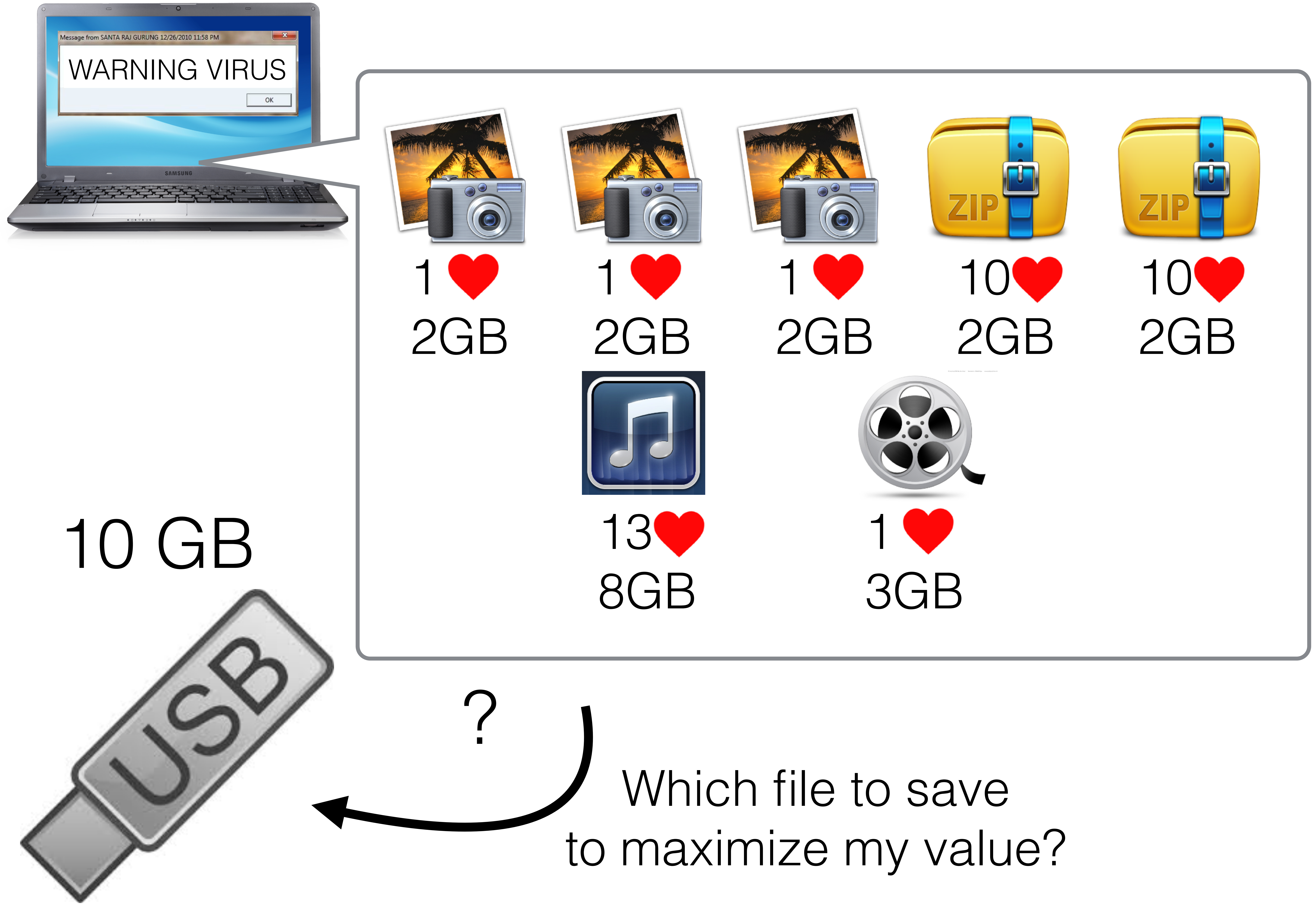
Motivation



WARNING VIRUS DETECTED!!!! AFTER 5 MINUTES YOUR COMPUTER WILL BE FORMATED!!!!

10 GB

The problem



Which file to save
to maximize my value?

Knapsack Problem

- the set of items: I
- is item i selected: $x_i \in \{0,1\}$
- Objective: Maximize $\sum_{i \in I} v_i x_i$
- Constraints: $\sum_{i \in I} w_i x_i \leq C$

Maximize value of selected items

Under capacity constraint

Is this problem NP-Hard?

- Yes if the related decision problem is NP-Complete

$$\sum_{i \in I} v_i x_i \geq V$$

$$\sum_{i \in I} w_i x_i \leq C$$

$$x_i \in \{0, 1\}$$

- We know *subset sum* is NP-Complete

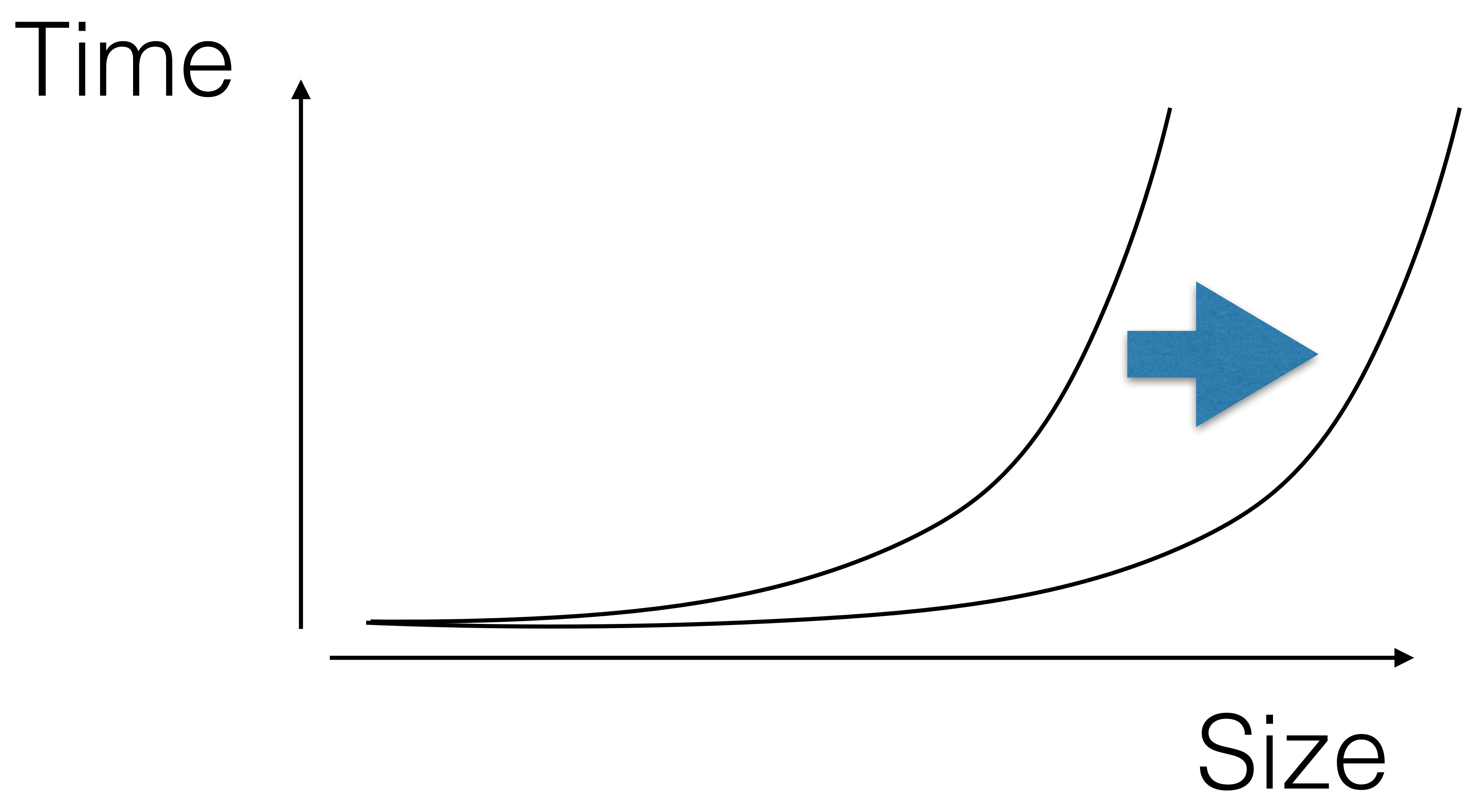
$$\texttt{Natural numbers } c_1, \ldots, c_n, K.$$

$$\texttt{Find } S \subseteq \{1, \ldots, n\} \texttt{ s.t. } \sum_{j \in S} c_j = K$$

- Exercise: Find a (polynomial) **reduction** from subset sum to knapsack (if you can solve knapsack efficiently, then you can solve subset sum efficiently).

Solving Knapsack: Brute-force

- Try every possible solutions
  - n items, 2^n solutions
  - n = 50, 1 ms to test one solution, > 30.000 years



Knapsack: Recursive Bruteforce

- Assume *I* = {*1,…,n*}

- Optimal objective of the problem with capacity *k* and items {*1,…,j*} is *O(k,j)*

$$\text{maximize} \quad \sum_{i \in \{1,\ldots,j\}} v_i x_i$$
$$\text{subject to} \quad \sum_{i \in \{1,\ldots,j\}} w_i x_i \leq k$$

- We are interested in *O(C,n)*

Knapsack Recursive Bruteforce

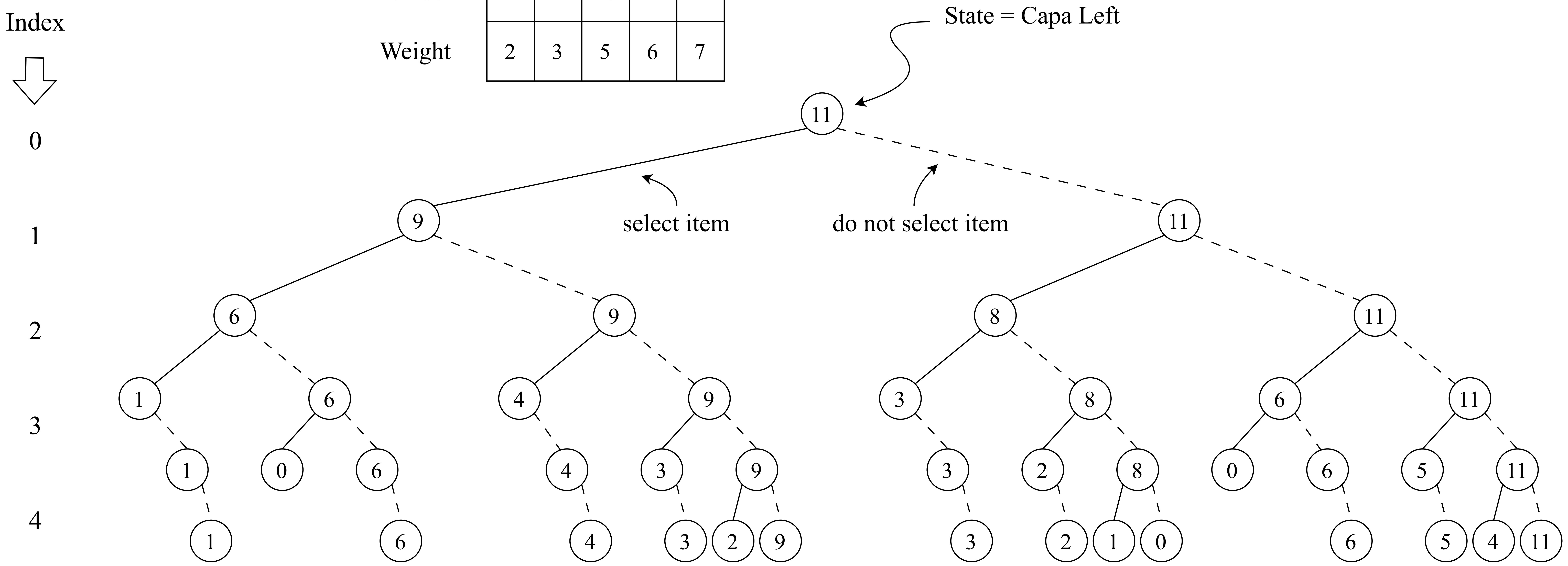
- Notation: $O(k,j)$ = optimal solution on items 0..j with capacity k
- Should we select item j (if $w_j \leq k$)?
  - If we don't select it the best solution is $O(k,j-1)$
  - If we select it the best solution $v_j + O(k-w_j, j-1)$
- Recursive equations:
  - *General case:*
    - $O(k,j) = \max(O(k,j-1)\ ,\ v_j + O(k-w_j, j-1))$ if $w_j \leq k$
    - $O(k,j) = O(k,j-1)$ otherwise
  - Base case:
    - $O(k,0) = 0$ for all k

Bellman Recurrence Equations

Knapsack Problem
Brute force recursive approach
Java Implementation

Pure brute force: backtrack if capa < 0

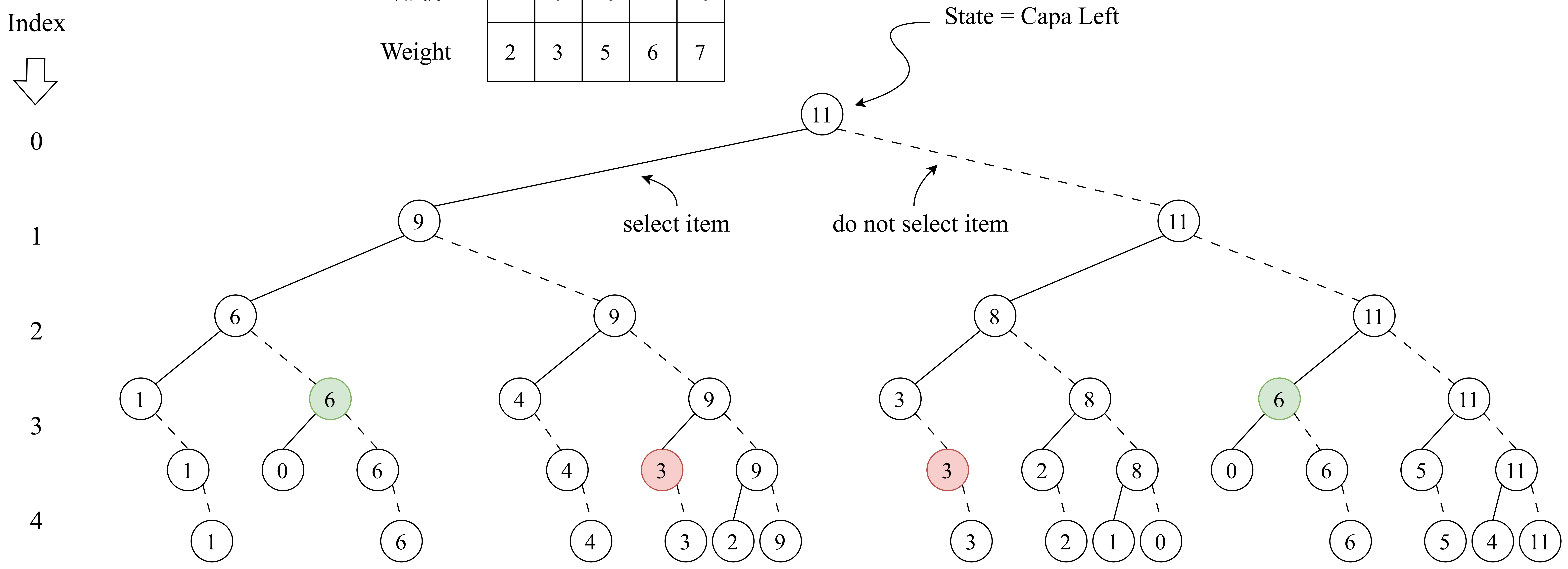
Index	0	1	2	3	4
Value	1	6	18	22	28
Weight	2	3	5	6	7



Knapsack Problem
Dynamic Programming

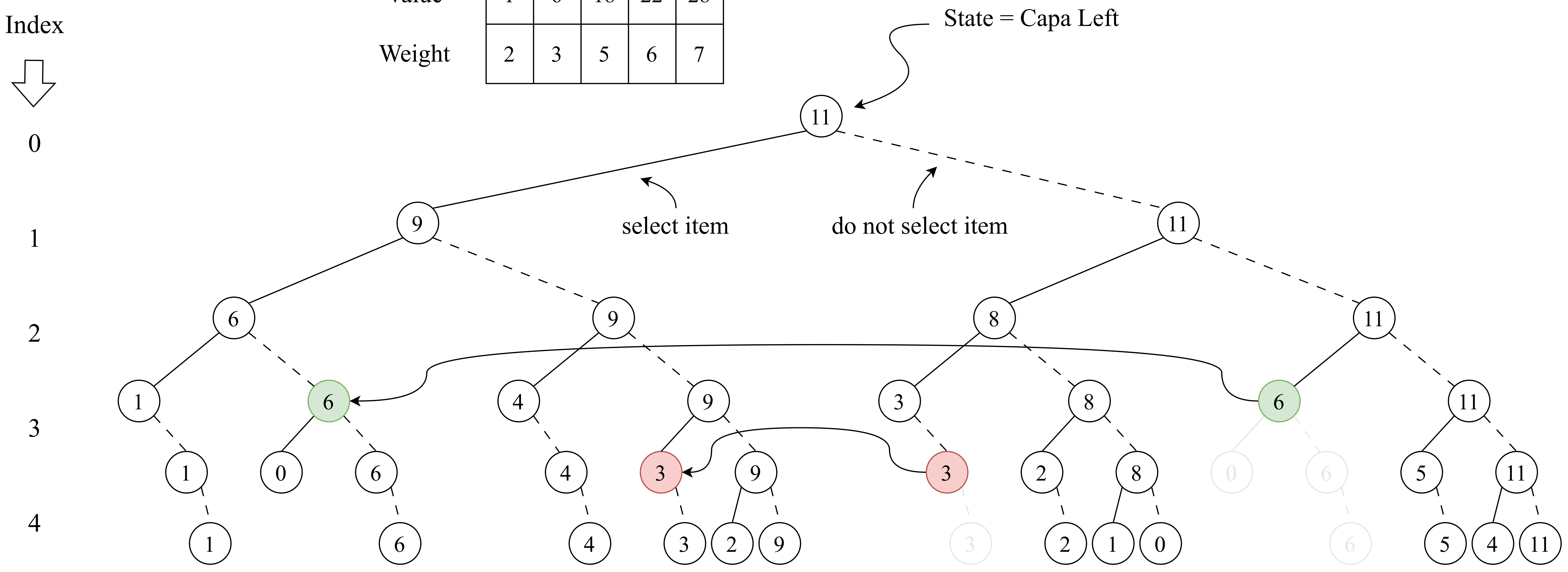
Some States are equivalent: DP idea don't recompute them

Index	0	1	2	3	4
Value	1	6	18	22	28
Weight	2	3	5	6	7



Store the state and retrieve them

Index	0	1	2	3	4
Value	1	6	18	22	28
Weight	2	3	5	6	7

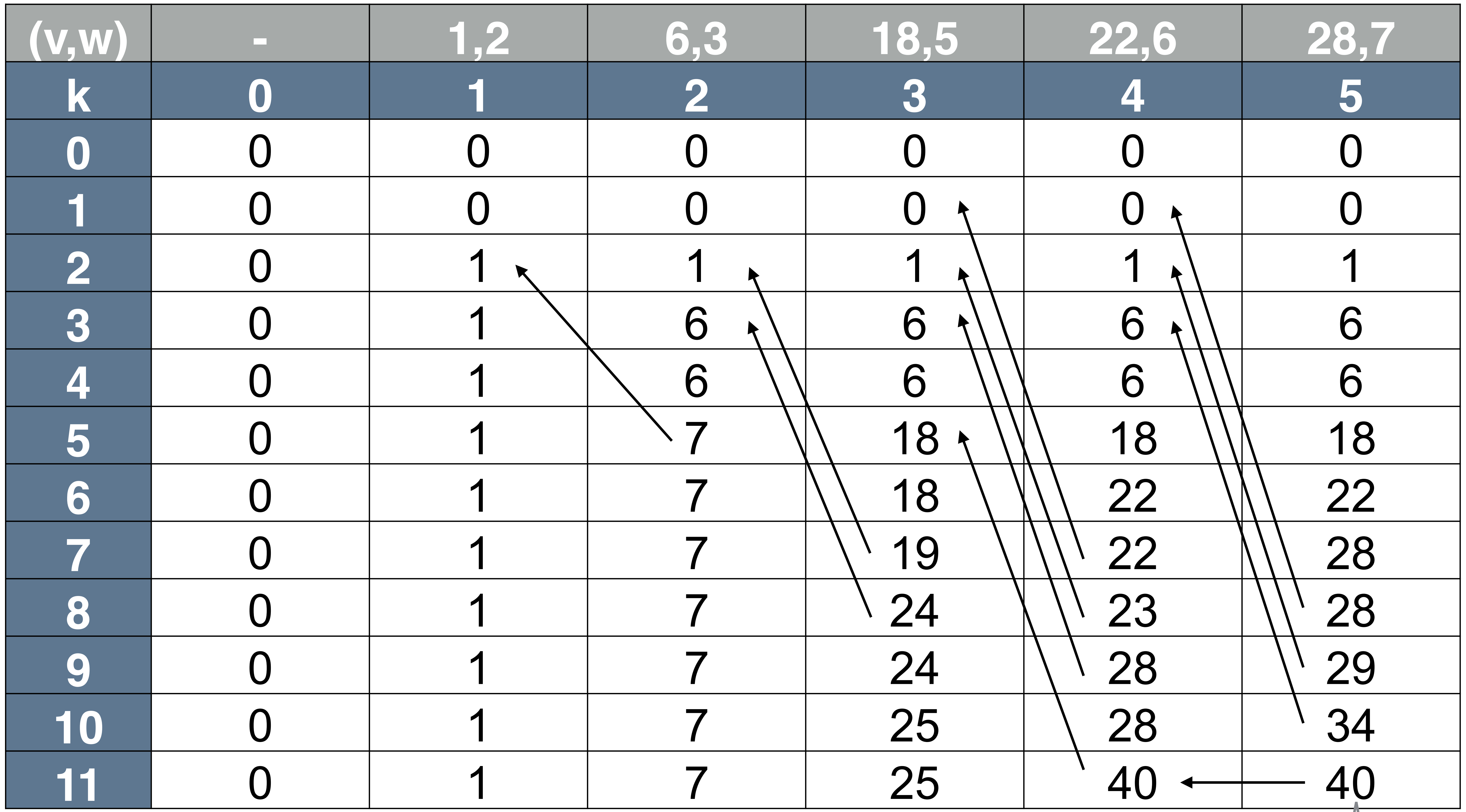


A MAP is needed

- For the knapsack we can also use a table since the capacity is fixed

- We need two dimensions:
  - Index (of item in the search tree)
  - Capacity left

Knapsack DP: Implem with Table



(v,w)	-	1,2	6,3	18,5	22,6	28,7
k	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	1	1	1	1	1
3	0	1	6	6	6	6
4	0	1	6	6	6	6
5	0	1	7	18	18	18
6	0	1	7	18	22	22
7	0	1	7	19	22	28
8	0	1	7	24	23	28
9	0	1	7	24	28	29
10	0	1	7	25	28	34
11	0	1	7	25	40	40

What is the time complexity?
How to retrieve the solution?

Optimal Value

- See code KnapsackTable

Knapsack DP: Implem with Table

- Time and Space Complexity: Θ*(C.n)*
- Is this polynomial ?

suggestion: divide C and w_i's by their greatest common divisor

- No! Because *log(C)* bits are necessary to represent *C*, the complexity is exponential wrt to the input size.
- We say it is pseudo-polynomial:
    - Can be considered as roughly polynomial for small value of *C*
    - But quickly becomes expensive to compute for large values of *C*
- *Hint: For large C, you can scale down (approximation)*

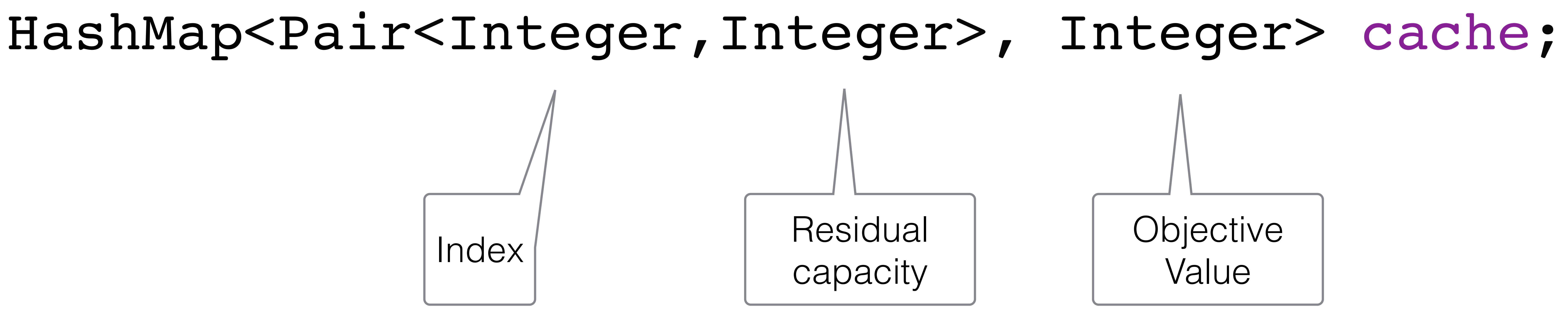
Knapsack is Weakly NP-Hard

An NP-complete (or NP-hard) problem is **weakly NP-complete** (or weakly NP-hard), if there is an algorithm for the problem whose running time is polynomial in the dimension of the problem and the magnitudes of the data involved (provided these are given as integers), rather than the base-two logarithms of their magnitudes.

http://en.wikipedia.org/wiki/Weakly_NP-complete

Not every NP-complete are weakly NP-complete. TSP is NP-complete in the strong sense, bin-packing as well.

- Alternatively we can use a standard hash-table for the MAP using



```java
HashMap<Pair<Integer,Integer>, Integer> cache;
```

- See code KnapsackHash.java

Knapsack DP: Implem with cache

- Time and Space
- Complexity: ~~$\Theta(C.n)$~~ $O(C.n)$

Only 30/72 cells are stored in the cache

(v,w)	-	1,2	6,3	18,5	22,6	28,7
k	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	1	1	1	1	1
3	0	1	6	6	6	6
4	0	1	6	6	6	6
5	0	1	7	18	18	18
6	0	1	7	18	22	22
7	0	1	7	19	22	28
8	0	1	7	24	23	28
9	0	1	7	24	28	29
10	0	1	7	25	28	34
11	0	1	7	25	40	40

Knapsack DP: An nV algorithm

- We have seen an O(nC) DP algo
- We can also design a O(nV) DP algo with $V = \sum_{i \in I} v_i$
- Interesting if C is large but values are small

$$maximize \sum_{i \in I} v_i x_i$$

$$subject\ to \sum_{i \in I} w_i x_i \leq C$$

$$x_i \in \{0, 1\}$$

Nice exam question?

- Exercise: Design a O(nV) DP algo. Hint: subproblem O(i,p) = minimum weight using only items 1..i with total value equal to p.

Optimization of a Range Partitioning

Problem Statement

Given

(1) an arrangement $S=[s_1, \dots, s_n]$ of nonnegative numbers

(2) an integer k,

The objective is to partition S into k or fewer ranges, to minimize the maximum sum over all the ranges without reordering the numbers.

Example:

$$S = [1, 2, 3, 4, 5, 6, 7, 8, 9] \text{ and } k = 3.$$

An optimal partition into contiguous ranges is

[1, 2, 3, 4, 5], [6, 7], **[8, 9]**

with the largest one having a sum of **17**.

Just to make sure ;-) ?

S = [1, 2, 6, 3, 1, 4, 5, 6, 7, 8, 5] and k = 4.

Optimal value?

- 13
- 9
- 10
- 15
- 12

Typical Exam Question

- Formulate this problem as a dynamic program. Write recurrence equations (don’t forget the base-cases)

- Sketch the code to solve it.

- What is the time complexity to solve this dynamic program (justify).

- Illustrate the execution and solution of your dynamic program on the following arrangement

  - S = [10,2,3,4,5,1,7,8,4].

Typical Exam mistakes

- Formulate this problem as a dynamic program.Write recurrence equations (don’t forget the base-cases)

A recurrence equation is given but

•It is not understandable 😢.

•The range of parameters are not specified 😢 and

•What they represent is not explained 😢 .

Example of bad (incomplete) answer

$$O(i,l) = \max_j \min\left((s_i + s_{i+1} \ldots + s_j), O(i, l-1)\right)$$

What is i,l ?

What is the range of j

What does O represent?

No base case, what are the ranges for i,l? Can it be negative?

- Unfortunately you should get a grade of zero for this answer.
- Can you fix this?

Choose the correct recurrence equation

Let $O(i, l)$ with $i \in [1..n]$, $l \in [2..k]$ denote the optimal value of the problem on the prefix sequence $[s_1, ..., s_i]$ using at most l partitions. $O(i, l) =$

🤔

$$1 : \max\left(O(i-1, l), O(i-1, l-1) + s_i\right)$$

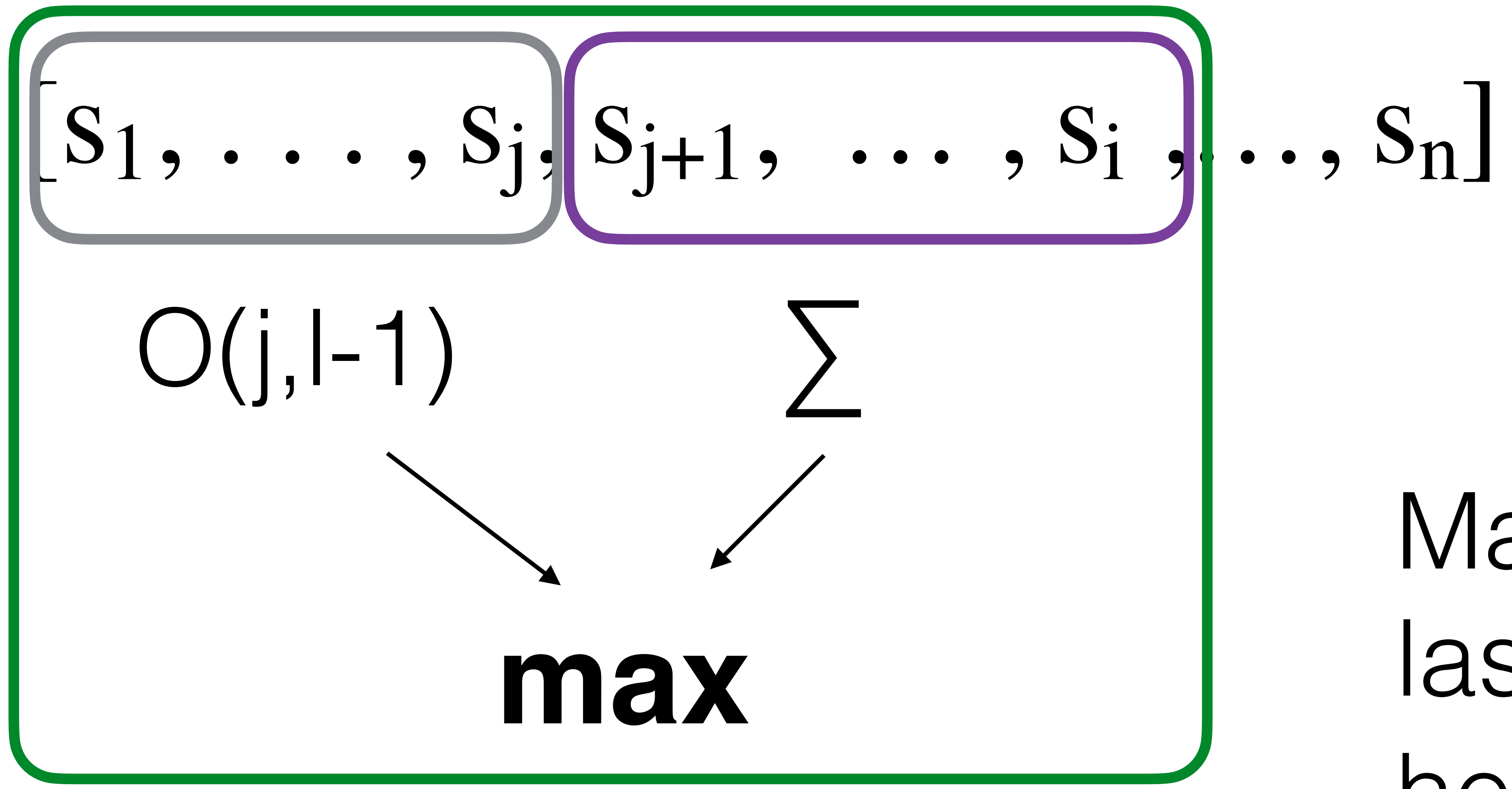
$$2 : \min\left(O(i-1, l), O(i-1, l-1) + s_i\right)$$

$$3 : \max_{j \in [1..i-1]} \min\left((s_{j+1} + s_{j+2} + \ldots + s_i), O(j, l-1)\right)$$

$$4 : \min_{j \in [1..i-1]} \max\left((s_{j+1} + s_{j+2} + \ldots + s_i), O(j, l-1)\right)$$

$$5 : \min_{j \in [1..i-1]} \max\left((s_{i+1} + s_{i+2} + \ldots + s_j), O(j, l-1)\right)$$

Explanation



Max is necessary in case the last partition [s_{j+1}, … , s_i] is the heaviest one

$$O(i,l) = \min_{j\in[1..i-1]} \boxed{\max\left((s_{j+1}+s_{j+2}+\ldots+s_i), O(j,l-1)\right)}$$

Base case

- O(i,l) i ∈ [1..n] and l ∈ [1..k].
- The base case is the one with one partition:
  - O(i,1) = s1+ . . . + si
- And the one with the sequence of length 1
  - O(1,l) = s1 for all l in [1..k].
- And this is not all, in your answer don’t forget to characterize the optimal solution… :
  - **The optimal solution is O(n,k)**

Table-Based Implementation

- O(5,2) = min(max(10,5),max(6,4+5),max(3,(3+4+5),max(1,(2+3+4+5))
- O(5,2) = min(10,9,12,14) = 9

k/s	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	1	3	6	10	15	21	28	36	45
2	1				**?**				
3	1								

$$O(i,l) = \min_{j \in [1..i-1]} \max\left((s_{j+1} + s_{j+2} + \ldots + s_i), O(j, l-1)\right)$$

Time-Complexity Analysis

$$O(i,l) = \min_{j \in [1..i-1]} \max\left((s_{j+1} + s_{j+2} + \ldots + s_i), O(j, l-1)\right)$$

- Time complexity to fill in the table ?

- Can we go faster to compute $s_{j+1} + s_{j+2} + \ldots + s_i$?

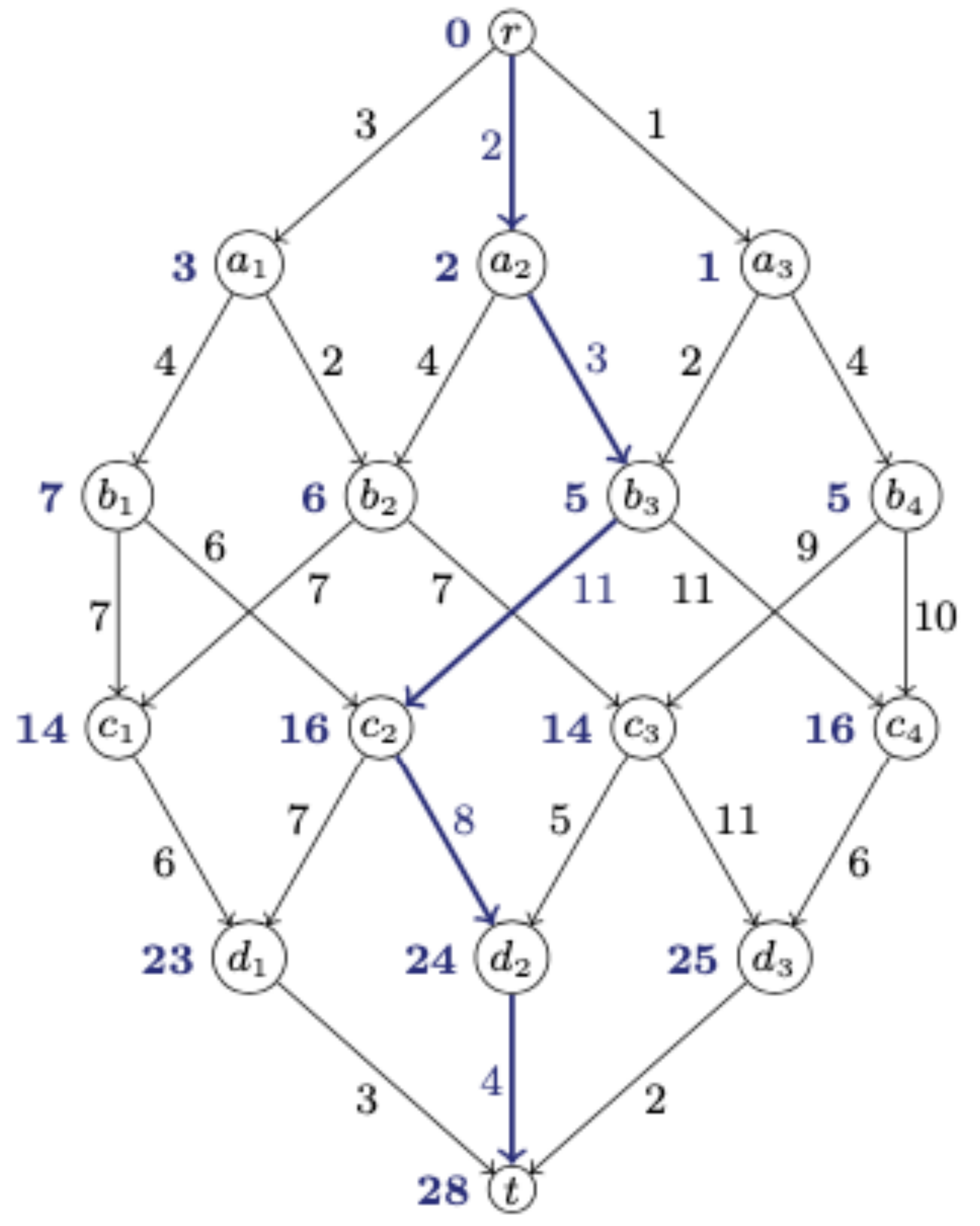
Range Partitioning

https://github.com/pschaus/linfo2266

Is it possible to make a generic Dynamic Solver ?

Yes!

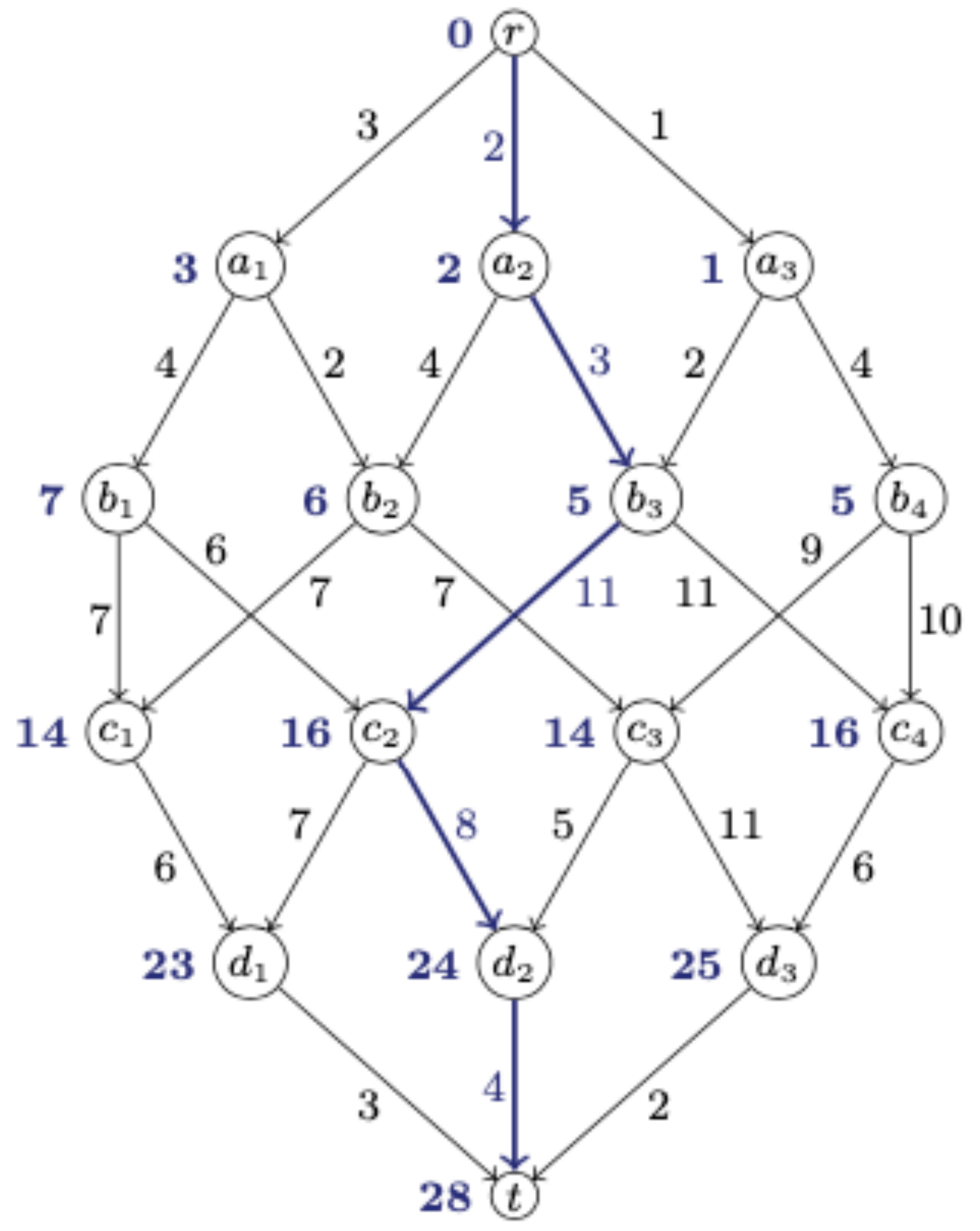
- Because every dynamic program can be reduced to a shortest (minimization) or (longest) path problem in a Directed Acyclic (Layered) Graph (DAG).



Longest path problem in a general graph is an NP-Hard problem but not in a DAG

Yes!

- Because every maximization dynamic program can be reduced to a longest path problem in a Directed Acyclic (Layered) Graph (DAG)

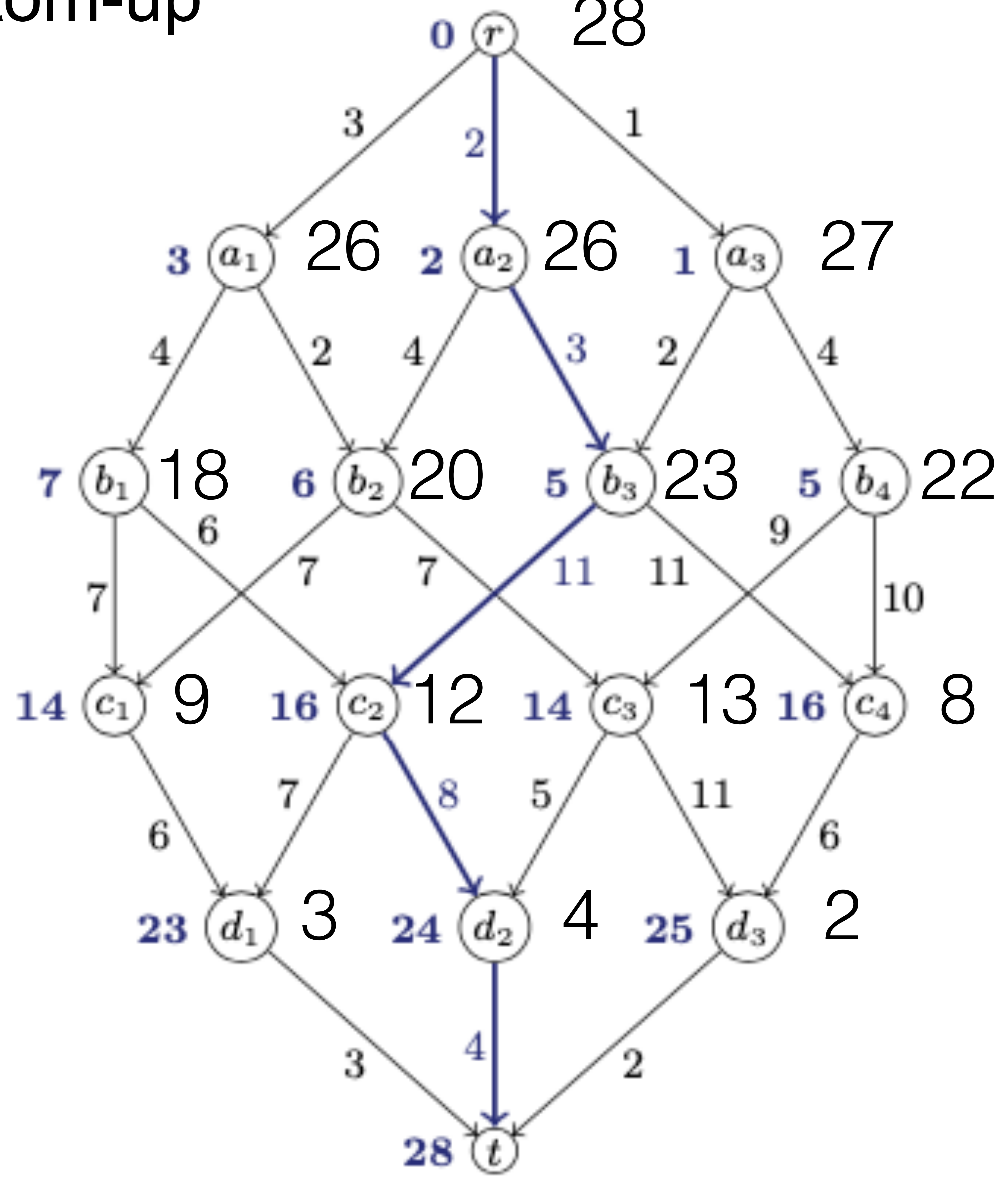


Let L(i) the longest path from i to the tail node t.
Can you give the DP recurrence equation ?

$$L(i) = \max_{j \in succ(i)} cost(i,j) + L(j)$$

Finding the longest path

- Linear time bottom-up



$$L(i) = \max_{j \in succ(i)} cost(i,j) + L(j)$$

The framework we propose

- Is composed of three classes:
  - State (= nodes of the DAG)
  - Transition = Edges (directed) of the DAG
  - The Model is able to generate successor states from a state and also identify the root r and the sink t

- A global Hash Table is used during the solving process

Best objective value for the state

```
HashMap<State, Double> table;
```

- Therefore State's must be *hashable*

State.java

```java
abstract class State {

    abstract int hash();

    abstract boolean isEqual(State state);

    @Override
    public int hashCode() {
        return this.hash();
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof State) {
            State state = (State) o;
            return isEqual(state);
        }
        return false;
    }

}
```

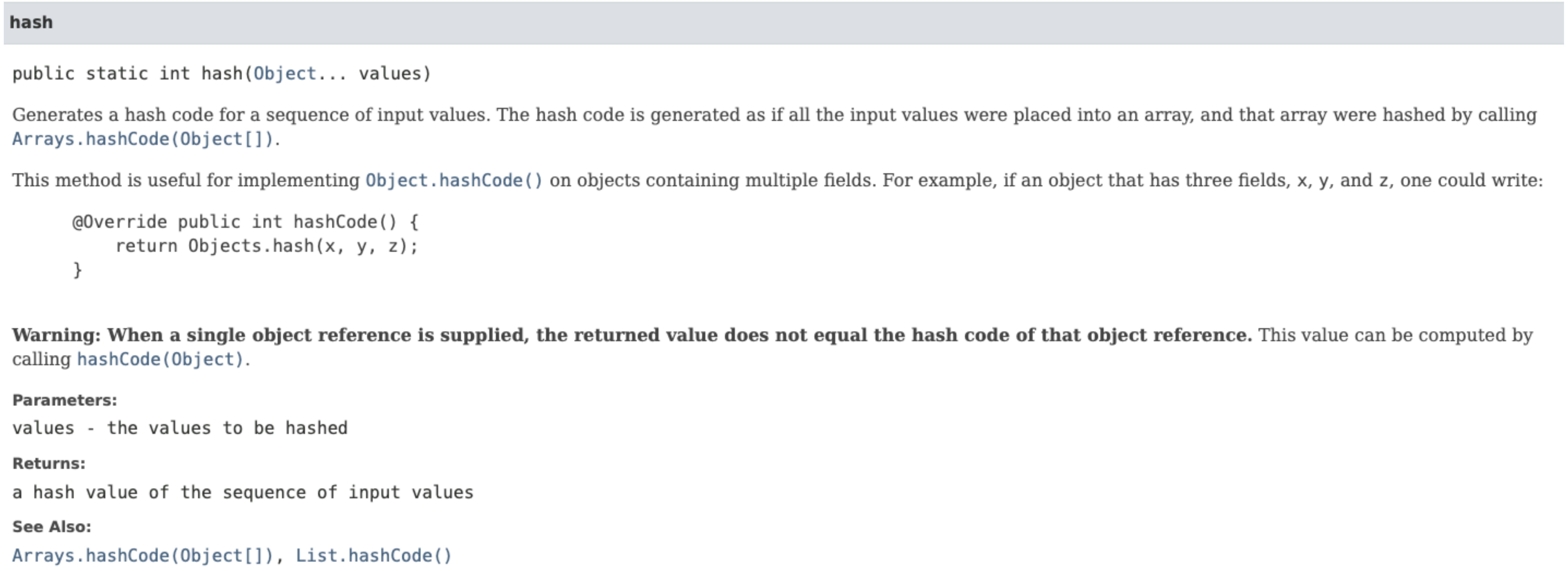
KnapsackState.java

```
public class KnapsackState extends State {

    int item, capacity;

    public KnapsackState(int index, int capacity) {
        this.item = index;
        this.capacity = capacity;
    }

    @Override
    int hash() {
        return Objects.hash(item, capacity);
    }

    @Override
    boolean isEqual(State s) {
        if (s instanceof KnapsackState) {
            KnapsackState state = (KnapsackState) s;
            return item == state.item && capacity == state.capacity;
        }
        return false;
    }

}
```

Useful Method

- Objects.hash

hash (2, 5)



hash

```
public static int hash(Object... values)
```

Generates a hash code for a sequence of input values. The hash code is generated as if all the input values were placed into an array, and that array were hashed by calling `Arrays.hashCode(Object[])`.

This method is useful for implementing `Object.hashCode()` on objects containing multiple fields. For example, if an object that has three fields, x, y, and z, one could write:

```
@Override public int hashCode() {
    return Objects.hash(x, y, z);
}
```

Warning: When a single object reference is supplied, the returned value does not equal the hash code of that object reference. This value can be computed by calling `hashCode(Object)`.

Parameters:

`values - the values to be hashed`

Returns:

`a hash value of the sequence of input values`

See Also:

`Arrays.hashCode(Object[])`, `List.hashCode()`

Transition.java (nothing to implement here)

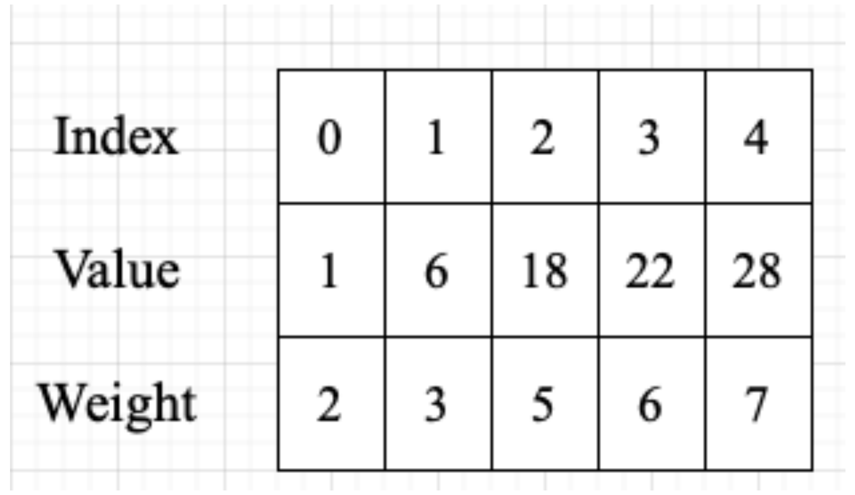
```
class Transition<S extends State> {

    private S successor;
    private int decision;
    private double value;

    public Transition(S successor, int decision, double value) {
        this.successor = successor;
        this.decision = decision;
        this.value = value;
    }

    public S getSuccessor() {
        return successor;
    }

    public int getDecision() {
        return decision;
    }

    public double getValue() {
        return value;
    }
}
```

Model

```
abstract class Model<S extends State> {

    abstract boolean isBaseCase(S state);

    abstract double getBaseCaseValue(S state);

    abstract S getRootState();

    abstract List<Transition<S>> getTransitions(S state);

    abstract boolean isMaximization();
}
```

Knapsack(Model).java

```java
public class Knapsack extends Model<KnapsackState> {
    KnapsackInstance instance;
    KnapsackState root;
    @Override
    boolean isBaseCase(KnapsackState state) { return state.item == instance.n || state.capacity == 0; }
    @Override
    double getBaseCaseValue(KnapsackState state) { return 0; }
    @Override
    List<Transition<KnapsackState>> getTransitions(KnapsackState state) {
        List<Transition<KnapsackState>> transitions = new LinkedList<>();
        // do not take the item
        transitions.add(new Transition<KnapsackState>(
            new KnapsackState(state.item + 1, state.capacity), 0,0));
        // take the item if remaining capacity allows
        if (instance.weight[state.item] <= state.capacity) {
            transitions.add(new Transition<KnapsackState>(
              new KnapsackState(state.item + 1, state.capacity - instance.weight[state.item]),1, instance.value[state.item]));
        }
        return transitions;
    }
    @Override
    KnapsackState getRootState() { return root; }
    @Override
    boolean isMaximization() { return true; }

}
```



Index	0	1	2	3	4
Value	1	6	18	22	28
Weight	2	3	5	6	7

DynamicProgramming.java

```java
public class DynamicProgramming<S extends State> {

    Model<S> model;                 // the dynamic programming model to solve
    HashMap<State, Double> table; // table to store the best value found for
each state

    public DynamicProgramming(Model<S> model) {
        this.model = model;
        this.table = new HashMap<>();
    }
    public Solution getSolution() {
        // TODO compute the solution for the root state of the model
    }
•}
```

Solution.java

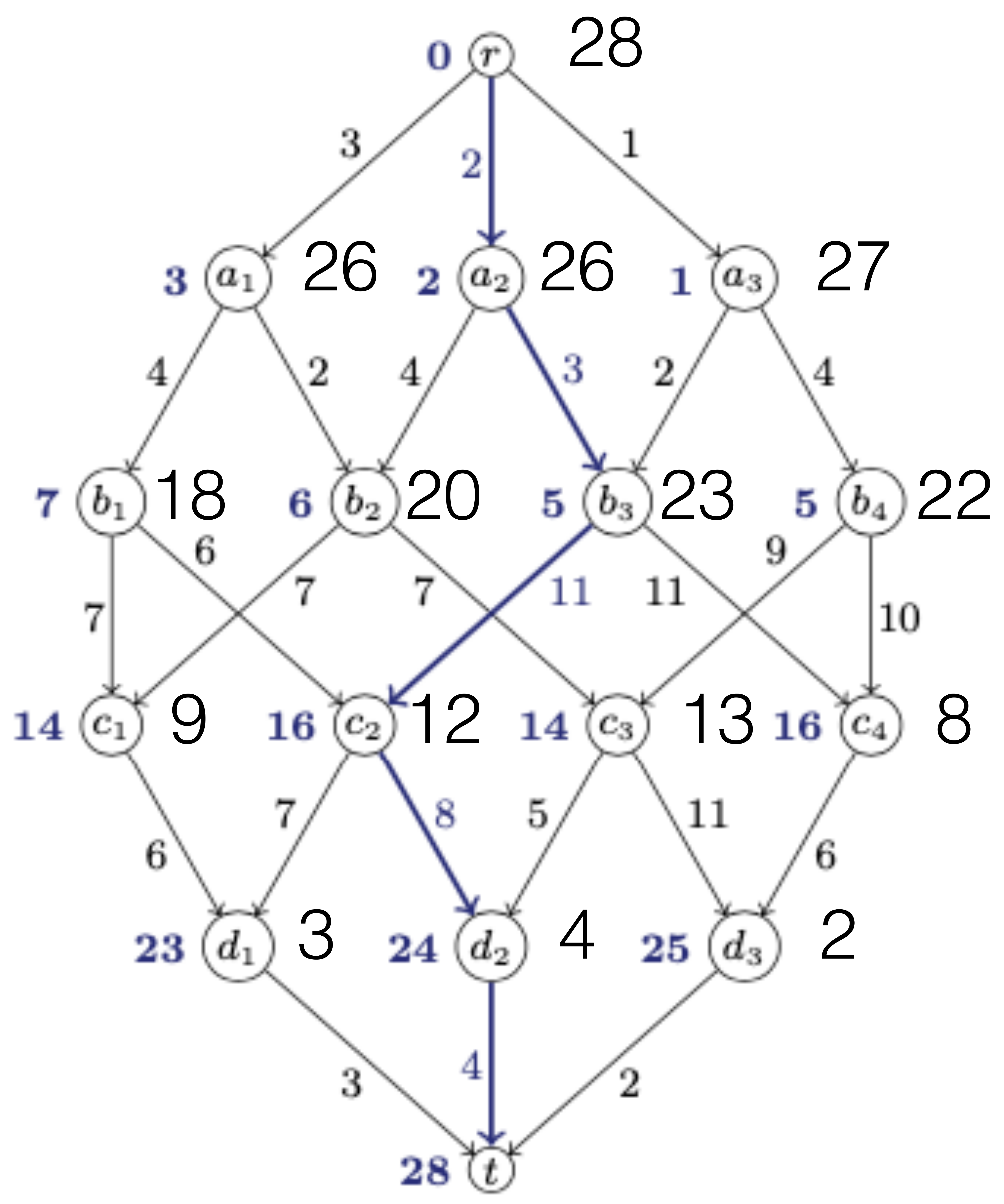
```java
public class Solution {

    private double value;
    private List<Integer> decisions;

    public Solution(double value, List<Integer> decisions) {
        this.value = value;
        this.decisions = decisions;
    }

    public double getValue() {
        return value;
    }

    public List<Integer> getDecisions() {
        return decisions;
    }

    public boolean isValid() {
        return true;
    }

}
```

How to recompute the solution from the table

```
HashMap<State, Double> table;
```

- Start from the root
- Generate the successor of the root and follow the one that has the same objective value minus the cost of the transition.
- Continue like this until you reach a terminal state.
- By following this path, you record the decisions into the solution.
- You will necessarily retrieve the optimal solution this way

Retrieving the longest path



Dynamic Programming

LINFO2266

Pierre Schaus