OPTIMIZATION TECHNIQUES

Integer and dynamic programming

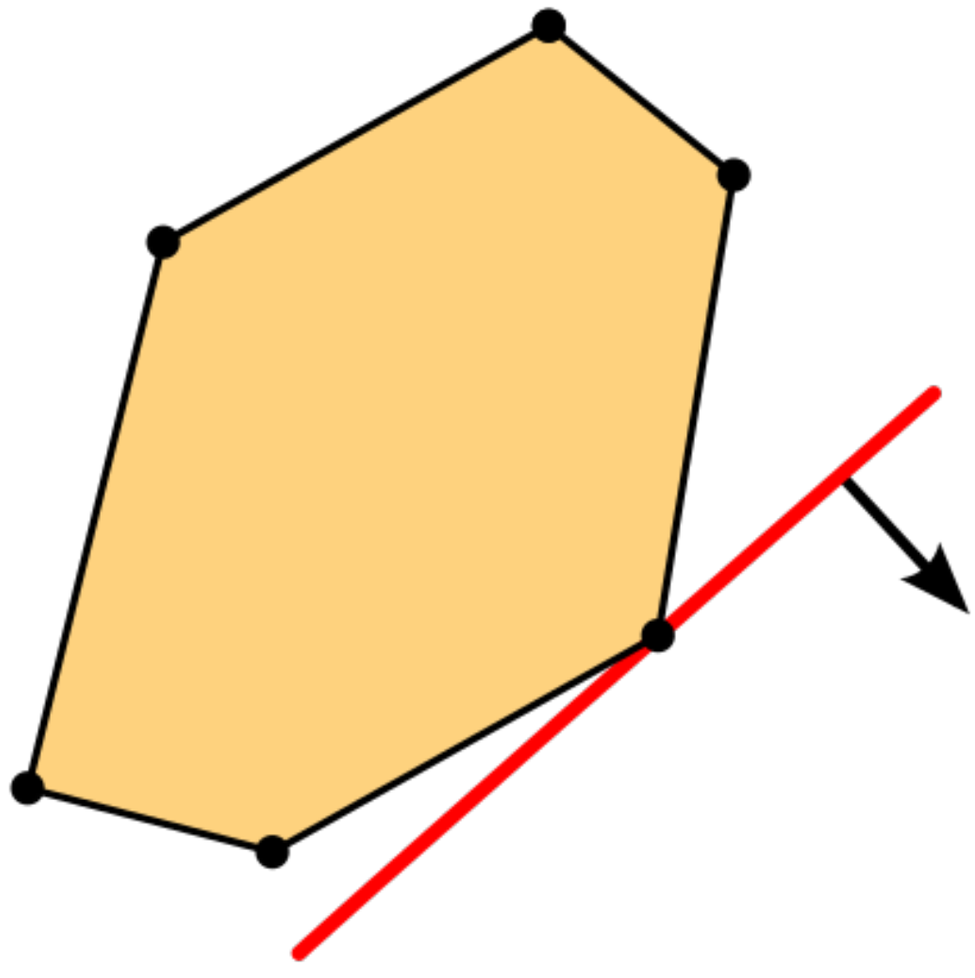
Prof. Giovanni Iacca

giovanni.iacca@unitn.it

UNIVERSITY OF TRENTO - Italy

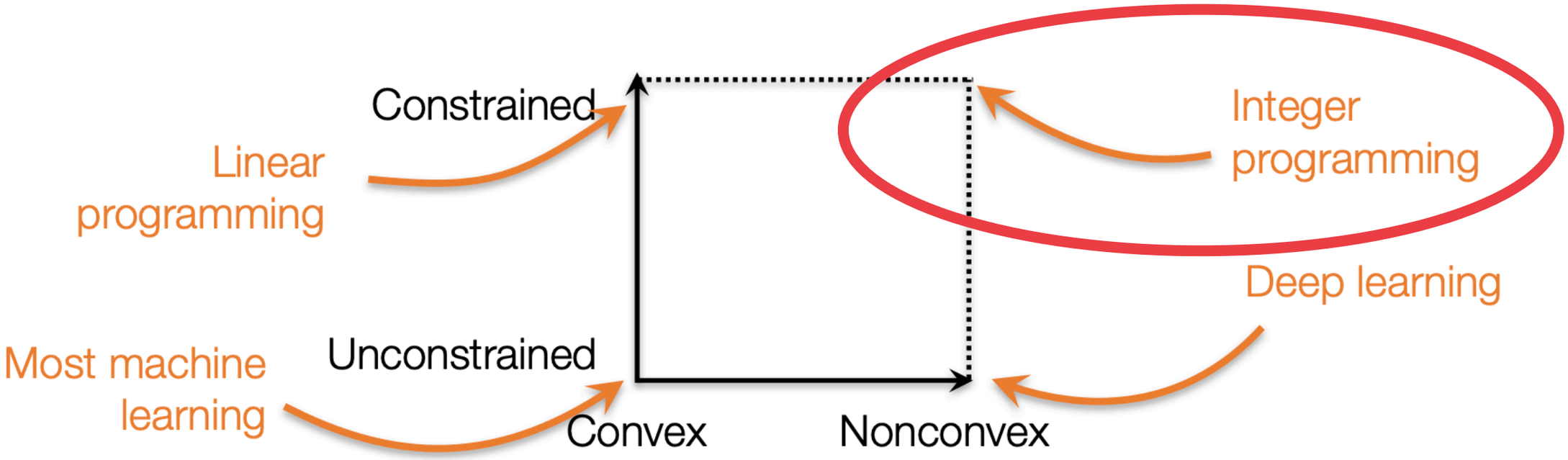
Information Engineering
and Computer Science Department

Integer programming



OPTIMIZATION

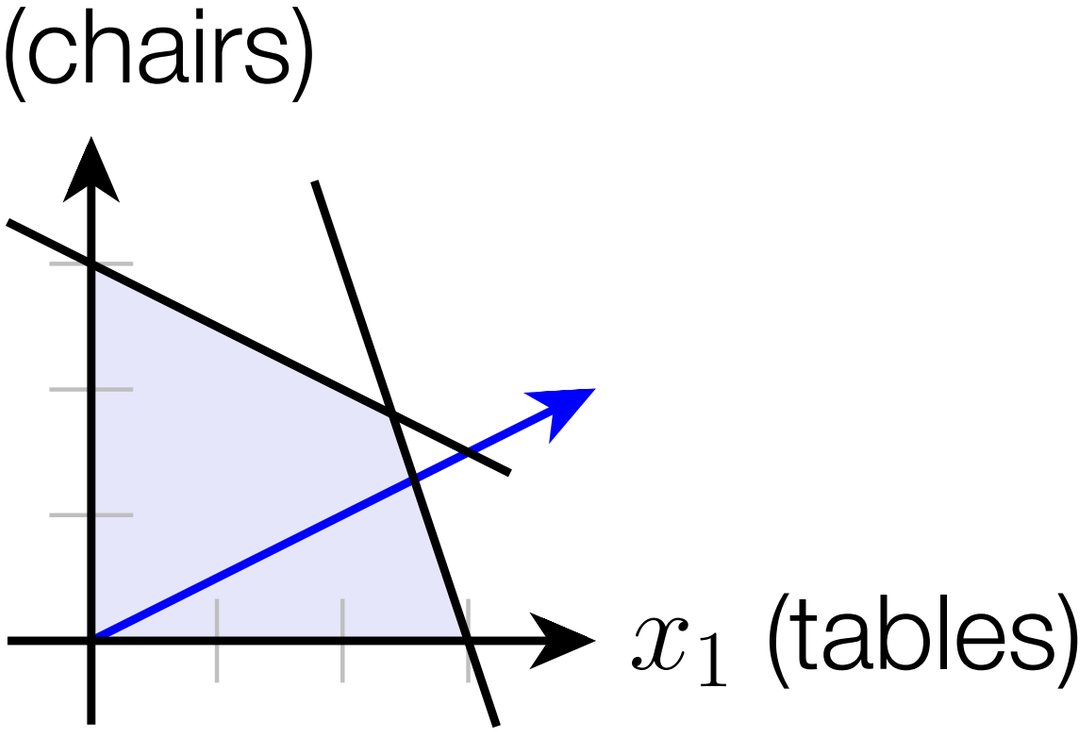
ONE WORD FOR SEVERAL MEANINGS

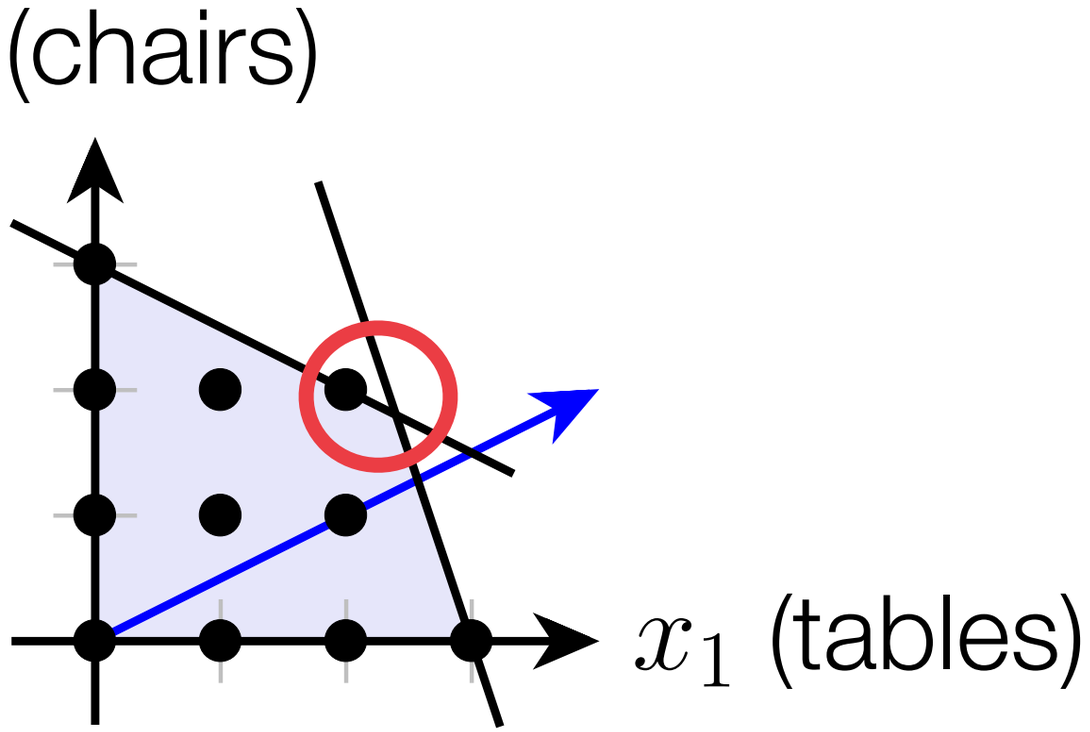


INTEGER (LINEAR) PROGRAMMING

EXAMPLE: MANUFACTURING

Let's consider the example seen in the previous lecture, buy adding a constraint of integerness: x_1 and x_2 indicate batches of production that cannot be fractioned, i.e., x_1 and x_2 are integer values.



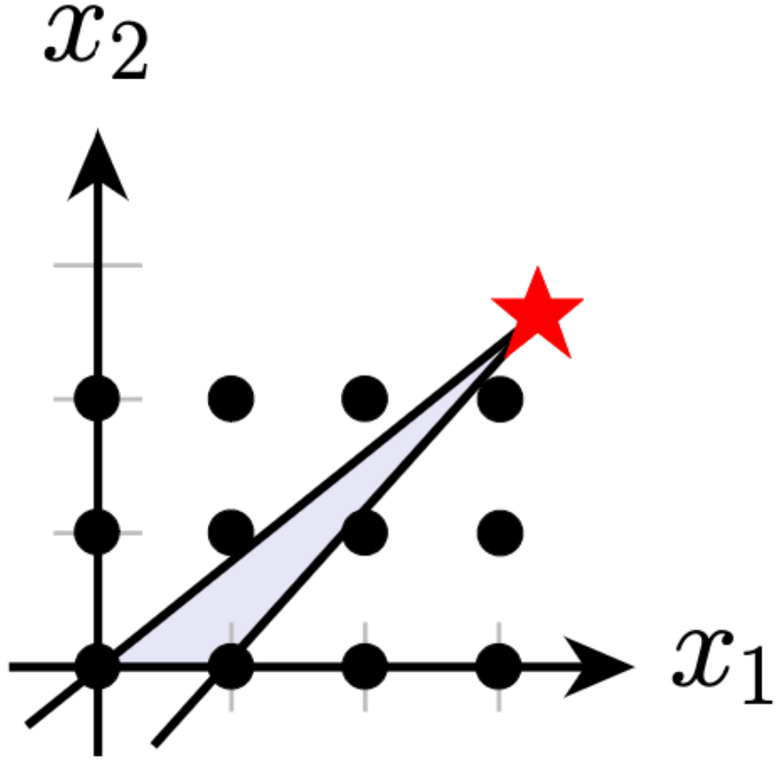


INTEGER (LINEAR) PROGRAMMING

CHALLENGES OF INTEGER PROGRAMMING

The above example was "easy" in that the rounded solution to the LP happened to also be a solution to the integer program.

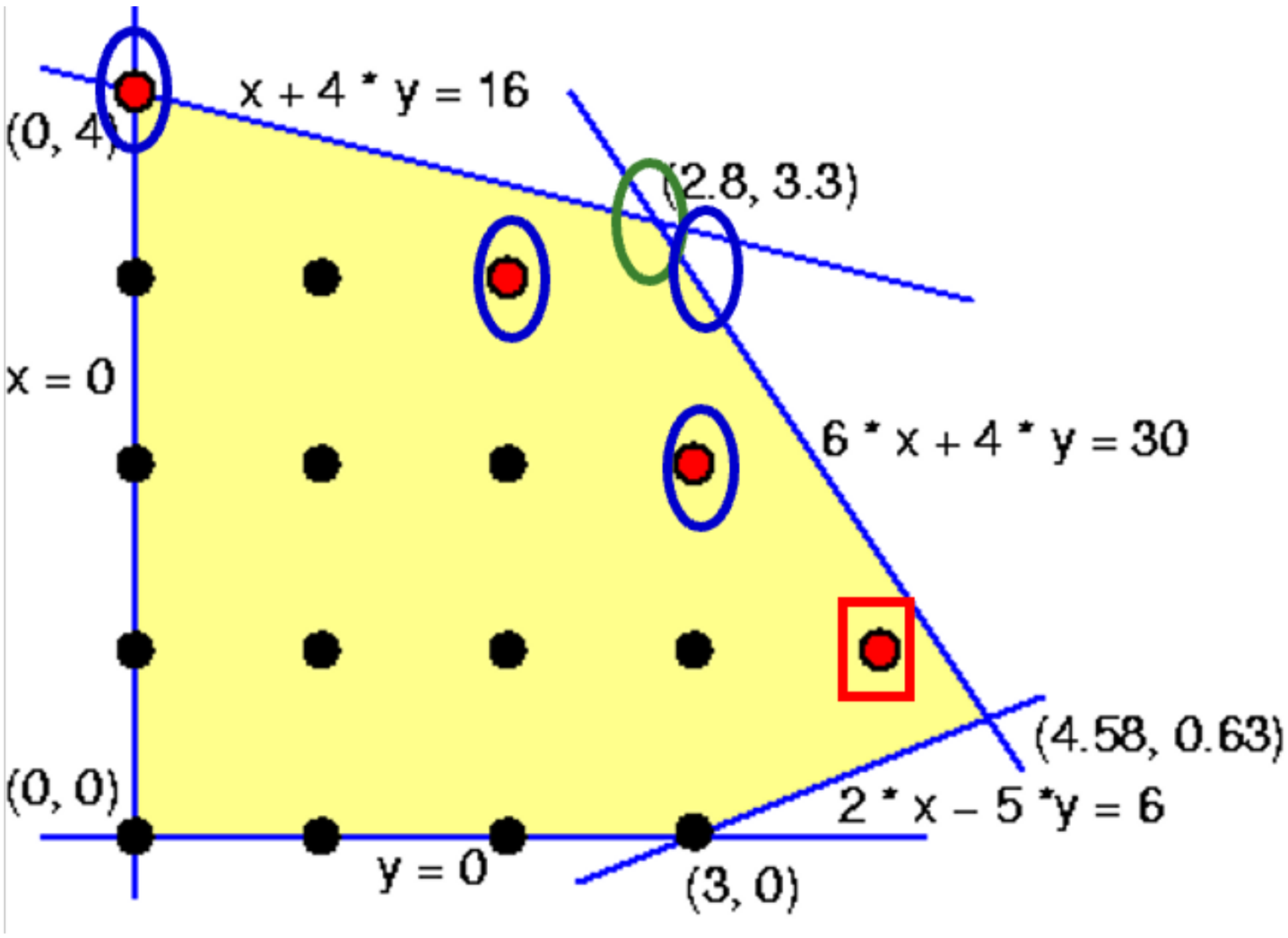
In general, integer solution can be arbitrarily far from the LP solution:



Can be hard to even find a feasible solution that is integer valued, e.g., imagine the task of finding an integer solution to some arbitrary set of linear equations *Ax= b.*

INTEGER (LINEAR) PROGRAMMING

CHALLENGES OF INTEGER PROGRAMMING



Round to nearest int (3,3)?

No, infeasible.

Round to nearest feasible int (2,3) or (3,2)?

No, suboptimal.

Round to nearest integer vertex (0,4)?

No, suboptimal.

INTEGER (LINEAR) PROGRAMMING

INEQUALITY FORM

An optimization problem like linear programming, except that variables are required to take on integer values:

$$\begin{aligned} \underset{x}{\text{minimize}} \;\; & c^T x \\ \text{subject to} \;\; & Gx \leq h \\ & x \in \mathbb{Z}^n \;\; (\text{integers}) \end{aligned}$$

Not a convex problem, because of integer constraint (set of all integers is not a convex set).

We can also consider **Mixed Integer (Linear) Programming** (MILP) problems, which contain both integer and non-integer variables:

$$\begin{aligned} \underset{x}{\text{minimize}} \;\; & c^T x \\ \text{subject to} \;\; & Gx \leq h \\ & x_i \in \mathbb{Z}, \qquad i \in \mathcal{I} \subseteq \{1, \ldots, n\} \end{aligned}$$

INTEGER (LINEAR) PROGRAMMING

INEQUALITY FORM

For simplicity, we will focus on binary integer programming, where the variables of x are all in $\{0,1\}$:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Gx \leq h \\ & x \in \{0,1\}^n \end{aligned}$$

This is just for ease of presentation, these methods can be adapted for general integer variables.

The techniques we present are actually largely applicable to any mixed integer programming problem with convex objective and constraints (other than integer constraint).

INTEGER (LINEAR) PROGRAMMING

SOLVING ILP PROBLEMS

Naïve solution (exhaustive search): given 2^n possible assignments of all the n variables in x, just try each one, return the solution with minimum objective value out of those that satisfy constraints.

In the worst case, we can't do any better than this, but often it is possible to solve the problem *much faster* in practice.

Key idea: relaxing integer constraints

I.e., consider an alternative LP problem where we relax the constraint $x \in \{0,1\}^n$ to be $x^* \in [0,1]^n$:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Gx \leq h \\ & x \in [0,1]^n \end{aligned}$$

Key point #1: if the solution to this LP x^* has all integer values, then it is also the solution to the integer program.
Key point #2: the optimal objective for the linear program is lower than that of the corresponding integer program.

Both points follow trivially from the fact that $\{0,1\}^n \subset [0,1]^n$.

INTEGER (LINEAR) PROGRAMMING

INTEGER SOLUTIONS

Integer solutions are more common than you may expect, as they occur whenever the vertices of the polytope all have integer values.

E.g., consider trivial optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & x \in [0,1]^n \end{aligned}$$

The optimal solution is:

$$x_i^\star = \begin{cases} 1 & \text{if } c_i \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

INTEGER (LINEAR) PROGRAMMING

BRANCH AND BOUND

The **branch and bound algorithm** is a greedy informed search applied using LP relaxation to provide bounds on the search tree. **LP relaxation** is a quickly-computable approximation which gives us a lower bound on the true solution.

Function: Solve-Relaxation($\mathcal{C}$):
- Solve linear program plus additional constraints in $\mathcal{C}$
- Return (objective value $f^\star$, solution $x^\star$, and constraint set $\mathcal{C}$)

Algorithm: Branch-and-Bound
- Push Solve-Relaxation($\{\}$) on to frontier set
- Repeat while frontier is not empty:
    1. Get lowest cost solution from frontier: $(f, x, \mathcal{C})$
    2. If x is integer valued, return x
    3. Else, choose some x_i not integer valued and add Solve-Relaxation($\mathcal{C} \cup \{x_i = 0\}$),Solve-Relaxation($\mathcal{C} \cup \{x_i = 1\}$), to the frontier

INTEGER (LINEAR) PROGRAMMING

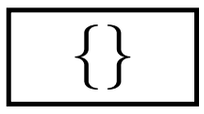
BRANCH AND BOUND - EXAMPLE

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 2x_1 + x_2 - 2x_3 \\ \text{subject to} \quad & 0.7x_1 + 0.5x_2 + x_3 \geq 1.8 \\ & x_i \in \{0,1\}, \qquad i = 1,2,3 \end{aligned}$$

LP relaxation

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 2x_1 + x_2 - 2x_3 \\ \text{subject to} \quad & 0.7x_1 + 0.5x_2 + x_3 \geq 1.8 \\ & x_i \in [0,1], \qquad i = 1,2,3 \end{aligned}$$

Search tree



Frontier

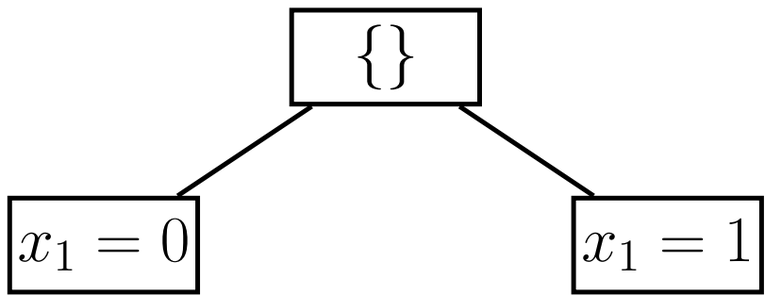
$$(f^\star = -0.143, x^\star = [0.43,1,1], \mathcal{C} = \{\})$$

INTEGER (LINEAR) PROGRAMMING

BRANCH AND BOUND - EXAMPLE

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 2x_1 + x_2 - 2x_3 \\ \text{subject to} \quad & 0.7x_1 + 0.5x_2 + x_3 \geq 1.8 \\ & x_i \in [0,1], \qquad i = 1,2,3 \end{aligned}$$

Search tree



Frontier

~~$(f^\star = -0.143, x^\star = [0.43,1,1], \mathcal{C} = \{\})$~~

$(f^\star = 0.2, x^\star = [1, 0.2, 1], \mathcal{C} = \{x_1 = 1\})$

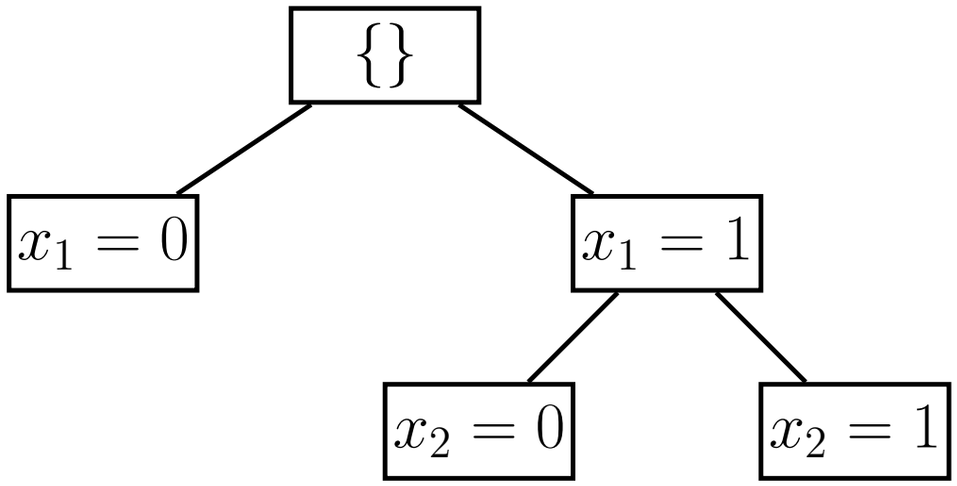
$(f^\star = \infty, x^\star = \emptyset, \mathcal{C} = \{x_1 = 0\})$

INTEGER (LINEAR) PROGRAMMING

BRANCH AND BOUND - EXAMPLE

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 2x_1 + x_2 - 2x_3 \\ \text{subject to} \quad & 0.7x_1 + 0.5x_2 + x_3 \geq 1.8 \\ & x_i \in [0,1], \qquad i = 1,2,3 \end{aligned}$$

Search tree



Frontier

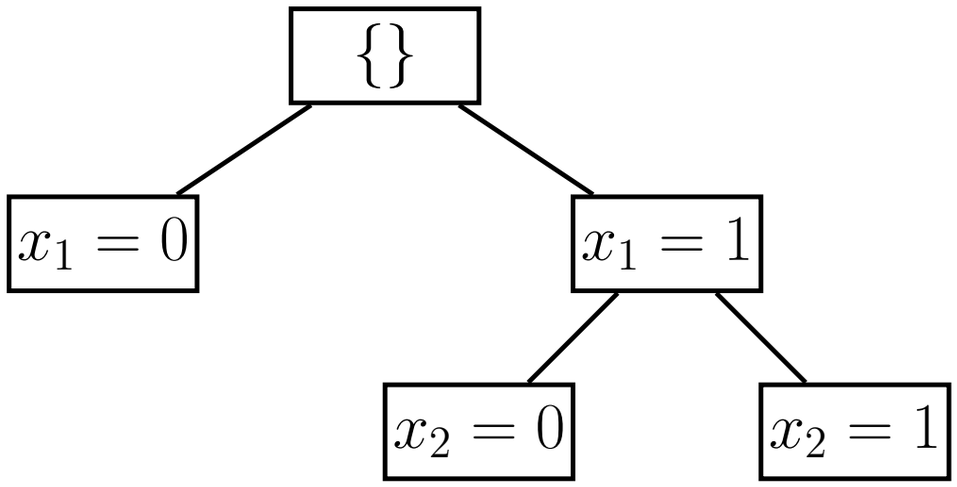
~~$(f^\star = -0.143, x^\star = [0.43,1,1], \mathcal{C} = \{\})$~~
~~$(f^\star = 0.2, x^\star = [1, 0.2, 1], \mathcal{C} = \{x_1 = 1\})$~~
$(f^\star = 1, x^\star = [1,1,1], \mathcal{C} = \{x_1 = 1, x_2 = 1\})$
$(f^\star = \infty, x^\star = \emptyset, \mathcal{C} = \{x_1 = 0\})$
$(f^\star = \infty, x^\star = \emptyset, \mathcal{C} = \{x_1 = 1, x_2 = 0\})$

INTEGER (LINEAR) PROGRAMMING

BRANCH AND BOUND - EXAMPLE

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 2x_1 + x_2 - 2x_3 \\ \text{subject to} \quad & 0.7x_1 + 0.5x_2 + x_3 \geq 1.8 \\ & x_i \in [0,1], \qquad i = 1,2,3 \end{aligned}$$

Search tree



Frontier

~~$(f^\star = -0.143, x^\star = [0.43,1,1], \mathcal{C} = \{\})$~~
~~$(f^\star = 0.2, x^\star = [1, 0.2, 1], \mathcal{C} = \{x_1 = 1\})$~~
$(f^\star = 1, x^\star = [1,1,1], \mathcal{C} = \{x_1 = 1, x_2 = 1\})$
$(f^\star = \infty, x^\star = \emptyset, \mathcal{C} = \{x_1 = 0\})$
$(f^\star = \infty, x^\star = \emptyset, \mathcal{C} = \{x_1 = 1, x_2 = 0\})$

INTEGER (LINEAR) PROGRAMMING

EXTENSIONS

- Often useful to also maintain an upper (feasible) bound, when possible.

- For problems with general integer constraint (not just binary constraints), the algorithm is virtually identical*, except that in this case we find a non-integer $\tilde{x}_j$ and add to the frontier (i.e., the cut in the search tree separating the completed nodes from the not-yet-explored nodes):

$$\text{Solve-Relaxation}(\mathcal{C} \cup \{x_i \geq \lceil \tilde{x}_j \rceil\}), \text{Solve-Relaxation}(\mathcal{C} \cup \{x_i \leq \lfloor \tilde{x}_j \rfloor\}$$

- Mixed integer problems (some but not all variables are non-integer): branch and bound can be applied by simply not branching on non-integer variables, and by resolving over non-integer variables after rounding integer variables.

- Unusual to use "pure" branch and bound to solve real-world problems: real solvers additionally use a concept called **cutting planes** to further restrict the allowable set of non-integer solutions ("Branch and cut").

* See: http://web.tecnico.ulisboa.pt/mcasquilho/compute/_linpro/TaylorB_module_c.pdf

INTEGER (LINEAR) PROGRAMMING

MANY PROBLEMS AND APPLICATIONS CAN BE TACKLED BY ILP

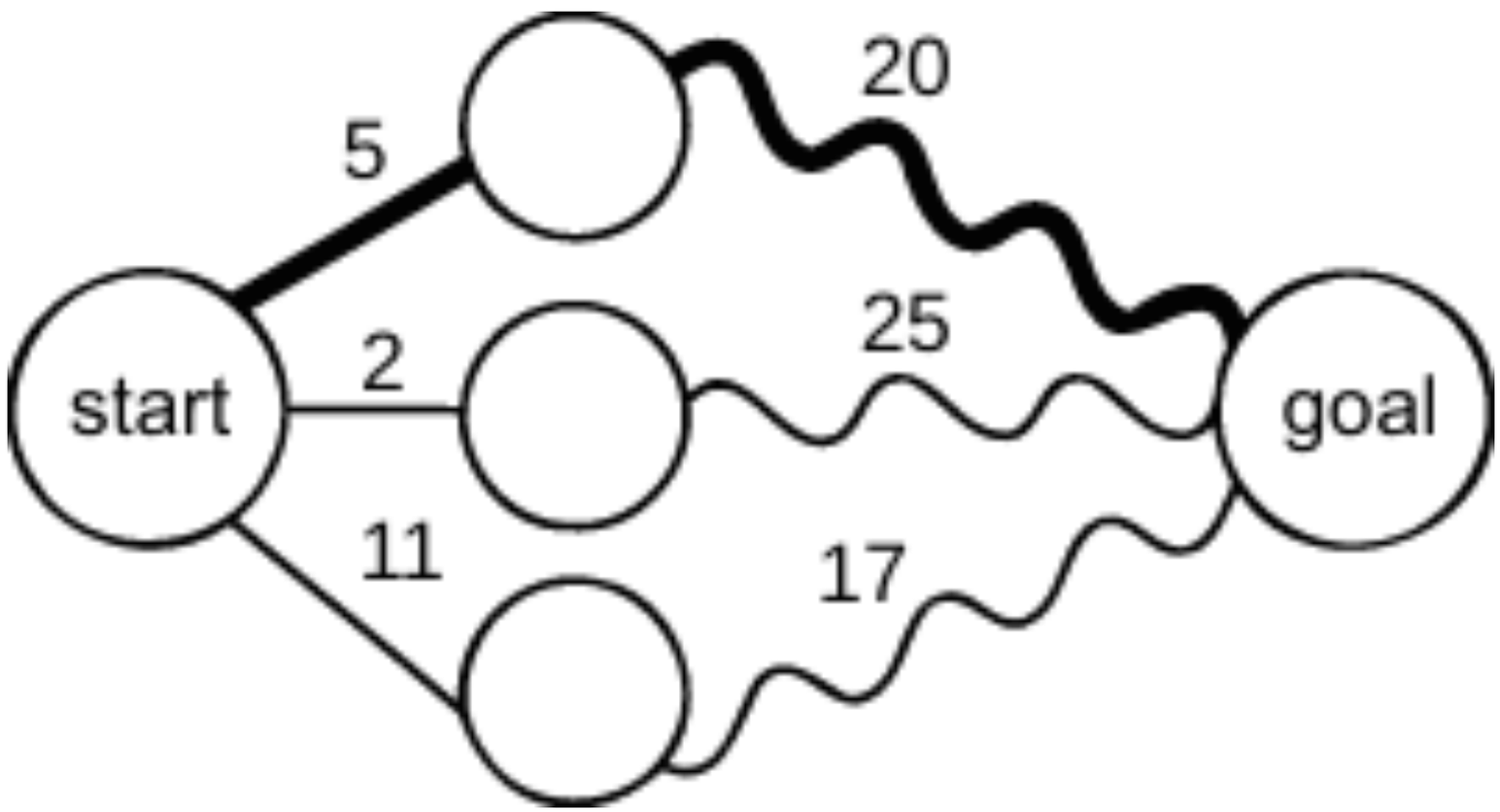
- Path planning with obstacles
- Many problems in game theory
- Constraint satisfaction problems
- (Exact) most likely assignment in graphical models
- Scheduling and unit commitment

NOTES

Extremely well-developed set of commercial solvers are available (free for academic use), e.g., CPLEX. Gurobi, or LocalSolver. For better efficiency these tools often use proprietary "pre-solve" problem simplification methods, relaxations based on simplex and other LP solvers, branch and bound, and cutting plane generation methods.

Open source notably lags behind in this area, with few exceptions such as SCIP (http://scip.zib.de/).

Dynamic programming



DYNAMIC PROGRAMMING

A BIT OF HISTORY

- Invented by American mathematician Richard Bellman (*Bellman equation, anyone?*) in the 1950s to solve optimization problems in control theory, and later assimilated by Computer Science.

- At the time, Bellman was employed at Rand Corporation, that had many, large military contracts… The problem was that the Secretary of Defense, Charles Wilson, was firmly "against research, especially mathematical research".

- …*"Thus, I thought* ***dynamic programming*** *was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."* (R. E. Bellman, Eye of the Hurricane, An Autobiography)

- As seen for LP/IP, in this case "programming" means "planning" (and, in particular, planning dynamically, i.e., over time, using tables/arrays to construct a solution), NOT programming code!

DYNAMIC PROGRAMMING

MAIN GIST

- More than an optimization algorithm: it's *a way of solving problems*, and and *algorithm design technique*. Many practical algorithms actually belong to the DP paradigm.

- A technique for solving problems that have:
  - Optimal sub-structure: i.e., an optimal solution to the problem/problem instance contains optimal solutions to its sub-problems. This property entails **recursion** and **self-reducibility**: the optimal solution to the problem can be obtained by combining the optimal solutions to all sub-problems (this property is also called **principle of optimality**).
  - Overlapping sub-problems: i.e., a recursive solution contains a "small" number of sub-problems, usually repeated many times.

- Main steps of DP:
  - Define sub-problems
  - Write down the recurrence that relates sub-problems, i.e., the order in which they are solved
  - Recognize and solve the base case(s)
  - Compute the answer to each sub-problem using the previously computed sub-problems. NOTE: this step is typically **polynomial**, once the other sub-problems are solved.

DYNAMIC PROGRAMMING

MAIN GIST

- In practice, DP does not repeatedly solve the same sub-problems, but solves them *only once* and stores the intermediate results in a *table*. Two approaches are possible:
  - **Memoization** (top-down approach): after computing a solution to a sub-problem, stores it in a table. Subsequent calls simply look up that solution in the table, to avoid redoing work.
  - **Tabulation** (bottom-up approach): we start from the bottom (i.e., the case base of recursion) and work our way forward, computing each new value using the previous states. Rather than using recursion, this involves a simple loop. ***This is the most common approach.***

- Whereas memoization fills entries on demand, tabulation systematically works its way up, filling in entries until it reaches the desired solution.

- Advantages of tabulation vs. memoization algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements

- Advantages of memoization algorithms vs. tabulation
  - Some sub-problems do not need to be solved

DYNAMIC PROGRAMMING

MAIN GIST

- Typically, first one has to formulate the recursive solution, and then turn it into recursion plus DP programming via memoization or tabulation. How to turn recursion into a tabular approach?
  1. Figure out what the variables are
  2. Use them to index the rows and columns
  3. Figure out what the base case is
  4. Handle the base case as the first one in the table
  5. Then figure out the inductive step and work up to the final answer

- Basically, DP usually *reduces time by increasing the amount of memory*: we solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table. The table is then used for finding the optimal solution to larger problems. Time is saved since each sub-problem is solved only once (not every sub-problem is new!).

- Intuitively, the running time of a DP algorithm depends on two factors:
  - Number of sub-problems overall
  - How many choices we look at for each sub-problem

  Informally, we can approximate the runtime as **#(overall sub-problems) x #(choices)**.

DYNAMIC PROGRAMMING

DIVIDE-AND-CONQUER VS DP

- A special case of DP-type technique is **divide-and-conquer**, which like DP solves problems by combining solutions to sub-problems:
  - Conquers the sub-problems by solving them recursively.
  - Combines the solutions to sub-problems into the solution for original problem.

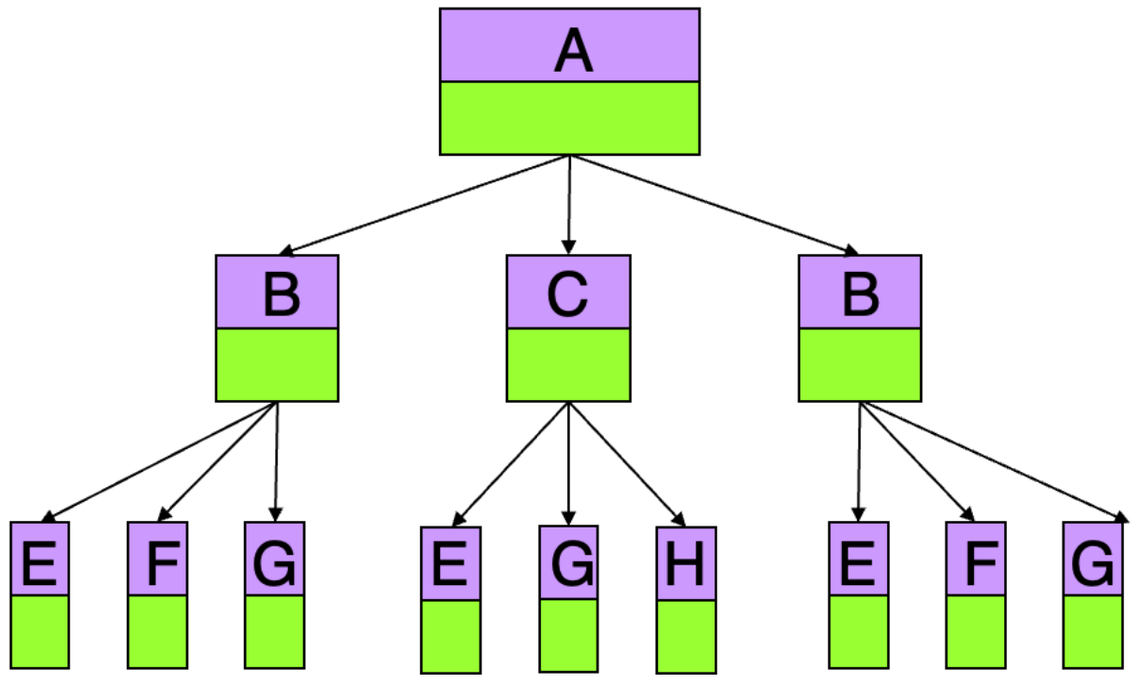
- However, unlike divide-and-conquer, in DP there are dependencies (which can be expressed as a graph) across sub-problems i.e., sub-problems share sub-sub-problems (recursively).

- In a problem with dependencies across sub-problems, a divide-and-conquer approach would *repeatedly solve the common sub-problems*. On the contrary, DP solves every sub-problem *just once* and stores the answer in a table.

- Other than that, the pillars of divide-and-conquer algorithms are similar to DP:
  1. Divide into sub-problems
  2. Conquer through recursion
  3. Recombine sub-results

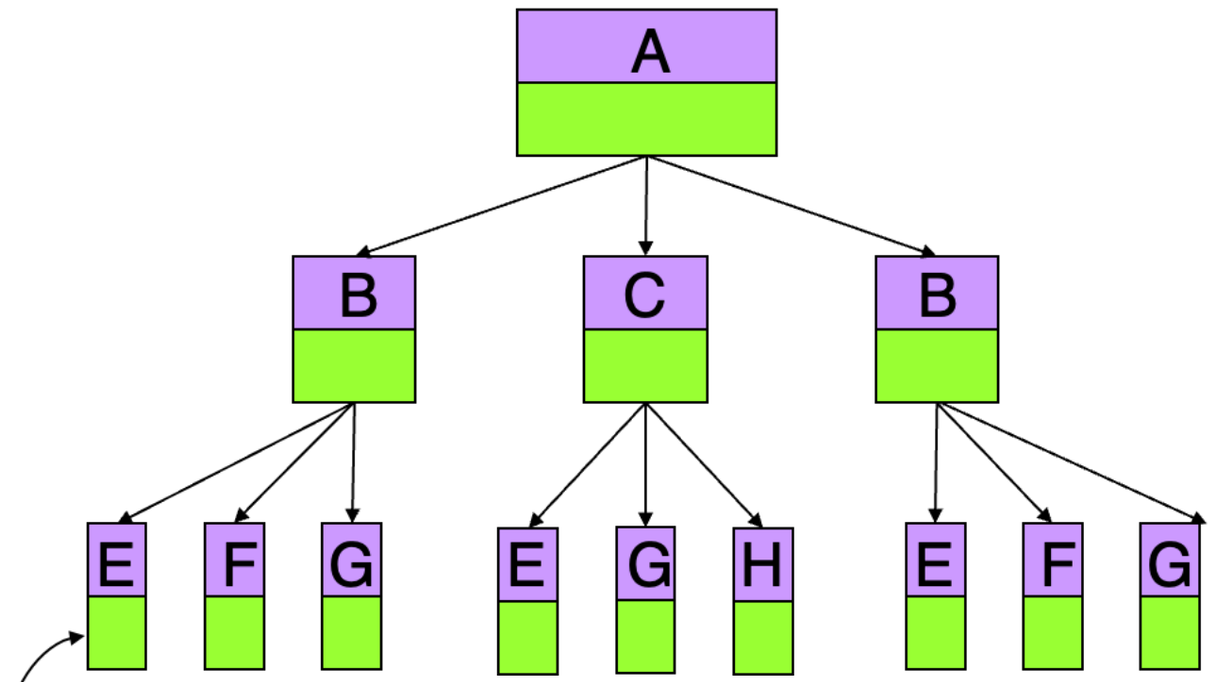
DYNAMIC PROGRAMMING

COMPUTING VIEW



DYNAMIC PROGRAMMING

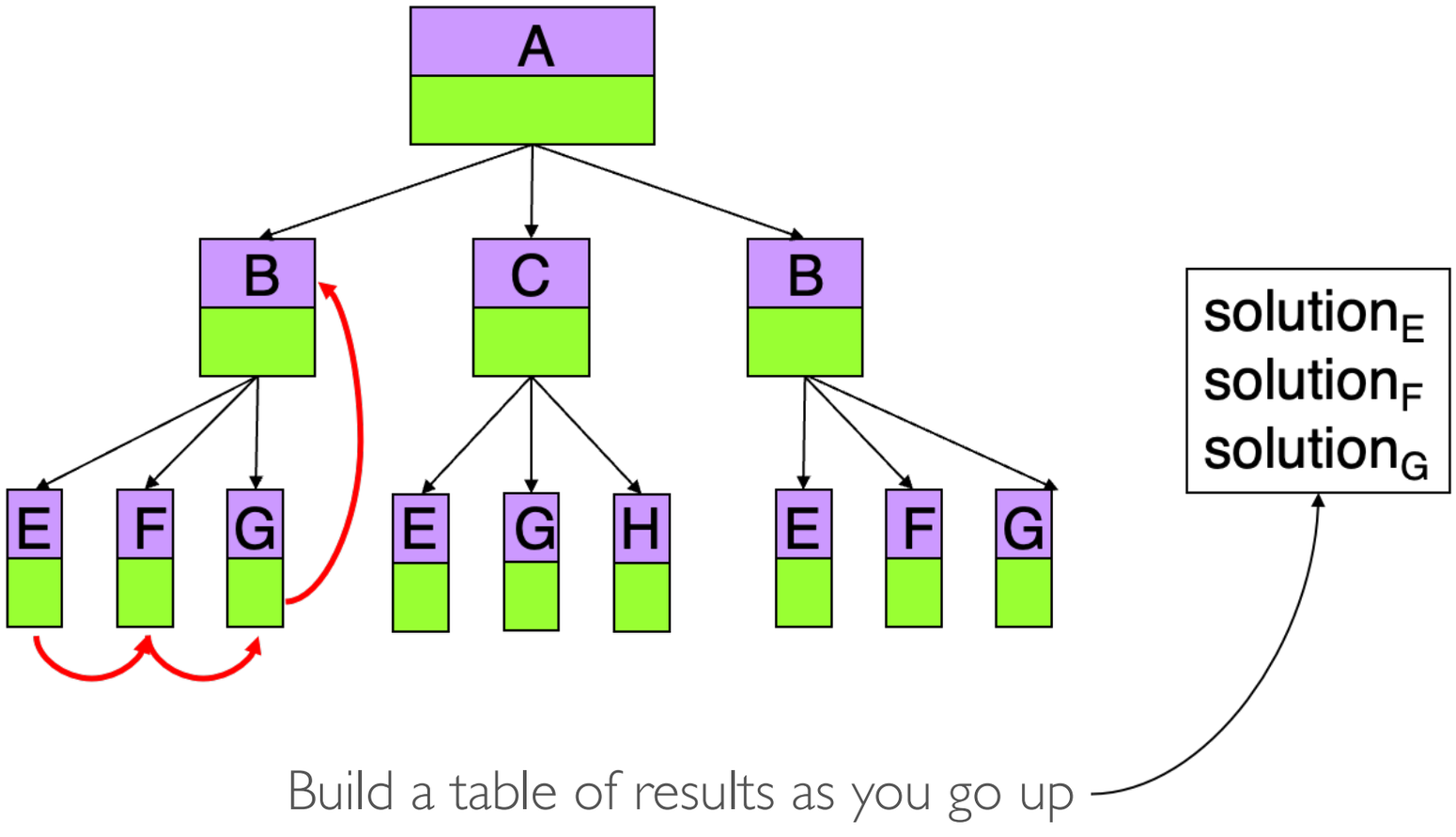
COMPUTING VIEW



Start solving sub-problems at the bottom

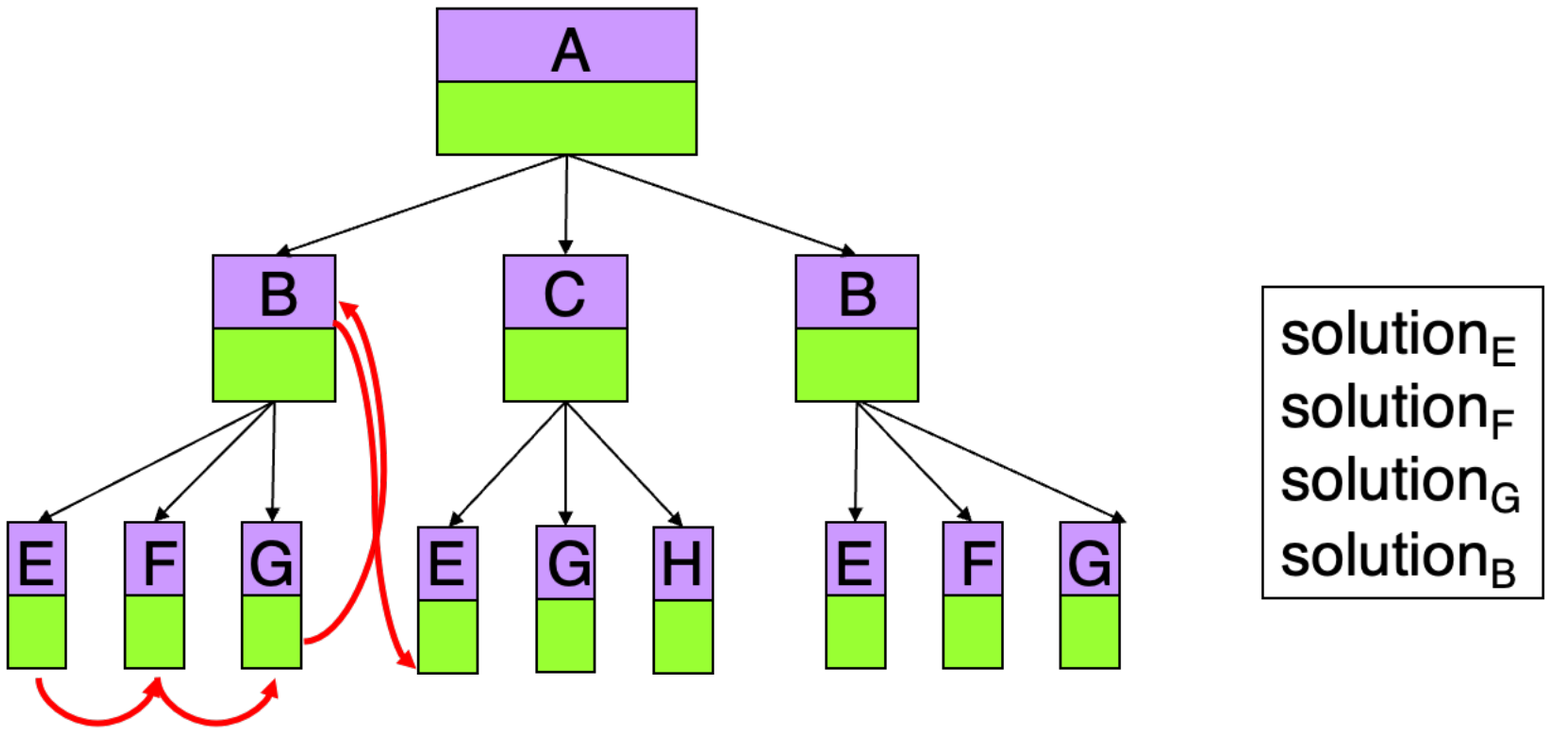
DYNAMIC PROGRAMMING

COMPUTING VIEW



DYNAMIC PROGRAMMING

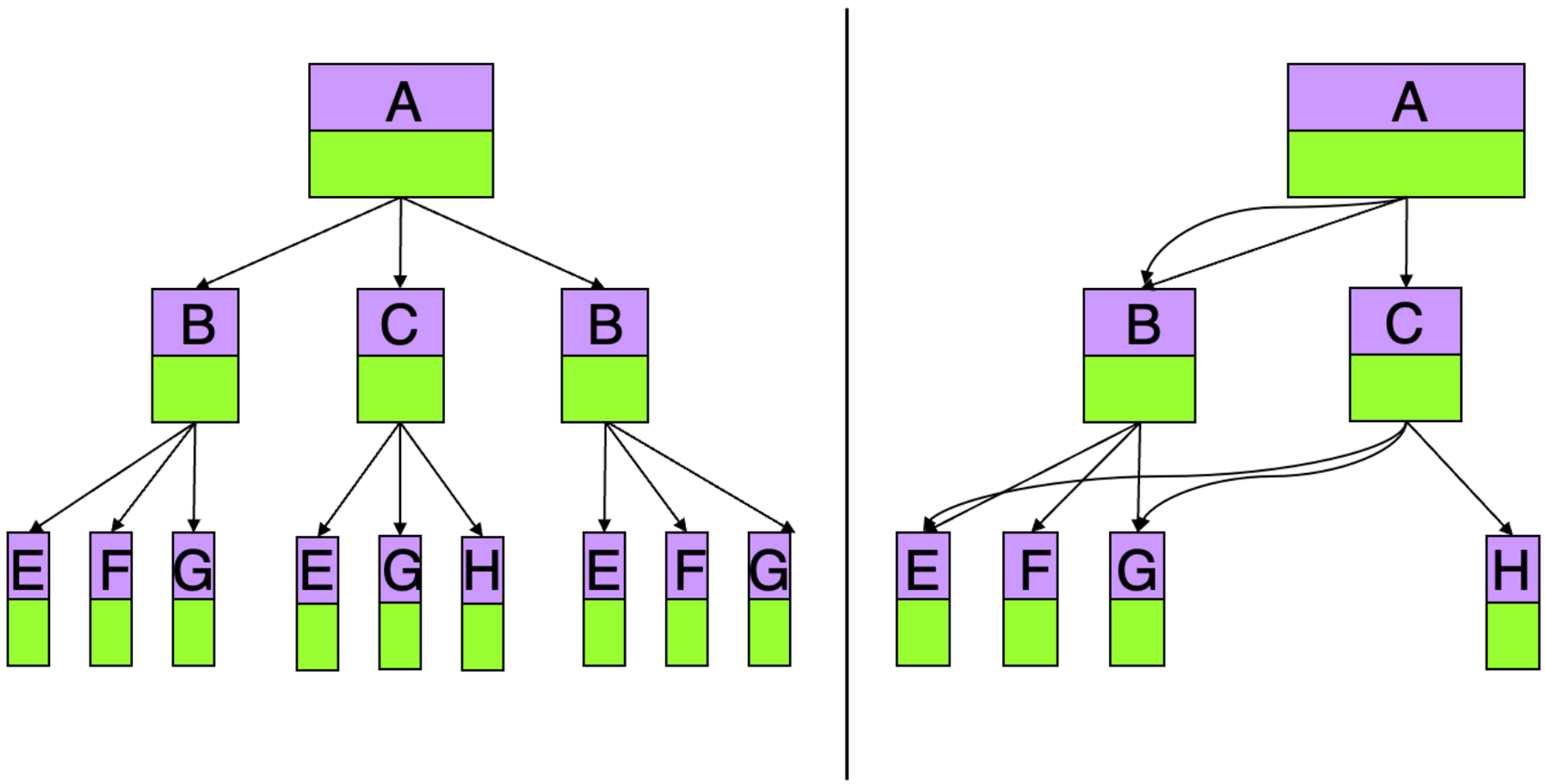
COMPUTING VIEW



Avoid recomputing intermediate results

DYNAMIC PROGRAMMING

COMPUTING VIEW



DYNAMIC PROGRAMMING

GENERAL DP PROBLEM STRUCTURE

- The solution to a DP problem is typically expressed as a minimum (or maximum) of possible alternative solutions.

- If *r* represents the cost of a solution composed of sub-problems $x_1, x_2, \ldots, x_l$, then *r* can be written as:

$$r = g(f(x_1), f(x_2), \ldots, f(x_l)).$$

- Here, *g* is called the *composition function*.

- If the optimal solution to each problem is determined by composing optimal solutions to the sub-problems and selecting the minimum (or maximum) of such composition, the formulation is said to be a DP formulation.

- The recursive DP equation is also called the *functional equation* or *optimization equation*.

- If the RHS has multiple recursive terms, the DP formulation is called polyadic (otherwise monadic).

DYNAMIC PROGRAMMING

EXAMPLE: FIBONACCI

- Let's consider the calculation of Fibonacci numbers:
  fibonacci(n) = fibonacci(n-2) + fibonacci(n-1)
- With seed values fibonacci(0) = 0, fibonacci(1) = 1, the series looks like:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

3

5

8

13

21

34

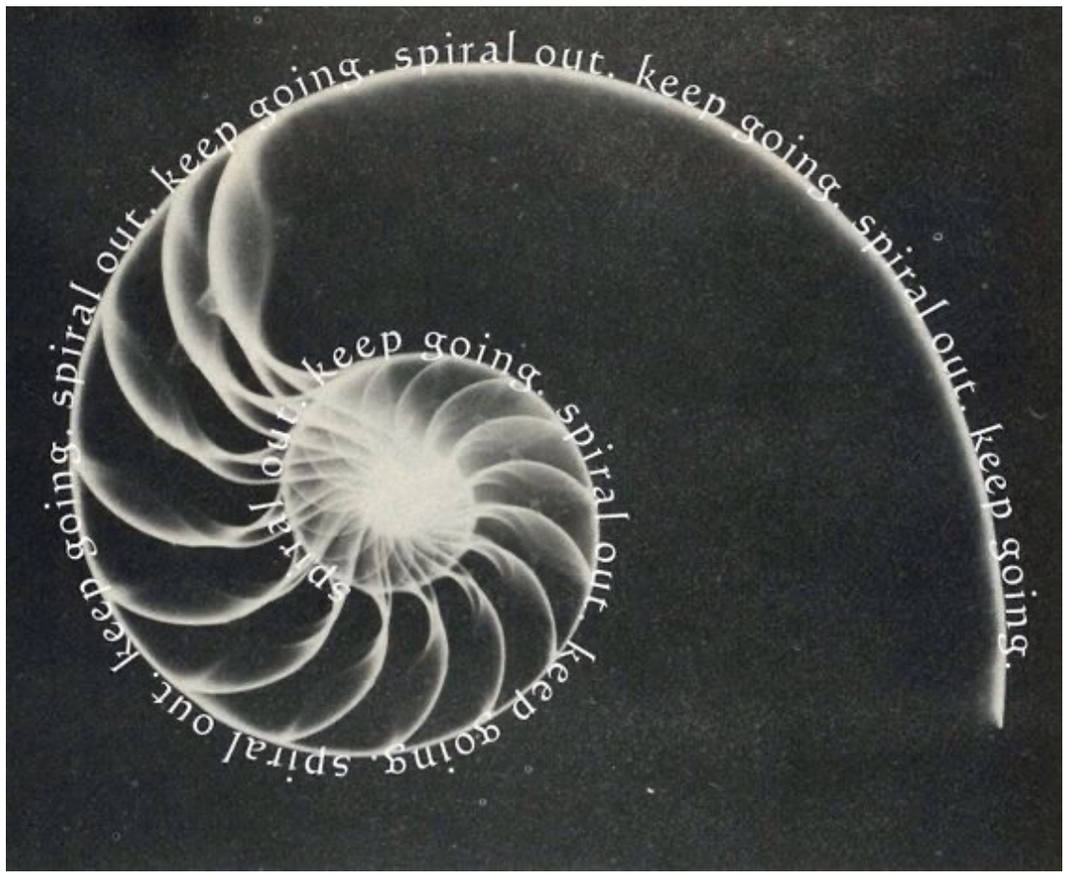
55

89

DYNAMIC PROGRAMMING

EXAMPLE: FIBONACCI

Black (1)
Then (1)
White are (2)
All I see (3)
In my infancy (5)
Red and yellow then came to be (8)
…

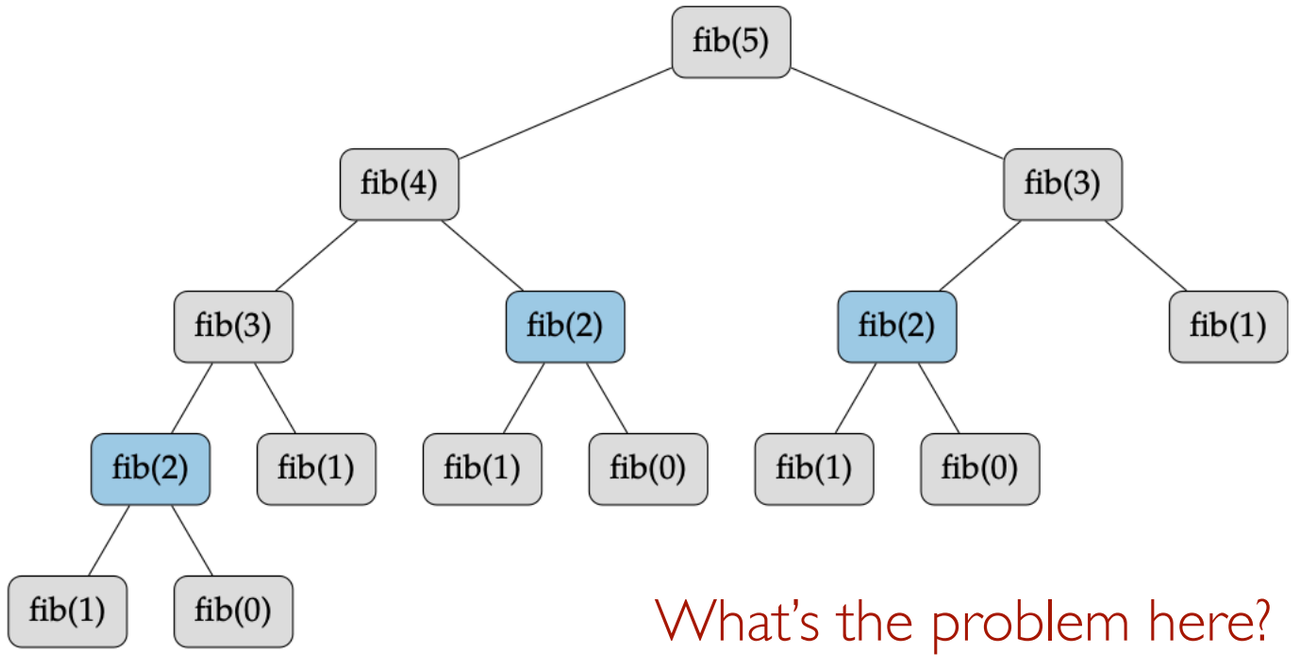


DYNAMIC PROGRAMMING

EXAMPLE: FIBONACCI / RECURSIVE SOLUTION

Recursive implementation

```
int fibonacci(int n)
{
   if (n == 0)
      return 0;
   if (n == 1)
      return 1;
   return fibonacci(n-1)+fibonacci(n-2);
}
```



What's the problem here?

$$F_n \sim \phi^n, \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

- Recursion is sometimes called "backward chaining": start with the goal you want, say fibonacci(7), choosing your sub-goals fibonacci(6), fibonacci(5), … on an as-needed basis.
  - *Reason backwards from goal to facts (start with goal and look for support for it)*
- Another option is "forward chaining": compute each value as soon as you can, fibonacci(0), fibonacci(1), fibonacci(2), fibonacci(3) … in hopes that you'll reach the goal.
  - *Reason forward from facts to goal (start with what you know and look for things you can prove)*
- Mixing forward and backward is possible
- In all cases, tabling results that you'll need later makes a lot of sense!

DYNAMIC PROGRAMMING

EXAMPLE: FIBONACCI / DP SOLUTION

Backward-chained implementation (top-down)

```
int fibonacci(int n, int[] M)
{
   if (n == 0)
      M[0] = 0;
   if (n == 1)
      M[1] = 1;
   // fibonacci(n-2) has not already been calculated
   if (M[n-2] is Empty)
      M[n-2] = fibonacci(n-2);
   // fibonacci(n-1) has not already been calculated
   if (M[n-1] is Empty)
      M[n-1] = fibonacci(n-1);
   // store the n-th Fibonacci no. in memory
   // use previous results
   M[n] = M[n-1] + M[n-2];

   return M[n];
}
```

Forward-chained implementation (bottom-up)

```
int fibonacci(int n)
{
   int[] f = new int[n+1];
   int f[0] = 0;
   int f[1] = 1;
   for (int i=2; i <=n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

DYNAMIC PROGRAMMING

EXAMPLE: KNAPSACK PROBLEM

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq K \\ & x_i \in \{0, 1\} . \end{aligned}$$

The naïve method is to consider all 2^n possible subsets of the n objects and choose the one that fits into the knapsack and maximizes the profit.

How can we turn it into a DP formulation? I.e., how can we build a recursive solution?

What's the structure of the smaller problems?

- How to use solution to smaller problems into a solution to larger ones?
- What do we have to store in table?
- What are the initial conditions?
- What's the order to solve the sub-problems?

DYNAMIC PROGRAMMING

EXAMPLE: KNAPSACK PROBLEM / DP SOLUTION

Consider a sub-instance defined by the first i items and capacity k ($k \leq K$).

Let $v[k, i]$ be the optimal value for such instance. Then:

$v[k, i] = v[k, i\text{-}1]$ if $w_i > k$

$v[k, i] = \max\{v[k, i\text{-}1], c_i + v[k\text{-}w_i, i\text{-}1]\}$ if $w_i \leq k$

Initial conditions: $v[k, 0] = 0$ and $v[0, i] = 0$

- $v[k, i]$ is the optimal cost for capacity k considering items 1 through i;
- note that indexing starts at 0

DYNAMIC PROGRAMMING

EXAMPLE: KNAPSACK PROBLEM / DP SOLUTION

Inputs:

c_i: Cost of item i

w_i: Weight of item i

K: Total available capacity

Outputs:

x^*: Optimal selections

$v(K, n)$: Corresponding cost

- $v[k, i]$ is the optimal cost for capacity k considering items 1 through i;
- note that indexing starts at 0

```
for k = 0 to K do
    v(k,0) = 0                                   No items considered; value is zero for any capacity
end for
for i = 1 to n do                                Iterate forward solving for one additional item at a time
    for k = 0 to K do
        if w_i > k then
            v(k,i) = v(k,i-1)                    Weight exceeds capacity; value unchanged
        else
            if c_i + v(k - w_i, i-1) > v(k,i-1) then    Take item
                v(k,i) = c_i + v(k - w_i, i-1)
                S(k,i) = 1
            else                                 Reject item
                v(k,i) = v(k,i-1)
            end if
        end if
    end for
end for
k = K                                            Initialize
x* = {}                                          Initialize solution x* as an empty set
for i = n to 1 by -1 do                          Loop to determine which items we selected
    if S_{k,i} = 1 then
        add i to x*                              Item i was selected
        k = k - w_i
    end if
end for
```

DYNAMIC PROGRAMMING

EXAMPLE: KNAPSACK PROBLEM / DP SOLUTION

$$w_i = [4, 5, 2, 6, 1]$$

$$c_i = [4, 3, 3, 7, 2]$$

$$K = 10$$

```
k = K
x* = {}
for i = n to 1 by −1 do
    if S_{k,i} = 1 then
        add i to x*
        k = k − w_i
    end if
end for
```

$V =$

	{}	{1}	{1,2}	{1,2,3}	{1,2,3,4}	{1,2,3,4,5}	
	0	0	0	0	0	0	K=0
	0	0	0	0	0	2	K=1
	0	0	0	3	3	3	K=2
	0	0	0	3	3	5	K=3
	0	4	4	4	4	5	K=4
	0	4	4	4	4	6	K=5
	0	4	4	7	7	7	K=6
	0	4	4	7	7	9	K=7
	0	4	4	7	10	10	K=8
	0	4	7	7	10	12	K=9
	0	4	7	7	11	12	K=10

$S =$

	0	1	2	3	4	5	
	0	0	0	0	0	0	K=0
	0	0	0	0	0	1	K=1
	0	0	0	1	0	0	K=2
	0	0	0	1	0	1	K=3
	0	1	0	0	0	1	K=4
	0	1	0	0	0	1	K=5
	0	1	0	1	0	0	K=6
	0	1	0	1	0	1	K=7
	0	1	0	1	1	0	K=8
	0	1	1	0	1	1	K=9
	0	1	1	0	1	1	K=10

DYNAMIC PROGRAMMING

MANY APPLICATIONS

- Many (mostly discrete) optimization problems. Some of the main application areas are:
  - Bioinformatics (e.g. finding secondary structures in RNA molecules)
  - Control and information theory
  - Segmented least squares
  - Operations research (e.,g., scheduling for assembly lines, packaging, and inventory management)
  - Computer science: theory, graphics, AI, compilers, systems, string manipulation, e.g.:
    - Sequence alignment (used in text diff, speech recognition, computational biology, etc.)
    - Finding longest common subsequences (LCS) in strings

Some famous DP algorithms:

- Unix diff for comparing two files
- Needleman–Wunsch/Smith–Waterman for sequence alignment
- Viterbi for hidden Markov models
- Smith-Waterman for genetic sequence alignment
- Bellman-Ford-Moore for shortest path
- Cocke-Kasami-Younger for parsing context free grammars
- De Boor for evaluating spline curves
- Avidan–Shamir for seam carving
- Knuth-Plass for word wrapping in $T_{\mathrm{E}}X$

DYNAMIC PROGRAMMING

CONCLUDING REMARKS

- DP relies on working recursively and saving the results of solving simpler problems. These solutions to simpler problems are then used to compute the solution to more complex problems.

- DP is an elegant design principle, but it's only usable on very specific types of problems.
  - In the case of optimization problems, DP is especially useful for combinatorial problems that would otherwise take exponential time.
  - Since exponential time is unacceptable for all but the smallest problems, DP is sometimes essential.
  - However, only problems that satisfy the **principle of optimality** are suitable for DP solutions.

- Top-down vs. bottom-up? Different people have different intuitions.

- Once understood, DP is relatively easy to apply, but many people have trouble understanding it. In any case, DP solutions can often be quite complex and tricky!

DYNAMIC PROGRAMMING

FURTHER READING

- https://www.geeksforgeeks.org/dynamic-programming/ (contains a large list of problems / DP algorithm implementations in several programming languages!)
- Lecture notes by Prof. Dimitri P. Bertsekas https://www.mit.edu/~dimitrib/DP_Slides_2015.pdf

Questions?