SCC.121: Fundamentals of Computer Science
Sorting, Trees and Graphs

Algorithmic Paradigms: Dynamic Programming

Today's Lecture

Aim:

- Introduce the concept of:
  - Exhaustive search
  - Backtracking
  - Problem substructure

- Describe the dynamic programming approach.

Algorithmic paradigms

<u>Generic framework that underlies a class of algorithms:</u>

- Recursion
- Divide and conquer
- (Sweep-line algorithms)
- Greedy algorithms

- **Exhaustive search,**
- **Backtracking,**
- **Dynamic programming**

Computational problems

- Problem:
  - Domain of inputs In
    - Integers,
    - Reals,
    - Binary strings,
    - Black and White Image (2D matrix of 0s and 1s),

Computational problems

- Problem:
  - Domain of inputs In
  - Domain of outputs Out
    - Integers,
    - Reals,
    - Binary strings,
    - Classification: $\{cat, dog\}$

Computational problems

- <u>Problem:</u>
  - Domain of inputs In
  - Domain of outputs Out
  - For each input $x \in In$, you have a set of valid outputs: $valid(x)$
    - Sorting: x is a binary string and $valid(x)$ is the set that contains the sorted version of x

Computational problems

- <u>Problem:</u>
  - Domain of inputs In
  - Domain of outputs Out
  - For each input $x \in In$, you have a set of valid outputs: $valid(x)$

- Some algorithm A solves a problem P if for a given input $x \in In$ of size n, $A(x)$ gives one of the valid outputs for x, or in other words, $\mathrm{A}(x) \in valid(x)$.

- Most algorithms you've seen until now compute $A(x)$ in a clever way.
- Because time and memory are important resources

Algorithmic Paradigm: Exhaustive Search

But in fact, one way to solve problems is to use “brute-force”:

- This algorithmic paradigm is sometimes called **brute-force search**, sometimes called **exhaustive search**, and **generate and test**.
- What does solving a problem by “brute-force” entail?
  - Given an input x,
  - Test for all possible output y, one by one, whether y is a valid output i.e., whether $y \in valid(x)$
- Another way to look at it:
  - Possible outputs = **candidate solutions**
  - Try all candidate solutions

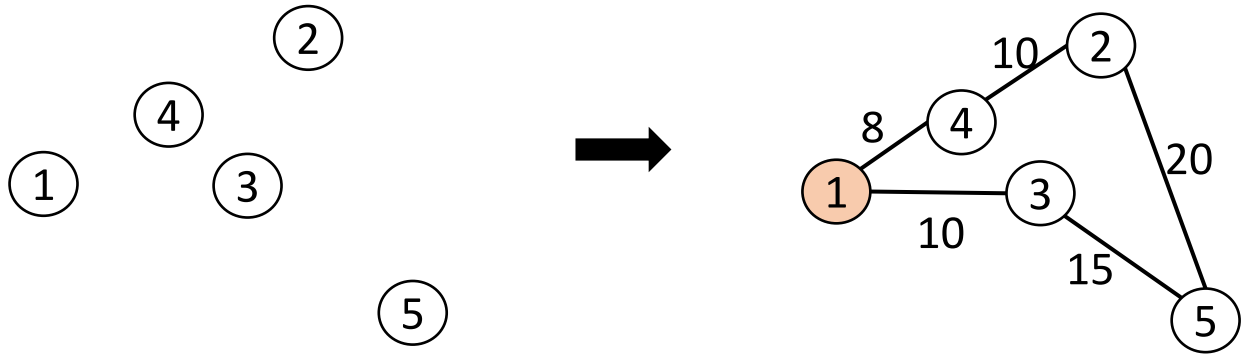
Exhaustive Search: Example 1

- Greatest common divisor of a and b:
  - Gcd(2,4) = 2
  - Gcd(5,7) = 1
  - Gcd(32,48) = 16

- Brute-force:
  - For all integers in $1, \ldots, \min\{a, b\}$, check if they divide a and b,
  - Keep the greatest such integer.

Exhaustive Search: Example 2

- <u>(Euclidean) Travelling Salesman Problem:</u>
  Given n nodes, distances between each pair and an origin city (node), compute smallest route starting at the origin city, going through all other city exactly once and coming back.

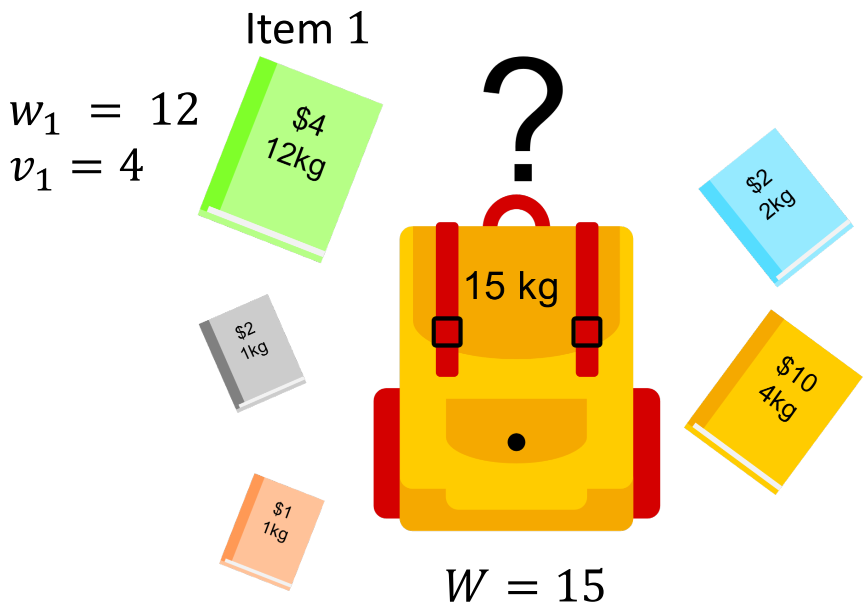


- <u>Brute-force</u>:
  For all possible orderings of the other $n-1$ nodes for the route, compute the route's distance and keep the route with smallest total distance.

- <u>Time complexity</u>: $O((n-1)!)$ possible routes (candidates)

Exhaustive Search: Example 3

- Knapsack: Given a set of items numbered 1 to n, such that item i has weight w_i and value v_i, and a total weight W, determine which items to put in the collection so that their total weight is < W and their total value is as large as possible.



Exhaustive Search: Example 3

- Knapsack: Given a set of items numbered 1 to n, such that item i has weight w_i and value v_i, and a total weight W, determine which items to put in the collection so that their total weight is < W and their total value is as large as possible.

- Brute-force:

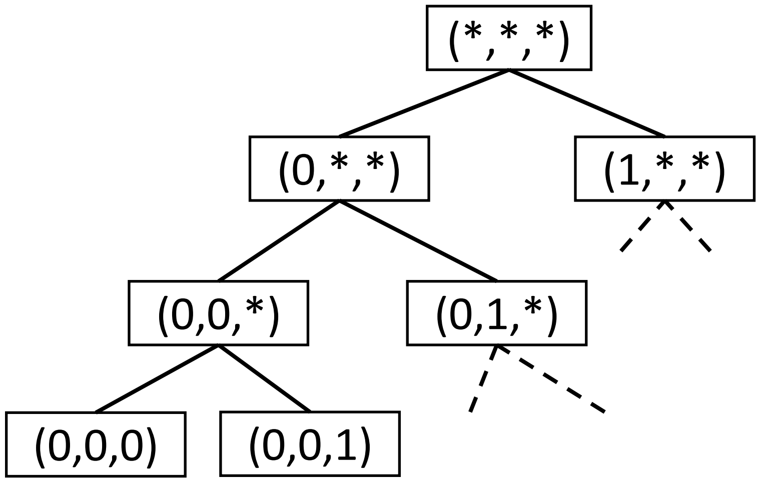
  For all subsets of these n items, compute the total weight and total value and keep the subset with largest total value, as long as its total weight is smaller than W.

- Time complexity: $O(2^n)$ possible subsets (candidates)

Candidate Space

- For a given problem, consider the set of all candidates:
  - That is a **candidate space**
  - Exhaustive search iterates through all candidates one by one and checks whether they are valid solutions.
  - Another possibility is to consider a **potential search tree** over the candidate space:
    - Tree root represents the candidate space (or also a generic candidate),
    - Its children represent disjoint subsets of the candidate space, obtained by **extending** the generic candidate,
    - And so on, until the leaves that represent the candidates are reached.

Search tree for Gcd(4,6): captures all possible solutions (binary encoded)

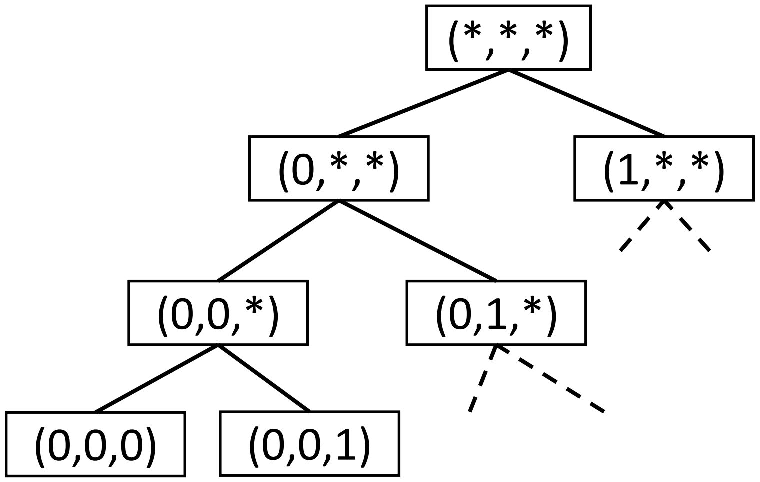


Algorithmic Paradigm: Backtracking

- Given a potential search tree approach:
  - Internal nodes allow to test multiple candidates at once, based on their common part,
  - There are many different ways to use this capability to test multiple candidates at once.

- In particular, you can use a depth-first search on this tree, and whenever you reach a (internal or leaf) node representing invalid candidates, you stop traversing this branch.

<u>Search tree for Gcd(4,6):</u> captures all possible solutions (binary encoded)

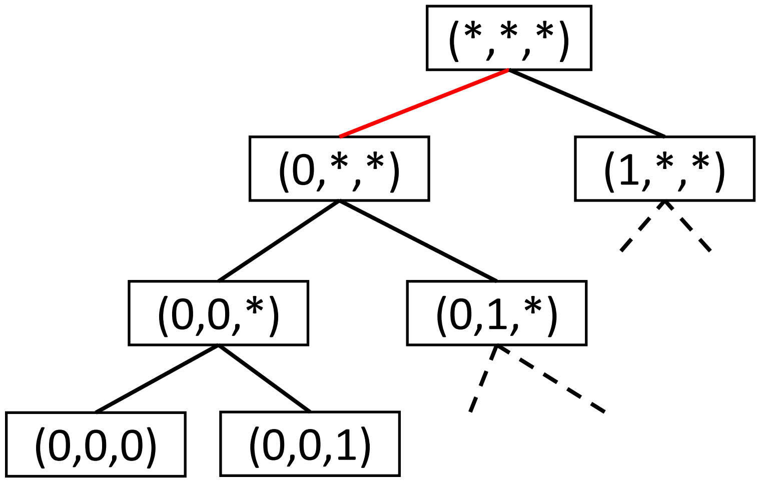


Algorithmic Paradigm: Backtracking

- Given a potential search tree approach:
  - Internal nodes allow to test multiple candidates at once, based on their common part,
  - There are many different ways to use this capability to test multiple candidates at once.

- In particular, you can use a depth-first search on this tree, and whenever you reach a (internal or leaf) node representing invalid candidates, you stop traversing this branch.

Search tree for Gcd(4,6): captures all possible solutions (binary encoded)

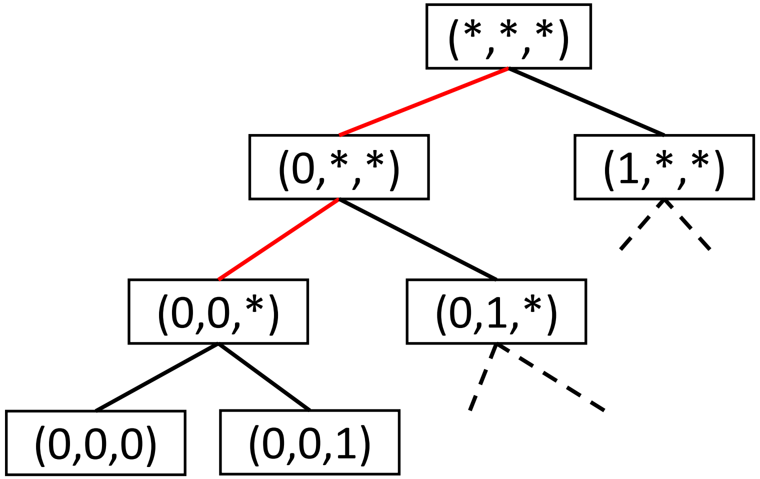


Algorithmic Paradigm: Backtracking

- Given a potential search tree approach:
  - Internal nodes allow to test multiple candidates at once, based on their common part,
  - There are many different ways to use this capability to test multiple candidates at once.

- In particular, you can use a depth-first search on this tree, and whenever you reach a (internal or leaf) node representing invalid candidates, you stop traversing this branch.

<u>Search tree for Gcd(4,6):</u> captures all possible solutions (binary encoded)

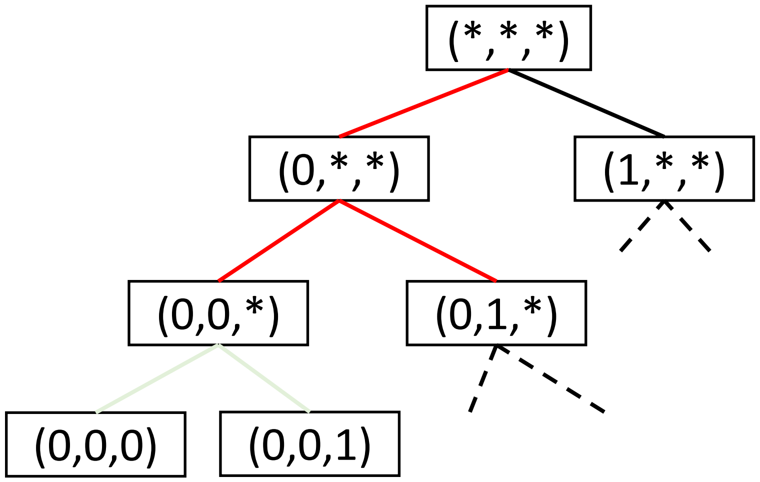


Algorithmic Paradigm: Backtracking

- Given a potential search tree approach:
  - Internal nodes allow to test multiple candidates at once, based on their common part,
  - There are many different ways to use this capability to test multiple candidates at once.

- In particular, you can use a depth-first search on this tree, and whenever you reach a (internal or leaf) node representing invalid candidates, you stop traversing this branch.

<u>Search tree for Gcd(4,6):</u> captures all possible solutions (binary encoded)



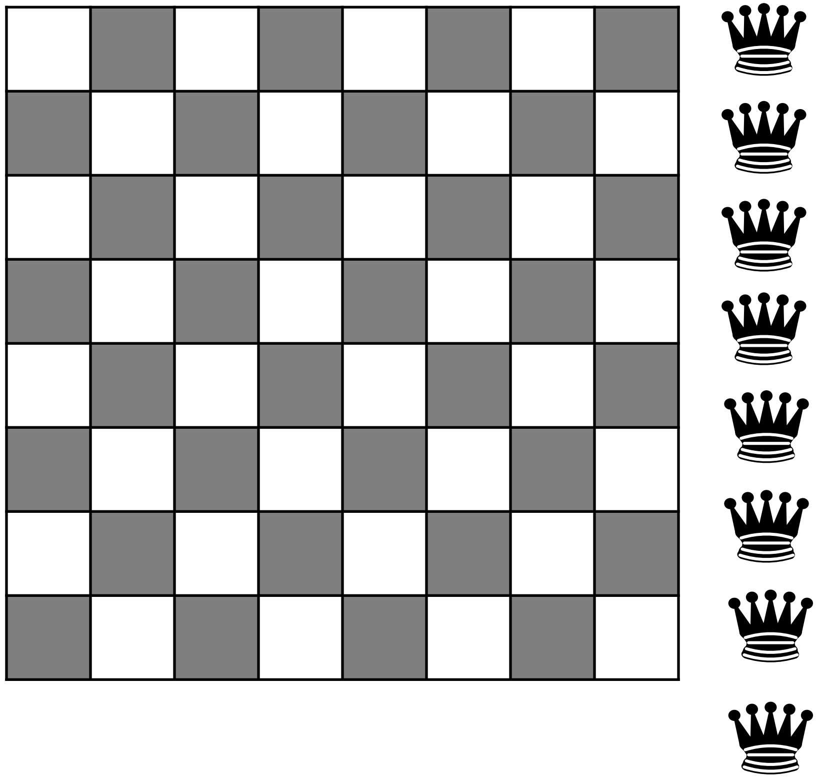
Beyond Backtracking

- In problems with additional constraints (e.g., knapsack), one can focus on eliminating subtrees of the search tree as early as possible.
  - These additional constraints, make even more subtrees unlikely to have a valid candidate.
  - Or one can compute some upper/lower bound on how good the candidates of a given subtree are -> **Branch and Bound paradigm**
  - One can use heuristics.

- <u>Related concepts:</u>
  - **Pruning,**
  - **Minimax principle** for the remaining possible moves in a chess game
  - **…**

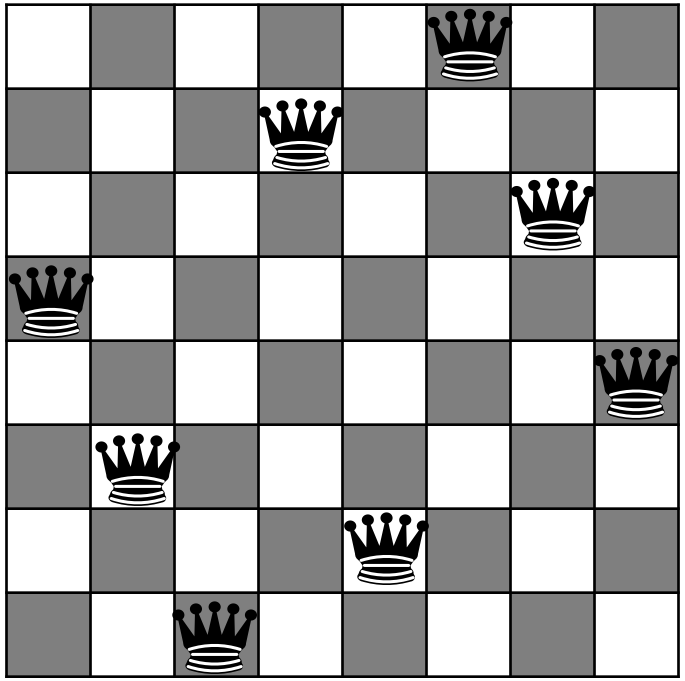
Usages of Backtracking

- Puzzle solving:
  - Eight queens puzzle



Usages of Backtracking

- Puzzle solving:
  - Eight queens puzzle



Usages of Backtracking

- Puzzle solving:
  - Eight queens puzzle,
  - Crosswords,
  - Sudoku,…

- Combinatorial optimization problems:
  - Knapsack,
  - Parsing, …

- Boolean satisfiability problem (SAT), constraint satisfaction problem, …

Beyond candidate space

- Previous paradigms focus on the candidate space.

- <u>Other paradigms focus on decomposing a "complicated" problem into "simpler" sub-problems:</u>
  - **Recursion:**
    Decompose some problem P on input of size n into one or more instances of same problem P on smaller inputs (of size $< n$)
  - **Divide-and-conquer:**
    Decompose some problem P on input of size n into two or more **disjoint** instances of same or related type.

Problem decomposition: First Aspect

- <u>Optimal substructure (of problem P):</u>
  - A solution to problem P can be obtained by the **combination of optimal solutions to its sub-problems.**
  - Describes/gives a recursive algorithm (and definition) for problem P.
  - Example of optimal substructure for the shortest path from node u to node v on some graph G :
    - Consider the shortest path from u to v.
    - Take any intermediate node w on that path.
    - Then the "shortest path from u to v" = "shortest path from u to w" + "shortest path from w to v"

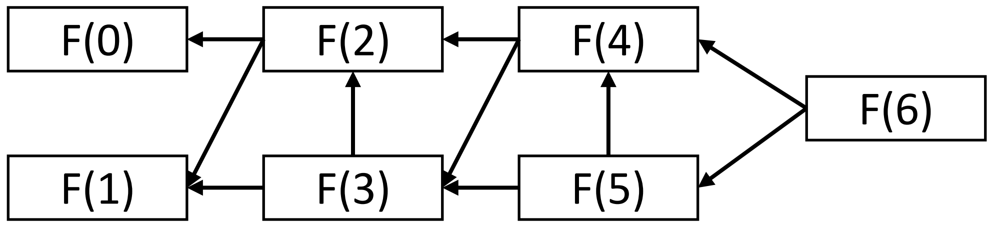
Problem decomposition: Second Aspect

- Overlapping/disjoint subproblems (of problem P):
  - If a solution to problem P can be obtained by the combination of optimal solutions to non-overlapping sub-problems -> **divide-and-conquer**

  - If a solution to problem P can be obtained by the combination of optimal solutions to overlapping sub-problems
    - Then the same subproblems will be solved over and over…
    - Example: Compute the Fibonacci numbers
      - $F(0) = 0$
      - $F(1) = 1$
      - $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$
      - That is, $F(2) = 1$, $F(3) = 2$, $F(4) = 3$, $F(5) = 5$, $F(6) = 8$, $F(7) = 13$, $F(8) = 21$,…

Problem decomposition: Second Aspect

- <u>Overlapping/disjoint subproblems (of problem P):</u>
  - If a solution to problem P can be obtained by the combination of optimal solutions to non-overlapping sub-problems -> **divide-and-conquer**
  - If a solution to problem P can be obtained by the combination of optimal solutions to overlapping sub-problems
    - Then the same subproblems will be solved over and over…
    - <u>Example</u>: Compute the Fibonacci numbers, $F(n) = F(n-1) + F(n-2)$

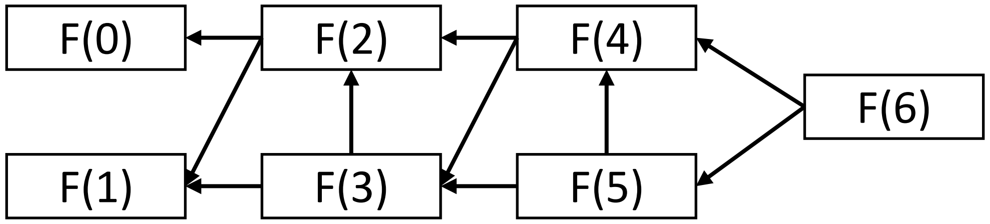


When computing $F(6)$, $F(4)$ is computed twice in the naïve recursive algorithm!

Problem decomposition: Second Aspect

- <u>Overlapping/disjoint subproblems (of problem P):</u>
  - If a solution to problem P can be obtained by the combination of optimal solutions to non-overlapping sub-problems -> **divide-and-conquer**

  - If a solution to problem P can be obtained by the combination of optimal solutions to overlapping sub-problems
    - Then the same subproblems will be solved over and over…
    - <u>Example</u>: Compute the Fibonacci numbers, $F(n) = F(n-1) + F(n-2)$



$$F(6) = F(5) + F(4)$$
$$F(6) = \big(F(4) + F(3)\big) + \big(F(3) + F(2)\big)$$
$$F(6) = \big(F(3) + F(2) + F(2) + F(1)\big) + (F(2) + F(1) + F(1) + F(0))$$

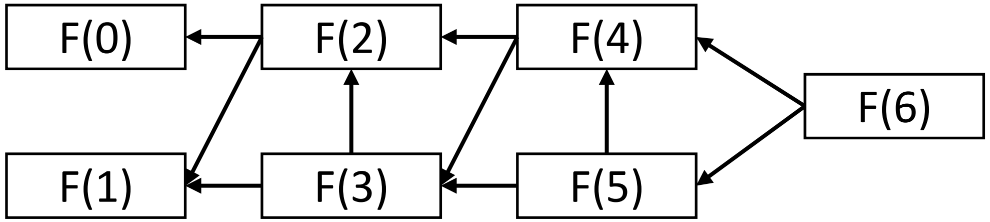
Algorithmic paradigm: Top-down dynamic programming

- Consider a problem with optimal substructure and overlapping subproblems.
- The **top-down dynamic programming approach** is:
  - Whenever a subproblem is solved, store the result in memory (e.g., in some array),
  - For every subsequent call to the subproblem, return the stored result.
- Storing computed values to avoid recomputing them is called **memoization.**

- **Computation vs memory tradeoff**

Top-down dynamic programming: Example

Computing the nth Fibonacci number



Algorithm:

- Keep an array Fib for the 0^{th} to $(n-1)$th Fibonacci numbers, with all entries null initially,
- Then, in algorithm, when $F(i)$ is called,
  - If $Fib[i]$ is null, compute $F(i)$ and store the result in Fib
  - Else, return $Fib[i]$

Algorithmic paradigm: Bottom-up dynamic programming

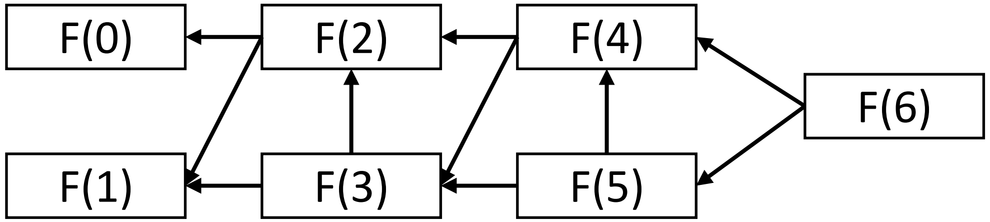
- Consider a problem with optimal substructure and overlapping subproblems.
- The **bottom-up dynamic programming approach** is:
  - Solve the smallest subproblems first, and store the result in memory,
  - Solve larger and larger subproblems by using the stored results

- **Can have better memory usage than top-down.**

Bottom-up dynamic programming: Example

Lancaster University

Computing the nth Fibonacci number

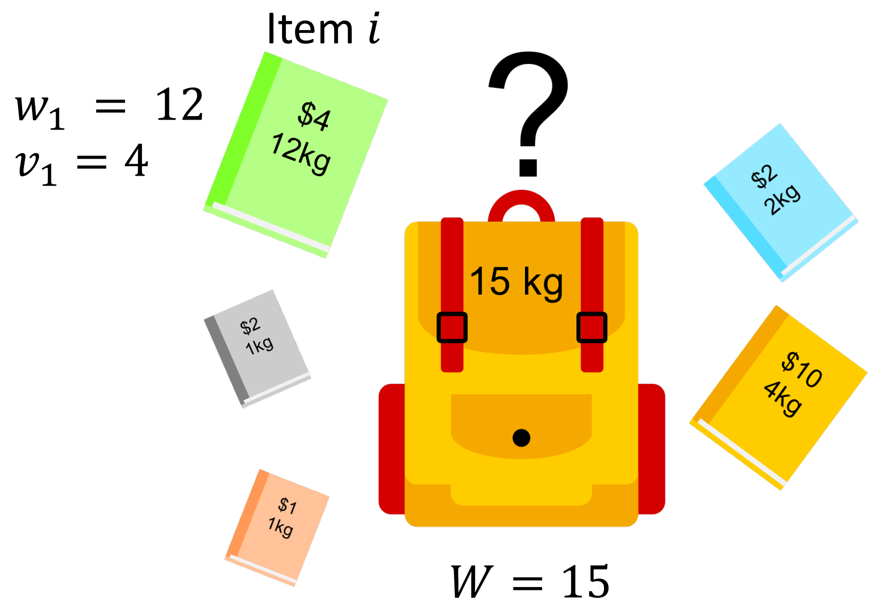


Algorithm:

- Have two variables currentFib and previous Fib, storing respectively the current Fibonacci number and the previous one, and initially set to 1 and 0.
- For $n - 1$ iterations,
  - Compute the next Fibonacci number (nextFib) by summing the two variables together,
  - Set previousFib = currentFib, currentFib = nextFib.

Dynamic programming: Knapsack

- Given a set of items numbered 1 to n, such that item i has weight w_i and value v_i, and a total weight W, determine which items to put in the collection so that their total weight is < W and their total value is as large as possible.



Dynamic programming: Knapsack

- Given a set of items numbered 1 to n, such that item i has weight w_i and value v_i, and a total weight W, determine which items to put in the collection so that their total weight is < W and their total value is as large as possible.
- Variable $x_i \in \{0,1\}$ denotes whether item i is in the collection
- **(Formal) Goal:** choose $x_1, \dots, x_n$ to maximise $\sum_{i=1}^{n} v_i x_i$,
  while keeping $\sum_{i=1}^{n} w_i\, x_i \leq W$

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \; = \; 0$ for any w
  - $m[j,w] \; = m[j-1,w]$ if $w_j > w$
  - $m[j,w] \; = \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1																
2																
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \;=\; 0$ for any w
  - $m[j, w] \;= m[j-1, w]$ if $w_j > w$
  - $m[j, w] \;= \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2																
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2																
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] = 0$ for any w
  - $m[j, w] = m[j-1, w]$ if $w_j > w$
  - $m[j, w] = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

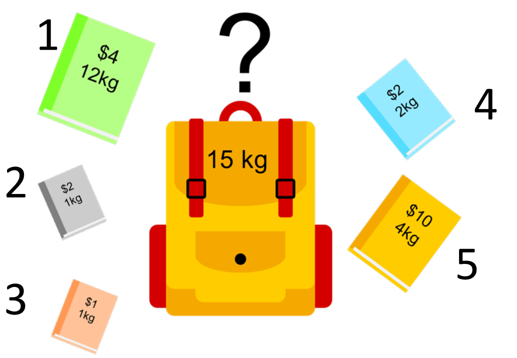
j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3																
4																
5																



$W = 15$

Dynamic programming: Knapsack

Lancaster University

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

Lancaster University

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3																
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] = 0$ for any w
  - $m[j,w] = m[j-1,w]$ if $w_j > w$
  - $m[j,w] = \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] = 0$ for any w
  - $m[j,w] = m[j-1,w]$ if $w_j > w$
  - $m[j,w] = \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

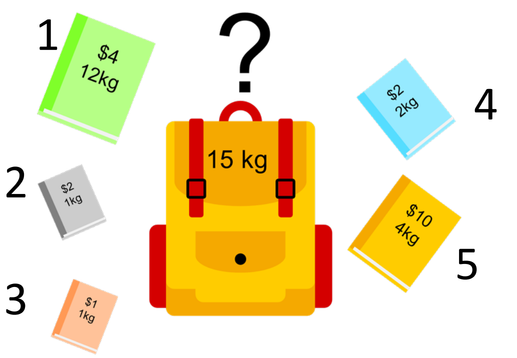
j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																



W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

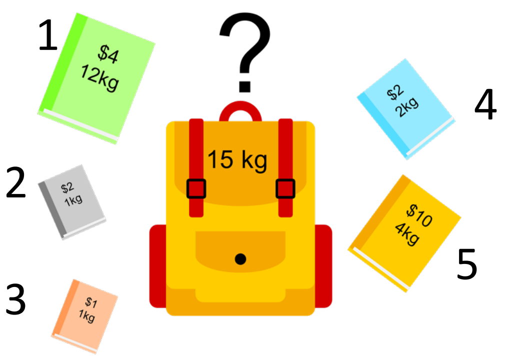
j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \le w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4																
5																



$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \; = \; 0$ for any w
  - $m[j,w] \; = m[j-1,w]$ if $w_j > w$
  - $m[j,w] \; = \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

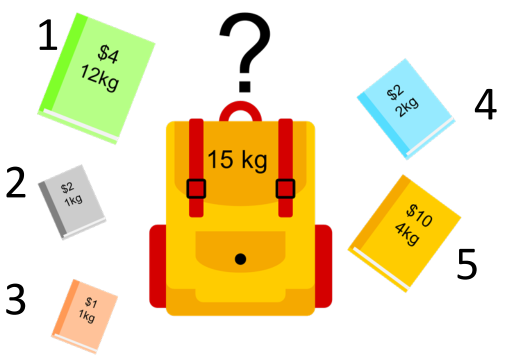
j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \;=\; 0$ for any w
  - $m[j, w] \;= m[j-1, w]$ if $w_j > w$
  - $m[j, w] \;= \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \le w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5																



$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \;=\; 0$ for any w
  - $m[j, w] \;= m[j-1, w]$ if $w_j > w$
  - $m[j, w] \;= \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \; = \; 0$ for any w
  - $m[j, w] \; = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \; = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \ = \ 0$ for any w
  - $m[j,w] \ = m[j-1,w]$ if $w_j > w$
  - $m[j,w] \ = \max\{m[j-1,w], m[j-1,w-w_j]+v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \ = \ 0$ for any w
  - $m[j, w] \ = m[j-1, w]$ if $w_j > w$
  - $m[j, w] \ = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \; = \; 0$ for any w
  - $m[j,w] \; = m[j-1,w]$ if $w_j > w$
  - $m[j,w] \; = \max\{m[j-1,w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \;=\; 0$ for any w
  - $m[j,w] \;= m[j-1,w]$ if $w_j > w$
  - $m[j,w] \;= \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] \;=\; 0$ for any w
  - $m[j, w] \;= m[j-1, w]$ if $w_j > w$
  - $m[j, w] \;= \max\{m[j-1, w], m\left[j-1, w-w_j\right] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j, w]$: the max value one can get over the first j items with total weight at most w
  - $m[0, w] = 0$ for any w
  - $m[j, w] = m[j-1, w]$ if $w_j > w$
  - $m[j, w] = \max\{m[j-1, w], m[j-1, w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

W = 15

Dynamic programming: Knapsack

- What are the subproblems of (0-1) knapsack?
  - Computing $m[j,w]$: the max value one can get over the first j items with total weight at most w
  - $m[0,w] \; = \; 0$ for any w
  - $m[j,w] \; = m[j-1,w]$ if $w_j > w$
  - $m[j,w] \; = \max\{m[j-1,w], m[j-1,w-w_j] + v_j\}$ if $w_j \leq w$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: {… }

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w - w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{\ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5, \ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

W = 15

Solution: $\{5, \ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = \ 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack\left(j-1, w-w_j\right)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$\mathrm{W} = 15$

Solution: $\{5{,}4, \ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5, 4, \dots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

W = 15

Solution: $\{5,4,3,\ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5, 4, 3, \dots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5,4,3,2,\ldots\}$

Dynamic programming: Knapsack

- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5,4,3,2, \dots\}$

Dynamic programming: Knapsack

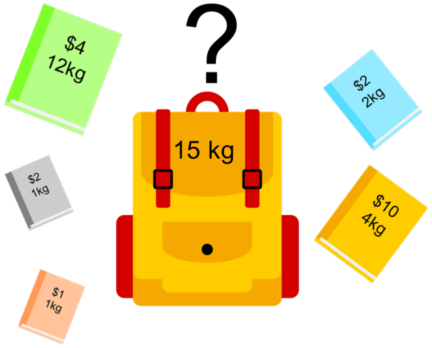
- How to compute the set of items of the solution, not just the total value?
  - Compute set of items recursively using $knapsack(j, w)$ function
  - If $j = 0$ then return empty set $\{\}$
  - Else if $m[j, w] > m[j-1, w]$ then $\{i\} \cup knapsack(j-1, w-w_j)$
  - Else return $knapsack(j-1, w)$

j \ w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	3	3	3	3	3	3	3	3	3	3	4	6	7	7
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

$W = 15$

Solution: $\{5,4,3,2\}$

0-1 Knapsack using dynamic programming

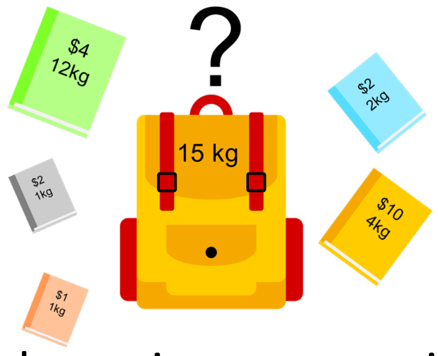


- Correctness
- Time complexity: $O(nW)$
- Space complexity: $O(nW)$

- <u>What about NP-completeness?</u>
  - Knapsack is NP-complete
  - But isn't it conjectured that NP-complete problems cannot be solved in polynomial time?
  - Input W contributes only $\log W$ to the problem input, unlike n.   Why?
    - W can be encoded using only $\log W$ bits.
    - Indeed, there are n items and each item contributes some value to the input, so that is $\Theta(n)$ bits in total.
  - Knapsack has **pseudo-polynomial runtime** and thus is said to be **weakly NP-complete**

Beyond 0-1 Knapsack

- Given a set of items numbered 1 to n, such that item i has weight w_i and value v_i, and a total weight W, determine which items to put in the collection so that their total weight is < W and their total value is as large as possible.



- <u>Unbounded Knapsack:</u>
  - No longer constrained to at most one of each item
  - $x_i \geq 0$ denotes how many items i you have in the collection
  - **Goal**: maximise $\sum_{i=1}^{n} v_i x_i$, but keeping $\sum_{i=1}^{n} w_i\, x_i \leq W$
- You can similarly define subproblems and use either bottom-up or top-down dynamic programming approach.

- And there are many other knapsack variations, as well as many other ways to solve (or approximate) them.

Applications of dynamic programming

- **Combinatorial optimization problem:**
  - Knapsack problems,
  - Travelling Salesman problems,…
- **Other computer science problems:**
  - Tower of Hanoi (puzzle),
  - Matrix chain multiplication (mathematic operation),…
- **But also for computational problems in other fields:**
  - Bioinformatics and computational biology
  - Economy,
  - Control theory,
  - Operations research ,..

Algorithmic paradigms: Summary

<u>Generic framework that underlies a class of algorithms:</u>

- **Exhaustive search,**
- **Backtracking,**

- Greedy algorithms
- Recursion
- Divide and conquer
- **Dynamic programming**

- (Sweep-line algorithms)

Summary

Today's lecture:

Introduced:

- Exhaustive (or brute-force) search and backtracking,
- Problem substructure,
- Dynamic programming.

- **Next Lecture:**
- **Any questions?**