TDDD95 Algorithmic Problem Solving 6hp, vt2026

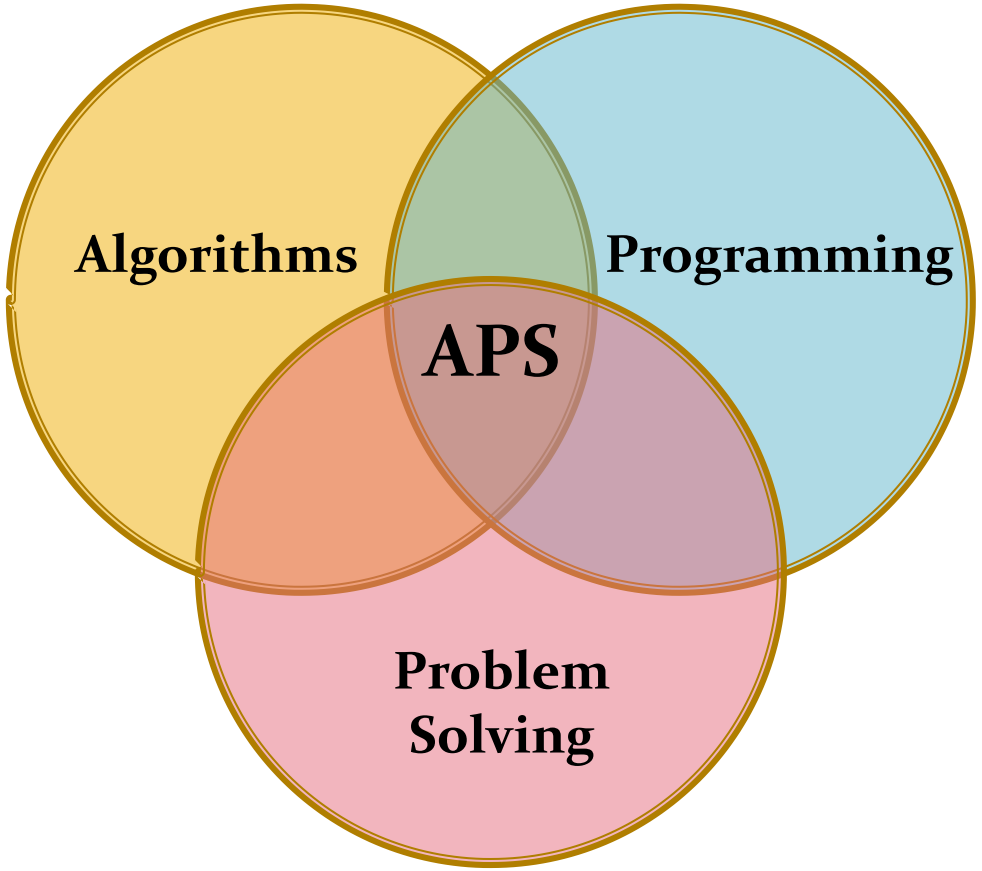
Fredrik Heintz
Dept of Computer and Information Science
Linköping University

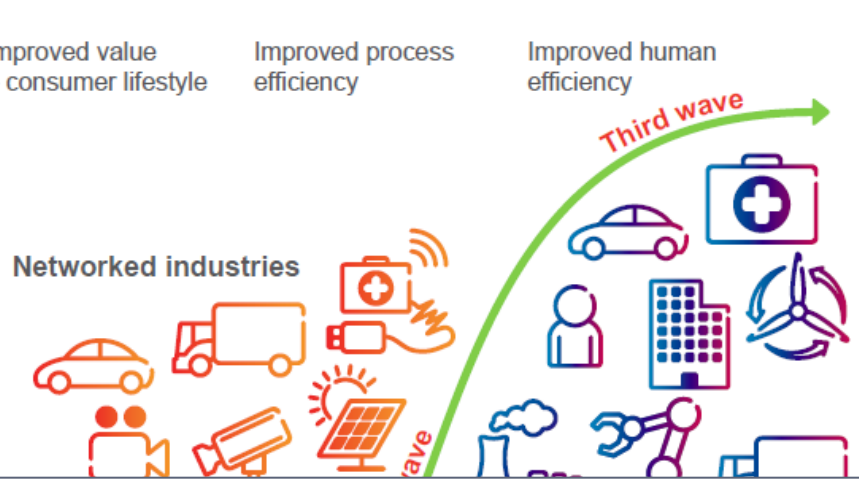
Outline

- What is algorithmic problem solving?
- Why is algorithmic problem solving important?
- What will be studied in this course?
- A method for algorithmic problem solving
- Common algorithmic problem-solving approaches
- Common data structures and algorithms
- Pragmatic algorithmic problem-solving using Kattis

What is Algorithmic Problem Solving?

- Algorithmic problem solving is about developing correct and working algorithms to solve classes of problems.
- The problems are normally very well defined, and you know there is a solution, but they can still be very hard.





Swedsoft

ERICSSON

Mjukvaran är själen i svensk industri

Those that really understand and take advantage of software technology owns the future!

TM

facebook

Spotify

Google™

LINKÖPINGS UNIVERSITET

LINKÖPINGS UNIVERSITET

acm icpc

IBM event sponsor

international collegiate programming contest

The 3*n* + 1 problem

Background

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

The Problem

Consider the following algorithm:

```
1.      input n
2.      print n
3.      if n = 1 then STOP
4.              if n is odd then  n ⟵ 3n + 1
5.              else  n ⟵ n/2
6.      GOTO 2
```

Given the input 22, the following sequence of numbers will be printed 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1{,}000{,}000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed (including the 1). For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between i and j.

The Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

You can assume that no operation overflows a 32-bit integer.

The Output

For each pair of input integers i and j you should output i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers i and j must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

Example: The 3n+1 problem

100 The $3n + 1$ problem

Background

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

The Problem

Consider the following algorithm:

1. input n
2. print n
3. if $n = 1$ then STOP
4. if n is odd then $n \leftarrow 3n + 1$
5. else $n \leftarrow n/2$
6. GOTO 2

Given the input 22, the following sequence of numbers will be printed

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1,000,000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed before and including the 1 is printed. For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between and including both i and j.

Example: The 3n+1 problem

The Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

The Output

For each pair of input integers i and j you should output i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers i and j must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

Example: The 3n+1 problem

- Follow the instructions in the problem!
- Memoization to speed it up.
- Table lookup to solve it in constant time.
- Gotchas:
  - *j* can be smaller than *i*.
  - *j* can equal *i*.
  - The order of *i* and *j* in output must be the same as the input, even when *j* is smaller than *i*.

Course Goals

The goals of the course are you should be able to:

- analyze the efficiency of different approaches to solving a problem to determine which approaches will be reasonably efficient in a given situation,
- compare different problems in terms of their difficulty,
- use algorithm design techniques such as greedy algorithms, dynamic programming, divide and conquer, and combinatorial search to construct algorithms to solve given problems,
- strategies for testing and debugging algorithms and data structures, and
- quickly and correctly implement a given specification of an algorithm or data structure.

Examination

- LAB1 4hp
  - individually solving the 4 lab assignments
  - optionally participating in problem solving sessions
- UPPG1 2hp,
  - individually solving the 13 weekly homework exercises, e.g.:
    - Data structures
    - Greedy Problems and Dynamic Programming
    - Graph Algorithms
    - Search
    - Math-related Problems
    - Computational Geometry.

Examination – Labs

Examination

The course is graded on a scale 3-5, and the course grade depends on your grade for the LAB1 and UPG1 parts. **The labs and exercises should be solved individually.** You are welcome to discuss the problems with other students, but your implementation should be individual.

LAB1

To pass the LAB1 part of the course, you must fulfill the following requirements:

1. Accumulate a sufficient number of LAB1 points. This also determines your LAB1 grade.

You accumulate points in LAB1 by solving labs and participating in **problem solving sessions**. Each problem you or your group solves in a problem solving session gives you 2 points, but you may only use the result from your two best individual sessions. Additionally, you obtain 1-2 point for solving each task from the **lab assignments**. The maximum possible number of points is therefore 8 (lab 1) + 3*9 (labs 2-4) + 12*2 (sessions) = 59. Your LAB1 grade is a function of your total points.

Points	Grade LAB1
20	3
30	4
40	5

We use **Webreg** to track your progress for labs and sessions.

Examination – Exercises

UPG1

To pass the UPG1 part of the course, you must fulfill the following requirements:

1. Solve at least one problem from each of the 13 **exercise sets**, before or after the corresponding deadline.
2. Accumulate a sufficient number of UPG1 points. This also determines your UPG1 grade.

The problems come in three difficulty classes: A, B, and C. You are awarded 1 point in the corresponding difficulty class for each problem you solve before its deadline, and 0.5 points if you solve it after the deadline. The table below specifies how many points you must accumulate for the different UPG1 grades.

A	B	C	Grade UPG1
13	0	0	3
13	7	0	4
13	7	6	5

You may of course replace easier problems with harder problems. For example, if you have 6 A points and 7 B points, you have 13 points at difficulty A or above, but not an *additional* 7 points at difficulty B or above. You would therefore get grade 3.

The exercises aren't tracked in Webreg. Instead, your total points will be calculated at the end of the course. If you want to check your current status, the easiest way is to calculate your points from your course summary, which you can access by going to your Kattis profile and pressing the course link. Alternatively, you can send an email to the course assistant.

Examination – Course grade

Course grade

To pass the course, you must obtain at least grade 3 in both LAB1 and UPG1. Your full course grade is then determined as (LAB1 grade + UPG1 grade) / 2.

	UPPG1		
LAB1	5	4	3
5	5	5	4
4	5	4	4
3	4	4	3

The Schedule – VT 1

- 21/1 How to solve algorithmic problems, intro seminar
- 22/1 Practice problem solving session: 13.15-17.00 with discussion
- 30/1 Deadline Ex 1 (Greedy and DP 1) – Seminar Ex1 and Data structures
- 6/2 Deadline Ex2 (Data structures) – Seminar Ex2 and Arithmetic
- 13/2 Deadline Ex3 (Arithmetic) – Seminar Ex3 and Problem solving approaches
- 19/2 Deadline Lab Assignment 1 (Data structures, Greedy/Dynamic, Arithmetic)
- **19/2 Problem solving session (individual based on Lab 1)**
- 20/2 Deadline Ex4 (Greedy and DP II) – Seminar Ex 4 and Graphs I
- 27/2 Deadline Ex5 (Graphs I) – Seminar Ex5 and Graphs II
- 6/3 Deadline Ex6 (Graphs II) – Seminar Ex6 and Graphs III
- 11/3 Deadline Ex 7 (Graphs III) - Seminar Ex7 and Strings I

The Schedule – VT 2

- 30/3 Deadline Lab Assignment 2 (Graphs)
- **30/3 Problem solving session (individual based on Lab 2)**
- 10/4 Deadline Ex8 (Strings I) – Seminar Ex8 and Strings II
- 17/4 Deadline Ex9 (Strings II) – Seminar Ex9 and Number Theory
- 24/4 Deadline Ex10 (Number Theory) – Seminar Ex 11 and Combinatorial Search
- 27/4 Deadline Lab Assignment 3 (Strings, String Matching and Number Theory)
- **27/4 Problem solving session (individual based on Lab 3)**
- 29/4 Deadline Ex11 (Combinatorial Search) – Seminar Ex12 and Computational Geometry
- 8/5 Deadline Ex12 (Computational Geometry)
- 13/5 Deadline Ex13 (Combinatorics and Probability Theory)
- 25/5 Deadline Lab Assignment 4 (Computational Geometry)
- **25/5 Problem solving session (individual based on Lab 4)**

Steps in solving algorithmic problems

- Estimate the difficulty
  - Theory (size of inputs, known algorithms, known theorems, …)
  - Coding (size of program, many cases, complicated data structures, …)
  - Have you seen this problem before? Have you solved it before? Do you have useful code in your code library?
- **Understand the problem!**
  - What is being asked for? What is given? How large can instances be?
  - Can you draw a diagram to help you understand the problem?
  - Can you explain the problem in your own words?
  - Can you come up with good examples to test your understanding?

Steps in solving algorithmic problems

- Determine the right algorithm or algorithmic approach
  - Can you solve the problem using brute force?
  - Can you solve the problem using a greedy approach?
  - Can you solve the problem using dynamic programming?
  - Can you solve the problem using search?
  - Can you solve the problem using a known algorithm in your code library?
  - Can you modify an existing algorithm? Can you modify the problem to suite an existing algorithm?
  - Do you have to come up with your own algorithm?
- **Solve the problem! ☺**

Time Limits and Computational Complexity

- The normal time limit for a program is a few seconds.
- You may assume that your program can do about 100M operations within this time limit.

n	Worst AC Complexity	Comments
≤ [10..11]	$O(n!)$, $O(n^6)$	Enumerating permutations
≤ [15..18]	$O(2^n \times n^2)$	DP TSP
≤ [18..22]	$O(2^n \times n)$	DP with bitmask technique
≤ 100	$O(n^4)$	DP with 3 dimensions and O(n) loop
≤ 450	$O(n^3)$	Floyd Warshall's (APSP)
≤ 2K	$O(n^2 \log_2 n)$	2-nested loops + tree search
≤ 10K	$O(n^2)$	Bubble/Selection/Insertion sort
≤ 1M	$O(n \log_2 n)$	Merge Sort, Binary search
≤ 100M	$O(n)$, $O(\log_2)$, $O(1)$	Simulation, find average

Important Problem Solving Approaches

- Simulation/Ad hoc
  - Do what is stated in the problem
  - Example: Simulate a robot
- Greedy approaches
  - Find the optimal solution by extending a partial solution by making locally optimal decisions
  - Example: Minimal spanning trees, coin change in certain currencies
- Divide and conquer
  - Take a large problem and split it up in smaller parts that are solved individually
  - Example: Merge sort and Quick sort
- Dynamic programming
  - Find a recursive solution and compute it “backwards” or use memoization
  - Example: Finding the shortest path in a graph and coin change in all currencies
- Search
  - Create a search space and use a search algorithm to find a solution
  - Example: Exhaustive search (breadth or depth first search), binary search, heuristic search (A*, best first, branch and bound)

Complete Search a.k.a. Brute Force

- When a problem is small or (almost) all possibilities have to be tried *complete search* is a candidate approach.
- To determine the feasibility of complete search estimate the number of calculations that have to be made in the worst case.
- *Iterative complete search* uses nested loops to *generate* every possible complete solution and *filter* out the valid ones.
  - Iterating over all permutations using next_permutation
  - Iterating over all subsets using bit set technique
- *Recursive complete search* extends a partial solution with one element until a complete and valid solution is found.
  - This approach is often called *recursive backtracking*.
  - *Pruning* is used to significantly improve the efficiency by removing partial solutions that can not lead to a solution as soon as possible. In the best case only valid solutions are generated.

Complete Search a.k.a. Brute Force

We have three different integers, x, y and z, which satisfy the following three relations:

- $x + y + z = A$
- $xyz = B$
- $x^2 + y^2 + z^2 = C$

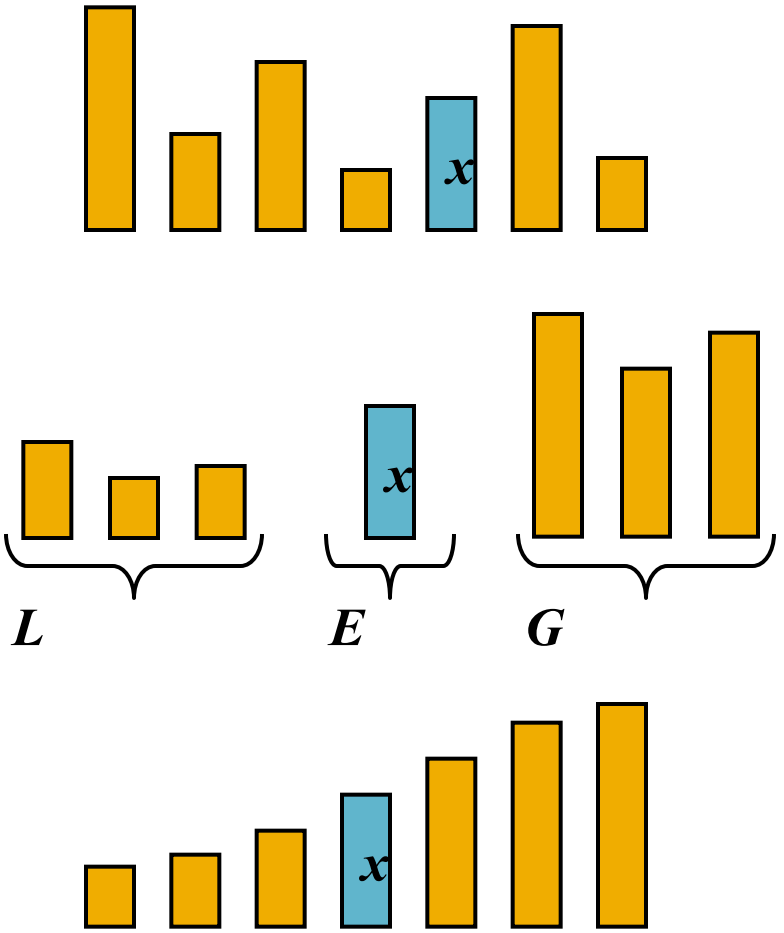
You are asked to **write a program** that solves for x, y and z for given values of A, B and C.

Divide and Conquer

- Divide and conquer is very common and powerful technique which divides a problem into smaller parts, solves each part recursively and then puts together the answer from the pieces.
- Many well known algorithms are based on divide and conquer such as quick sort, merge sort and binary search.
- Binary search is a very versatile and useful technique which can be used to
  - find a particular value in a sorted range,
  - find the parameters of a (convex) function that gives a particular value,
  - find the minimum/maximum value of a function.
- Binary search can be implemented either using built in functions (lower_bound/upper_bound), iterating until the difference between the end points is small enough or iterate a constant but sufficiently large number of times.

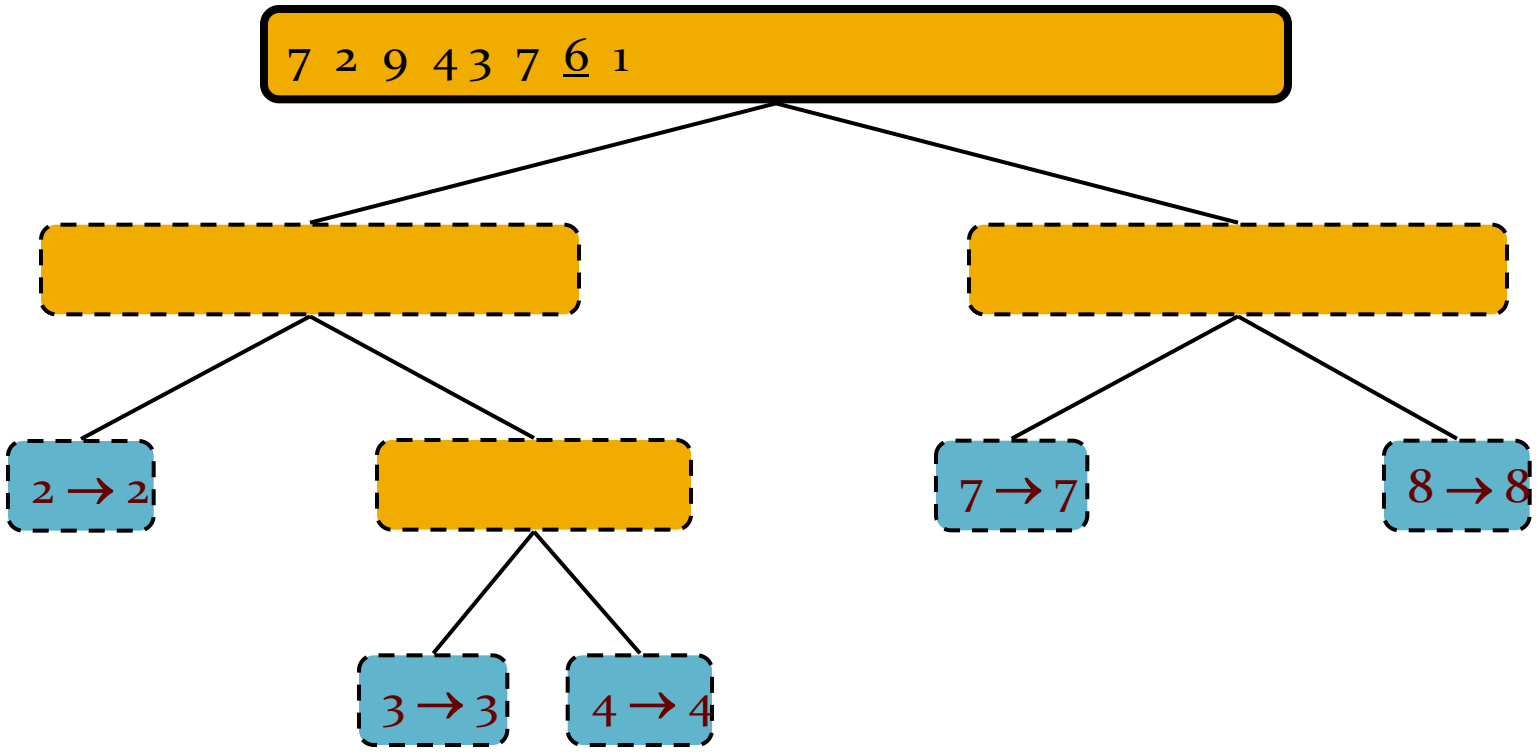
Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



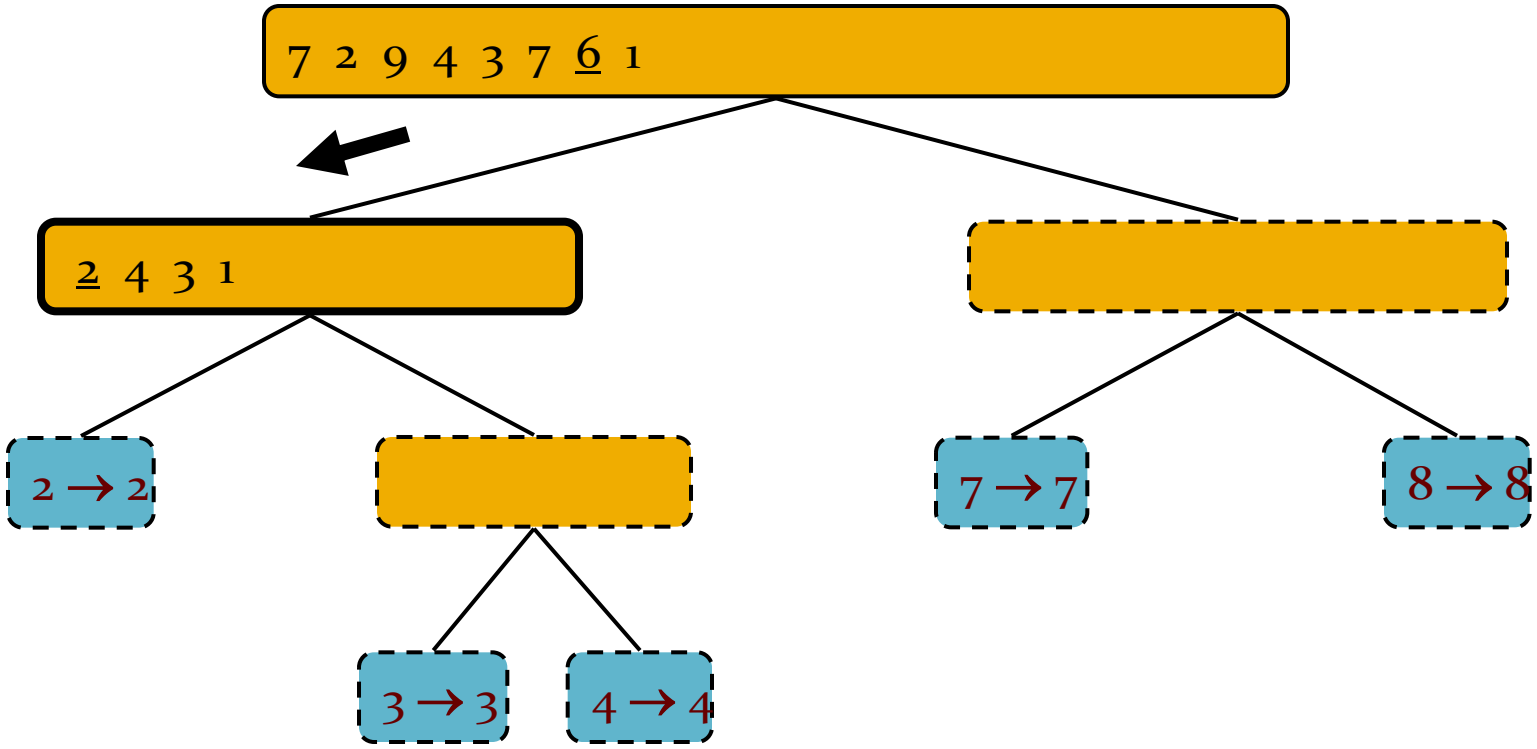
Quick-Sort Example

- Pivot selection



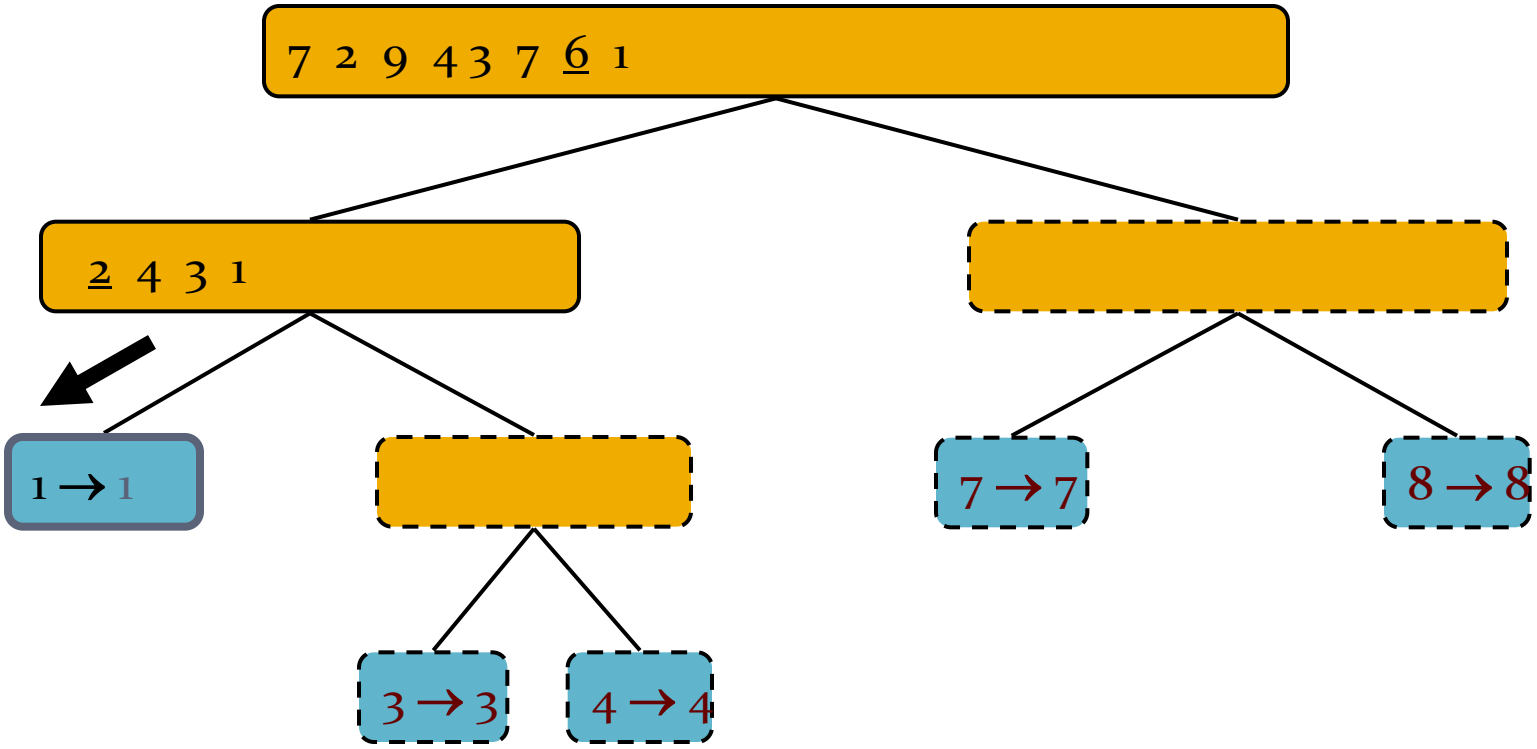
Quick-Sort Example

- Partition, recursive call, pivot selection



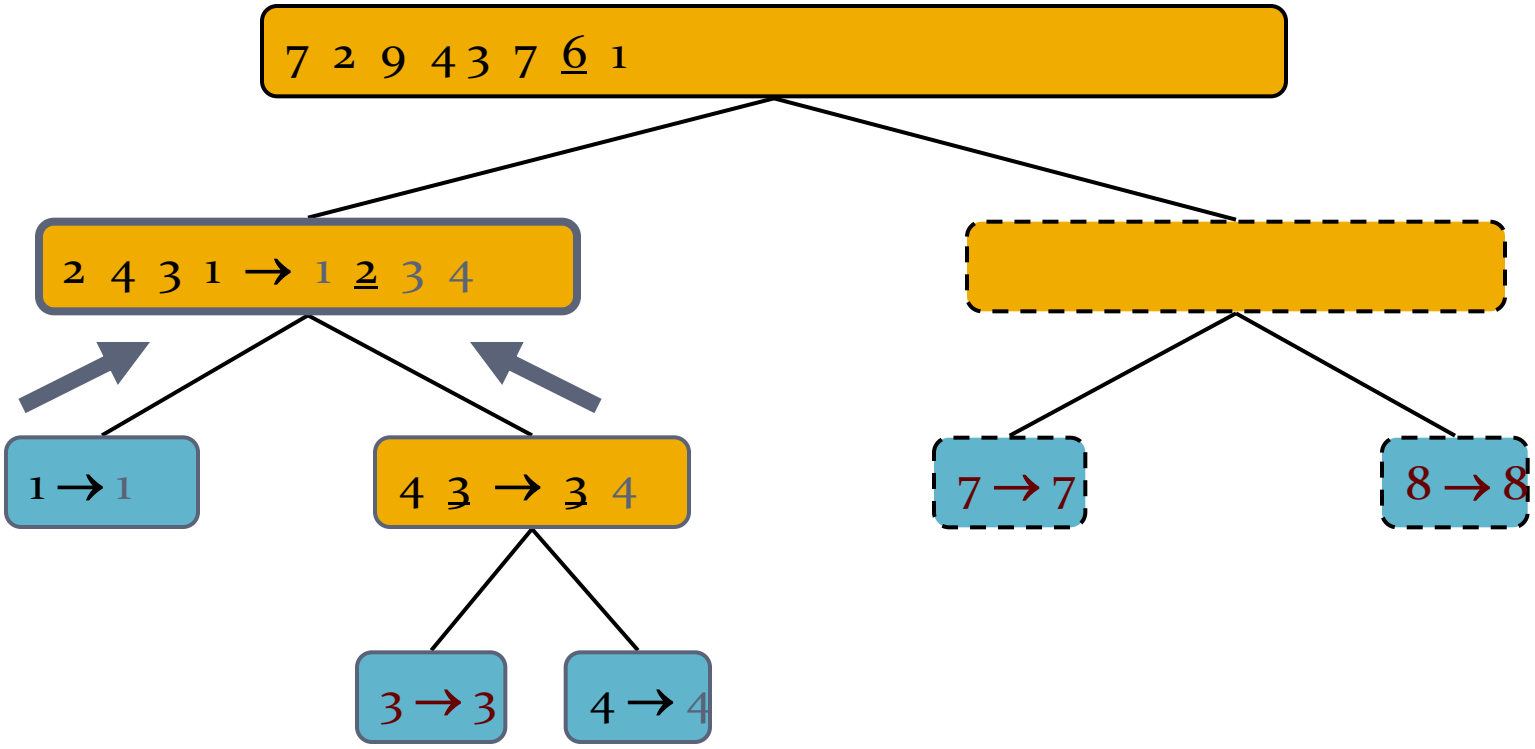
Quick-Sort Example

- Partition, recursive call, base case



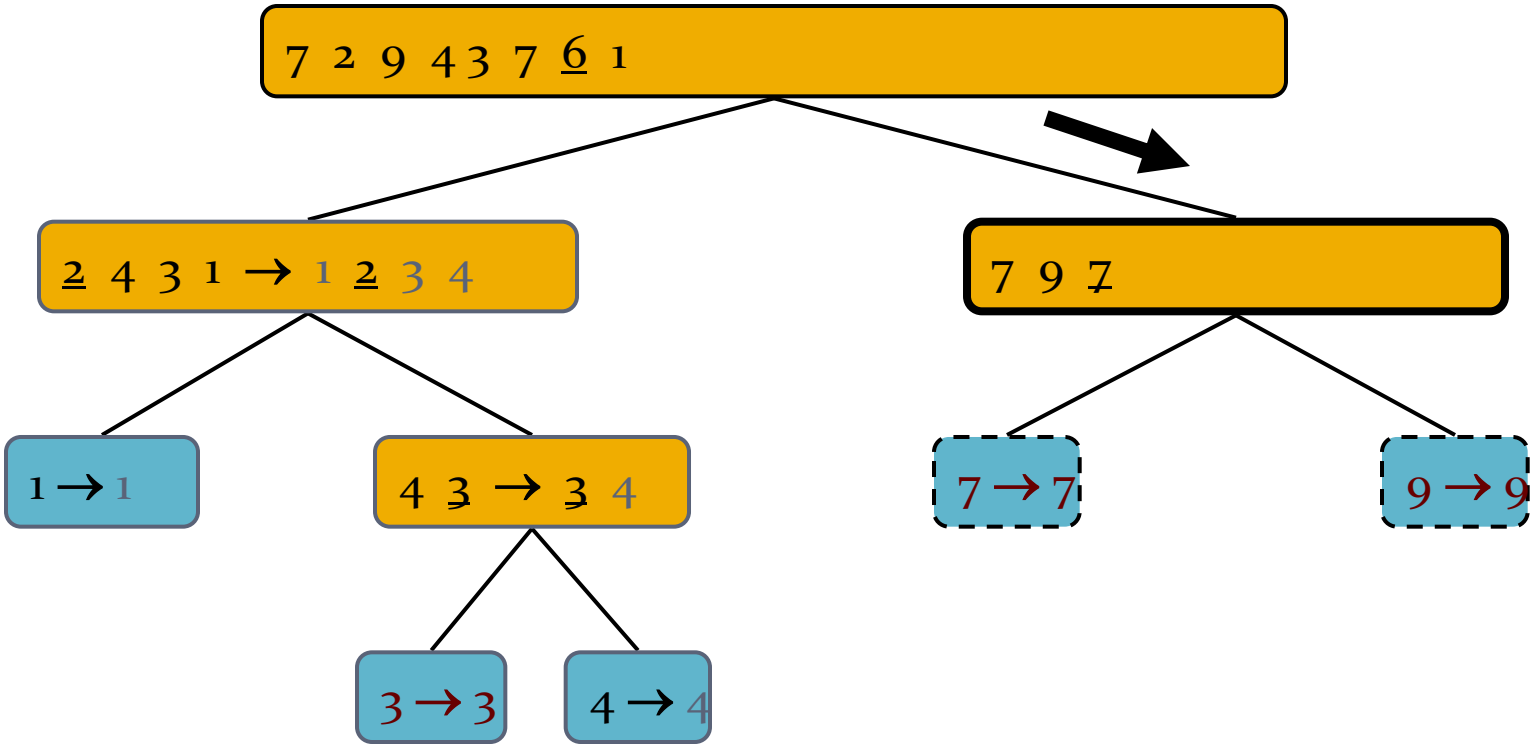
Quick-Sort Example

- Recursive call, …, base case, join



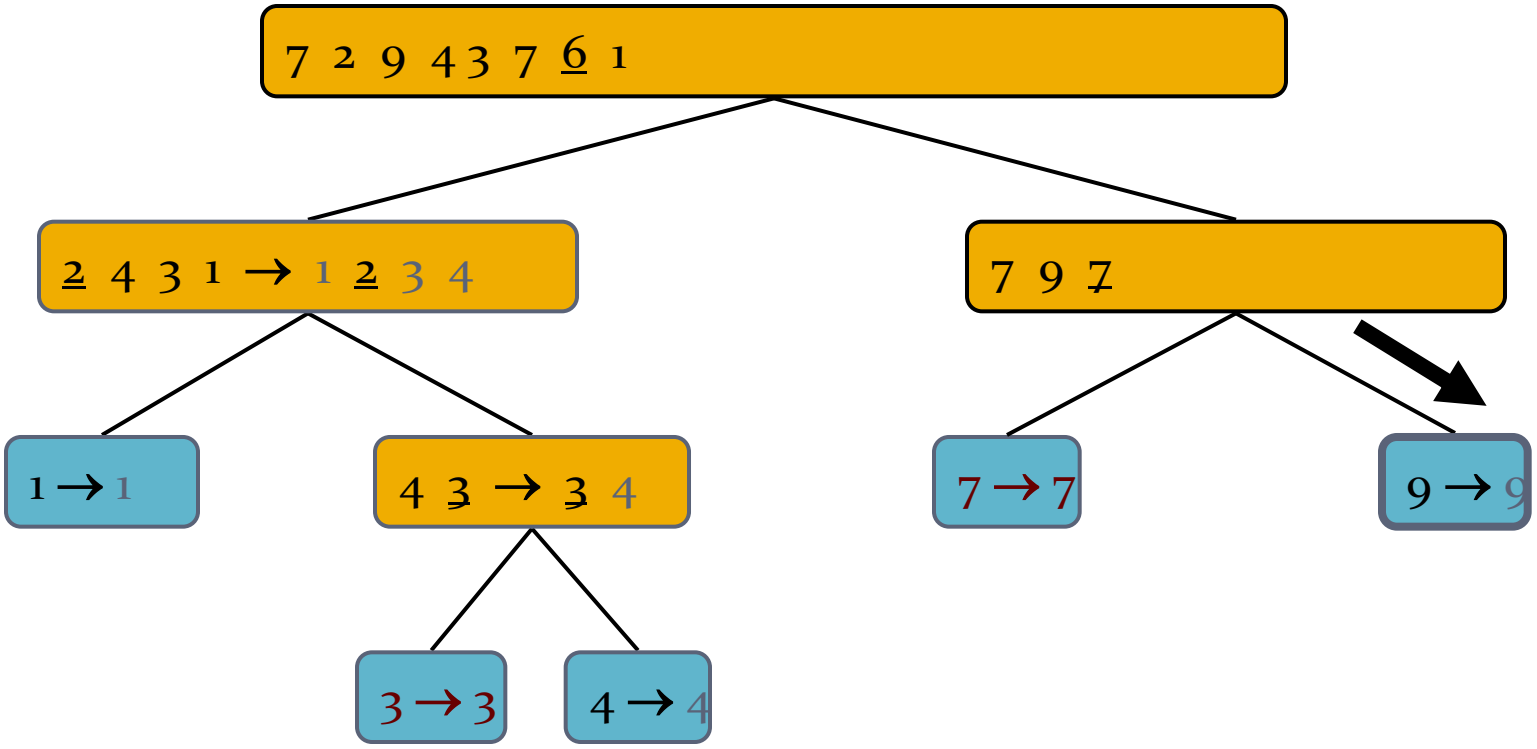
Quick-Sort Example

- Recursive call, pivot selection



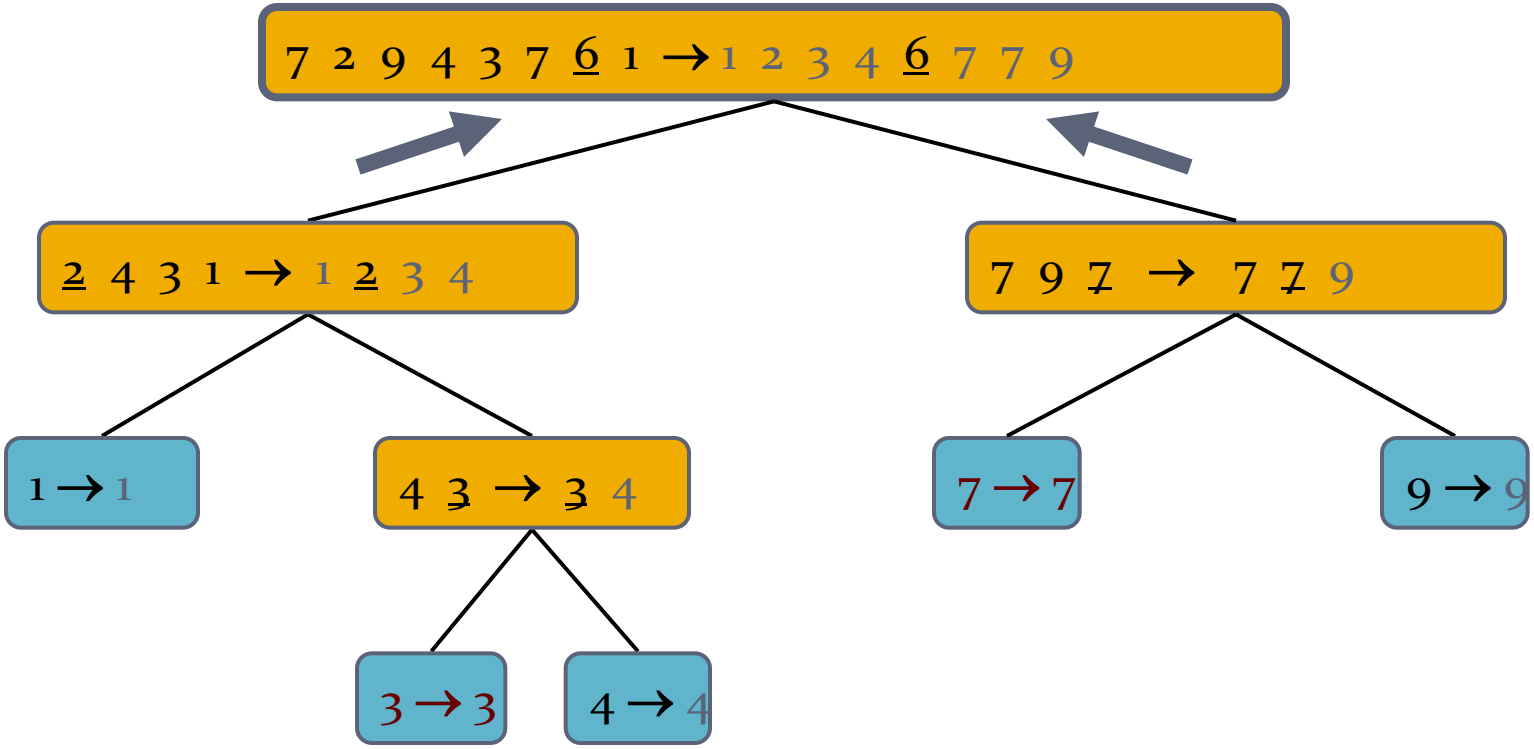
Quick-Sort Example

- Partition, …, recursive call, base case



Quick-Sort Example

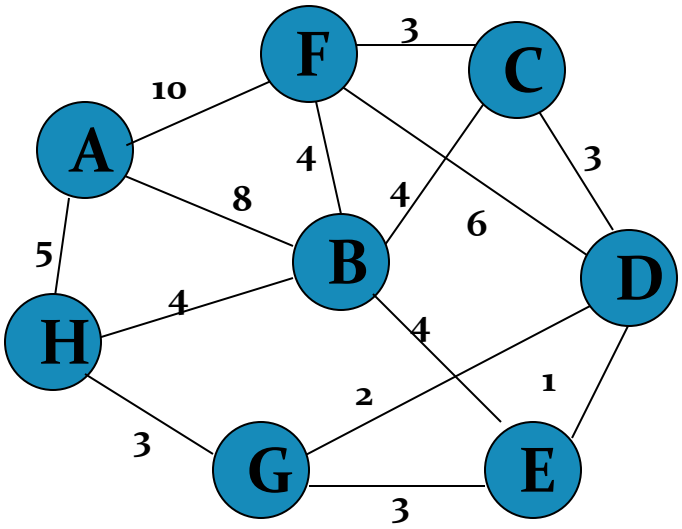
- Join, join



Greedy

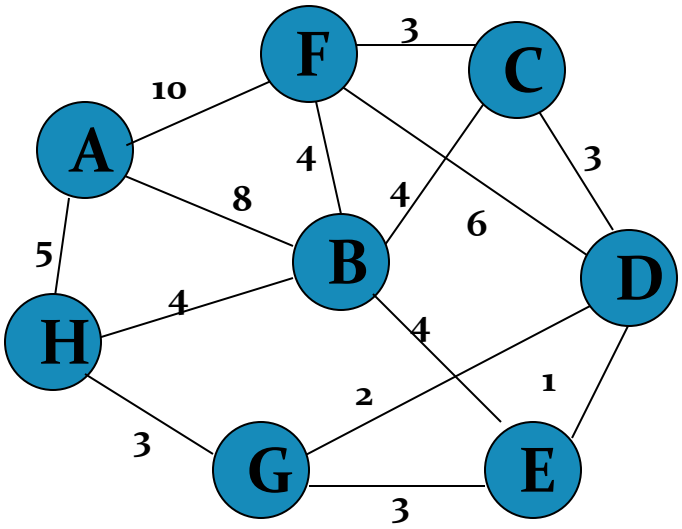
- An algorithm is said to be greedy if it makes a locally optimal choice in each step towards the globally optimal solution.
- For a greedy algorithm to give a globally optimal result a problem must have two properties:
  - It has optimal sub-structures, i.e. an optimal solution contains the optimal solutions to sub problems.
  - It has the greedy choice property, i.e. if we extend a partial solution by making a locally optimal choice we will get the optimal complete solution without reconsidering previous choices.
- Classical examples: Coin change in some currencies, interval coverage and load balancing.
- Greedy algorithms can be very useful as heuristics for example in branch-and-bound search algorithms.
- In combinatorics matroids and the generalization greedoids characterize classes of problems with greedy solutions.

Kruskal's MST Algorithm



Consider an undirected, weight graph

Kruskal's MST Algorithm

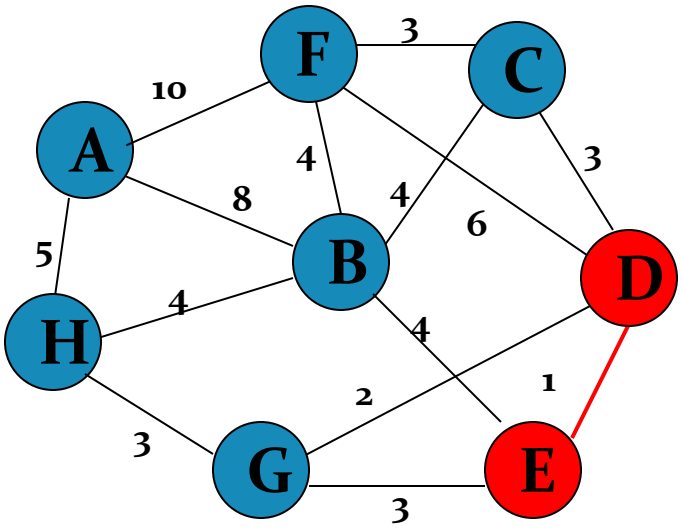


Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

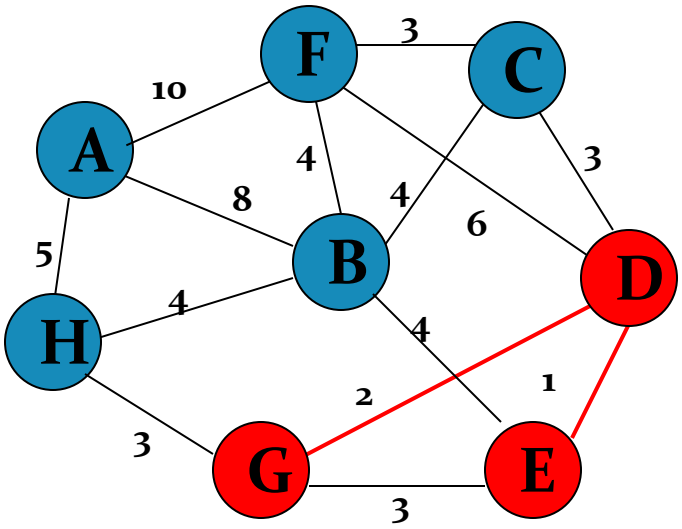


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

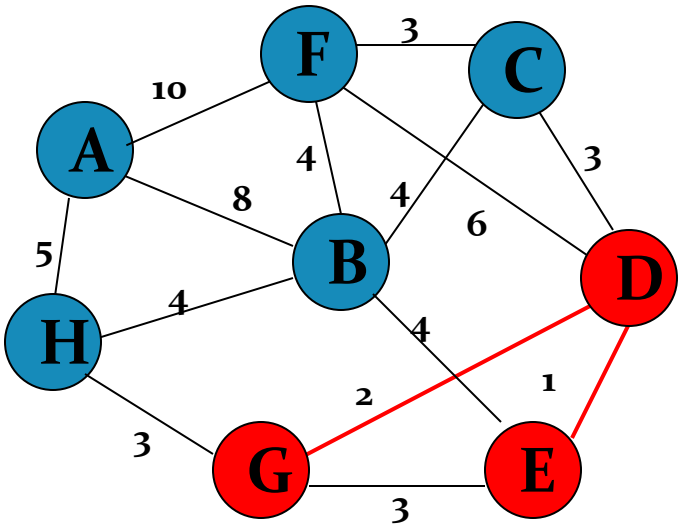


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm



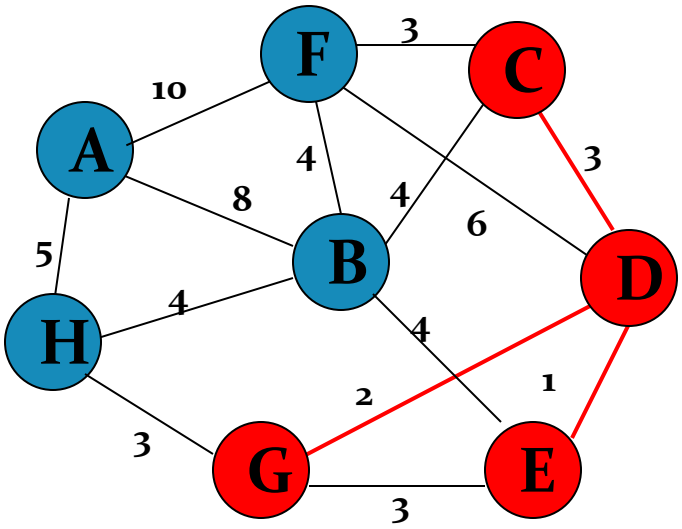
Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Accepting edge (E,G) would create a cycle

Kruskal's MST Algorithm

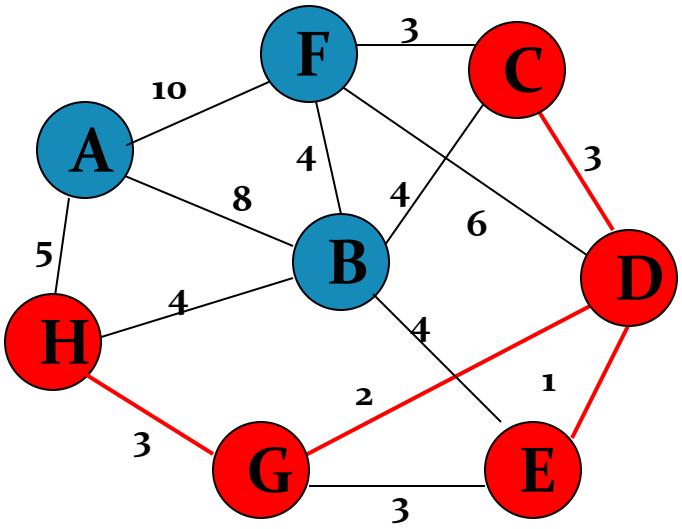


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

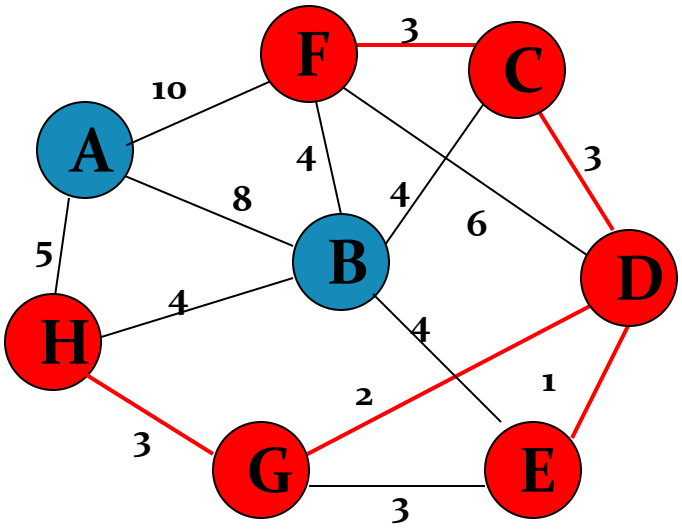


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

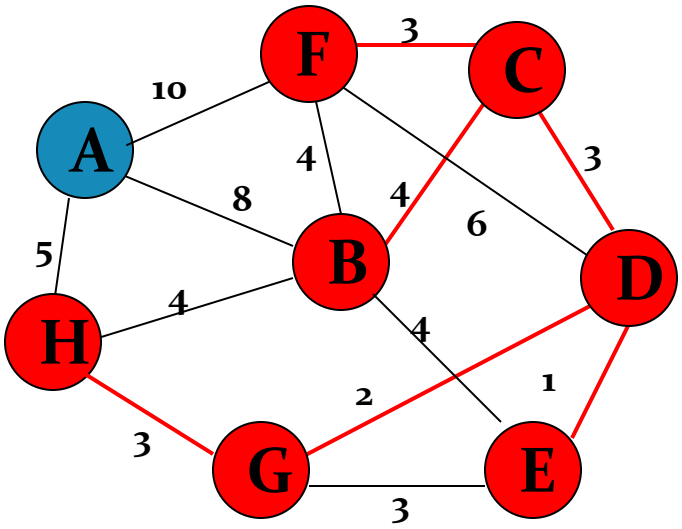


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

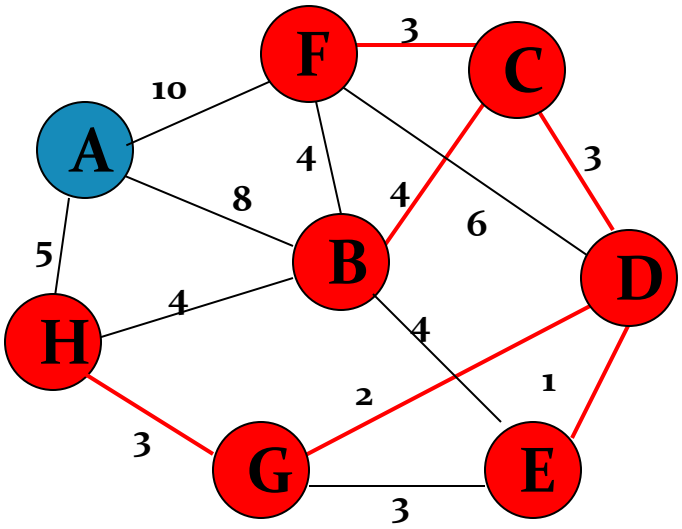


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

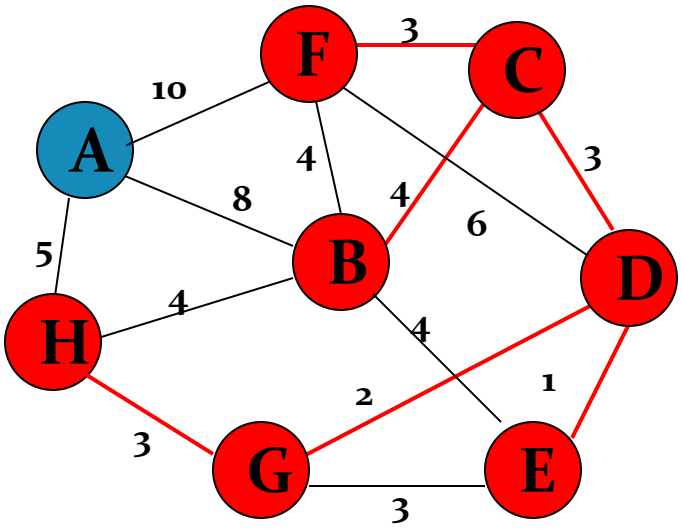


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

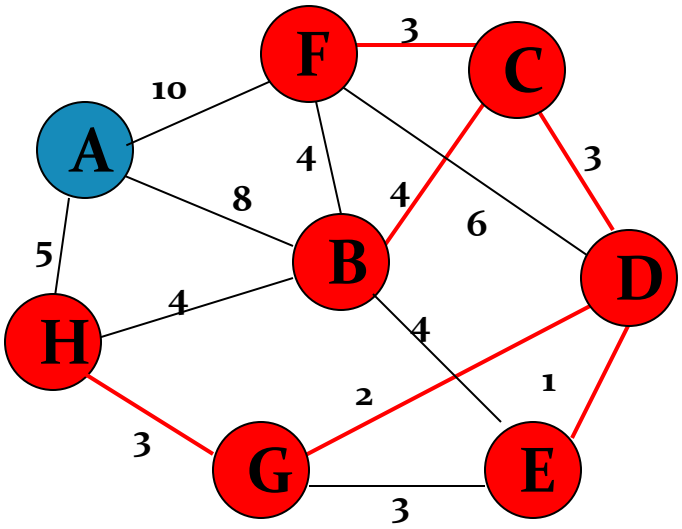


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

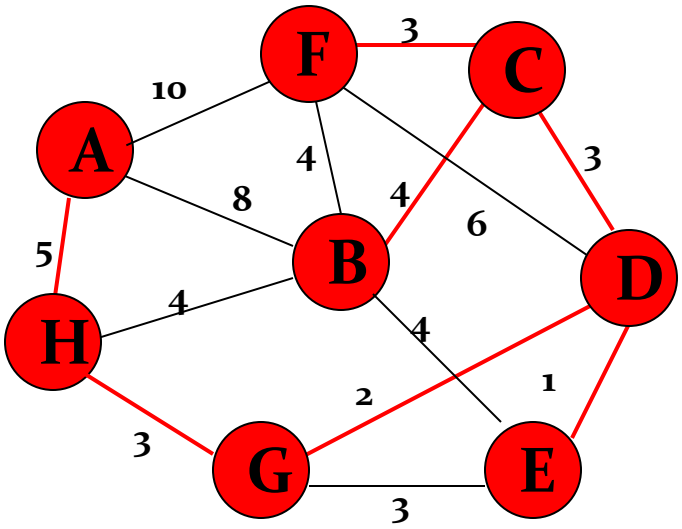


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm

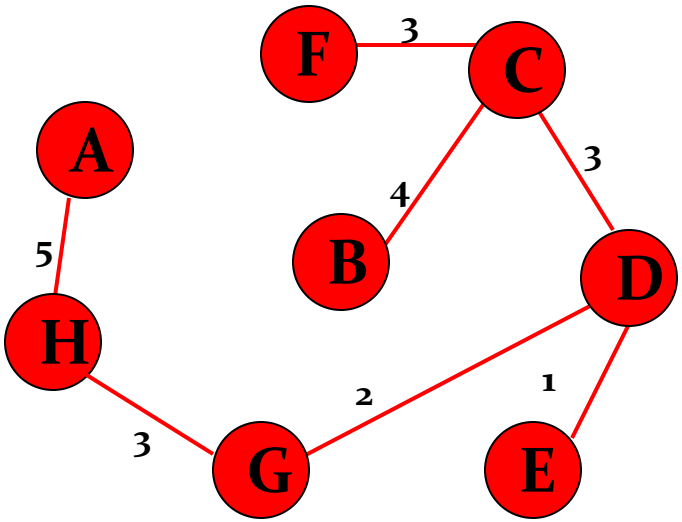


Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's MST Algorithm



Select first |V|−1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = $\Sigma\, d_v = 21$

Dynamic Programming

- Dynamic Programming is a problem-solving approach which computes the answer for every possible *state* exactly once.
- For DP to be suitable a problem must have two properties:
  - It has optimal sub-structures, i.e., an optimal solution contains the optimal solutions to sub problems.
  - Overlapping sub-problems, i.e. the same subproblem occurs many times.
- Top-down (memoization) vs Bottom-up
  - Top-down: no need to consider the order of computations, only compute states actually used, natural transition from complete search,
  - Bottom-up: no recursion, computes every state, table size can be reduced if only the previous row of states is used then only two rows are required.
- Displaying the optimal solution
  - Store the previous state for each solution
  - Use the DP table and the optimal sub-structures property to compute the path.

Subset Sum

Given:

- an integer bound W, and
- a collection of n items, each with a positive, integer weight w_i,

find a subset S of items that:

$$maximizes \sum_{i \in S} w_i \ while \ keeping \sum_{i \in S} w_i \leq W.$$

Motivation: You have a CPU with W free cycles and want to choose the set of jobs (each taking w_i time) that minimizes the number of idle cycles.

Subset Sum

Notation:

- Let S* be an optimal choice of items (e.g. a set {1,4,8}).
- Let *OPT(n, W)* be the value of the optimal solution.
- We design a dynamic programming algorithm to compute *OPT(n, W)*.

Subproblems:

- To compute OPT(n, W): We need the optimal value for subproblems consisting of the first *j* items for every knapsack size *0 <= w <= W*.
- Denote the optimal value of these subproblems by *OPT(j, w)*.

Subset Sum

Recurrence: How do we compute $OPT(j, w)$ in terms of solutions to smaller subproblems?

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

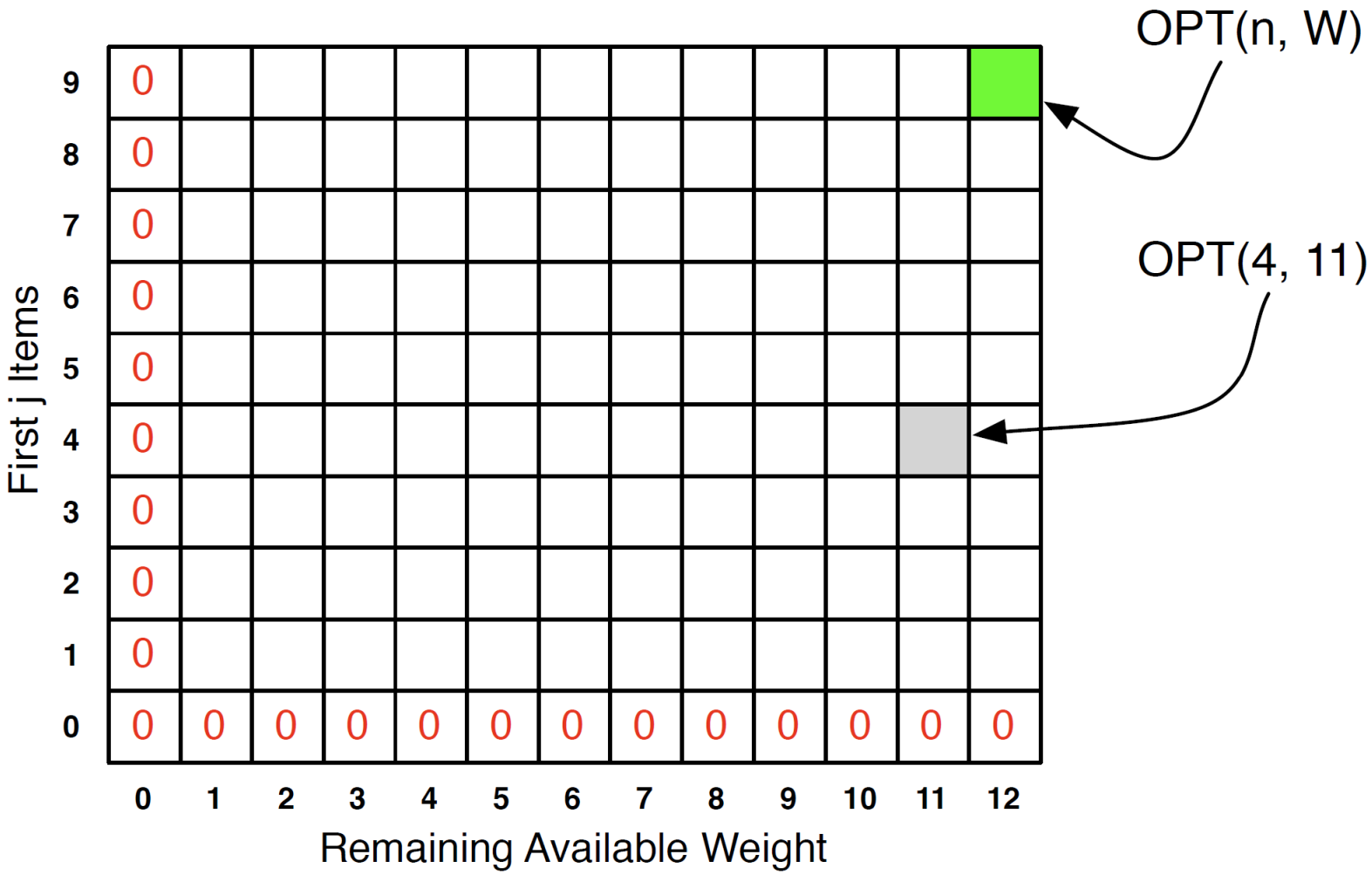
$$OPT(0, W) = 0 \qquad \text{If no items, 0}$$

$$OPT(j, 0) = 0 \qquad \text{If no space, 0}$$

Special case: if $w_j > W$ then $OPT(j, W) = OPT(j-1, W)$.

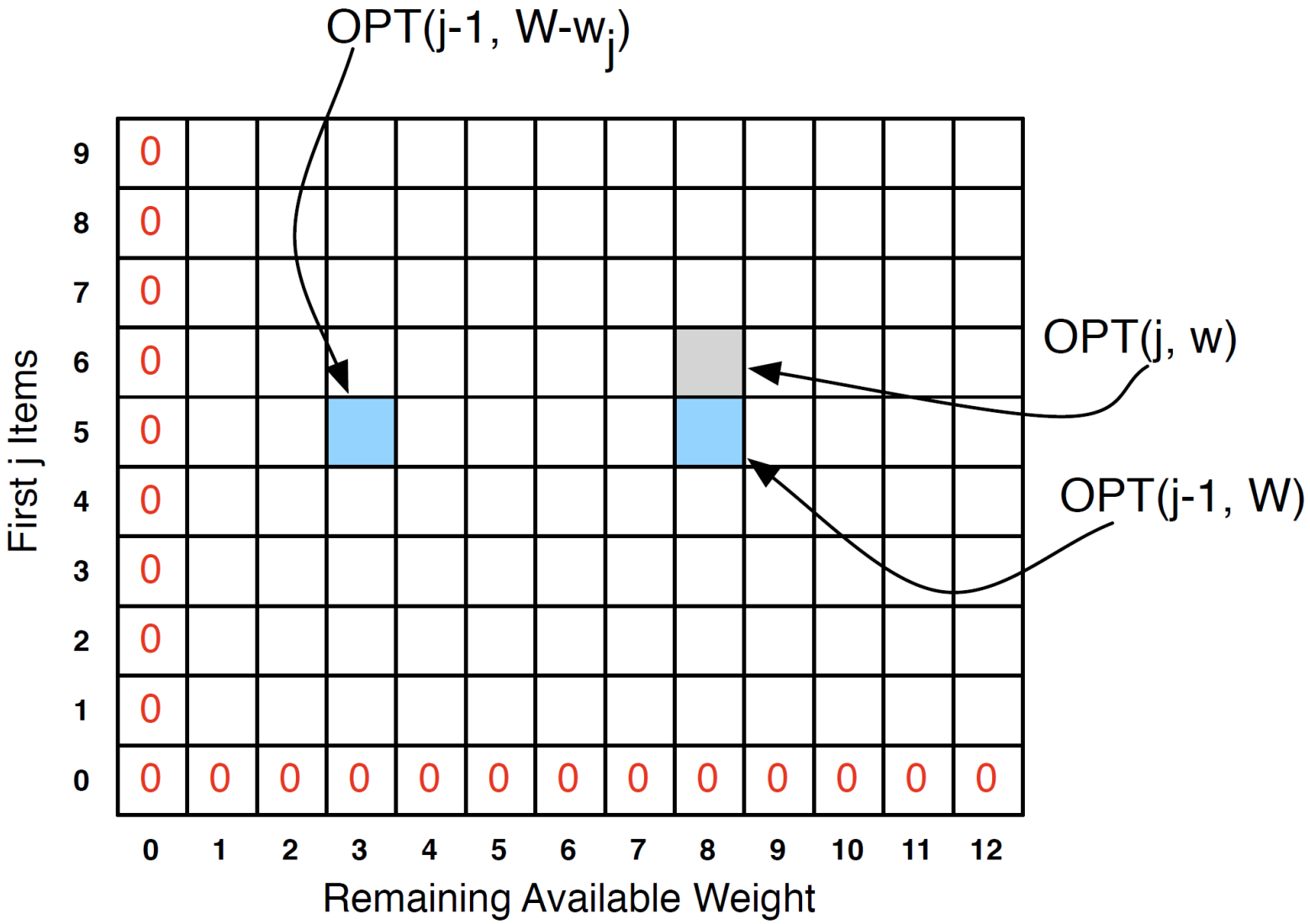
Subset Sum

The table of solutions



Subset Sum

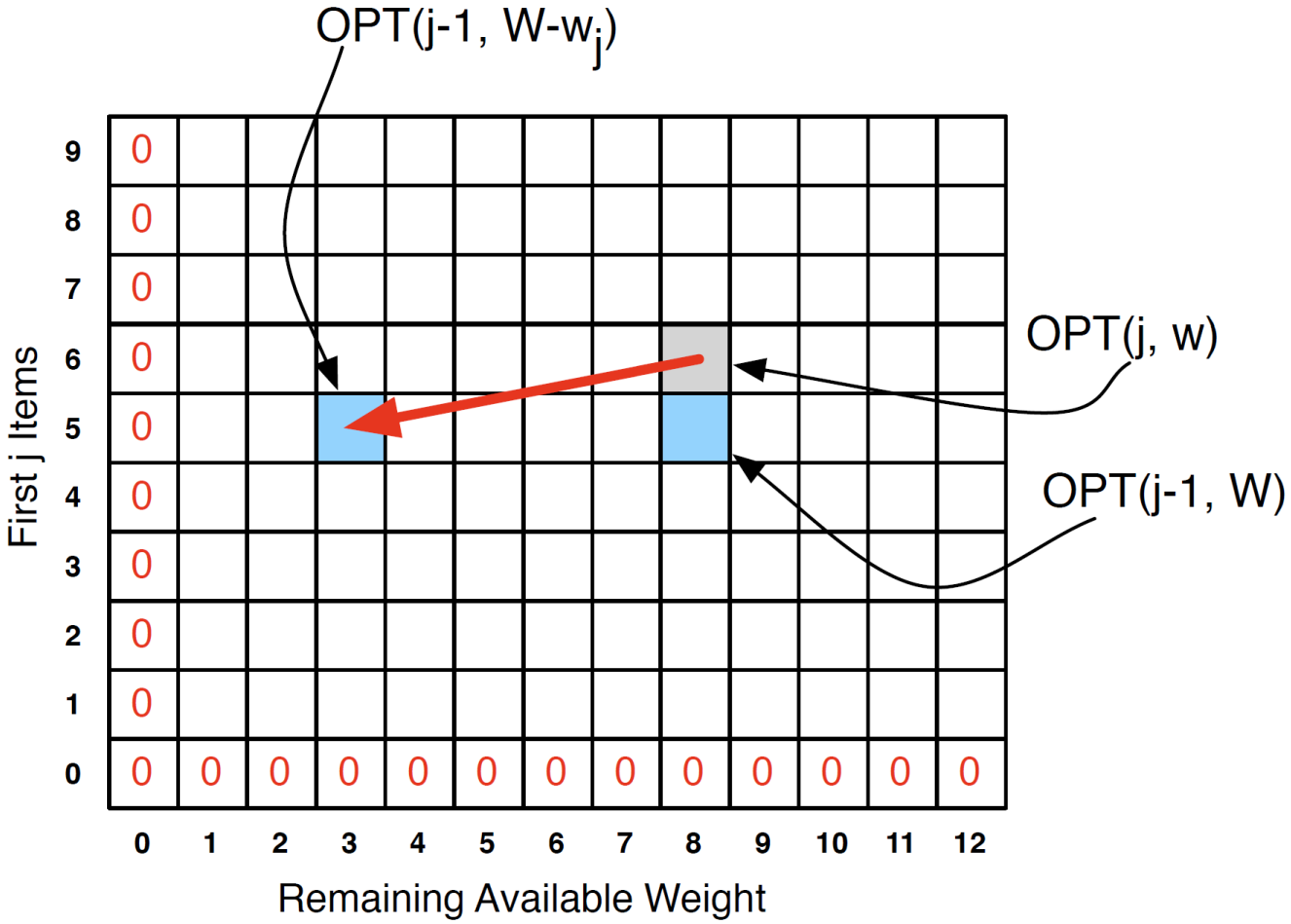
Filling in a box using smaller problems



Subset Sum

Remembering Which Subproblem Was Used

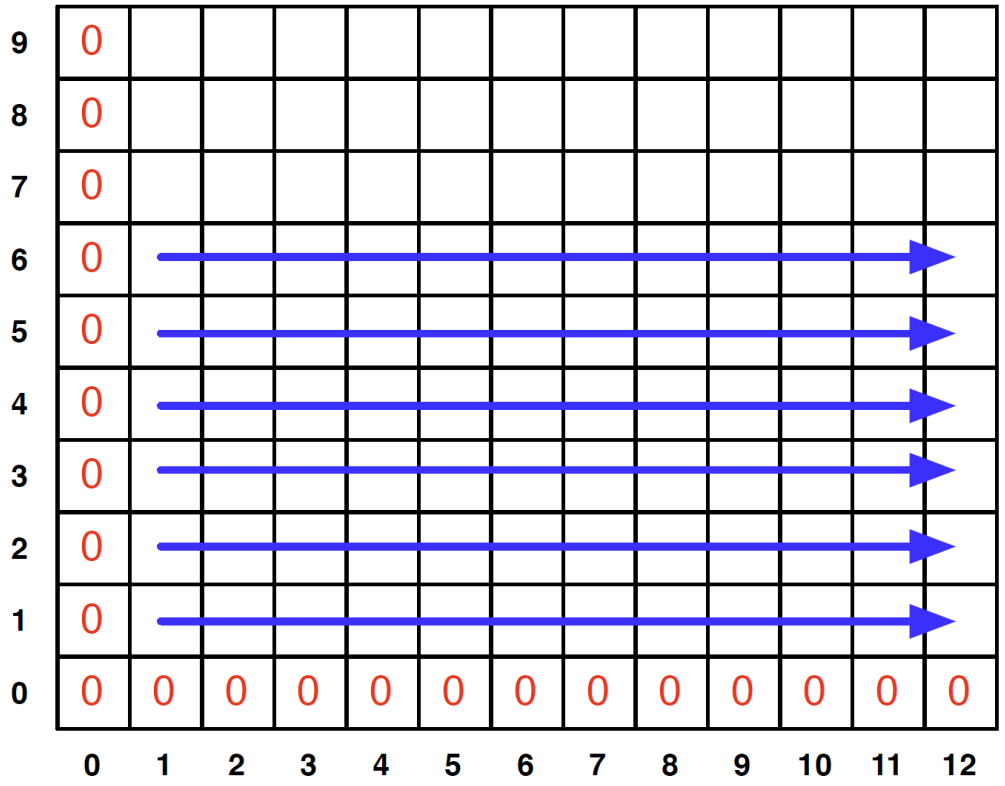
When we fill in the gray box, we also record which subproblem was chosen in the maximum:



Subset Sum

Filling in the Matrix

Fill matrix from bottom to top, left to right.



When you are filling in box, you only need to look at boxes you've already filled in.

Subset Sum

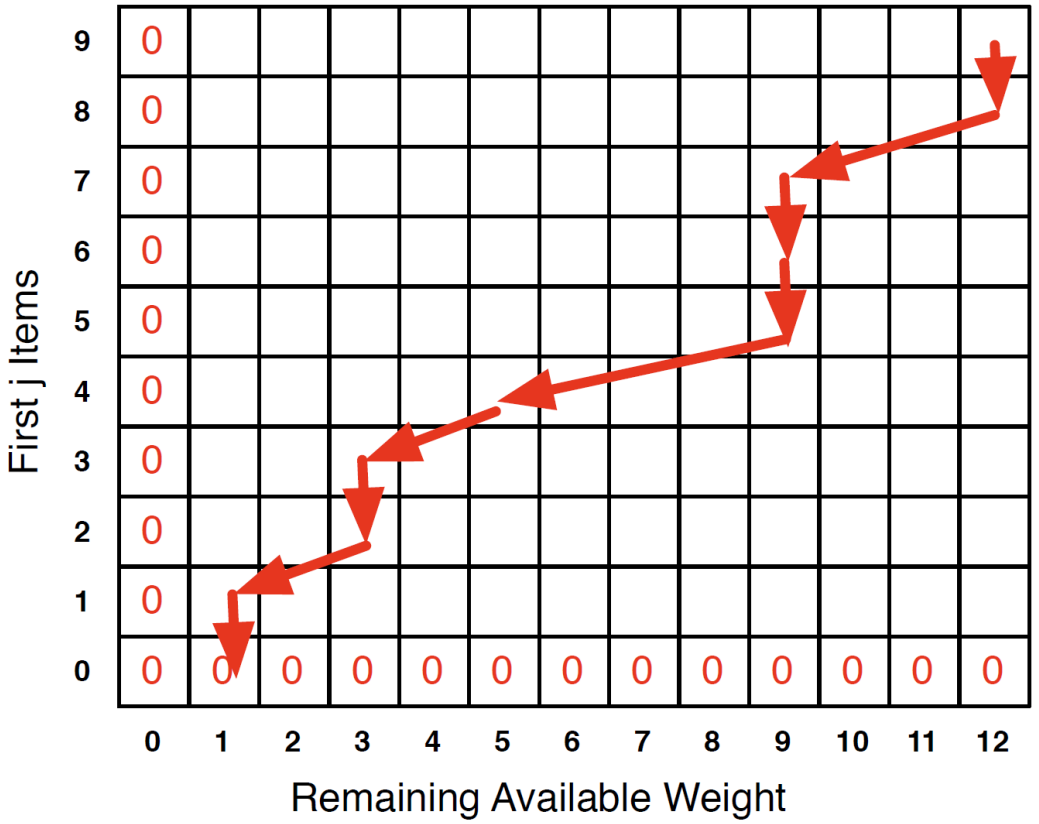

```
SubsetSum(n, W):
    Initialize M[0,r] = 0 for each r = 0,...,W
    Initialize M[j,0] = 0 for each j = 1,...,n

    for j = 1,...,n:                          for every row
        For r = 0,...,W:                      for every column
            If w[j] > r:                      case where item can't fit
                M[j,r] = M[j-1,r]
            M[j,r] = max(                     which is best?
                        M[j-1,r],
                        w[j] + M[j-1, W-w[j]]
                        )
    return M[n,W]
```

Subset Sum

Finding The Choice of Items

Follow the arrows backward starting at the top right:

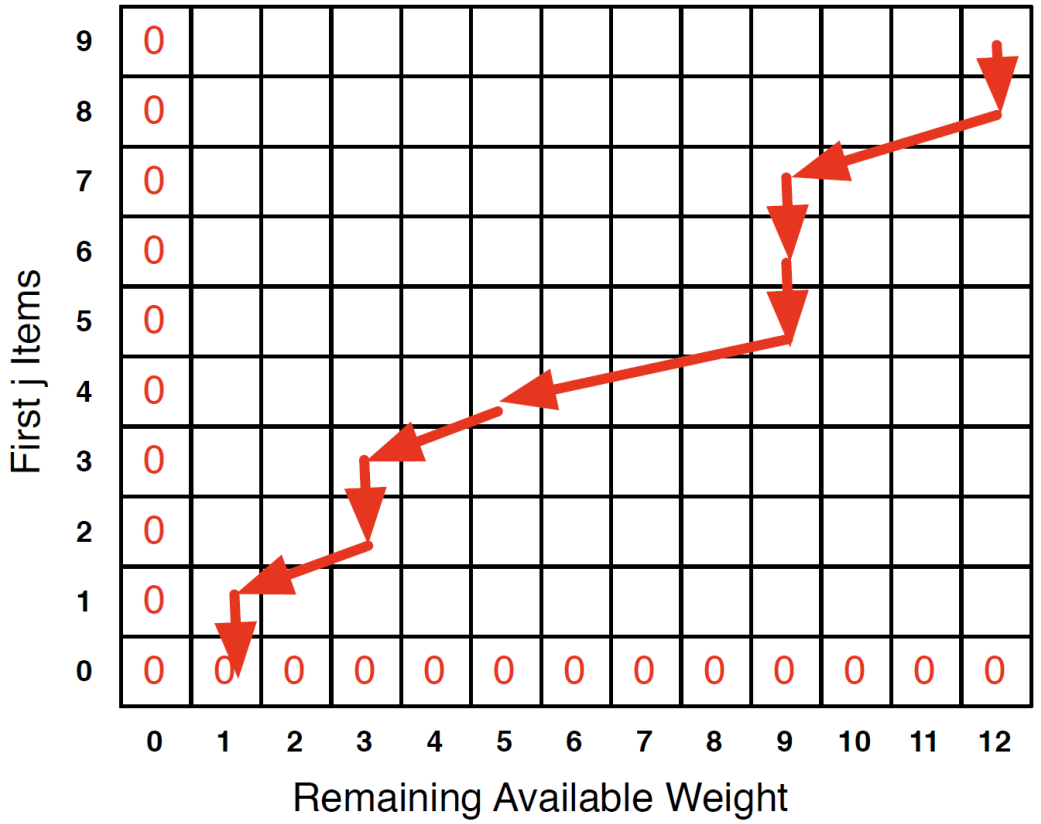


Which items does this path imply?

Subset Sum

Finding The Choice of Items

Follow the arrows backward starting at the top right:



Which items does this path imply? **8, 5, 4, 2**

General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems.

2. There are only a polynomial # of subproblems.

3. There is a "natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.

General DP Principles

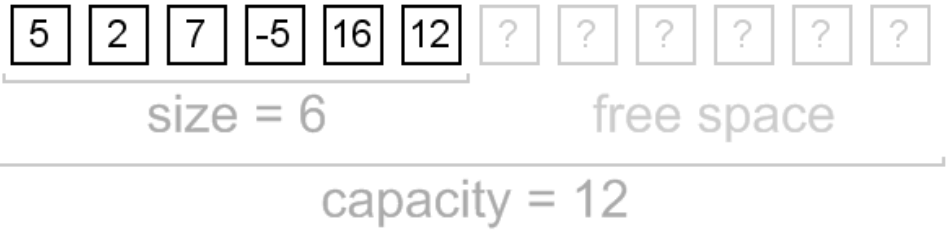
1. Optimal value of the original problem can be computed easily from some subproblems.
   OPT(j, w) = max of two subproblems
2. There are only a polynomial # of subproblems.
   {(j, w)} for j = 1, …, n and w = 0, …, W
3. There is a "natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.
   Considering items {1, 2, 3} is a smaller problem than considering items {1, 2, 3, 4}

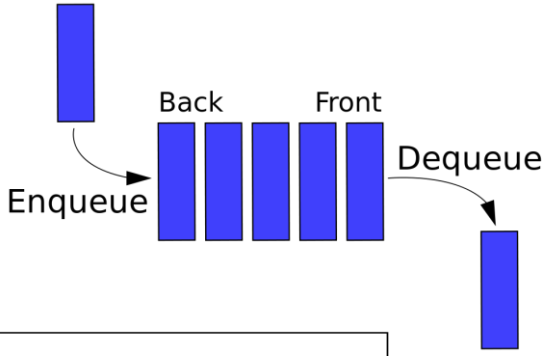
Classical DP Problems

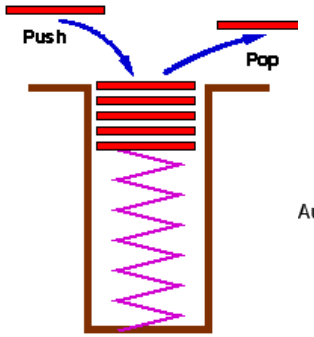
- Max 1D sum
- Max 2D sum
- Longest increasing subsequence (LIS)
  - Longest decreasing subsequence (LDS)
- 0-1 Knapsack (subset sum)
- Coin Change (general version)
- Travelling Salesman Problem (TSP)

	1D RSQ	2D RSQ	LIS	Knapsack	CoinChange	TSP
State	(i)	(i,j)	(i)	(id,remW)	(v)	(pos,mask)
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all j<i	take/ignore	all *n* coins	all *n* cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Common Data Structures

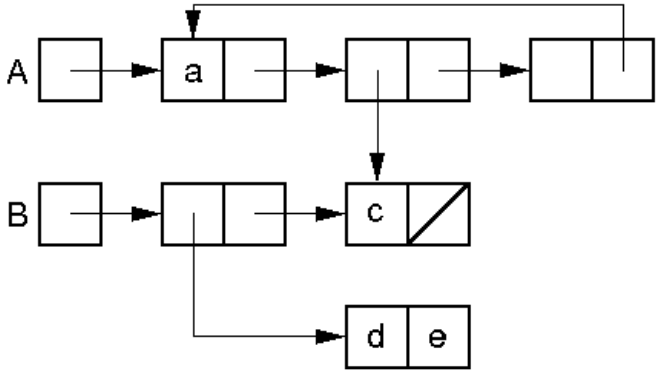


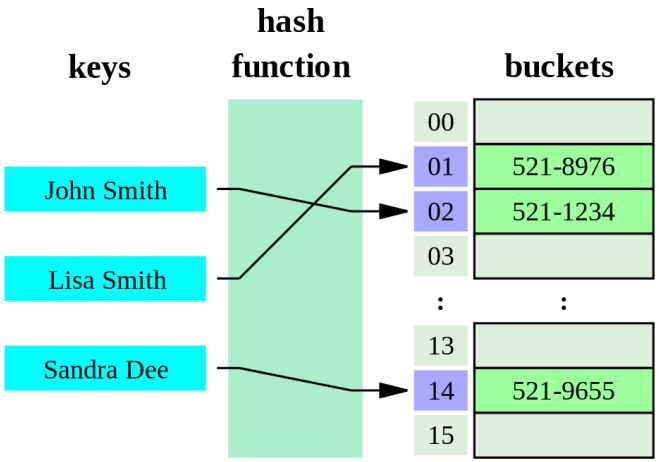


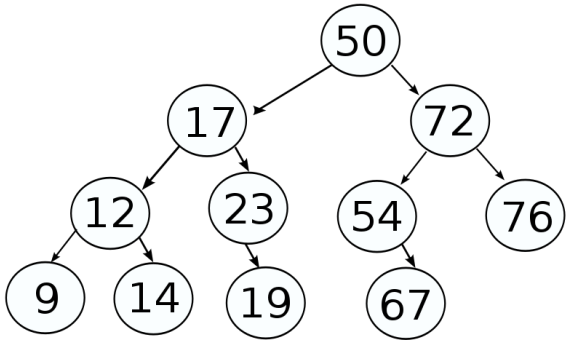


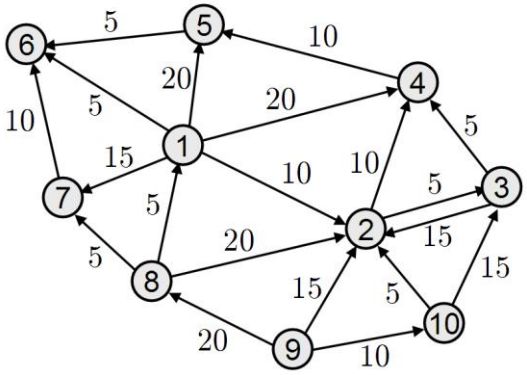
KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3









Important Data Structures and Algorithms

- Data structures
  - Standard library data structures
    - Vector, stack, queue, heap, priority queue, sets, maps
  - Other data structures
    - Graph (adjacency list and adjacency matrix), Union/find, Segment tree, Fenwick tree, Trie
- Sorting
  - Quick sort, Merge sort, Radix sort, Bucket sort
- Strings
  - String matching (Knuth Morris Pratt, Aho-Corasick), pattern matching, trie, suffix trees, suffix arrays, recursive decent parsing

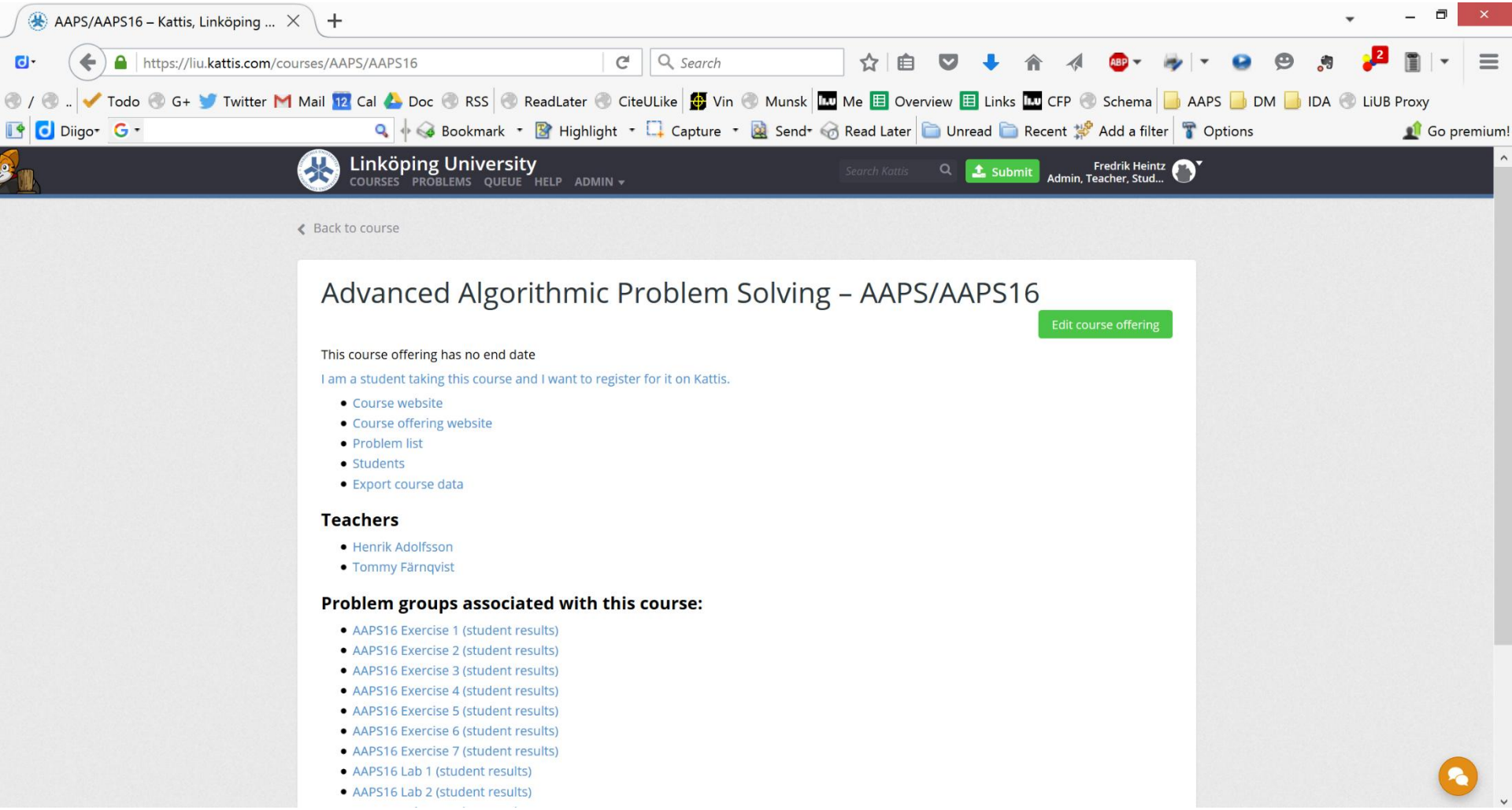
Important Data Structures and Algorithms

- Dynamic programming
  - Longest common subsequence, Longest increasing subsequence, 0/1 Knapsack, Coin Change, Matrix Chain Multiplication, Subset sum, Partitioning
- Graphs
  - Traversal (pre-, in- and post-order), finding cycles, finding connected components, finding articulation points, topological sort, flood fill, Euler cycle/Euler path, SSSP - Single source shortest path (Dijkstra, Bellman-Ford), APSP – All pairs shortest path (Floyd Warshall), transitive closure (Floyd Warshall), MST – Minimum spanning tree (Prim, Kruskal (using Union/find)), Maximal Bipartite Matching, Maximum flow, Maximum flow minimal cost, Minimal cut
- Search
  - Exhaustive search (depth-first, breadth-first search, backtracking), binary search (divide and conquer), greedy search (hill climbing), heuristic search (A*, branch and bound), search trees

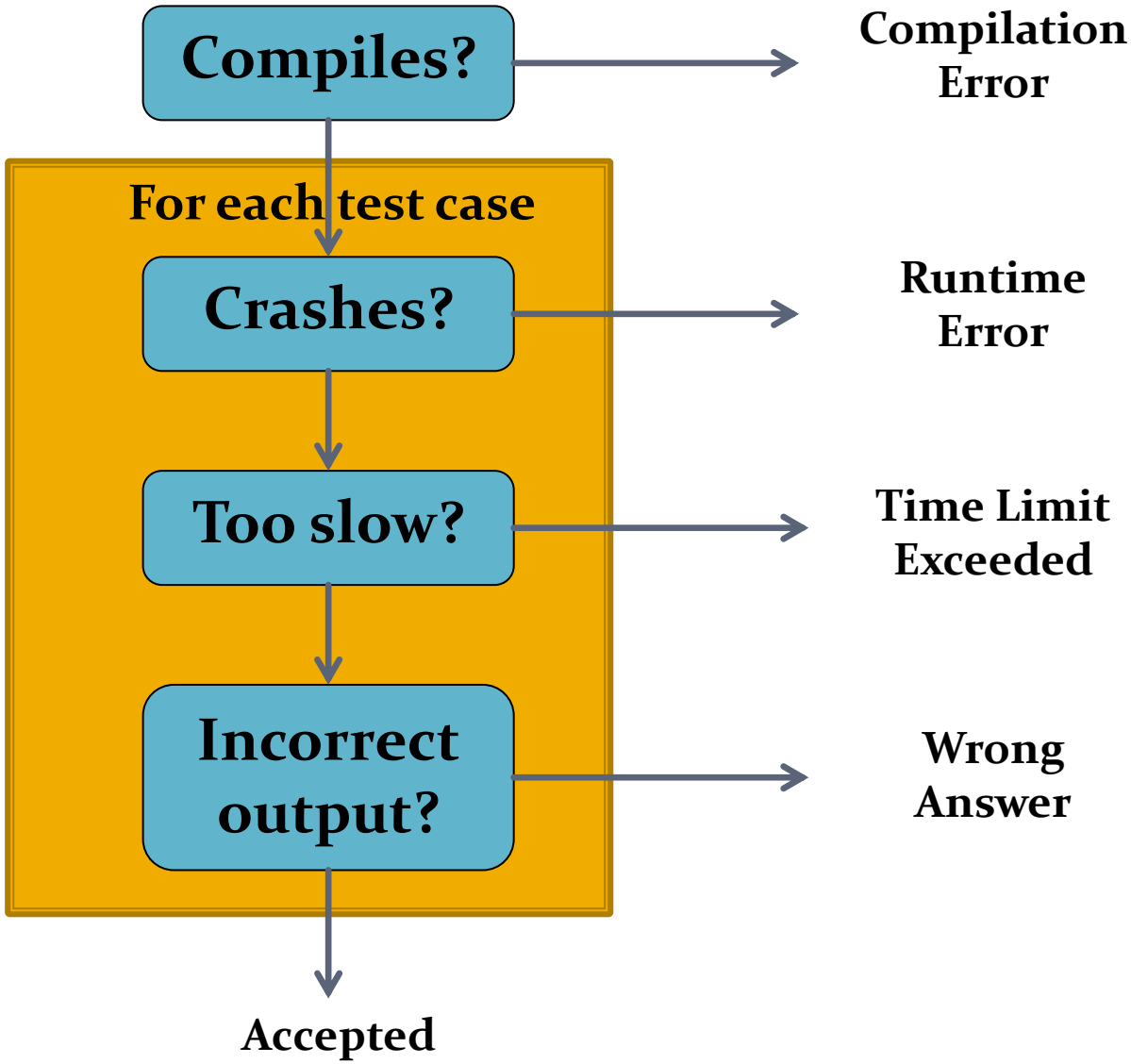
Important Data Structures and Algorithms

- Mathematics
  - Number theory (prime numbers, greatest common divisor (GCD), modulus), big integers, combinatorics (count permutations), number series (Fibonacci numbers, Catalan numbers, binomial coefficients), probabilities, linear algebra (matrix inversion, linear equations systems), finding roots to polynomial equations, diofantic equations, optimization (simplex)
- Computational geometry
  - Representations of points, lines, line segments, polygons, finding intersections, point localization, triangulation, Voronoi diagrams, area and volume calculations, convex hull (Graham scan), sweep line algorithms

Kattis (https://liu.kattis.com)



How Kattis checks a program



UVA Online Judge

UVa Online Judge

http://uva.onlinejudge.org/

Home

Google™ Custom Search | Search

Main Menu

- Home
- My Account
- Contact Us
- TOOLS on the Old UVa OJ Site
- ACM-ICPC Live Archive
- Logout

Online Judge

- Quick Submit
- Migrate submissions
- My Submissions
- My Statistics
- My uHunt with Virtual Contest Service
- Browse Problems

UVa OJ fundraising campaing

As you may already discovered by the widget shown on the left, we have started a fundraising campaing to create a whole new UVa Online Judge. Please, take a couple of minutes to read the reasons for this on the campaing website, by clicking on the widget.

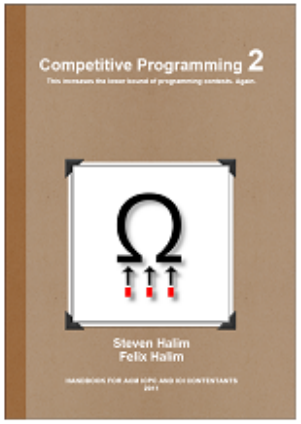
Welcome to the UVa Online Judge

Here you will find hundreds of problems. They are like the ones used during programming contests, and are available in HTML and PDF formats. You can submit your sources in a variety of languages, trying to solve any of the problems available in our database.

See the new Contest Rankings section at the Live Rankings link.

Now you can use the new Quick access, info and search option on the left menu for and easier navigation. (The tool will be updated next days

Categorized set of problems



This book contains a collection of relevant data structures, algorithms, and programming tips written for University students who want to be more competitive in the ACM International Collegiate Programming Contest (ICPC), high school students who are aspiring to be competitive in the International Olympiad in Informatics

Programming languages

- Preferred languages are C, C++, Java, and Python.
- C++ or Java is strongly recommended, use the language that you are most familiar with and want to learn more about.
- Get to know their standard libraries.
- Get to know input and output. Remember that I/O in Java is very slow, use Kattio. Remember that cout/cerr also is relatively slow, learn how to use scanf/printf if you use C++.
- Learn to use an appropriate IDE such as eclipse, emacs, or vim
- Create a problem template to speed up problem solving and to create a common format for your problems.


```
/* -*- Mode: C++ -*- */

/**
 * XXX NAME   C++   "Approach"
 *
 *    Started:
 *   Finished:
 * Total time:
 *
 * Submission 1:
 *
 * Comments:
 *
 * Lessons learned:
 */

#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <functional>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <map>
#include <set>
#include <queue>
#include <stack>
#include <string>
#include <utility>
#include <vector>

using namespace std;
typedef vector<int> vi;

int
main(int argc, char* argv[])
{

  return 0;
}
```

Testing and debugging

- Always create an example input (.in) and example output (.out) file with verbatim copies of the example input and output from the problem statement!
- For most problems it is enough to diff your output with the example output:
  ./prog < prog.in | diff - prog.out
- Create additional tests, such as:
  - Extreme inputs, i.e. smallest and largest values (0, 1, "", empty line, 2^31-1)
  - Small inputs that you can compute by hand
  - Potentially tricky cases such as when all inputs are equal, in the case of floating points numbers when you have to round both up and down
  - Very large cases, randomly generated to test that your program computes an answer fast enough (even though you might not know the correct answer).
- Use a correct but slow algorithm to compute answers.
- Print intermediate information, such as values of relevant variables.
  cout << "a=" << a << "; b=" << b << endl;
  Remember to remove the debug output before submitting! (or use cerr)

Summary

- What is algorithmic problem solving?
- Why is algorithmic problem solving important?
- What will be studied in this course?
- A method for algorithmic problem solving
- Common algorithmic problem-solving approaches
- Common data structures and algorithms
- Pragmatic algorithmic problem-solving using Kattis