Competitive Programming

Pascal Garcia

INSA de Rennes

October 13, 2014

1. Introduction
   - Competitive Programming
   - Tips to be competitive
   - Ressources
   - Exercices
2. Development Environment
   - Editor
   - Typical Input/Output Routines
   - Testing
   - Debugging
3. Proof Techniques
   - Proof by Contradiction
   - Pigeonhole Principle
   - Proof by Induction
4. Complexity
   - Asymptotic Order of Growth
   - Common Running Times
5. How to Solve It
   - Four Steps Plan
   - Understand the Problem
   - Devising a Plan
   - Executing the Plan

- Looking Back
- Exercices

6. Data Structures and Libraries
   - Linear Data Structures
   - Non-Linear Data Structures
7. Problem Solving Tools
   - Design of Algorithms by Induction
   - Invariants
   - Reduction
   - Complete Search
   - Binary Search
   - Greedy
   - Divide and Conquer
   - Dynamic Programming
8. Specific Domains
   - Graph
   - Mathematics
   - String Processing
   - Computational Geometry
   - Simplex
9. Harder problems

Preface

This course is based on several books and online ressources (listed from slides 27 to 29). I used extracts from those different sources in this presentation.

The main ressource I used is the book:

Competitive Programming 3 by Felix Halim and Steven Halim

which I highly recommand.

What is Competitive Programming?

Definition

Given known computer science problems, solve them as quickly as possible!

A problem usually contains the following components:

- Background story/problem description.
- Input and output description.
- Sample inputs and outputs.
- Hints (sometimes).

Why Studying Competitive Programming?

Allow you to improve your skills for:

- Estimating the complexity of a given task.
- Choosing the correct algorithms and data structures (given the complexity of the task at hand).
- Testing and debugging your program efficiently.

Moreover, each problem you solve gives you a model for similar problems. With practice, you will get a kind of "problem pattern database" in your head allowing you to **relate new problems to already solved ones**.

Bonus: help you for interviews in big software companies (Google, Facebook, Microsoft, . . .).

Big bonus: it's fun!

Problem Example

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Problem description:
Let (x, y) be the coordinates of a student's house on a 2D plane. There are $2N$ students and we want to **pair** them into N groups.
Let d_i be the distance between the houses of 2 students in group i. Form N groups such that $cost = \sum_{i=1}^{N} d_i$ is **minimized**. Output the minimum $cost$.
Constraints: $1 \leq N \leq 8$ and $0 \leq x, y \leq 1000$.

Sample input:
$N = 2$; coordinates of the $2N = 4$ houses are: $(1, 1)$, $(8, 6)$, $(6, 8)$, and $(1, 3)$.

Sample output:
$cost = 4.83$.

Type of Programmers

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Uncompetitive programmer A (a.k.a. the blurry one):

Step 1: Reads the problem and become confused (this problem is new for him).

Step 2: Tries to code something: Reading the non-trivial input and output.

Step 3: Realizes that all his attempts are not **Accepted** (**AC**):

- **Greedy**: Repeatedly pairing the two remaining students with the shortest separating distances gives the **Wrong Answer** (**WA**).
- Naïve **Complete Search**: Using recursive backtracking and trying all possible pairings yields **Time Limit Exceeded** (**TLE**)

Type of Programmers

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Uncompetitive programmer B (give up):

Step 1: Reads the problem and realizes that he has seen this problem before. But also remembers that he has not learned how to solve this kind of problem...
He is not aware of the **Dynamic Programming** (**DP**) solution.

Step 2: Skips the problem and reads another problem in the problem set.

Type of Programmers

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Uncompetitive programmer C (slow):

Step 1: Reads the problem and realizes that it is a difficult problem: **"minimum weight perfect matching on a small general weighted graph"**. However, since the input size is small, this problem is solvable using DP. The DP state is a **bitmask** that describes a matching status, and matching unmatched students i and j will turn on two bits i and j in the bitmask.

Step 2: Codes I/O routines, write recursive top-down DP, tests, debugs (only using printf).

Step 3: After three hours, his solution obtains AC (passed all the secret test data).

Type of Programmers

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Competitive programmer D:

Completes all the steps taken by uncompetitive programmer C in ≤ 30 minutes.

Type of Programmers

UVa Online Judge (http://uva.onlinejudge.org) problem number 10911

Competitive programmer E:

A very competitive programmer (e.g. the red coders in TopCoder) would solve this "well known" problem in ≤ 15 minutes.

Solution

```
//includes ommitted
int N, target;
double dist[20][20], memo[1 << 16];
double matching(int bitmask)
{
 if (memo[bitmask] > -0.5) return memo[bitmask];
 if (bitmask == target) return memo[bitmask] = 0;
 double ans = 2000000000.0;
 int p1, p2;
 for (p1 = 0; p1 < 2 * N - 1; p1++)
  if (!(bitmask & (1 << p1))) break;

 for (p2 = p1 + 1; p2 < 2 * N; p2++)
 {
  if (!(bitmask & (1 << p2)))
   ans = min(ans,
             dist[p1][p2] + matching(bitmask | (1 << p1) | (1 << p2)));
 }
 return memo[bitmask] = ans;
}
```

Solution

```
int N, target;
double dist[20][20], memo[1 << 16];
double matching(int bitmask) { ... }
int main()
{
 int i, j, caseNo = 1, x[20], y[20];
 while (scanf("%d", &N), N)
 {
  for (i = 0; i < 2 * N; i++)
   scanf("%*s␣%d␣%d", &x[i], &y[i]);
  for (i = 0; i < 2 * N - 1; i++)
   for (j = i + 1; j < 2 * N; j++)
    dist[i][j] = dist[j][i] = hypot(x[i] - x[j], y[i] - y[j]);

  for (i = 0; i < (1 << 16); i++) memo[i] = -1.0;
  target = (1 << (2 * N)) - 1;
  printf("Case␣%d:␣%.2lf\n", caseNo++, matching(0));
 }
 return 0;
}
```

Exercice 1

The greedy approach of the uncompetitive programmer A does not work. Find a counter example.

Exercice 2

Analyze the time complexity of the naïve complete search solution by uncompetitive programmer A to understand why it receives the TLE verdict.

Six Tips to Be More Competitive

1. Code faster!
2. Quickly identify problem types.
3. Do algorithm analysis.
4. Master programming languages.
5. Master the art of testing code and debugging.
6. Practice and more practice.

We will discuss briefly those point on the next slides, but we will cover them in more depth (except the first point) during the course.

Code Faster!

- Use **auto-completion** and **code templates**
- Type code faster :-). Try this typing test:

  `http://www.typingtest.com`

  and read the instructions there on how to improve your typing skill.

Quickly Identify Problem Types

We can categorize the problems types as follows (we will discuss about this categories in the remaining of the course):

1. Ad hoc.
2. Complete search.
3. Binary search.
4. Divide and conquer.
5. Greedy.
6. Dynamic programming.
7. Graph.
8. Mathematics.
9. String processing.
10. Computational geometry.
11. Simplex.

Note that some problems belongs to more than one category and that the use of data structure is compulsory in your program.

When you have identified the problem type, you can try to remember the most similar problem of this type you have already solved.

The more problem you know about each type, the faster you will solve new problems.

Do Algorithm Analysis

You need to carrefully understand the constraints of the problem and the complexity of the algorithm you want to use to solve it.

For example, it can be enough to use a complete search solution if the constraints are easy. Whereas, if the constraints are hard, you have to use a better algorithm.

Master Programming Languages

Generally, several languages are supported in programming contest, including C/C++ and Java. Which programming language you choose depends on your taste, but some problems can be easier to solve depending and your language and the librairies it provides. I will use essentially `C++` in this course, but you can choose `Java` (the language has to be supported by *Uva Online Judge*).

The following problem is easier to solve with `Java` than `C++`:

Problem description:
Suppose that a problem requires you to compute 25!. The answer is very large: $15,511,210,043,330,985,984,000,000$. This far exceeds the largest built-in primitive integer data type (`unsigned long long`: $2^{64}-1$ in `C++`). As there is no built-in arbitrary-precision arithmetic library in `C/C++` as of yet, we would have needed to implement one from scratch (more realistically copy/paste one from the web :-)). The `Java` code, however, is exceedingly simple (solution on next slide).

Solution

```
import java.util.Scanner;
import java.math.BigInteger;

class MainFactorial
{
 public static void main(String[] args)
 {
  BigInteger fac = BigInteger.ONE;
  for (int i = 2; i <= 25; i++)
   fac = fac.multiply(BigInteger.valueOf(i));
  System.out.println(fac);
 }
}
```

Exercice 3

Generate all possible subsets of $\{0, 1, 2, \ldots, N-1\}$, for $N = 10$. Each subset will be printed on a different line with each number of the subset separated by a blank.

Exercice 4

Write a function that takes three integer parameters representing respectively the day, the month and the year, and returns one of the string: "Monday", . . . , "Sunday". The string represents the day corresponding to the date.

Master the Art of Testing Code and Debugging

To avoid losing points during a competition, you need to submit a correct program. To do so, You will have to:

1. Design some comprehensive and tricky test cases. The sample input-output given in the problem description is by nature trivial and therefore usually not a good means for determining the correctness of your code.
2. Debug your program using a debugger (avoid using `printf`).

Practice and More Practice

Practice problems and **learn** from problems' solutions.

Online Ressources

Competitive programming site:

- Uva Online Judge: `http://uva.onlinejudge.org`
- uHunt: `http://uhunt.felix-halim.net`
- UVA toolkit: `http://uvatoolkit.com`
- TopCoder: `www.topcoder.com`
- Codeforces: `http://www.codeforces.com`
- CodeChef: `http://www.codechef.com`
- Google Code Jam: `http://code.google.com/codejam`

Algorithm courses:

- Algorithms, Part I, Princeton University: `https://www.coursera.org`
- Algorithms, Part II, Princeton University: `https://www.coursera.org`

Books

- Competitive Programming 3. Steven Alim and Felix Alim.
- Programming Challenges. The Programming Contest Manual. Steven S. Skiena and Miguel A. Revilla.
- Introduction to Algorithms: A Creative Approach. Udi Manber.
- Algorithms and Programming. Alexander Shen.
- Algorithm Design. Jon Kleinberg and Éva Tardos.
- Introductory Combinatorics. Richard A. Brualdi.
- Computational Geometry in C. Joseph O'Rourke.
- Data Structures and Network Algorithms. Robert E. Tarjan.
- Data Structures and Algorithms in C++. Michael T. Goodrich, Roberto Tamassia and David Mount.
- The C++ Standard Library. Tutorial and Reference. N. Josuttis.
- Hacker's Delight. Henry S. Warren.

Books

- How to Solve It. G. Polya.
- How to Prove It: A Structured Approach. Daniel J. Velleman.
- Principles of Mathematical Problem Solving. Martin J. Erickson and Joe Flowers.
- The Art and Craft of Problem Solving. Paul Zeitz.
- How to Think Like a Mathematician. A Companion to Undergraduate Mathematics. Kevin Houston.
- Discrete Mathematics with Applications. Susanna S. Epp.
- The Moscow Puzzles. Boris A. Kordemsky.
- Fifty Challenging Problems in Probability with Solutions. Frederick Mosteller.
- Challenging Problems in Algebra. Alfred S. Posamentier and Charles T. Salkind.
- Challenging Problems in Geometry. Alfred S. Posamentier and Charles T. Salkind.
- The Art of Debugging with GDB, DDD, and Eclipse. Norman Matloff and Peter Jay Salzman.
- Debugging. The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems.

Exercice 5

Try to solve the Ad Hoc problem:

UVa Online Judge (http://uva.onlinejudge.org) problem number 195

Exercice 6

Try to solve the Ad Hoc problem:

UVa Online Judge (http://uva.onlinejudge.org) problem number 608

Other problems:

UVa Online Judge (http://uva.onlinejudge.org) problem number 10189
UVa Online Judge (http://uva.onlinejudge.org) problem number 12239
UVa Online Judge (http://uva.onlinejudge.org) problem number 12060

Editor

To be more efficient, use an editor with:

- **Auto-completion**.
- **Code template**.
- **Integrated debugger**.

Use your editor of choice. I use *Emacs* with the configuration given in the following slides.

Auto-complete

Auto-complete is an auto-completion extension for *Emacs*:

`http://cx4a.org/software/auto-complete/`

1. Create a directory, for example: `~/.emacs.d/auto-complete`
2. Extract the source in this directory
3. Add to your `.emacs`:

```
;; add the new directory to search path
(add-to-list 'load-path "~/.emacs.d/auto-complete")
(require 'auto-complete-config) ;; load the package
;; load support for c++
(require 'auto-complete-clang)
(add-to-list 'ac-dictionary-directories "~/.emacs.d/auto-complete/
dict") ;; add directory for dictionaries
(ac-config-default)
;; to pop up quickly help for c++
(setq ac-quick-help-delay 0)
;; to fix bad interaction with auto-complete and yasnippet
(setq ac-source-yasnippet nil)
;; short-cut to launch c++ completion
(global-set-key (kbd "M-z") 'ac-complete-clang)
```

Code Template

I use *Yasnippet* a code template extension for *Emacs*:

`https://github.com/capitaomorte/yasnippet`

1. Create a directory, for example: `~/.emacs.d/yasnippet`
2. Extract the source in this directory
3. Add to your `.emacs`:

```
;; add the new directory to search path
(setq load-path (cons "~/.emacs.d/yasnippet/" load-path))
;; directories of templates
(setq yas-snippet-dirs "~/.emacs.d/yasnippet/snippets")
;; load package
(require 'yasnippet)
;; to use in all buffers
(yas-global-mode 1)
```

Code Template: Snippet

The directory `~/.emacs.d/yasnippet/snippets` contains the code templates.

For example, in the `~/.emacs.d/yasnippet/snippets/cc-mode` which contains the code templates common to *C++* and *C*, I have defined:

```
# name: if (...) { ... } else { ... }
# key: ife
# --
if (${1:condition})
{
  $2
}
else
{
  $0
}
```

Code Template: Snippet

```
1   # name: if (...) { ... } else { ... }
2   # key: ife
3   # --
4   if (${1:condition})
5   {
6     $2
7   }
8   else
9   {
10    $0
11  }
```

- **key**: (`ife` here) is the abbreviation you type to expand a snippet just before hitting the key that runs `yas-expand` (bounded to the `TAB key`.
- **name**: This is a one-line description of the snippet. It will be displayed in the menu.
- **Line 4 through 11**: The template to be expanded.
- **Tab stops**: They are fields that you can navigate back and forth by `TAB` and `S-TAB`. They are written by `$` followed with a number. `$0` has the special meaning of the exit point of a snippet. That is the last place to go when you've traveled all the fields.

Debugger Integration

You can compile the current buffer with:

```
<ESC>x compile (or M-x compile)
```

To change the default compile command (`make -k`), add to your `.emacs`:

```
;; load compile package
(require 'compile)
;; to change the compilation command for c++ buffer
(add-hook 'c++-mode-hook
    (lambda ()
      (unless (or (file-exists-p "makefile")
      (file-exists-p "Makefile"))
        (set (make-local-variable 'compile-command)
       (concat "g++␣-std=c++11␣-g␣" ;; -g for debugging
         (buffer-file-name))))))
```

To launch gdb from the current buffer:

```
<ESC>x gdb (or M-x gdb)
```

followed by:

```
<ESC>x gdb-many-windows (or M-x gdb-many-windows)
```

Usefull Configurations and Commands

- **Switching between buffer**. `iswitchb` is an extansion to navigate more easily between buffers (`C-x b`). Add to your `.emacs`:

```
;; activate iswitchb
(iswitchb-mode 1)

;; prevent switching to another frame
(setq iswitchb-default-method 'samewindow)

;; using the arrow keys to select a buffer
(defun iswitchb-local-keys ()
      (mapc (lambda (K)
        (let* ((key (car K)) (fun (cdr K)))
              (define-key iswitchb-mode-map (edmacro-parse-keys key
) fun)))
      '(("<right>" . iswitchb-next-match)
        ("<left>"  . iswitchb-prev-match)
        ("<up>"    . ignore             )
        ("<down>"  . ignore             ))))
(add-hook 'iswitchb-define-mode-map-hook 'iswitchb-local-keys)
```

Usefull Configurations and Commands

- **Moving between windows**. To navigate between windows add the following to your `.emacs`:

```
;; move to left window
(global-set-key [M-C-left] 'windmove-left)
;; move to right window
(global-set-key [M-C-right] 'windmove-right)
;; move to upper window
(global-set-key [M-C-up] 'windmove-up)
;; move to downer window
(global-set-key [M-C-down] 'windmove-down)
```

- **Moving inside buffer**. To jump part of text, add to your `.emacs`:

```
(global-set-key [C-right] 'forward-word)
(global-set-key [C-left] 'backward-word)
(global-set-key [M-left] 'beginning-of-line)
(global-set-key [M-right] 'end-of-line)
(global-set-key [home] 'beginning-of-line)
(global-set-key [end] 'end-of-line)
```

Usefull Configurations and Commands

- **Add line number to all buffers**. Add to your `.emacs`:

```
(global-linum-mode 1)
```

- **Add a theme**. Try some theme with `M-x load-theme` (use `TAB` after to get a list of theme). Add your choosen theme in your `.emacs`:

```
(load-theme 'whiteboard t)
```

- **Set default directory**. When you open a file, you can select the default directory. Add the following to your `.emacs`:

```
(setq default-directory "~/")
```

- **Show matching parenthesis**. Add the following to your `.emacs`:

```
(show-paren-mode 1)
```

Usefull Configurations and Commands

- **Deletion when copying on selection**. Add the following to your `.emacs`:

```
(delete-selection-mode t)
```

- **Switch between frame**. Add the following to your `.emacs`:

```
(global-set-key (kbd "M-a") 'other-frame)
```

- **Set a box shape for the cursor**. Add the following to your `.emacs`:

```
(set-default 'cursor-type 'box)
```

.emacs Recap

```
;; add yasnippet BEFORE auto-complete
(setq load-path (cons "~/.emacs.d/yasnippet/" load-path))
(setq yas-snippet-dirs "~/.emacs.d/yasnippet/snippets")
(require 'yasnippet)
(yas-global-mode 1)

(add-to-list 'load-path "~/.emacs.d/auto-complete")
(require 'auto-complete-config)
(require 'auto-complete-clang)
(add-to-list 'ac-dictionary-directories "~/.emacs.d/auto-complete/dict")
(ac-config-default)
(setq ac-quick-help-delay 0)
(setq ac-source-yasnippet nil)
(global-set-key (kbd "M-z") 'ac-complete-clang)

(global-set-key [M-C-left] 'windmove-left)
(global-set-key [M-C-right] 'windmove-right)
(global-set-key [M-C-up] 'windmove-up)
(global-set-key [M-C-down] 'windmove-down)
```

.emacs Recap

```
(global-set-key [C-right] 'forward-word)
(global-set-key [C-left] 'backward-word)
(global-set-key [M-left] 'beginning-of-line)
(global-set-key [M-right] 'end-of-line)
(global-set-key [home] 'beginning-of-line)
(global-set-key [end] 'end-of-line)

(iswitchb-mode 1)
(setq iswitchb-default-method 'samewindow)
(defun iswitchb-local-keys ()
      (mapc (lambda (K)
        (let* ((key (car K)) (fun (cdr K)))
              (define-key iswitchb-mode-map (edmacro-parse-keys key) fun
)))
      '(("<right>" . iswitchb-next-match)
        ("<left>"  . iswitchb-prev-match)
        ("<up>"    . ignore             )
        ("<down>"  . ignore             ))))
(add-hook 'iswitchb-define-mode-map-hook 'iswitchb-local-keys)
```

.emacs Recap

```
(require 'compile)
(add-hook 'c++-mode-hook
    (lambda ()
      (unless (or (file-exists-p "makefile")
      (file-exists-p "Makefile"))
        (set (make-local-variable 'compile-command)
       (concat "g++␣-std=c++11␣-g␣" ;; -g for debugging
         (buffer-file-name))))))

(global-linum-mode 1)
(load-theme 'whiteboard t)
(setq default-directory "~/")
(show-paren-mode 1)
(delete-selection-mode t)
(global-set-key (kbd "M-a") 'other-frame)
(set-default 'cursor-type 'box)
```

Typical Input/Output Routines for Multiple Test Cases

In a programming contest, the correctness of your code is usually determined by running your code against several test cases.
Rather than using many individual test case files, programming contest problems usually uses **one** test case file with multiple test cases included.

We use a very simple problem as an example of a multiple-test-cases problem: Given two integers in one line, output their sum in one line. We will illustrate three possible input/output formats:

- The number of test cases is given in the first line of the input.
- The multiple test cases are terminated by special values (usually zeroes).
- The multiple test cases are terminated by the EOF (end-of-file) signal.

Number of Test Cases Given in the First Input Line

```
                                  // +-------+--------+
int TC, a, b;                     // | Input | Output |
scanf("%d", &TC);                 // +-------+--------+
while (TC--)                      // | 3     | 3      |
{                                 // | 1 2   | 12     |
  scanf("%d␣%d", &a, &b);         // | 5 7   | 9      |
  printf("%d\n", a + b);          // | 6 3   |        |
}                                 // +-------+--------+
```

Test Cases Terminated by Special Values

We stop when both integers are 0.

```
int a, b;                                   // +-------+--------+
while (scanf("%d␣%d", &a, &b), (a || b))    // | Input | Output |
{                                           // +-------+--------+
  printf("%d\n", a + b);                    // | 1 2   | 3      |
}                                           // | 5 7   | 12     |
                                            // | 6 3   | 9      |
                                            // | 0 0   |        |
                                            // +-------+--------+
```

Test Cases Terminated by the EOF

```
int a, b;                                   // +-------+--------+
while (scanf("%d␣%d", &a, &b) == 2)         // | Input | Output |
{                                           // +-------+--------+
  printf("%d\n", a + b);                    // | 1 2   | 3      |
}                                           // | 5 7   | 12     |
                                            // | 6 3   | 9      |
                                            // +-------+--------+
```

Case Number and Blank Lines after Each Case

Some problems with multiple test cases require the output of each test case to be numbered sequentially. Some also require a blank line after each test case.

Let's modify the simple problem above to include the case number in the output (starting from one) with this output format: '**Case [NUMBER]: [ANSWER]**' following by a blank line for each test case (we assume the input is terminated by EOF).

```
int a, b, c = 1;                                  // +-------+------------+
while (scanf("%d␣%d", &a, &b) != EOF)             // | Input | Output     |
{                                                 // +-------+------------+
  printf("Case␣%d:␣%d\n\n", c++, a + b);          // | 1 2   | Case 1: 3  |
}                                                 // | 5 7   |            |
                                                  // | 6 3   | Case 2: 12 |
                                                  // |       |            |
                                                  // |       | Case 3: 9  |
                                                  // |       |            |
                                                  // +-------+------------+
```

Case Number and Blank Lines only Between Cases

Some problems require us to output blank lines only **between** test cases.

```
int a, b, c = 1;                                  // +-------+------------+
while (scanf("%d␣%d", &a, &b) != EOF)             // | Input | Output     |
{                                                 // +-------+------------+
  if (c > 1) printf("\n");                        // | 1 2   | Case 1: 3  |
  printf("Case␣%d:␣%d\n", c++, a + b);            // | 5 7   |            |
}                                                 // | 6 3   | Case 2: 12 |
                                                  // |       |            |
                                                  // |       | Case 3: 9  |
                                                  // +-------+------------+
```

Variable Number of Inputs I

Let's change the problem slightly. For each test case (each input line), we are now given an integer k ($k >= 1$), followed by k integers. Our task is now to output the sum of these k integers (we assume the input is terminated by EOF and we do not need case numbering).

```
int k, ans, v;
while (scanf("%d", &k) != EOF)
{
  ans = 0;
  while (k--)
  {
   scanf("%d", &v);
   ans += v;
  }
  printf("%d\n", ans);
}
```

```
// +-------------+--------+
// | Input       | Output |
// +-------------+--------+
// | 1 1         | 1      |
// | 2 3 4       | 7      |
// | 3 8 1 1     | 10     |
// | 4 7 2 9 3   | 21     |
// | 5 1 1 1 1 1 | 5      |
// +-------------+--------+
```

Variable Number of Inputs II

We modify slightly the input format for the problem of the previous slide. Instead of an integer k at the beginning of each test case, you are now required to sum all integers in each test case (each line).

```
int main()
{
  int ans, v;
  char s[100];
  while (scanf("%[^\n]%*c", s) != EOF)
    {
      ans = 0;
      char* integer = strtok(s, "␣");
      while (integer != NULL)
      {
        sscanf(integer, "%d", &v);
        ans += v;
        integer = strtok(NULL, "␣");
      }
      printf("%d\n", ans);
    }
  return 0;
}
```

Variable Number of Inputs III

We want to read the input for uva problem number 10511:

Input:
In the first line of input there will be an integer T, giving the number of test cases. The next line will be blank. Each of the T test cases consists of up to 1000 lines, each naming a resident, a party, and a list of clubs to which the resident belongs. Names are alphanumeric and separated by a single space. Each resident appears in exactly one input line. Input lines do not exceed 80 characters. Every set of input ends with a blank line.

Sample Input:

```
2

fred dinosaur jets jetsons
john rhinocerous jets rockets
mary rhinocerous jetsons rockets
ruth platypus rockets

fred dinosaur jets jetsons
john rhinocerous jets rockets
mary rhinocerous jetsons rockets
ruth platypus rockets
```

Variable Number of Inputs III

```
int main(int argc, char *argv[])
{
  int TC;
  cin >> TC >> ws;
  while (TC--)
    {
      string s;
      while (getline(cin, s) && !all_of(s.begin(), s.end(),
                  [](char c) { return isspace(c); }))
      {
        stringstream ss(s);
        string resident, party, club;
        ss >> resident >> party;
        while (ss >> club)
        {
        }
      }
    }
  return 0;
}
```

Exercice 7

Write a program to read the following format from the standard input and write the numbers read to the standard output:

- The first line countains the number of elements to read.
- The others lines countain a real number between $[0, 1[$ and terminated by three dots.

For example:

```
3
0.12345555...
0.9873221...
0.001...
```

Testing

The sample input-output given in the problem description is by nature trivial and therefore usually not a good means for determining the correctness of your code.

You can always generate by hand input/output pairs to test your program. But it can be tedious and time consumming to generate tests with this approach.

If the constraints of the problem are too difficult for a simple solution, we can generate input/output pairs for testing with the following approach:

1. Generate a simple solution (often a brute-force approach which is generally easy to design). Generate few tests by hand for this simple solution to validate it.
2. Generate lots of small random inputs.
3. Feed the simple solution with the random inputs.
4. Test your solution with the input/output pairs obtained above.

Testing: Removing duplicates

Problem: Given a list of integers, create a new list without duplicates (there is a method in `class list` that does exactly that, but we code it for learning purpose).

We come-up with the following function:

```
list<int> removing_duplicates(const list<int>& l)
{
  list<int> res(l.begin(), l.end());
  res.sort();
  auto it = res.begin();
  while (it != res.end())
    {
      auto current = it;
      it++;
      if (it != res.end() && *current == *it)
      {
        res.erase(current);
      }
    }
  return res;
}
```

Testing: Removing duplicates

To test it, we create two functions:

1. A generator of random lists.
2. A brute-force solution to the problem.

Testing: Generating Random Lists

```
#include <list>
#include <random>
#include <chrono>

template <typename T> inline
list<T> generate_random_list(int min_size, int max_size, T min_value, T
max_value)
{
  list<T> res;
  default_random_engine dre(chrono::system_clock::now().time_since_epoch
().count());
  uniform_int_distribution<T> s(min_size, max_size);
  uniform_int_distribution<T> di(min_value, max_value);
  int size = s(dre);
  for (int i=0; i<size; ++i)
    {
      res.push_back(di(dre));
    }
  return res;
}
```

Testing: Brute-Force Solution

```
list<int> removing_duplicates_brute_force(const list<int>& l)
{
  list<int> res;
  for (const auto& v : l)
  {
      if (find(res.begin(), res.end(), v) == res.end())
      {
          res.push_back(v);
      }
  }
  // to compare the results between the two methods, we sort the list
  res.sort();
  return res;
}
```

Testing

To test:

1. We generate a file, for example `test_removing_duplicates.txt`, containing random lists.
2. Then, we test if there is some difference between the results of `removing_duplicates_brute_force` and `removing_duplicates` on `test_removing_duplicates.txt`.

If we suppose that:

- `remove_brute_force` is the executable for our brute-force version.
- `remove_deluxe` is the executable for our faster version.

We can test the difference with the following command:

```
$ diff <(./remove_deluxe < test_removing_duplicates.txt) <(./
remove_brute_force < test_removing_duplicates.txt)
```

Exercice 8

Problem: Given an unsorted vector of integers, give the k-th smallest element.

Input: The number `TC` of test cases and `TC` lines that contains the number k followed by the array elements. For example:

```
3
1 10 2 3 4 100 42
3 1 2 3 4 5 6 7 8 9 10
5 0 8 7 6 5 4 3 2 1 9
```

Output: The k-th element for each array. For the above input:

```
2
3
4
```

We proposed a fast solution in the following slides. Test it with the approach described above.

Proposed Solution

We come-up with the following solution:

```
int select(const vector<int>& v, int k)
{
  vector<int> aux = v;
  random_shuffle(aux.begin(), aux.end() - 1);
  int lo = 0, hi = v.size() - 2;
  while (hi > lo)
    {
      int j = partition(aux, lo, hi);
      if (j < k) lo = j + 1;
      else if (j > k) hi = j - 1;
      else return aux[k];
    }
  return aux[k];
}
```

Proposed Solution

```
void swap(vector<int>& v, int i, int j)
{
  int tmp = v[i];
  v[i] = v[j];
  v[j] = tmp;
}
int partition(vector<int>& v, int lo, int hi)
{
  int i = lo, j = hi+1;
  while (true)
    {
      while (v[++i] < v[lo])
        if (i == hi) break;
      while (v[lo] < v[--j]);
      if (i >= j) break;
      swap(v, i, j);
    }
  swap(v, lo, j);
  return j;
}
```

Debugging

When your program fails on some test, you have to find the bugs. A debugger is way more powerful and efficient than adding `printf` to your code. In this part:

- We give general guideline to debug.
- We give an overview of `gdb` using an example.

Guidelines

The following list give some general advices for debugging:

1. Understand the requirements.
2. Simplify the test case.
3. Read the right error message.
4. Check the plug.
5. Separate fact from interpretation.
6. Divide and conquer.
7. Change one thing at a time.
8. Get a fresh view.
9. If you didn't fix it it ain't fixed.

Understand the Requirements

Make sure you understand the requirements before you begin to debug and fix anything:

- Is there a standards document or a specification to look at?
- Maybe the software is not malfunctioning after all. It could be a misinterpretation instead of a bug.

Simplify the Test Case

You do this in order to:

- Rule out factors that do not play a role.
- Make the test case easier to debug. Who wants to deal with data containers filled with hundreds or thousands of items?

Read the Right Error Message

Something went wrong and you face a screen full of error messages. Which ones do you focus on?

Many people don't give the correct answer: *The ones that come out first!*

The first problem may have left the program in a corrupt state.

Check the Plug

Check the seemingly obvious:

- Were all parts of the software up and running when the problem occurred?
- Permissions OK?
- Enough disk quota?
- Is there enough space on all relevant file systems (for example in /tmp)
- Does the system have enough memory?

Separate Facts from Interpretation

Don't jump to conclusions. Maintain a list of things you know for a fact, and why. Ask yourself: "Can I prove it". Is the behavior reproducible?

Is what you consider a fact really a fact?
For example:

"It fails when I select a blue item but it always works for red items"

a bug report may state.

So misbehavior depends on the color? Maybe not. It could be that the user selected the blue item with a mouse click and everything else via the keyboard, by specifying its name.

Divide and Conquer

- Narrow your search.
- Reduce the places a bug can hide.
- Check the obvious first.

To cite *Sherlock Holmes*:

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"
—The Sign Of The Four.

Change One Thing at a Time

Generally, Do not change more than one thing at a time if possible. Then check if it makes sense and, if not, revert back before trying out the next idea.

Get a Fresh View

When you are stuck, go and find somebody to talk to. Make sure to draw a clear line between the facts – and why they are – and your theories. Chances are good that your theories may be less than perfect.

If You Didn't Fix It It Ain't Fixed

Occasionally, a bug will just disappear after you modified some statements. **Unless you have a good explanation why your fix is effective, you are better off to assume that the bug still exists** and will hit again in the future.

Your source code change may merely change the probability for the bug to re-occur.

Using GDB

We describe in the following slides a very simple debugging session with gdb.

We use gdb on the following program:

```
1   #include <cstdio>
2   #include <cstdlib>
3
4   int factorial(int n)
5   {
6     int result = 1;
7     if (n == 0)
8       return result;
9     result = factorial(n-1) * n;
10    return result;
11  }
```

Using GDB

```
12
13  int main(int argc, char **argv)
14  {
15    int n, result;
16    if (argc != 2)
17      {
18        fprintf(stderr, "usage:␣factorial␣n,␣n␣>=0\n");
19        return 1;
20      }
21    n = atoi(argv[1]);
22    result = factorial(n);
23    printf("factorial␣%d␣=␣%d\n", n, result);
24    return 0;
25  }
```

Using GDB

First, we compile the program `Factorial.cpp` with the option `-g` for debugging:

```
> g++ -g Factorial.cpp -o factorial
```

Then we launch gdb:

```
> gdb Factorial
```

We get:

```
<some bla bla ...>
(gdb)
```

Using GDB

We launch the program with -1 as argument:

```
(gdb) run -1
Starting program: /Users/pgarcia/Enseignement/CompetitiveProgramming/
Code/a.out -1

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000e5a in factorial (n=<error reading variable: Cannot
access memory at address 0x7fff5f3aefec>)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:5
(gdb)
```

This gdb output tells us that the program hit a memory error in line 5 of the function `factorial`, as it was trying to make yet another call to itself. gdb shows the actual source code line where the error occurred.

Using GDB

The gdb debugger refers to the stack frames by numbers, where the current frame has stack frame number 0, and the frame of the `main()` function has the highest number. The command for a stack trace is: `bt`, `backtrace`, or `where`:

```
(gdb) backtrace
#0  0x0000000100000e5a in factorial (
    n=<error reading variable: Cannot access memory at address 0
x7fff5f3aefec>)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:5
#1  0x0000000100000e7c in factorial (n=-181603)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:9
#2  0x0000000100000e7c in factorial (n=-181602)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:9
<... lots of frames>
```

You can navigate up and down the call stack, and inspect the values of function arguments and local variables. You can use the commands `up` or `down` to move through the stack. You can also use the command: `frame n` to go to the frame numbered `n`.

Using GDB

We take the factorial example again, and apply the debugger to find out why the program returns an incorrect value for $n >= 13$. Our program returns the value 1932053504 for 13!, whereas the correct value is $13! = 6227020800$.

```
> gdb factorial
<...>
(gdb) start 13
Temporary breakpoint 1 at 0x100000e97: file /Users/pgarcia/Enseignement/
CompetitiveProgramming/Code/DebugFactorial.cpp, line 16.
Starting program: /Users/pgarcia/Enseignement/CompetitiveProgramming/
Code/a.out 13

Temporary breakpoint 1, main (argc=2, argv=0x7fff5fbff2e0)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:16
(gdb)
```

The `start` command run the program with the specified argument and stop right at the beginning of main.

Using GDB

We do the `next` command over each line until we reach line 22, where the `factorial()` function is called. The purpose is to check that the command line argument is parsed correctly, and that an integer value of 13 is passed to the `factorial()` function. We then step into the function `factorial()`:

```
(gdb) next
(gdb) next
(gdb) print n
$2 = 13
(gdb) step
factorial (n=13)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:6
(gdb)
```

Using GDB

We need to check that the recursion works, so we place a breakpoint in line 7, to check that for $\mathtt{n} = 0$, the function does not call itself again. We also place a breakpoint in line 10, so we can look at the return value of each call to `factorial`. Rather than stepping, we then let the program continue to the first breakpoint:

```
(gdb) break 8
Breakpoint 2 at 0x100000e6a: file /Users/pgarcia/Enseignement/
CompetitiveProgramming/Code/DebugFactorial.cpp, line 8.
(gdb) break 10
Breakpoint 3 at 0x100000e83: file /Users/pgarcia/Enseignement/
CompetitiveProgramming/Code/DebugFactorial.cpp, line 10.
(gdb) continue
Continuing.

Breakpoint 2, factorial (n=0)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:8
(gdb)
```

Using GDB

We print the variables `n` and `result`, to check that the function works properly. We can also use the `display` command on the variable result, so we do not have to repeat typing the `print` command:

```
(gdb) print n
$1 = 0
(gdb) print result
$2 = 1
(gdb) display result
1: result = 1
(gdb)
```

As we keep repeating the `continue` command, we get a printout of the variable `result` at line 10 in the program, for $\texttt{n} = 0, 1, .., 13$. We pause when `n` is 13:

Using GDB

```
(gdb) continue
Continuing.
Breakpoint 3, factorial (n=1)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:10
1: result = 1
<...>
(gdb) continue
Continuing.
Breakpoint 3, factorial (n=12)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:10
1: result = 479001600
(gdb) continue
Continuing.
Breakpoint 3, factorial (n=13)
    at /Users/pgarcia/Enseignement/CompetitiveProgramming/Code/
DebugFactorial.cpp:10
1: result = 1932053504
(gdb)
```

Using GDB

```
(gdb) print 13 * 479001600
$5 = 193053504
```

The result for $\mathtt{n} = 12$ is still correct, but that the multiplication with the value 13 calculates an incorrect result. We repeat the calculation done in line 9, by printing the expression `13 * 479001600`, and note that it reproduces the calculation error. The reason for this is that the value for 13! is too large to be stored in a 32- bit variable of type `int`, so the multiplication causes an incorrect result due to an integer overflow.

Exercice 9

Debug the program of exercice 8.

Proof Techniques

We give in this section a review of three proof techniques:

- Proof by contradiction
- Pigeonhole principle
- Proof by induction

Those proof techniques are usefull for solving some programming problems and/or proving that our solution is correct.

Proof by Contradiction

Contradiction

Suppose we wish to prove some statement A. We can use contradiction by showing that if A were false, then some impossible statement would have to be true.

Example: Prove that there are infinitely many prime numbers.

Proof: Suppose there aren't infinitely many primes and look for a contradiction. Let there be k prime numbers, namely $p_1, p_2, \ldots, p_k$. Consider the number:

$$Z = p_1 p_2 p_3 \ldots p_{k-1} p_k + 1$$

According to our hypothesis, Z is not prime and so must be divisible by a prime number. So Z must be divisible by one of the p_i and so $1 = Z - p_1 p_2 p_3 \ldots p_{k-1} p_k$ too, contradicting our assumption that there are only k prime numbers, so there cannot be a finite number of primes.

Exercice 10

We are given the following problem (Code Chef (http://www.codechef.com) problem Subarray GCD):

Problem:
Given an array `[v1, v2, ..., vn]` you have to print the size of the largest contiguous subarray such that the *gcd* of all integers in that subarray is 1. Minimum size of the subarray is 2.

Exercice 10

We are pretty sure that the following code solve the problem:

```
int main(int argc, char *argv[])
{
  int TC;
  scanf("%d", &TC);
  while (TC--)
  {
    int N, res = 0;
    scanf("%d", &N);
    for (int i = 0; i < N; ++i)
    {
      int v;
      scanf("%d", &v);
      res = gcd(res, v);
    }
    if (res == 1) printf("%d\n", N);
    else printf("-1\n");
  }
  return 0;
}
```

Exercice 10

To prove that our solution is correct, show by contradiction that: if the *gcd* of all elements of the array is not 1 then it implies that there are no subarrays whose *gcd* is 1

Pigeonhole Principle

Pigeonhole Principle

The Pigeonhole Principle states that given $kn + 1$ objects which are placed in n boxes, there must be some box with at least $k + 1$ objects.

Example: Prove that if we select 5 points within the boundaries of a unit square, then some pair of them are no more than $\sqrt{2}/2$ apart.

Proof: We can apply the Pigeonhole Principle by dividing the square into four squares with side length $1/2$. By the Pigeonhole Principle at least two of the points must fall in the same square. The farthest apart two points can be in a $1/2$ unit square is $\sqrt{2}/2$, where the points are on opposite ends of a diagonal. Since there must be two points in one of the little squares, there must somewhere be a pair of the five points at most $\sqrt{2}/2$ apart.

Exercice 11

We are given the following problem (Code Chef (http://www.codechef.com) problem Divisible Subset):

Problem:
You are given a multiset of N integers. Please find a nonempty subset of it for which the sum of the subset's elements is divisible by N. Otherwise, state that this subset doesn't exist.

Example:
For $N = 5$ and the multiset: $\{1, 1, 3, 4, 6\}$, this is true for the subset $\{1, 4\}$.

We claim that there is always such a subset. Moreover, there is always a consecutive subsegment of any sequence of numbers that corresponds to the multiset's elements with the sum divisible by the multiset's size. Prove it using the Pigeonhole Principle

Proof by Induction

We give in this part a review of mathematical induction because:

- We will use this principle to prove statements in the rest of the course.
- We will use a variation of this principle to solve algorithmic problems (*Design of Algorithms by Induction*).
- For some problems, we have to guess a pattern for a given sequence, and we can prove by induction that our guess is correct.

Mathematical Induction

Principle of Mathematical Induction

Let $P(n)$ be a property that is defined for integers n, and let a be a fixed integer. Suppose the following two statements are true:

1. $P(a)$ is true.
2. For all integers $k \geq a$, if $P(k)$ is true then $P(k+1)$ is true.

Then the statement:

$$\text{For all integers } n \geq a,\ P(n)$$

is true.

Method of Proof by Mathematical Induction

Method of Proof by Mathematical Induction

Consider a statement of the form, "For all integers $n \geq a$, a property $P(n)$ is true." To prove such a statement, perform the following two steps:

- Step 1 (basic step): Show that $P(a)$ is true.
- Step 2 (inductive step): Show that for all integers $k \geq a$, if $P(k)$ is true then $P(k+1)$ is true. To perform this step:

Suppose that $P(k)$ is true where k is any arbitrarily chosen integer with $k \geq a$.

[*This supposition is called the* **inductive hypothesis**.]

Then:

Show that $P(k+1)$ is true.

Example 1

For all integers $n \geq 8$, n cents can be obtained using 3 cents and 5 cents coins.

Proof (by mathematical induction):

Let the property $P(n)$ be the sentence:

$$\boxed{n \text{ cents can be obtained using 3 cents and 5 cents coins.}}$$

1. **Show that $P(8)$ is true:**
   $P(8)$ is true because 8 cents can be obtained using one 3 cents coin and one 5 cents coin.

2. **Show that for all integers $k \geq 8$, if $P(k)$ is true then $P(k+1)$ is also true:**
   Suppose that k is any integer with $k \geq 8$ such that:

   $$k \text{ cents can be obtained using 3 cents and 5 cents coins.}$$

   We must show that:

   $$(k+1) \text{ cents can be obtained using 3 cents and 5 cents coins.}$$

Example 1

For all integers $n \geq 8$, n cents can be obtained using 3 cents and 5 cents coins.

2. **Show that for all integers $k \geq 8$, if $P(k)$ is true then $P(k+1)$ is also true:**
   - **Case 1. There is a 5 cents coin among those used to make up the k cents.** In this case, replace the 5 cents coin by two 3 cents coins; the result will be $(k+1)$ cents.
   - **Case 2. There is not a 5 cents coin among those used to make up the k cents.** In this case, because $k \geq 8$, at least three 3 cents coins must have been used. So remove three 3 cents coins and replace them by two 5 cents coins; the result will be $(k+1)$ cents.

Example 2

For all integers $n \geq 0$, $2^{2n} - 1$ is divisible by 3.

Proof (by mathematical induction):

Let the property $P(n)$ be the sentence:

$$\boxed{2^{2n} - 1 \text{ is divisible by } 3.}$$

1. **Show that $P(0)$ is true.** We must show that $2^{2\times 0} - 1$ is divisible by 3. But $2^{2\times 0} - 1 = 0$ and because $0 \times 3 = 0$, $2^{2\times 0} - 1$ is divisible by 3. Hence $P(0)$ is true.

2. **Show that for all integers $k \geq 0$, if $P(k)$ is true then $P(k+1)$ is also true:** Suppose that k is any integer with $k \geq 0$ such that:

$$2^{2k} - 1 \text{ is divisible by } 3.$$

By definition of divisibility, this means that $2^{2k} - 1 = 3r$ for some integer r. We must show that:

$$2^{2(k+1)} - 1 \text{ is divisible by } 3.$$

Example 2

For all integers $n \geq 0$, $2^{2n} - 1$ is divisible by 3.

2. **Show that for all integers $k \geq 0$, if $P(k)$ is true then $P(k+1)$ is also true:**
   We have:

$$
\begin{aligned}
2^{2(k+1)} - 1 &= 2^{2k+2} - 1 \\
&= 2^{2k} \times 2^2 - 1 \\
&= 2^{2k} \times 4 - 1 \\
&= 2^{2k} \times (3+1) - 1 \\
&= 2^{2k} \times 3 + (2^{2k} - 1) \\
&= 2^{2k} \times 3 + 3r \text{ (by inductive hypothesis)} \\
&= 3(2^{2k} + r)
\end{aligned}
$$

   And because $2^{2k} + r$ is an integer, $2^{2(k+1)} - 1$ is divisible by 3.

Example 3: Wrong Usage

We will **incorrectly** prove by induction that in any sequence of real numbers, all elements are equal!

In any sequence of real numbers, all elements are equal.

Incorrect Proof (by mathematical induction):

Let the property $P(n)$ be the sentence:

For any real numbers $a_1, a_2, \ldots, a_n$, we have $a_1 = a_2 = \ldots = a_n$.

- **Show that $P(1)$ is true.** Indeed the sequence contains only one real number.

Example 3: Wrong Usage

- **Show that for all integers $k \geq 1$, if $P(k)$ is true then $P(k+1)$ is also true:**

  Suppose that k is any integer with $k \geq 0$ such that: $P(k)$ is true, i.e., that any k real numbers must be equal.
  We seek to show that $P(k+1)$ is true as well, i.e., any $k+1$ real numbers must also be equal.
  Let $a_1, a_2, \ldots, a_{k+1}$ be given real numbers. Applying the induction hypothesis to the first k of these numbers, $a_1, a_2, \ldots, a_k$, we obtain:

  $$a_1 = a_2 = \ldots = a_k \qquad (1)$$

  Similarly, applying the induction hypothesis to the last k of these numbers, $a_2, a_3, \ldots, a_k, a_{k+1}$, we get:

  $$a_2 = a_3 = \ldots = a_k = a_{k+1} \qquad (2)$$

  Combining (1) and (2) gives:

  $$a_1 = a_2 = \ldots = a_k = a_{k+1} \qquad (3)$$

  Thus we have proved $P(k+1)$.

Exercice 12

What is wrong with the preceding proof?

Prove by induction that:

For all n 3, if n distinct points on a circle are connected in consecutive order by straight lines, then the interior angles of the resulting polygon add up to: (n 2)180°.

For example, with n = 4:

Example

Any integer strictly greater than 1 is divisible by a prime.

Proof (by strong mathematical induction):

Let the property $P(n)$ be the sentence:

$$\boxed{n \text{ is divisible by a prime number.}}$$

1. **Show that $P(2)$ is true:** 2 is divisible by 2 and 2 is a prime number.
2. **Show that for all integers $k \geq 2$, if $P(i)$ is true for all integers i from 2 through k, then $P(k+1)$ is also true:**
   Let k be any integer with $k \geq 2$ and suppose that:

   $$i \text{ is divisible by a prime number for all integers } i \text{ from 2 through } k.$$

   We must show that:

   $$(k+1) \text{ is divisible by a prime number.}$$

Example

Any integer strictly greater than 1 is divisible by a prime.

2. **Show that for all integers $k \geq 2$, if $P(i)$ is true for all integers i from 2 through k, then $P(k+1)$ is also true:**
   - **Case 1. $k+1$ is prime.** In this case $k+1$ is divisible by a prime number, namely itself.
   - **Case 2. $k+1$ is not prime.** In this case $k+1 = ab$ where a and b are integers with $1 < a < k+1$ and $1 < b < k+1$. Thus, in particular, $2 \leq a \leq k$, and so by inductive hypothesis, a is divisible by a prime number p. In addition because $k+1 = ab$, we have that $k+1$ is divisible by a. Hence, since $k+1$ is divisible by a and a is divisible by p, by transitivity of divisibility, $k+1$ is divisible by the prime number p.

     Therefore, regardless of whether $k+1$ is prime or not, it is divisible by a prime number.

Exercice 15

Consider the following sequence:

$$s_n = \begin{cases} 0 & \text{if } n = 0 \\ 4 & \text{if } n = 1 \\ 6s_{n-1} - 5s_{n-2} & \text{else} \end{cases}$$

1. Find the first four terms of this sequence.
2. Guess a closed-form solution for this sequence and proof by strong induction that your guess is correct.

Exercice 16

Solve Code Chef (http://www.codechef.com) problem Yet Another Number Game (Practice Easy). Try to find a pattern and use induction to show that this pattern is correct.

Structural Induction

We introduce structural induction, which is a version of mathematical induction that is used to prove properties of recursively defined sets.

Recursively Defined Sets

To define a set of objects recursively, you identify a few core objects as belonging to the set and give rules showing how to build new set elements from old. More formally, a recursive definition for a set consists of the following three components:

1. Base: A statement that certain objects belong to the set.
2. Recursion: A collection of rules indicating how to form new set objects from those already known to be in the set.
3. Restriction: A statement that no objects belong to the set other than those coming from 1 and 2.

Example

Parenthesis Structures

Here is a recursive definition to generate the set P of legal configurations of parentheses:

1. Base: () is in P.
2. Recursion:
   - If E is in P, so is (E).
   - If E and F are in P, so is EF.
3. Restriction: No configurations of parentheses are in P other than those derived from 1 and 2 above.

Exercice 17

Derive the fact that (())() is in P.

Structural Induction for Recursively Defined Sets

Structural Induction for Recursively Defined Sets

Let S be a set that has been defined recursively, and consider a property that objects in S may or may not satisfy. To prove that every object in S satisfies the property:

1. Show that each object in the base for S satisfies the property.
2. Show that for each rule in the recursion, if the rule is applied to objects in S that satisfy the property, then the objects defined by the rule also satisfy the property.

   Because no objects other than those obtained through the base and recursion conditions are contained in S, it must be the case that every object in S satisfies the property.

Exercice 18

Show by structural induction that every element of the set S of parenthesis defined above, contains an equal number of left and right parenthesis.

Complexity

When we deal with a problem, it is always a good idea to use the **Ockham's razor**:

Ockham's razzor

Pluralitas non est ponenda sine necessitate.
"Multiplicity is not to be posited without necessity".

In our setting, this means that if it is possible to solve a problem with a simple algorithm, it is better to use it because:

- We do less bugs because the solution is simple.
- We come up faster with a correct one.

One problem, is the possible time complexity of the simple algorithm. For example, a brute-force like algorithm is often very simple to come up with, but usually, due to its complexity, it is limited to small inputs.

We will review complexity analysis in this part to be able to select a correct approach for a given problem. We strive for a simple solution but our approach must be able to cope with the input size constraints of the problem.

Note that we only speak about **time complexity** in this part, but our discussion is general and can be easily adaptated for **space complexity**.

Complexity as a Function of the Input Size

We define **complexity as a function of the input size**. We want to understand the behavior of our program as the input size grows. Moreover, we are interested in the **worst case behavior** of our algorithm.

For example, if our input is an array of size n, we will consider for each input size, the worst time complexity of our algorithm for the set of all arrays of size n.
Why? Because :

- Best case behavior is useless because our algorithm can be very fast for one entry and very slow for all others. So it does not give us a garantee.
- Average case behavior can be usefull (is used for example for randomized algorithms), but it gives us less garantees than worst case behavior and is more difficult to analyze.

Complexity as a Function of the Input Size

Suppose that we have some algorithms that can have a time complexity of: n, $n \log_2 n$, n^2, n^3, 1.5^n, 2^n or $n!$. We give in the following table, the running time for those algorithms on inputs of increasing size, for a processor running a million (simple) instructions per seconds.

n	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 4 s
30	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	18 mn	10^{25} yrs
50	$<$ 1 s	$<$ 1 s	$<$ 1 s	$<$ 1 s	11 mn	36 yrs	long
100	$<$ 1 s	$<$ 1 s	$<$ 1 s	1 s	12892 yrs	10^{17} yrs	long
1000	$<$ 1 s	$<$ 1 s	1 s	18 mn	long	long	long
10000	$<$ 1 s	1 s	2 mn	12 days	long	long	long
100000	$<$ 1 s	2 s	3 hours	32 yrs	long	long	long
1000000	1 s	20 s	12 days	31710 yrs	long	long	long

Detailed Analysis

Consider the following code for *insertion sort*:

```
void insertion_sort(vector<int>& v)
{
  int n = v.size();
                                        //cost times
  for (int i = 1; i < n; i++)           // c1    n
  {
    int key = v[i];                     // c2    n − 1
    int j = i - 1;                      // c3    n − 1
    while (j >= 0 && v[j] > key)        // c4    Σ_{i=1}^{n−1} t_i
    {
      v[j + 1] = v[j];                  // c5    Σ_{i=1}^{n−1} (t_i − 1)
      j--;                              // c6    Σ_{i=1}^{n−1} (t_i − 1)
    }
    v[j + 1] = key;                     // c7    n − 1
  }
}
```

Detailed Analysis

The running time $T(n)$ of insertion sort is then the sum of running times for each statement. We obtain:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1}(t_i - 1) + c_6 \sum_{i=1}^{n-1}(t_i - 1)$$

$$+c_7(n-1)$$

After simplification we get:

- Best case, $t_i = 1$: $an + b$ for constants a and b that depend on the costs c_i.
- Average case, $t_i = n/2$: $an^2 + bn + c$ for constants a, b and c that depend on the costs c_i.
- **Worst case**, $t_i = n$: $an^2 + bn + c$ for constants a, b and c that depend on the costs c_i.

Detailed Analysis

The previous analysis is tedious, and more precise than we need. Moreover the costs c_i depend of the particular language, system and machine we use and low order terms are not significative when n becomes large.

For all these reasons, we want to express the growth rate of the running time in a way that is insensitive to constant factors and low order terms.
For example, if we take the worst case running time of the insertion $an^2 + bn + c$ we want to say that **it grows like** n^2.

Asymptotic Upper Bound

Let $g(n)$ be a positive function. We note $O(g(n))$ the following set:

Asymptotic Upper Bound

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

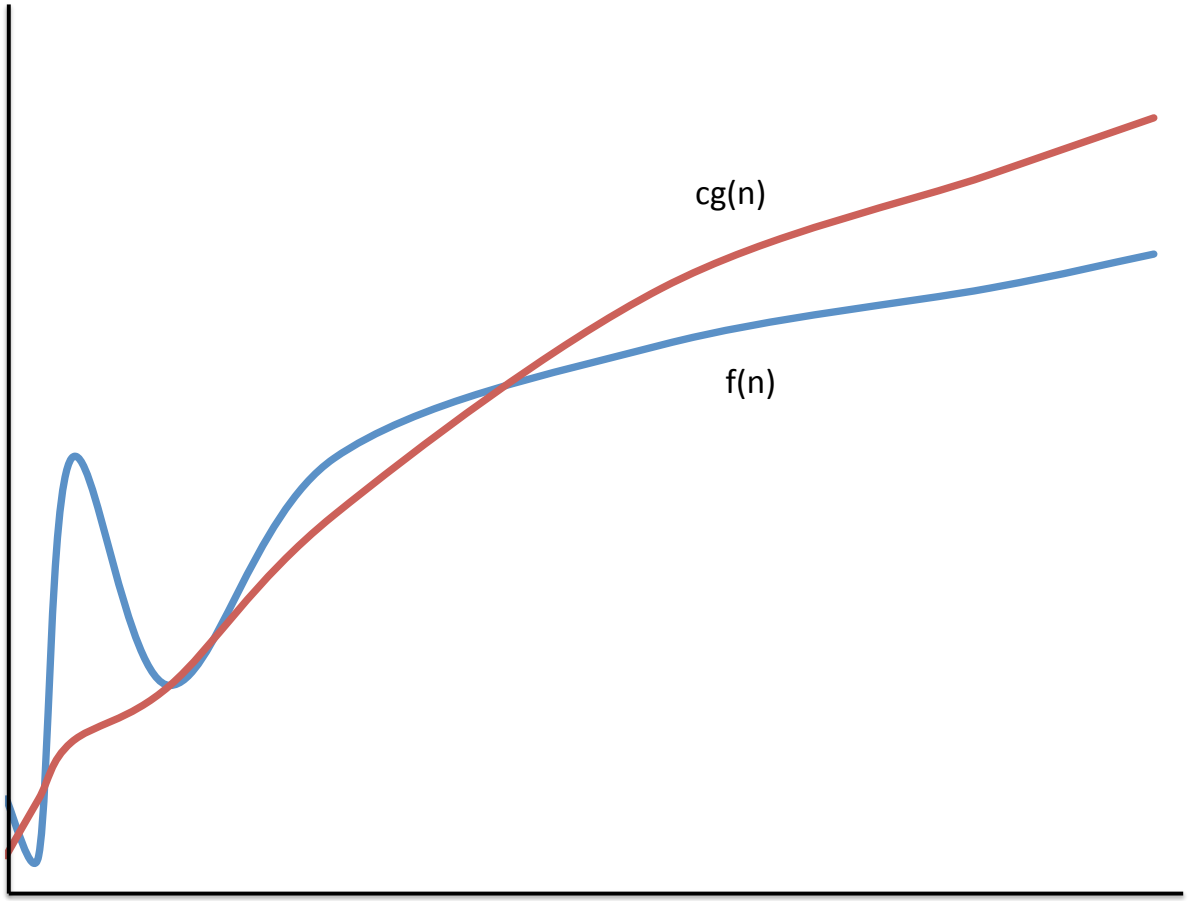
We use O to give an upper bound on a function within a constant factor and we note $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

For example:

- $23n^2 + 100n + 512 = O(n^2)$.
- $23n^2 + 100n + 512 = O(n^4)$.
- $0.5n = O(n)$.

Intuitively, $f = O(g)$ approximately means that $f \leq g$.

Asymptotic Upper Bound: Graphical Representation



f(n) = O(g(n))

Asymptotic Lower Bound

Let $g(n)$ be a positive function. We note $\Omega(g(n))$ the following set:

Asymptotic Lower Bound

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \in \mathbb{R}^* \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

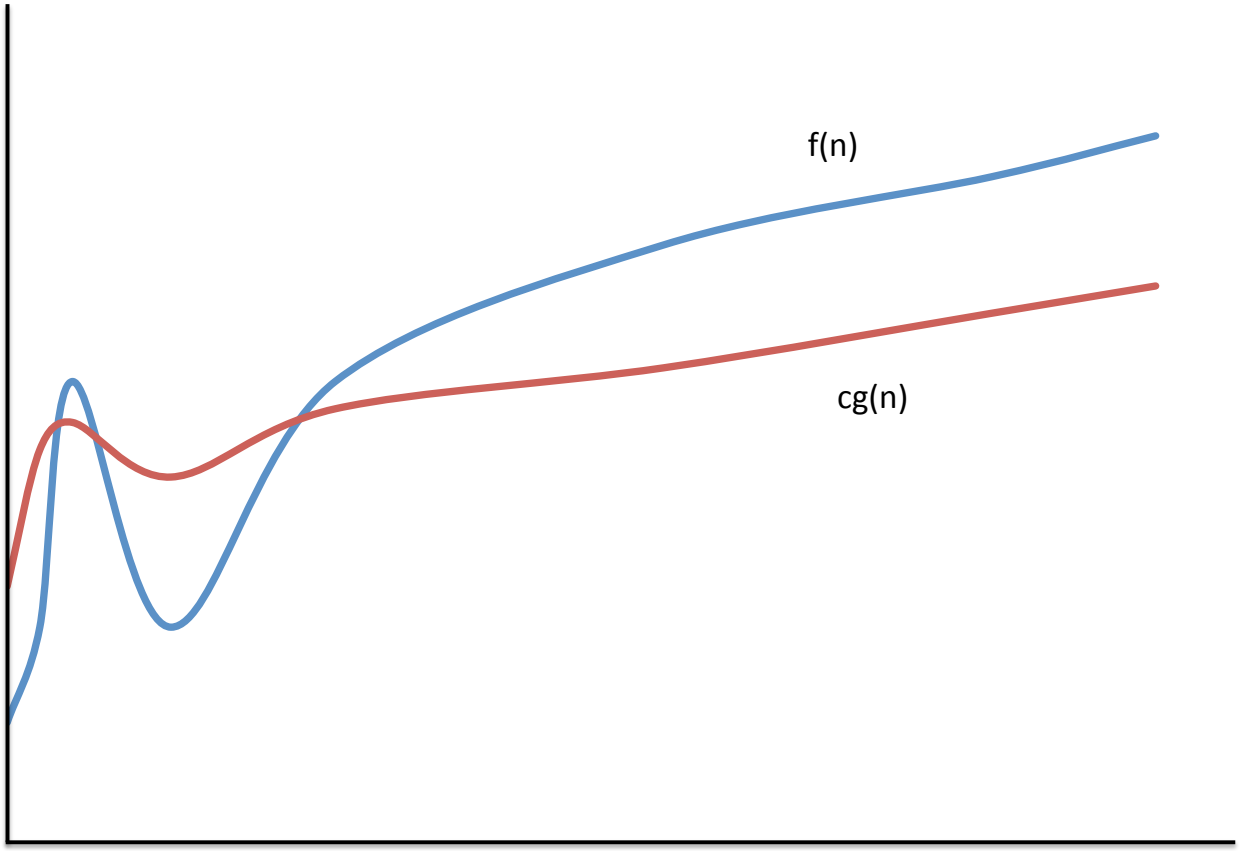
We use Ω to give an upper bound on a function within a constant factor and we note $f(n) = \Omega(g(n))$ instead of $f(n) \in \Omega(g(n))$.

For example:

- $n^2 = \Omega(0.5n^2)$.
- $23n^2 + 100n + 512 = \Omega(n^2)$.
- $n = \Omega(\ln n)$.

Intuitively, $f = \Omega(g)$ approximately means that $f \geq g$.

Asymptotic Lower Bound: Graphical Representation



f(n) = Ω(g(n))

Asymptotic Tight Bound

Let $g(n)$ be a positive function. We note $\Theta(g(n))$ the following set:

Asymptotic Tight Bound

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1 \in \mathbb{R}^*,\ c_2 \in \mathbb{R}^* \text{ and } n_0 \in \mathbb{N} \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

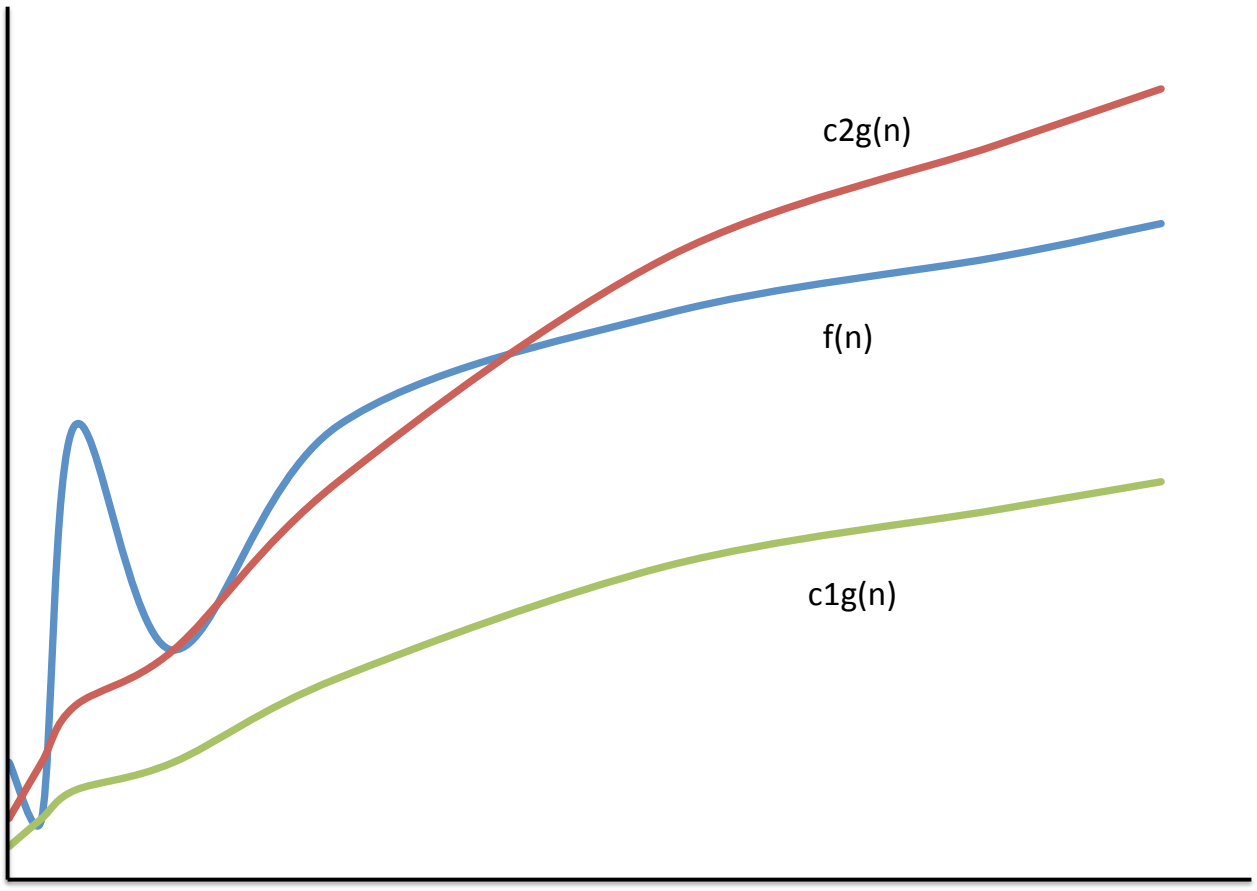
We use Θ to give a tight bound on a function within a constant factor and we note $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$.

For example:

- $n^2 = \Theta(0.5n^2)$.
- $23n^2 + 100n + 512 = \Theta(n^2)$.
- $n = \Theta(1000n)$.

Intuitively, $f = \Theta(g)$ approximately means that $f = g$.

Asymptotic Tight Bound: Graphical Representation



f(n) = Θ(g(n))

Properties

Here is a list of usefull properties on asymptotic bounds:

Let f and g be two functions such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ then $f(n) = \Theta(g(n))$

- If $f = O(g)$ and $f = \Omega(g)$ then $f = \Theta(g)$
- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$
- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$
- Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and h be functions such that $f_i = O(h)$ for all i. Then $f_1 + f_2 + \ldots + f_k = O(h)$.
- Suppose that f and g are two functions (taking non negative values) such that $g = O(f)$. Then $f + g = \Theta(f)$.

Properties

Let f be a polynomial of degree d, in which the coefficient $a_d > 0$ then $f = O(n^d)$.

For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$

For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$

Exercice 19

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$
\begin{aligned}
f_1(n) &= 10^n \\
f_2(n) &= n^{1/3} \\
f_3(n) &= n^n \\
f_4(n) &= \log_2 n \\
f_5(n) &= 2^{\sqrt{\log_2 n}}
\end{aligned}
$$

Exercice 20

Let f and g be two functions that takes nonnegative values, and suppose that $f = O(g)$. Show that $g = \Omega(f)$.

Common Running Times

When analyzing an algorithm, **it helps to have a rough sense of the "landscape" of different running times**. We present in this part different common running times and representative algorithms of them, to help identify quickly the running time of a new algorithm.

With this knowledge, you should be able to see if one approach can cope with the constraints of the problem at hand.

Sublinear Time

Sublinear time is running times that are asymptotically smaller than linear. Since it takes linear time just to read the input, these **situations tend to arise in a model of computation where the input can be "querie" indirectly rather than read completely**, and the goal is to minimize the amount of querying that must be done.

Perhaps the best-known example of this is the **binary search algorithm**. Given a sorted array A of n numbers, we would like to determine the index of a given number p in the array, or -1 if this number is not present. We could do this by reading the entire array, but we would like to do it much more efficiently. The following slide describes the binary search algorithm.

Binary Search

```
int binary_search(int key, const vector<int>& v, int lo, int hi)
{
  if (lo > hi) return -1;
  int mid = lo + (hi - lo) / 2;
  if      (key < v[mid]) return binary_search(key, v, lo, mid - 1);
  else if (key > v[mid]) return binary_search(key, v, mid + 1, hi);
  else return mid;
}

int binary_search(int key, const vector<int>& v)
{
  return binary_search(key, v, 0, v.size() - 1);
}
```

Exercice 21

In the preceding `binary_search` function, why did we use:

```
int mid = lo + (hi - lo) / 2;
```

instead of:

```
int mid = (lo + hi) / 2;
```

Complexity of Binary Search

As binary search, we often are interested in algorithms expressed in a recursive way. When we analyze them, we get a recurrence: a description of the running time on an input of size n as a function of n and the running time on inputs of smaller sizes.

Let $T(n)$ be the running time of the binary search algorithm of slide 124. We can express $T(n)$ as a reccurence relation:

$$T(n) = \begin{cases} T(n/2) + c_1 & \text{if } n \geq 1 \\ c_2 & \text{if } n = 0 \end{cases}$$

Different methods exist to solve those kind of relations. We present a simple one in the next slide.

Solving by Guess and Inductive Proof

One way to solve recurrences is to make a guess and then prove the guess correct inductively. If we get into trouble proving our guess correct (e.g., because it was wrong), often this will give us clues as to a better guess.

Let's try with $T(n)$, supposing that it is linear, we have:

$$T(n) \leq cn + c'$$

where c and c' are positive constants. The base case is true if $c' \geq c_2$. Suppose that $T(n) \leq cn + c'$ for all $n' < n$. We have:

$$\begin{aligned} T(n) &= T(n/2) + c_1 \\ &\leq cn/2 + c' + c_1 \\ &= cn + (c' + c_1 - cn/2) \end{aligned}$$

if we pose $c = 2c_1$ we get the result. Because $T(n) \leq cn + c'$ we have $T(n) = O(n)$.

Solving by Guess and Inductive Proof

We can try to find a better bound. Let's suppose that it is logarithmic:

$$T(n) \leq c \log(n+1) + c'$$

where c and c' are positive constants.

Note: For now on, we write "log" for "$\log_2$".

The base case is true if $c' \geq c_2$. Suppose that $T(n) \leq c \log(n+1) + c'$ for all $n' < n$. We have:

$$\begin{array}{rcl} T(n) & = & T(n/2) + c_1 \\ & \leq & c \log((n+1)/2) + c' + c_1 \\ & = & c \log(n+1) - c + c' + c_1 \end{array}$$

if we pose $c \geq c_1$ we get the result. Because $T(n) \leq c \log(n+1) + c'$ we have $T(n) = O(\log n)$.

Exercice 22

Find a bound for the recurrence:

$$T(n) = \begin{cases} 7T(n/7) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Iterative Version of Binary Search

A non-recursive version of binary search is given below (the complexity is the same, but faster in practice):

```
int binary_search(int key, const vector<int>& v)
{
  int lo = 0;
  int hi = v.size() - 1;
  while (lo <= hi)
  {
    int mid = lo + (hi - lo) / 2;
    if      (key < v[mid]) hi = mid - 1;
    else if (key > v[mid]) lo = mid + 1;
    else return mid;
  }
  return -1;
}
```

Exercice 23

Try to solve the problem:

UVa Online Judge (http://uva.onlinejudge.org) problem number 11057

Summary on Sublinear Time

The binary search algorithm is an example of a class of algorithms where:

- We **successively reduce the size of the input by a given factor** (two for binary search) in each iteration.
- **In each iteration we do a constant amount of work**.

Those kind of algorithms give rise to a **logarithmic complexity**.

Linear Time

An algorithm that runs in $O(n)$, or **linear**, time has a very natural property: **its running time is at most a constant factor times the size of the input**.

One basic way to get an algorithm with this running time is to process the input in a single pass, spending a constant amount of time on each item of input encountered. Other algorithms achieve a linear time bound for more subtle reasons.

To illustrate some of the ideas here, we consider two simple linear time algorithms and a more complexe one as examples.

Linear Time Algorithm Example 1

Computing the maximum: Computing the maximum of n numbers, can be done in the basic "one pass style".

```
int get_maximum(const vector<int>& v)
{
  int n = v.size();
  int max = v[0];
  for (int i = 1; i < n; i++)
  {
    if (max < v[i])
    {
      max = v[i];
    }
  }
  return max;
}
```

In this algorithm we do constant time per element of the array for a total running time of $O(n)$.

Linear Time Algorithm Example 2

Merging two sorted arrays: We are given two arrays of numbers a_1 and a_2, such that:

- Each is already arranged in ascending order.
- The total size of the two arrays is n.

We would like to merge these into a single array also arranged in ascending order. For example, merging the arrays:

```
2, 3, 11, 19
```

and

```
4, 9, 16, 25
```

results in the output:

```
2, 3, 4, 9, 11, 16, 19, 25.
```

Linear Time Algorithm Example 2

```
vector<int> merge(const vector<int>& v1, const vector<int>& v2)
{
  vector<int> res;
  int n1 = v1.size(), n2 = v2.size();
  int index1 = 0, index2 = 0;
  while (index1 < n1 && index2 < n2)
  {
    if (v1[index1] <= v2[index2]) res.push_back(v1[index1++]);
    else res.push_back(v2[index2++]);
  }
  while (index1 < n1) res.push_back(v1[index1++]);
  while (index2 < n2) res.push_back(v2[index2++]);
  return res;
}
```

An element of each input vector is added only one time to the output `res` vector and because we do constant time per element, we get a total running time of $O(n)$.

$O(n \log n)$ Time

$O(n \log n)$ is a very common running time, and in the *Divide and Conquer* section we will see one of the main reasons for its prevalence: it is the running time of any algorithm that splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two solutions in linear time.

Sorting is perhaps the most well-known example of a problem that can be solved this way. The **merge sort** algorithm given below is an example of such an algorithm.

```
void sort(vector<int>& v, vector<int>& aux, int lo, int hi)
{
  if (hi <= lo) return;
  int mid = lo + (hi - lo) / 2;
  sort(v, aux, lo, mid);
  sort(v, aux, mid + 1, hi);
  merge(v, aux, lo, mid, hi)
}

void sort(vector<int>& v)
{
  vector<int> aux(v.begin(), v.end());
  sort(v, aux, 0, v.size() - 1);
}
```

$O(n \log n)$ Time

```
void merge(vector<int>& v, vector<int>& aux, int lo, int mid, int hi)
{
  // copy to aux
  for (int k = lo; k <= hi; k++) {
    aux[k] = v[k];
  }
  // merge back to v
  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++) {
    if      (i > mid)           v[k] = aux[j++];
    else if (j > hi)            v[k] = aux[i++];
    else if (aux[j] < aux[i])   v[k] = aux[j++];
    else                        v[k] = aux[i++];
  }
}
```

We also frequently encounters $O(n \log n)$ as a running time simply because there are many algorithms whose most expensive step is to sort the input.

Exercice 24

We are given a set of n timestamps $t_1, t_2, \ldots, t_n$ on which copies of a file arrived at a server, and we would like to find the largest interval of time between the first and last of these timestamps during which no copy of the file arrived.

Propose an $O(n \log n)$ algorithm to solve this problem.

Quadratic Time

Here's a basic problem: Suppose you are given n points in the plane, each specified by (x, y) coordinates, and you would like to find the pair of points that are closest together. The natural brute-force algorithm for this problem would, enumerate all pairs of points, compute the distance between each pair, and then choose the pair for which this distance is smallest:

```
pair<Point, Point> closest_points(const vector<Point>& points)
{
  auto res = make_pair(Point(0, 0), Point(0, 0));
  double best_so_far = numeric_limits<double>::max();
  for (int i=0; i < points.size(); ++i)
    for (int j=i+1; j < points.size(); ++j)
    {
      double d = Point::distance(points[i], points[j]);
      if (d < best_so_far)
      {
        best_so_far = d;
        res = make_pair(points[i], points[j]);
      }
    }
  return res;
}
```

Quadratic Time

In this algorithm, we iterate over all distinct pairs of points. There is $\binom{n}{2}$ where n is the number of points. This quantity is bounded by $\frac{n^2}{2}$ so the compexity of this algorithm is in $O(n^2)$.

Quadratic time arises naturally from a pair of nested loops: An algorithm consists of a loop with O(n) iterations, and each iteration of the loop launches an internal loop that takes O(n) time. Multiplying these two factors of n together gives the $O(n^2)$ running time.

Note that the previous algorithm is not the best one for solving this problem.

Cubic Time

More elaborate sets of nested loops often lead to algorithms that run in $O(n^3)$ time. Consider, for example, the following problem:

We are given a list of n vectors $v_1, v_2, \ldots, v_n$, each of which can contain up to n elements. We would like to know if a given vector v is disjoint from at least one of the v_i. The following algorithm solves this problem:

```
bool disjoint(const vector<int>& v1, const vector<int>& v2)
{
  for (int i1 : v1)
    for (int i2 : v2)
      if (i1 == i2) return false;
  return true;
}
bool exists_disjoint(const list<vector<int>>& l, const vector<int>& v)
{
  for (const auto& vi : l)
    {
      if (disjoint(vi, v)) return true;
    }
  return false;
}
```

Cubic Time

Each of the vectors has maximum size $O(n)$, so the innermost loop of the `disjoint` fonction takes time $O(n)$. Looping over the elements of the `vector v1` involves $O(n)$ iterations around this innermost loop; and looping again over the elements of the `list l` involves $O(n)$ iterations around this. Multiplying these three factors of n together, we get the running time of $O(n^3)$.

Cubic time arises naturally from three nested loops each of complexity $O(n)$.

Exercice 25

Propose a better algorithm for the preceding problem that runs in $O(n^2 \log n)$ time.

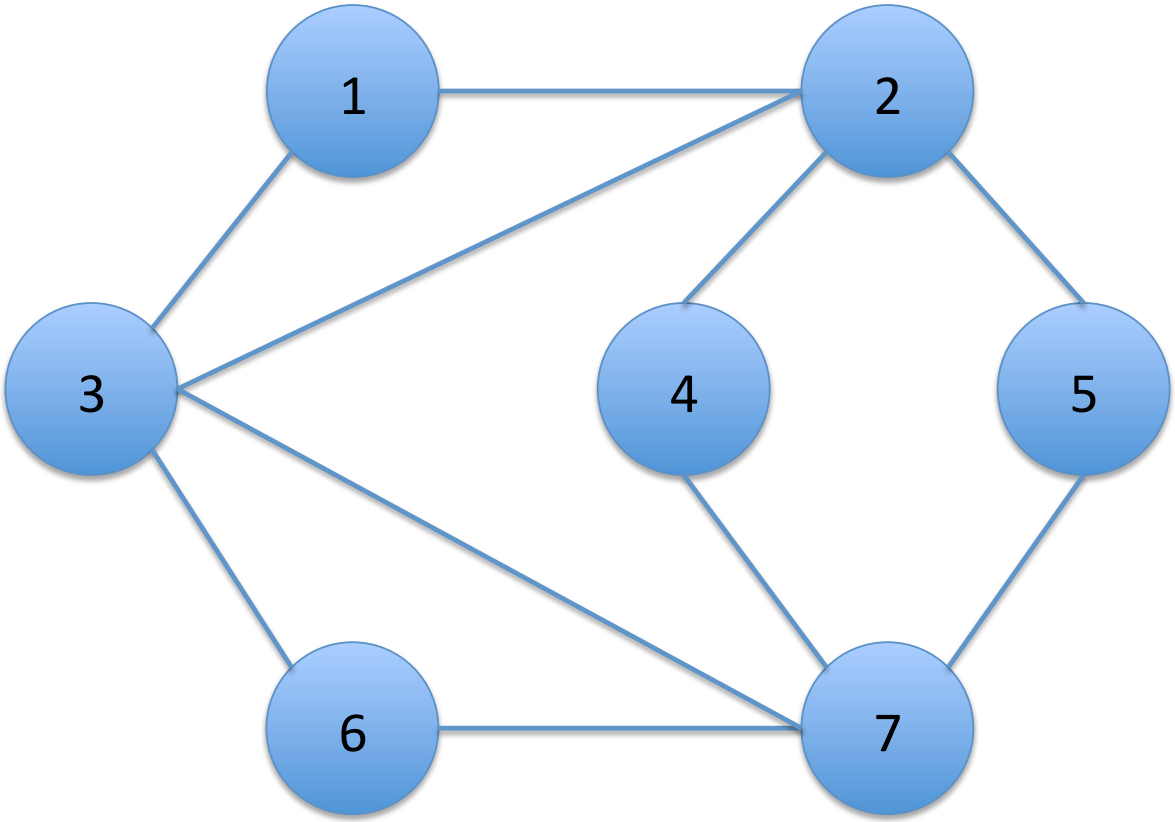
$O(n^k)$ Time

In the same way that we obtained a running time of $O(n^2)$ by performing brute-force search over all pairs formed from a set of n items, **we obtain a running time of $O(n^k)$ for any constant k when we search over all subsets of size k.**

To illustrate this complexity, we consider the **Independent Set problem**. Given a graph $G = (V, E)$, we say a set of nodes $S \subset V$ is independent if no two nodes in S are joined by an edge. The Independent Set Problem is then the following: Given G, find an independent set that is as large as possible. For example, the maximum size of an independent set in the graph in the next slide is four, achieved by the four-node independent set $\{1, 4, 5, 6\}$.

We use a variant of this problem where we fixe the size k of the independent set we seek.

$O(n^k)$ Time



$O(n^k)$ Time

A pseudo-code solution of the independent set problem of size k (k is fixed) is given below:

```
Foreach subset S of k nodes
  Check whether S constitutes an independent set
  If S is an independent set then
    Stop and declare success
  Endif
Endforeach
If no k-node independent set was found then
  Declare failure
Endif
```

Exercice 26

Give the complexity of our solution to the independent set problem of size k.

$O(n^k)$ Time

Independent Set is an example of a problem believed to be computationally hard, and in particular it is believed that no algorithm to find k-node independent sets in arbitrary graphs can avoid having some dependence on k in the exponent.

Beyond Polynomial Time

If we are given a graph and want to find an **independent set of maximum size** (rather than testing for the existence of one with a given number of nodes), nobody knows an algorithm that improve significantly on brute-force search, which in this case would look as follows:

```
Foreach subset S of nodes
  Check whether S constitutes an independent set
  If S is a larger independent set than the largest seen so far then
    Record the size of S as the current maximum
  Endif
Endforeach
```

Exercice 27

Give the complexity of the pseudo-code for the independent set of maximum size problem.

Beyond Polynomial Time

The $O(2^n)$ complexity arises naturally as a running time for a search algorithm that must consider all subsets.

An other common complexity that is worst than $O(2^n)$ is $O(n!)$. **Search spaces of size $n!$ tend to arise for one of two reasons:**

- First, $n!$ is the number of ways to match up n items with n other items (as the search space size of the *Stable Matching Problem*).
- Second, $n!$ also arises in problems where the search space consists of all ways to arrange n items in order (as the search space size of the *Traveling Salesman Problem*).

High Level Approach to Problem Solving

In this section, we present **Polya's four-point plan for problem-solving** drawn form G. Polya's book: *How to Solve it*.

We illustrate its problem-solving method with mathematical problems, but those ideas can be applied to computer science and other domains as well.

Microsoft used to give Polya's book to all of its new programmers.

Three Examples

We illustrate the method with these three problems:

1. How many zeroes are at the end of 1000!.
2. Suppose that X and Y are two finite sets. Find a formula that relates $|X|$, $|Y|$, $|X \cup Y|$ and $|X \cap Y|$.
3. Show that the equation $x^2 + y^2 = z^n$ has positive integer solutions for every $n = 1, 2, 3, \ldots$.

Polya's Four Steps Plan

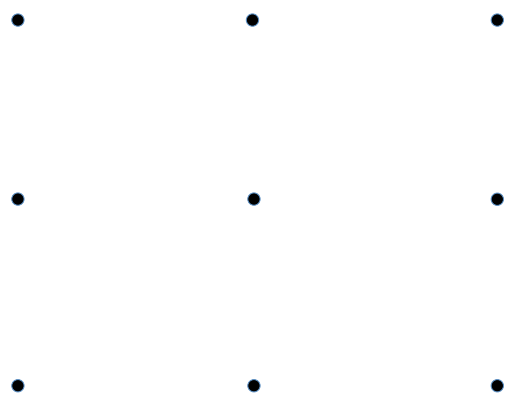
For solving any type of problem, Polya gives a four-point plan:

1. Understand the problem.
2. Devise a plan.
3. Execute the plan.
4. Look back.

Understand All the Words and Symbols in the Problem

You have to cleary understand the problem description. The key to a solution may be to use the properties of some particular definition. If you don't know that the definition provides that property, then you are unlikely to solve the problem. So **ensure you know and understand all the necessary definitions and constraints of the problem description**.

For example, consider the following problem: You have to connect all nine points below with an unbroken path of four straight lines.



Nothing in the problem description constrains you to stay within the bounds of the square. When you realize this, the problem is easy to solve.

Draw a Picture

The human mind is very good at working with images. Pictures are excellent for understanding the problem and developing intuition about a problem and subsequently suggesting a solution to it. In fact, a diagram is often essential in problems from geometry or physics.

In the case of the second problem drawing a Venn diagram is very illuminating.

Devising a Plan

Once the problem is understood construct a plan to solve it.

The following slides give some general ideas to devise a plan to tackle the problem.

Work with Initial and Special Cases

Many problems will have some form of index. For example in the third problem the index is n.

It can be any natural number and we have to find integers x, y and z for every possible n.

In these indexed problems you should try to solve the problem in the initial cases, e.g. $n = 1$, 2 and 3. This will not necessarily give you the general answer but allows insight and a "feel" for the problem:

- If $n = 1$ we have to find solutions to $x^2 + y^2 = z$. This is easy in this case.
- For $n = 2$ we have to solve $x^2 + y^2 = z^2$. This is the famous Pythagoras equation, for which we have many solutions.
- For $n = 3$ we have to solve $x^2 + y^2 = z^3$. Because it seems to be less obvious, we can try to use the $n = 2$ case to provide a solution.

Work with a Concrete Case

In a similar way to the idea on the previous slide, for an abstract problem look at a concrete case.

In a problem concerned with sets take a specific set. So for the second problem take a set where $X = \{a, b, c, d\}$ and $Y = \{b, d, e, f, g\}$. In this case we see that:

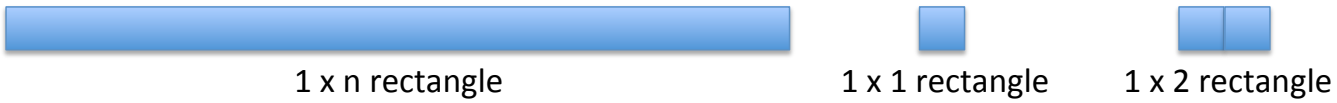
- $|X| = 4$, $|Y| = 5$.
- $|X \cup Y| = |\{a, b, c, d, e, f, g\}| = 7$ and $|X \cap Y| = |\{b, d\}| = 2$.

So we need a formula that relates 4, 5, 2 and 7. Play around with some other examples if you don't see a pattern. Vary X and Y by an element or two and see how the cardinalities change.

The examination of specific cases is intended to provide insight and frequently unlocks the problem.

Exercice 28

How many ways can a $1 \times n$ rectangle be filled by nonoverlapping 1×1 and 1×2 rectangles?



Solve an Easier Problem

This is similar to working with a special case. **With the experience from solving an easier problem it might be possible to solve the general one**.

So for example, in the first problem it may be easier to work with 10! rather than 1000!. Try it and see what happens. It may suggest some ideas. In the third problem consider what happens if z is odd or even.

Work Backwards and Forwards

In finding the solution, you do not have to begin with the assumptions and work towards the conclusion.You can start with the conclusion and try to think what this would imply.

In this way one can work towards the assumptions. Once the solution becomes clear rewrite with the assumptions going to the conclusion.

Work Backwards and Forwards

For example, if 1000! has exactly n zeroes at the end what does that imply?

It means that $1000! = y \times 10n$ for some y where y does not end in a zero. Thus, from 1000! we need to take out factors of 10.

Now we look at 1000! and see from the definition how factors of 10 might arise.

Proceeding in this way, i.e. backwards and forwards, we can produce an argument that "meets in the middle".

Think About a Similar Problem

The way to become good at solving problems is to solve problems. Similar problems often have similar solutions, so consider problems with similar assumptions or conclusions that you have solved before and see if the same method will work.

In the $n = 2$ case of second problem there is a way of constructing an infinite number of solutions; these solutions are called Pythagorean triples. It may be worth investigating these to see if the method of finding Pythagorean triples can be adapted to the solution of our problem.

Find an Equivalent Problem

Reformulating the problem can be helpful. For instance, suppose that the problem was to show that two functions were equal. Suppose that f and g are functions of a real variable. Define a new function h by:

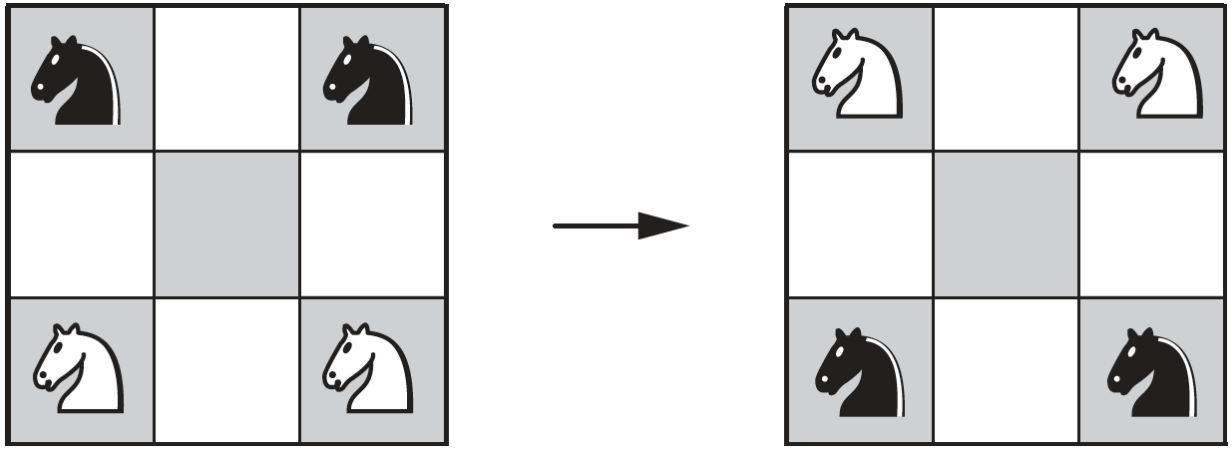
$$h(x) = f(x) - g(x)$$

Then "$f(x) = g(x)$ for all x" is equivalent to "$h(x) = 0$ for all x".

Now look for theorems with the conclusion that a function is the zero function, maybe that will give the answer.

Exercice 29

There are four knights on the 3×3 chessboard: the two white knights are at the two bottom corners, and the two black knights are at the two upper corners of the board. The goal is to switch the knights in the minimum number of moves so that the white knights are at the upper corners and the black knights are at the bottom corners.



Break the Problem Into Pieces

Another approach is to **break the problem into pieces; the hope is that each piece is an easier problem**, preferably a mere exercise. Hence, divide the problem into as many natural cases as is possible.

Now consider the third problem; it involves natural numbers. The set of natural numbers can be divided into two: even numbers and odd numbers. Let's look at the case of even numbers, that is, $n = 2m$ for some natural number m. The equation becomes:

$$x^2 + y^2 = z^{2m} = (z^m)^2$$

Thus we have something of the form $x^2 + y^2 = w^2$ (where $w = z^m$). This looks like the case where $n = 2$. Maybe we can use the solution we got in that case to help us.

Executing the Plan

Once the plan has been made, it needs to be carried out. By now you should have a good idea of why a result is true, next work out precisely what convinced you and polish that.

Check each step carefully and ensure that it is justified. Any argument is only as strong as its weakest link. If only one small step is incorrect, then the whole argument is false. Now is the time to avoid using intuition. If you think something is true, then prove it beyond doubt by using small steps of logic.

Looking Back

Even though you have produced a solution, the problem-solving plan is not finished:

- Test your solution.
- Reflect on what you have done.

Check the Answer

Testing your answer is important. First, does the answer make sense. Is it of the right order? If you have calculated that your car is travelling at a million miles an hour, or that the Sun is 299 km from the Earth, then it is unlikely that your answer is correct.

If you have at your disposal correct answers for a given set of data, test your solution on those data.

Simple tests are often available. For example:

- The angles in a triangle should add up to 180 degrees.
- Another simple test involves the idea of parity. This is easier to see in action than give a good definition. The equation $51^2 = 34^2 + 46^2$ is false. This is easy to see without any calculation: The left-hand side is odd and the right-hand side is even. That's parity!

Find Another Solution

Even though a solution has been found there may be a better one, so attempt to **solve the problem in a different way**.

Compare yours with model solutions, if they are available.

Reflect

Reflection really pays in solving problems. Think about what solved the problem and ask questions:

- How is the solution similar to others?
- How is it different?
- Was a certain theorem or technique used and does it keep getting used?

Think about what keeps cropping up and put it in your armoury, ready to attack future problems.

Exercice 30

Design a program to find the number of zeroes at the end of $n!$ where n can be as large as 10^9.
If you solve this problem you can easily solve Code Chef (http://www.codechef.com) problem Factorial.

Exercice 31

Show that the equation $x^2 + y^2 = z^n$ has positive integer solutions for every $n = 1, 2, 3, \ldots$.

Exercice 32

Prove that the product of four consecutive numbers in $\mathbb{N}^*$ cannot be the square of an integer.

Data Structures and Libraries

A data structure is a means of **storing and organizing data**.

Different data structures **have different strengths**.

When designing an algorithm, it is important to pick one that allows for efficient: Insertions, searches, deletions, updates, depending on what your algorithm needs.

We present in this section, linear and non-linear data structures with related problems.

Array

Arrays allow **O(1) time access and modification to any elements**. You can use C-style arrays:

```
int a[1024];
```

or the class `array` in the standard library:

```
array<int, 1024> a;
```

C-style arrays and `std::array` are not resizable.

Array Example

The following example illustrate some simple use of arrays. We use in this example, some methods provided by the standard library. It is a good practice to look for functionalities already defined in the standard library to avoid reinvented the wheel.

```
#include <iostream>
#include <algorithm>
#include <array>
using namespace std;

struct comparator {
  bool operator() (int i, int j) { return i>j; }
} comp;

template <typename T> inline void print_elements(const T& coll)
{
  for (const auto& elem : coll) {
      cout << elem << '␣';
    }
  cout << endl;
}
```

Array Example

```
...
int main () {
  int myints[] = {10, 20, 30, 40, 50, 60, 70};
  array<int, 10> myarray {};
  copy(myints, myints+7, myarray.begin());
  print_elements(myarray);

  sort(myarray.begin(), myarray.end(), comp);
  print_elements(myarray);

  myarray.fill(42);
  print_elements(myarray);
  return 0;
}
```

The resulting output is:

```
10 20 30 40 50 60 70 0 0 0
70 60 50 40 30 20 10 0 0 0
42 42 42 42 42 42 42 42 42 42
```

Vector

A `vector` models a **dynamic array**. Consequently, their size can change dynamically, with their storage being handled automatically by the container.

Compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Vector Example

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
  vector<int> v(1024);
  v = {1, 2, 3, 4, 5, 4, 3, 2, 1};
  sort(v.begin(), v.end());
  cout << "looking␣for␣a␣3" << endl;
  if (binary_search(v.begin(), v.end(), 3))
    cout << "found!" << endl;
  else
    cout << "not␣found." << endl;
  return 0;
}
```

The resulting output is:

```
looking for a 3
found!
```

Problem Using Vector Example

We solve the following problem UVa Online Judge (http://uva.onlinejudge.org) problem number 11093:

Problem:
Along a circular track, there are N gas stations, which are numbered clockwise from 1 up to N. At station i, there are p_i gallons of petrol available. To race from station i to its clockwise neighbor one need q_i gallons of petrol.
Consider a race where a car will start the race with an empty fuel tank. Your task is to find whether the car can complete the race from any of the stations or not. If it can then mention the smallest possible station i from which the lap can be completed.

Input:
First line of the input contains one integer T the number of test cases. Each test case will start with a line containing one integer N, which denotes the number of gas stations. In the next few lines contain $2 * N$ integers. First N integers denote the values of p_is (petrol available at station i), subsequent N integers denote the value of q_is (amount of patrol needed to go to the next station in the clockwise direction).

Problem Using Vector Example

UVa Online Judge (http://uva.onlinejudge.org) problem number 11093

```
const int MAX_N = 100010;
int petrols[MAX_N];
int main(int argc, char *argv[])
{
  int TC;
  scanf("%d", &TC);
  for (int tc = 1; tc <= TC; ++tc)
  {
    printf("Case␣%d:␣", tc);
    int N;
    scanf("%d", &N);
    for (int i = 1; i <= N; ++i)
    {
      scanf("%d", &petrols[i]);
    }
  //...
}
```

Problem Using Vector Example

UVa Online Judge (http://uva.onlinejudge.org) problem number 11093

```
    //...
    int sum = 0;
    for (int i = 1; i <= N; ++i)
    {
      int d;
      scanf("%d", &d);
      petrols[i] -= d;
      sum += petrols[i];
    }
    if (sum < 0)
    {
      printf("Not possible\n");
      continue;
    }
    //...
}
```

Problem Using Vector Example

UVa Online Judge (http://uva.onlinejudge.org) problem number 11093

:

```
      //...
      sum = 0;
      int res = 1;
      for (int i = 1; i <= N; ++i)
      {
        sum += petrols[i];
        if (sum < 0)
        {
          sum = 0;
          res = i + 1;
        }
      }
      printf("Possible␣from␣station␣%d\n", res);
    }
  return 0;
}
```

Exercice 33

Try to solve the problem (1D Array):

UVa Online Judge (http://uva.onlinejudge.org) problem number 12150

Other problem:

UVa Online Judge (http://uva.onlinejudge.org) problem number 10038

Exercice 34

Try to solve the problem (2D Array):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11040

Other problem:

UVa Online Judge (http://uva.onlinejudge.org) problem number 12398

STL `algorithm`

Two operations are commonly performed on arrays: **sorting** and **searching**. These two operations (and others) are well supported in *C++* in the header `<algorithm>` who defines a collection of functions especially designed to be used on ranges of elements.

Exercice 35

Try to solve the problem (STL `algorithm`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 00146

List

Lists are sequence containers that allow **O(1) time insert and erase operations anywhere within the sequence**, and iteration in both directions.

The main drawback of lists compared to arrays is that they **lack direct access to the elements by their position**; For example, to access an element in a list one has to iterate from a known position (like the beginning or the end) to the position of that element, which takes linear time on average.

List Example

The following example illustrate some simple use of lists.

```
#include <list>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

void print_lists(const list<int>& l1, const list<int>& l2)
{
  cout << "list1:␣";
  copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout, "␣"));
  cout << endl << "list2:␣";
  copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout, "␣"));
  cout << endl;
}
```

List Example

```
...
int main()
{
  // create two empty lists
  list<int> list1, list2;
  // fill both lists with elements
  for (int i=0; i<6; ++i)
  {
    list1.push_back(i);
    list2.push_front(i);
  }
  print_lists(list1, list2);
  // insert all elements of list1 before the first element
  // with value 3 of list2. find() returns an iterator to
  // the first element with value 3
  list2.splice(find(list2.begin(),list2.end(), 3), list1);
  print_lists(list1, list2);
  ...
```

List Example

```
  ...
  // sort second list, assign to list1 and remove duplicates
  list2.sort();
  list1 = list2;
  list2.unique();
  print_lists(list1, list2);
  // merge both sorted lists into the first list
  list1.merge(list2);
  print_lists(list1, list2);
}
```

The resulting output is:

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0
list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0
list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5
list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

Exercice 36

Try to solve the problem (`list`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11988

Stack

The class `stack` implements a stack:

- With **push**, you can **insert** in $\mathbf{O(1)}$ an element at the top of the stack.
- With **pop**, you can **remove** in $\mathbf{O(1)}$ an element from the top of the stack.

This behavior is usually referred to as "Last In First Out" (*LIFO*).

Stack Example

The following example illustrate some simple use of stacks.

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
  stack<int> st;
  // push three elements into the stack
  st.push(1);
  st.push(2);
  st.push(3);
  // pop and print two elements from the stack
  cout << st.top() << '␣';
  st.pop();
  cout << st.top() << '␣';
  st.pop();
  // modify top element
  st.top() = 77;
  ...
```

Stack Example

```
int main()
{
  ...
  // push two new elements
  st.push(4);
  st.push(5);
  // pop one element without processing it
  st.pop();
  // pop and print remaining elements
  while (!st.empty())
  {
    cout << st.top() << '␣';
    st.pop();
  }
  cout << endl;
}
```

The resulting output is:

```
3 2 4 77
```

Exercice 37

Try to solve the problem (`stack`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 01062

Queue

The class queue implements a queue:

- With **push**, you can **insert** in $\mathbf{O(1)}$ an element at the end of the queue.
- With **pop**, you can **remove** in $\mathbf{O(1)}$ an element from the start of the queue.

This behavior is usually referred to as "First In First Out" (*FIFO*). A queue serves as a classic data buffer.

Queue Example

The following example illustrate some simple use of queues.

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
  queue<string> q;
  // insert three elements into the queue
  q.push("These␣");
  q.push("are␣");
  q.push("more␣than␣");
  // read and print two elements from the queue
  cout << q.front();
  q.pop();
  cout << q.front();
  q.pop();
  // insert two new elements
  q.push("four␣");
  q.push("words!");
  ...
```

Queue Example

```
int main()
{
  ...
  // skip one element
  q.pop();
  // read and print two elements
  cout << q.front();
  q.pop();
  cout << q.front() << endl;
  q.pop();
  // print number of elements in the queue
  cout << "number␣of␣elements␣in␣the␣queue:␣" << q.size()
       << endl;
}
```

The resulting output is:

```
These are four words!
number of elements in the queue: 0
```

Dequeue

A `deque` (pronounced "deck") is very similar to a `vector`, a `stack` and a `queue`. It manages its elements with a dynamic array, **provides random access**, and has almost the same interface as a `vector`.

The difference is that with a deque, the dynamic array is open at both ends. **A deque has O(1) amortized running time for insertions and deletions at both the end and the beginning**:

- `push_back(elem)`: Appends a copy of `elem` at the end.
- `pop_back()`: Removes the last element (does not return it).
- `push_front(elem)`: Appends a copy of `elem` at the beginning.
- `pop_front()`: Removes the first element (does not return it).

Dequeue Example

The following example illustrate some simple use of dequeues.

```
#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
  // create empty deque of strings
  deque<string> coll;
  // insert several elements
  coll.assign(3, string("string"));
  coll.push_back("last␣string");
  coll.push_front("first␣string");
  // print elements separated by newlines
  copy(coll.cbegin(), coll.cend(),
       ostream_iterator<string>(cout, "\n"));
  cout << endl;
  ...
```

Dequeue Example

```
int main()
{
  ...
  // remove first and last element
  coll.pop_front();
  coll.pop_back();
  // insert "another" into every element but the first
  for (unsigned i=1; i<coll.size(); ++i)
  {
    coll[i] = "another␣" + coll[i];
  }
  // change size to four elements
  coll.resize (4, "resized␣string");
  // print elements separated by newlines
  copy(coll.cbegin(), coll.cend(),
       ostream_iterator<string>(cout, "\n"));
}
```

Dequeue Example

The resulting output is:

```
first string
string
string
string
last string

string
another string
another string
resized string
```

Exercice 38

Try to solve the problem (queue):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11034

Bitmask

We can use integers to represent a small set of boolean values or a small list or array of constants. All set operations, list or vector manipulation then involve only bitwise operations.

For example:

- We can represent the set of number: $\{29, 1, 5, 12\}$ with the following 32 bits integer:

  00100000000000000001000000100010

- We can represent the following 15-puzzle configuration:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

  with the following 64 bits integer:

```
Most Significant Bits
0000 1111 1110 1101 1100 1011 1010 1001
                 Least Significant Bits
1000 0111 0110 0101 0100 0011 0010 0001
```

Bitmask: Common Operations

We describe in the next slides some common operations on bitmasks.

- **Check if the j-th bit of S is on**: j-th bit is on iff: `S & (1 << j) != 0`.
  For example, if `S` $= 50$ et `j` $= 3$:

```
bit number:          5|4|3|2|1|0

50 =                 1 1 0 0 1 0
(1 << 3) = 8   =     0 0 1 0 0 0
                    -------------
50 & (1 << 3) =      0 0 0 0 0 0

So bit number 3 is off
```

Bitmask: Common Operations

- **Turn on the j-th bit of S**: `S |= (1 << j)`.
  For example, if `S` = 50 et `j` = 3:

```
bit number:          5|4|3|2|1|0

S               =    1 1 0 0 1 0
(1 << 3)        =    0 0 1 0 0 0
                    -------------
S |= (1 << 3)   =    1 1 1 0 1 0
```

- **Turn off the j-th bit of S**: `S &= ~(1 << j)`.
  For example, if `S` = 58 et `j` = 3:

```
bit number:           5|4|3|2|1|0

S               =     1 1 1 0 1 0
~(1 << 3)       =     1 1 0 1 1 1
                     -------------
S &= ~(1 << 3)  =     1 1 0 0 1 0
```

Bitmask: Common Operations

- **Toggle the j-th bit of S**: `S ^= (1 << j)`.
  For example, if `S` = 50 et `j` = 3:

```
bit number:          5|4|3|2|1|0

S               =    1 1 0 0 1 0
(1 << 3)        =    0 0 1 0 0 0
                    -------------
S ^= (1 << 3)   =    1 1 1 0 1 0
```

  If `S` = 58 et `j` = 3:

```
bit number:          5|4|3|2|1|0

S               =    1 1 1 0 1 0
(1 << 3)        =    0 0 1 0 0 0
                    -------------
S ^= (1 << 3)   =    1 1 0 0 1 0
```

Bitmask: Common Operations

- **Get the value of the least significant bit that is on in S**: `S & (-S)`.
  For example, if `S` = 40:

```
bit number:        5|4|3|2|1|0

S              =    1 0 1 0 0 0
-S             =    0 1 1 0 0 0
                   -------------
S & (-S)       =    0 0 1 0 0 0
```

- **To turn on all bits in a set of size n**: `(1 << n) - 1`.
  For example, if `n` = 5:

```
bit number:        5|4|3|2|1|0

(1 << 5)       =    1 0 0 0 0 0
1              =    0 0 0 0 0 1
                   -------------
(1 << 5) - 1   =    0 1 1 1 1 1
```

Bitmask: Common Operations

- **Determine if S is a power of 2**: `(S & (S - 1)) == 0`.
  For example, if `S` = 32:

```
bit number:        5|4|3|2|1|0

S              =   1 0 0 0 0 0
(S - 1)        =   0 1 1 1 1 1
                  -------------
S & (S - 1)    =   0 0 0 0 0 0
```

if `S` = 23:

```
bit number:        5|4|3|2|1|0

S              =   0 1 0 1 1 1
(S - 1)        =   0 1 0 1 1 0
                  -------------
S & (S - 1)    =   0 1 0 1 1 0
```

Bitmask: Common Operations

- **Turn off the last bit set to 1 in S**: `S & (S - 1)`.
  For example, if `S = 101100`:

```
bit number:        5|4|3|2|1|0

S              =   1 0 1 1 0 0
(S - 1)        =   1 0 1 0 1 1
                  -------------
S & (S - 1)    =   1 0 1 0 0 0
```

Exercice 39

How to turn on the last bit set to 0 in a given number `S`?

Exercice 40

Try to solve the problem (one-liner bit manipulation expression):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11173

Exercice 41

Try to solve the problem (`bitmask`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 10264

Bitset

A bitset stores bits (elements with only two possible values: 0 or 1, true or false, ...). The class emulates an array of `bool` elements, but optimized for space allocation: generally, each element occupies only one bit (which, on most systems, is eight times less than the smallest elemental type: char).

Each bit position can be accessed individually: for example, for a given bitset named `foo`, the expression `foo[3]` accesses its fourth bit, just like a regular array accesses its elements.

Note that **you can't change the number of bits in a bitset**. The number of bits is the template parameter. If you need a container for a variable number of bits or Boolean values, you can use the class `vector<bool>`.

Bitset Example

The following example illustrate some simple use of bitsets.

```
#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;
int main() {
  // print some numbers in binary representation
  cout << "267␣as␣binary␣short:␣"
       << bitset<numeric_limits<unsigned short>::digits>(267)
       << endl;

  cout << "267␣as␣binary␣long:␣"
       << bitset<numeric_limits<unsigned long>::digits>(267)
       << endl;

  cout << "10,000,000␣with␣24␣bits:␣"
       << bitset<24>(1e7) << endl;
  ...
}
```

Bitset Example

```
int main()
{
  ...
  // write binary representation into string
  string s = bitset<42>(12345678).to_string();
  cout << "12,345,678␣with␣42␣bits:␣" << s << endl;

  // transform binary representation into integral number
  cout << "\"1000101011\"␣as␣number:␣"
       << bitset<100>("1000101011").to_ullong() << endl;
}
```

Bitset Example

Depending on the number of bits for `short` and `long`, the program might produce the following output:

```
267 as binary short: 0000000100001011
267 as binary long: 00000000000000000000000000000000
                    00000000000000000000000100001011
10,000,000 with 24 bits: 100110001001011010000000
12,345,678 with 42 bits: 000000000000000000101111000110000101001110
"1000101011" as number: 555
```

Exercice 42

Try to solve the problem (`bitset`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11926

Non-Linear Data Structures

For some problems, for example if you need a **dynamic collection of keys or key/value pairs**, linear data structures are not the best way to organize data.

Some non-linear data structures can give you $\mathbf{O}(\log \mathbf{n})$ **insertion, search and deletion of keys or key/value pairs**. The preceding linear data structures cannot garantee those running time for all operations.

We describe in this part, **tree-based data structures** and **hash tables**.

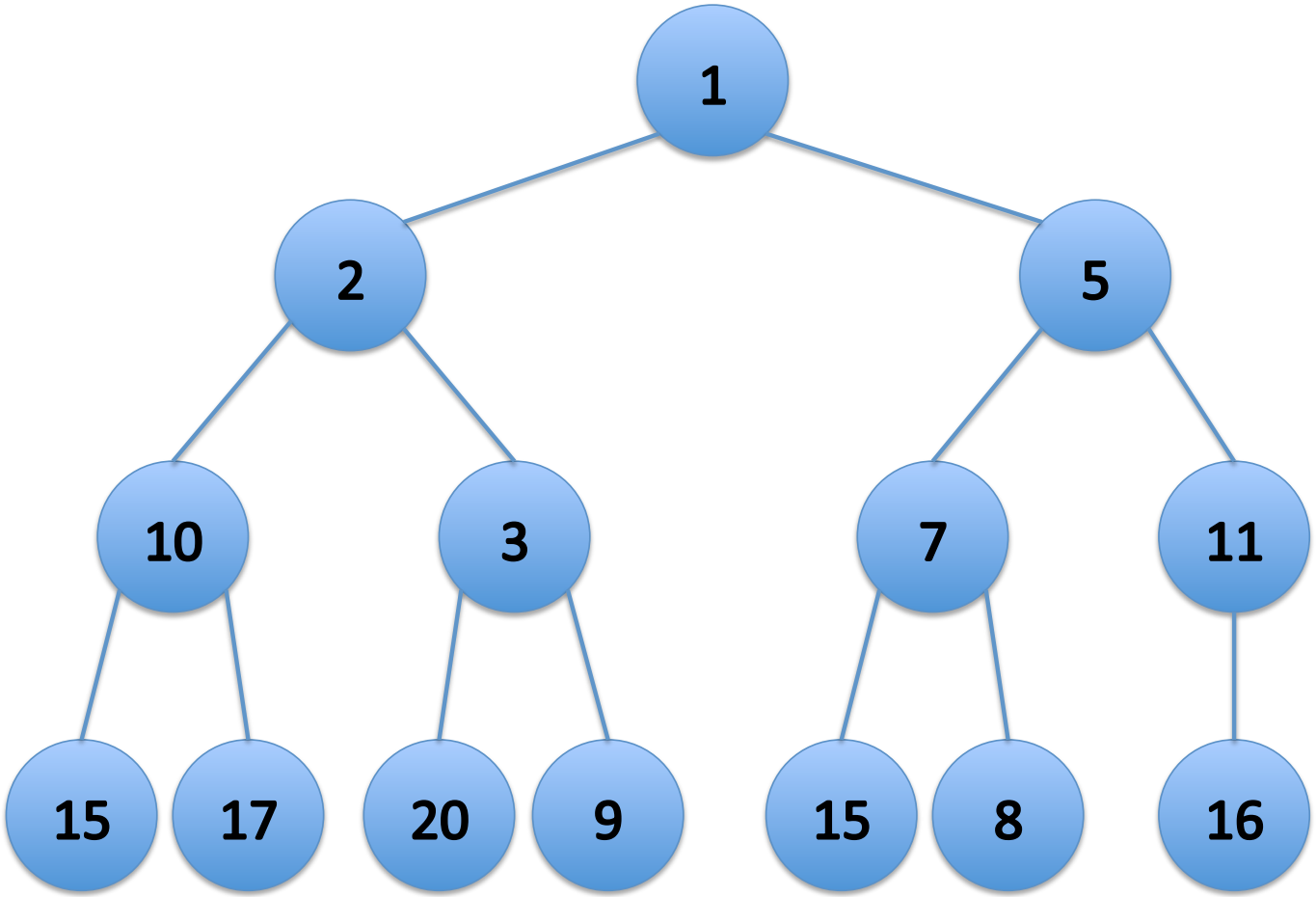
Heap

A **heap** is a tree-based data structure, that contains **keys** (each `key` can be associated with a **value**), based on the heap order property:

Heap order

For every node i, $\texttt{key}(i) \geq \texttt{key}(\texttt{parent}(i))$.

Heap Example



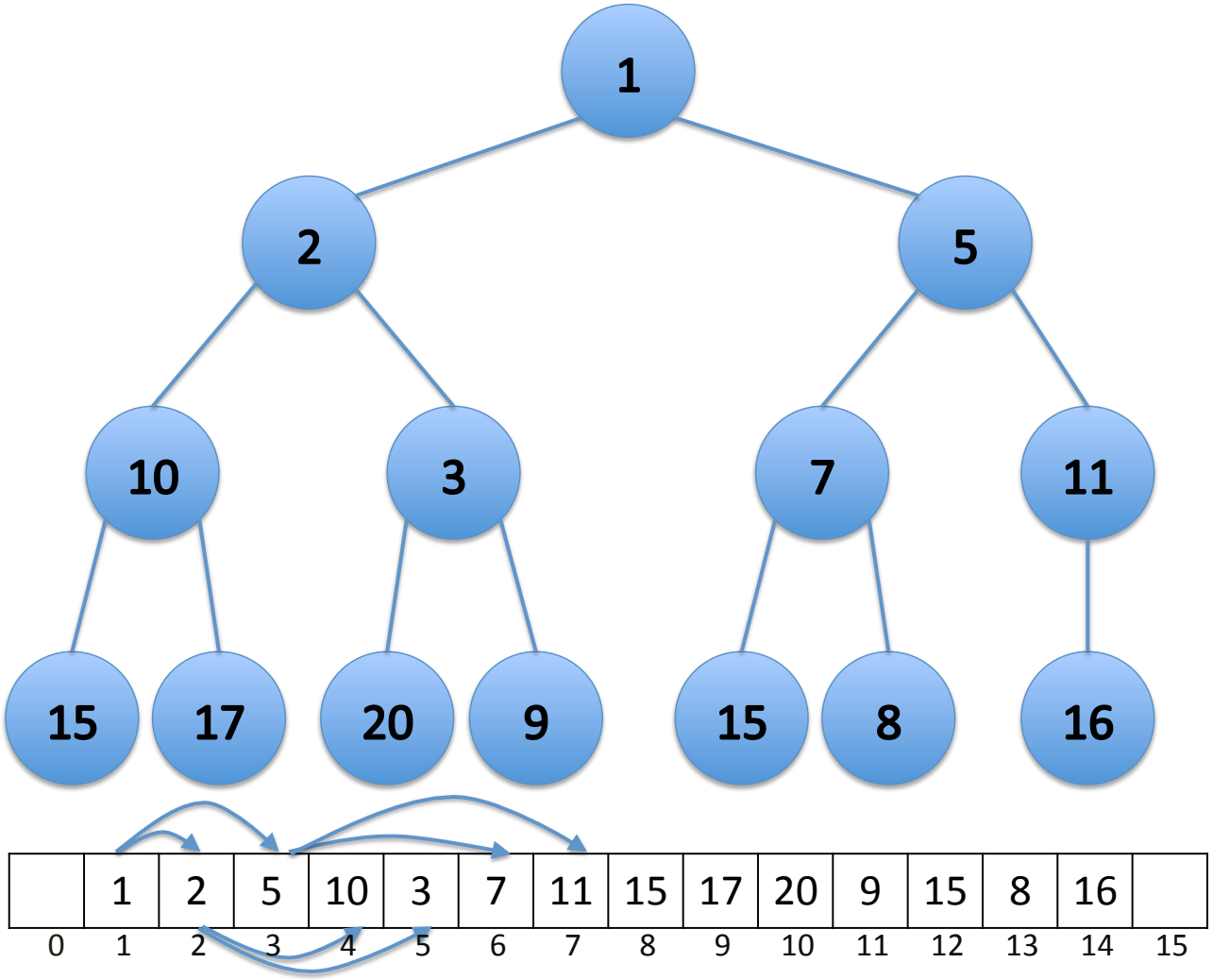
Heap Implementation

To implement a heap, we can avoid using pointers, if a bound N is known in advance on the total number of elements that will ever be in the heap at any one time:

- We can use an array H of size $N + 1$ to represent the heap.
- The heap nodes correspond to the positions in this array.
- $H[1]$ is the root.
- If there is n elements in the heap, they are in position $1 to n$.
- For any node at position i, the children are the nodes at positions:
  - `leftChild(i) = 2i`.
  - `rightChild(i) = 2i + 1`.

  And the parent at position:
  - `parent(i) =` $\lfloor i/2 \rfloor$.

Heap Implementation Example



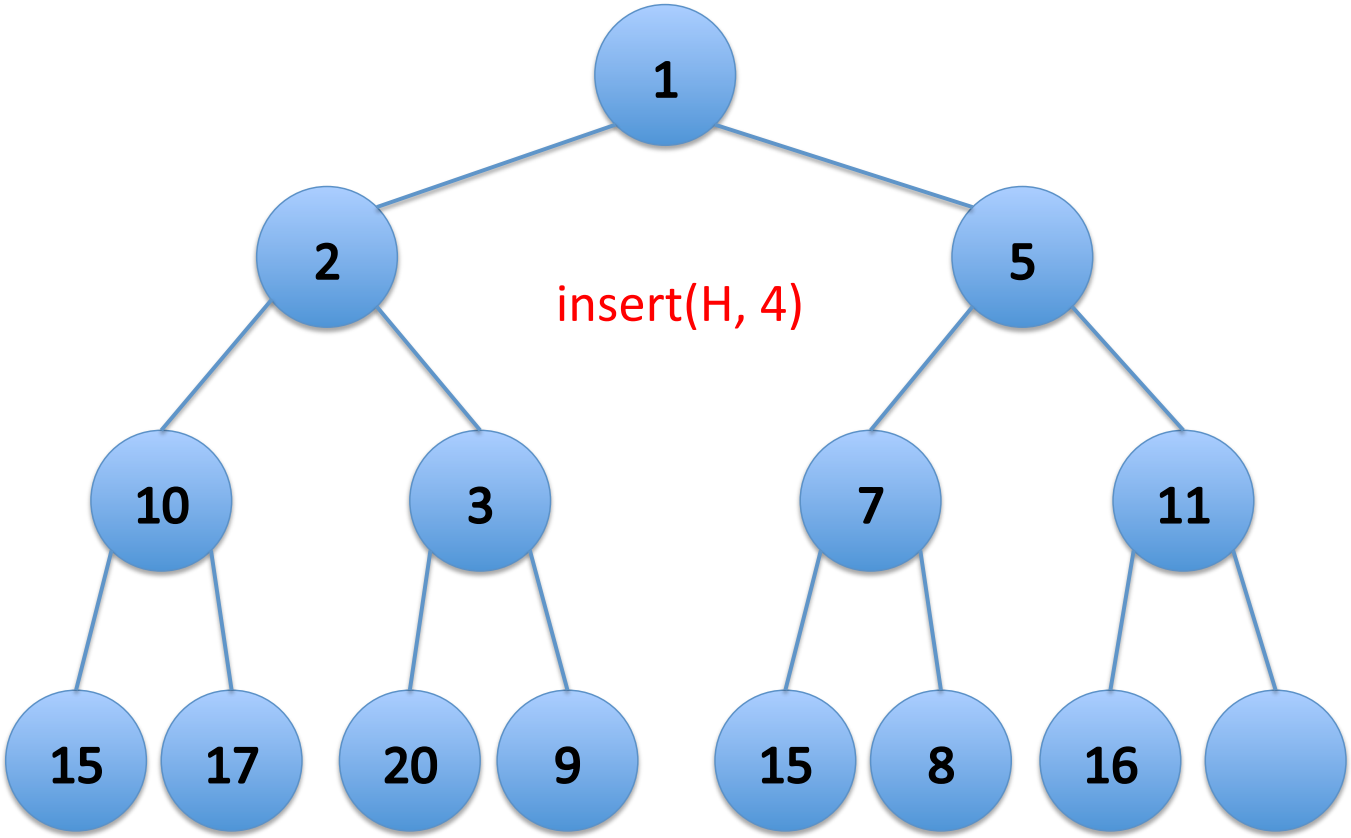
Heap Operations

An heap has, at least, the following operations:

- **insert(H, v)**: insert item `v` (a key or key/value pair) into the heap `H`.
- **findMin(H)**: return the minimum item of the heap `H` but does not remove it.
- **deleteMin(H)**: remove the minimum item of the heap `H`.

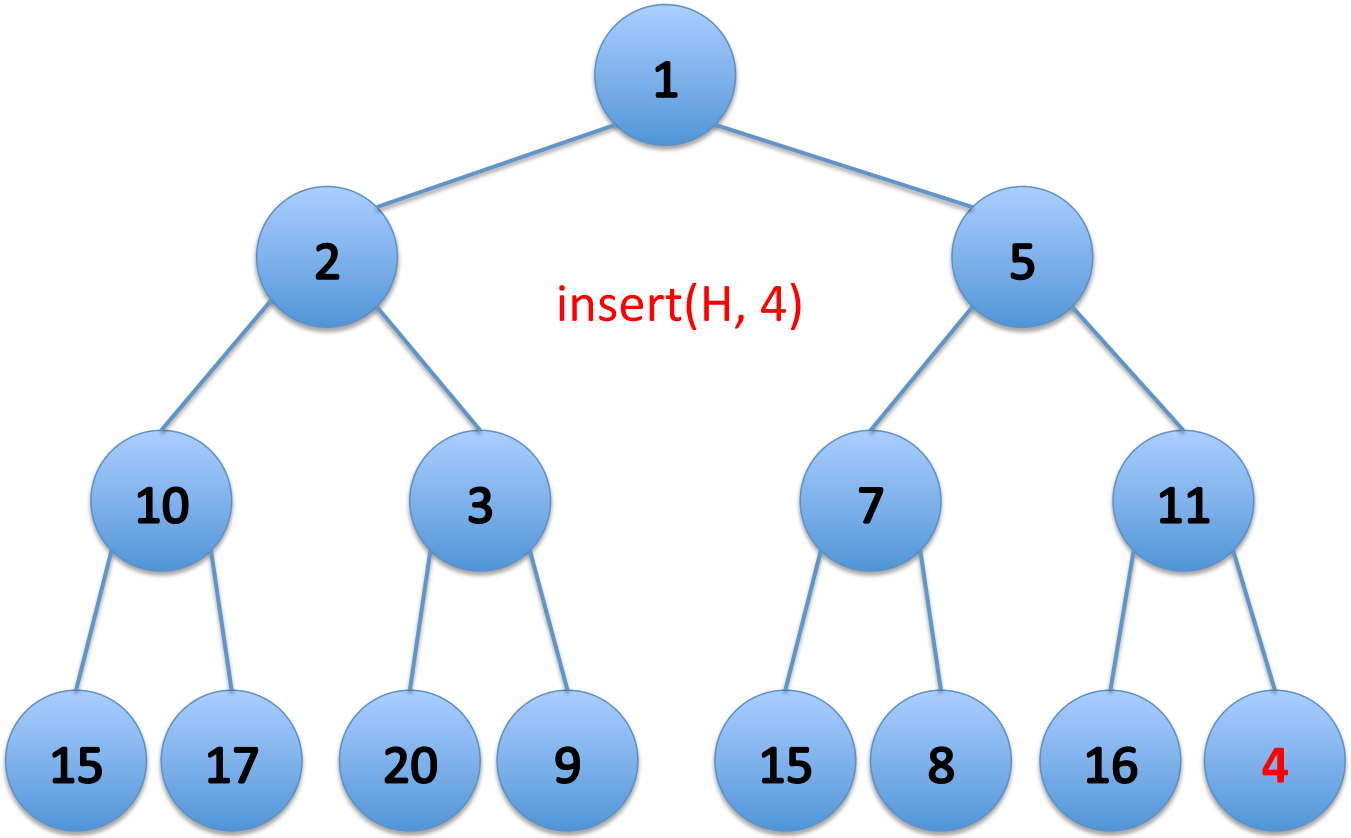
We illustrate, graphically, in the following slides those operations.

Heap Operation: insert



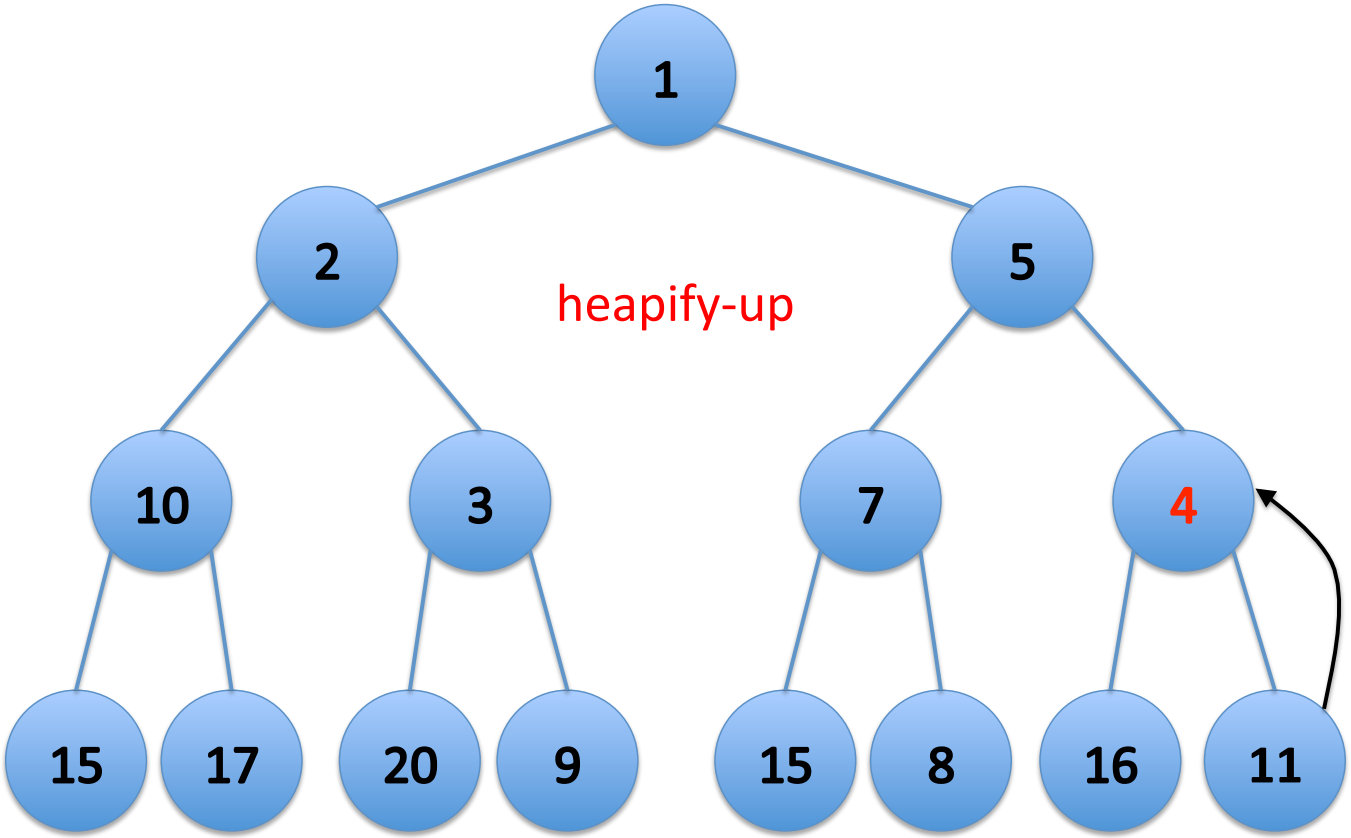
	1	2	5	10	3	7	11	15	17	20	9	15	8	16	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: insert



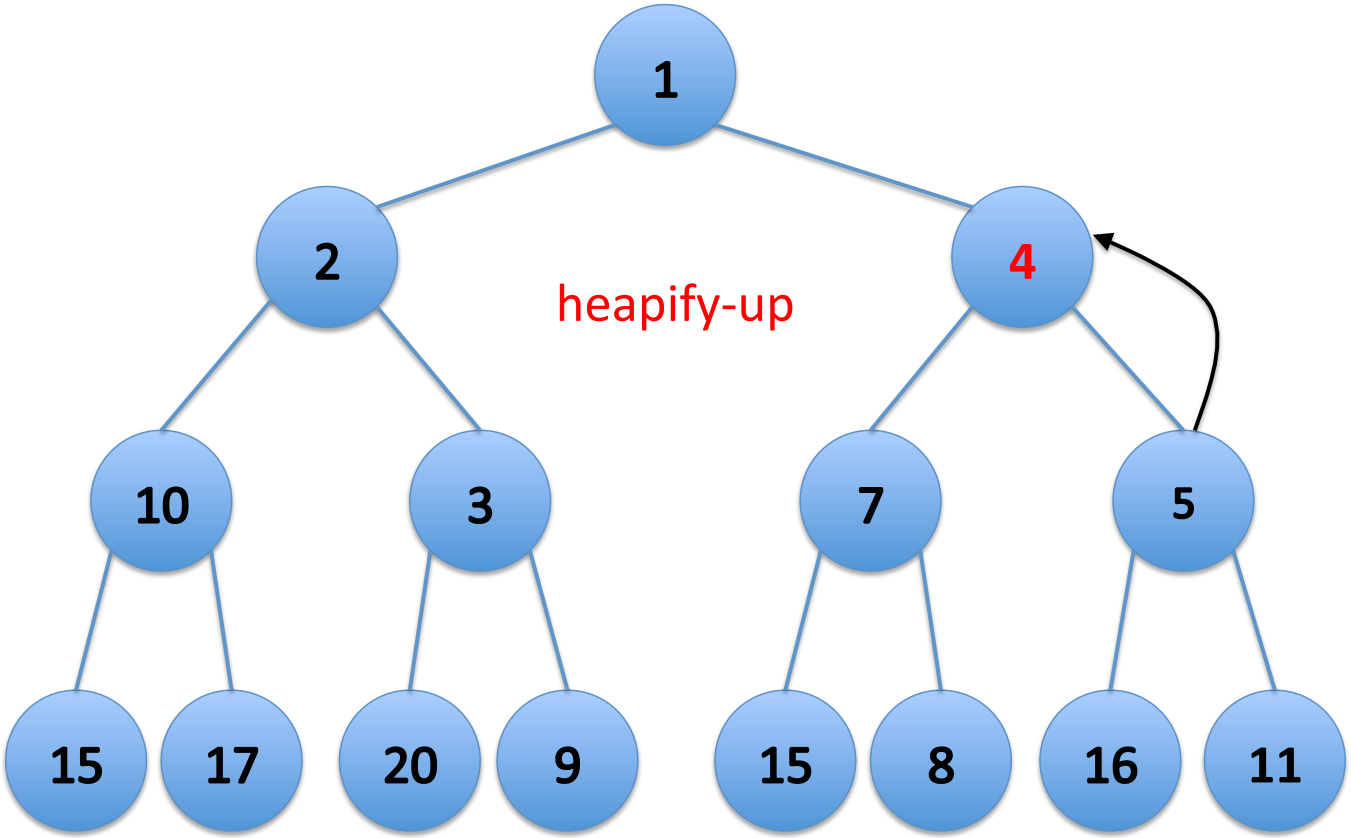
	1	2	5	10	3	7	11	15	17	20	9	15	8	16	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: insert



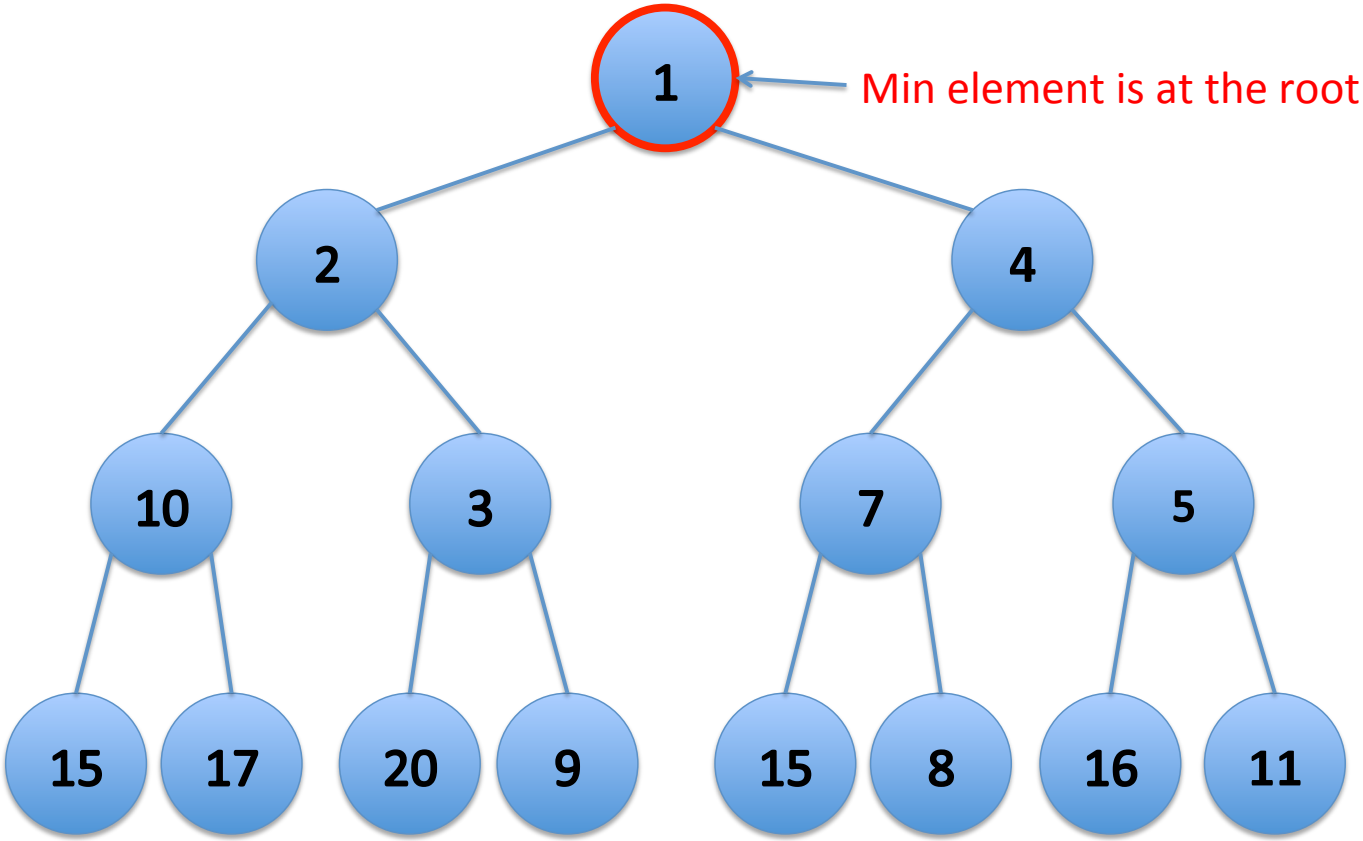
	1	2	5	10	3	7	4	15	17	20	9	15	8	16	11
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: insert



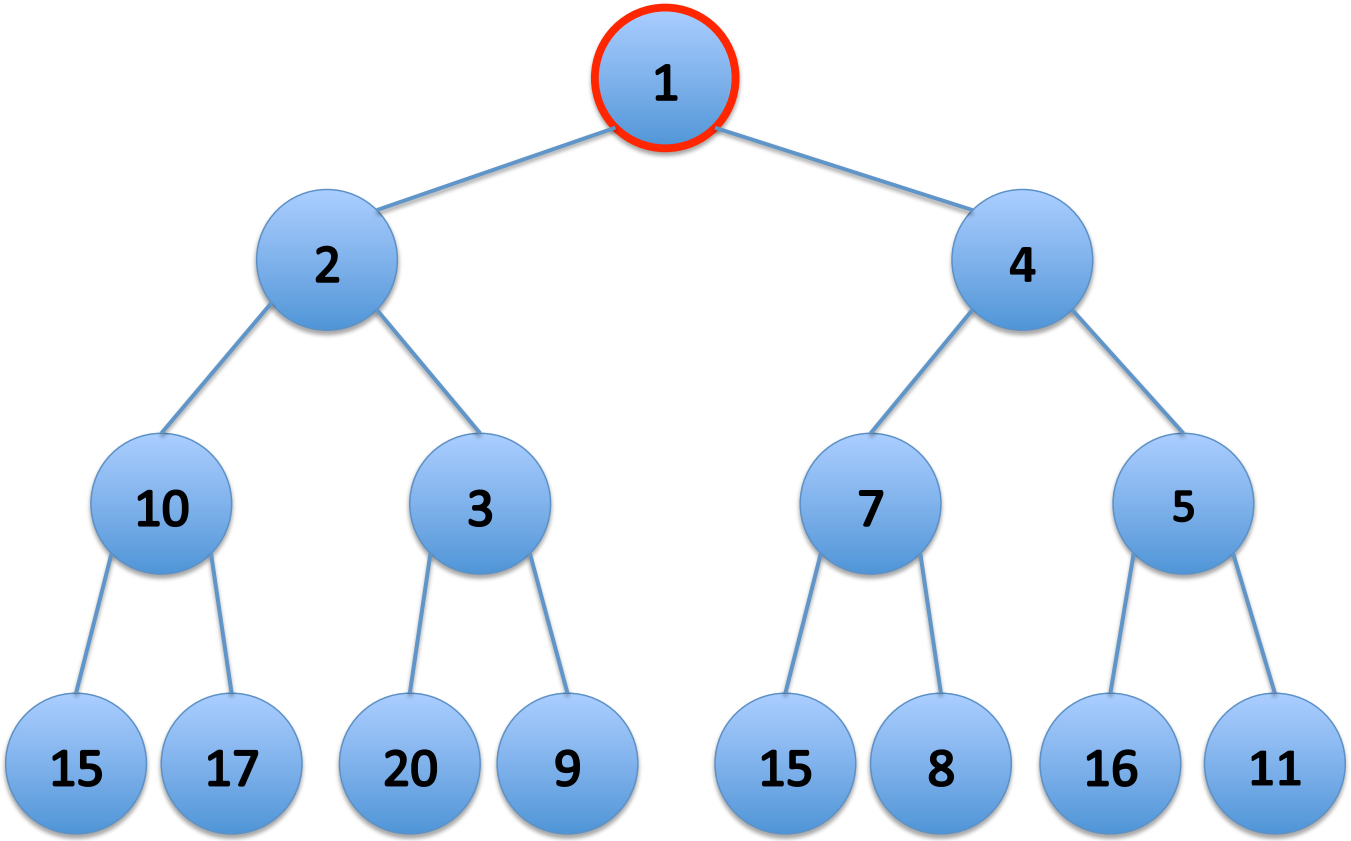
	1	2	4	10	3	7	5	15	17	20	9	15	8	16	11
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: findMin



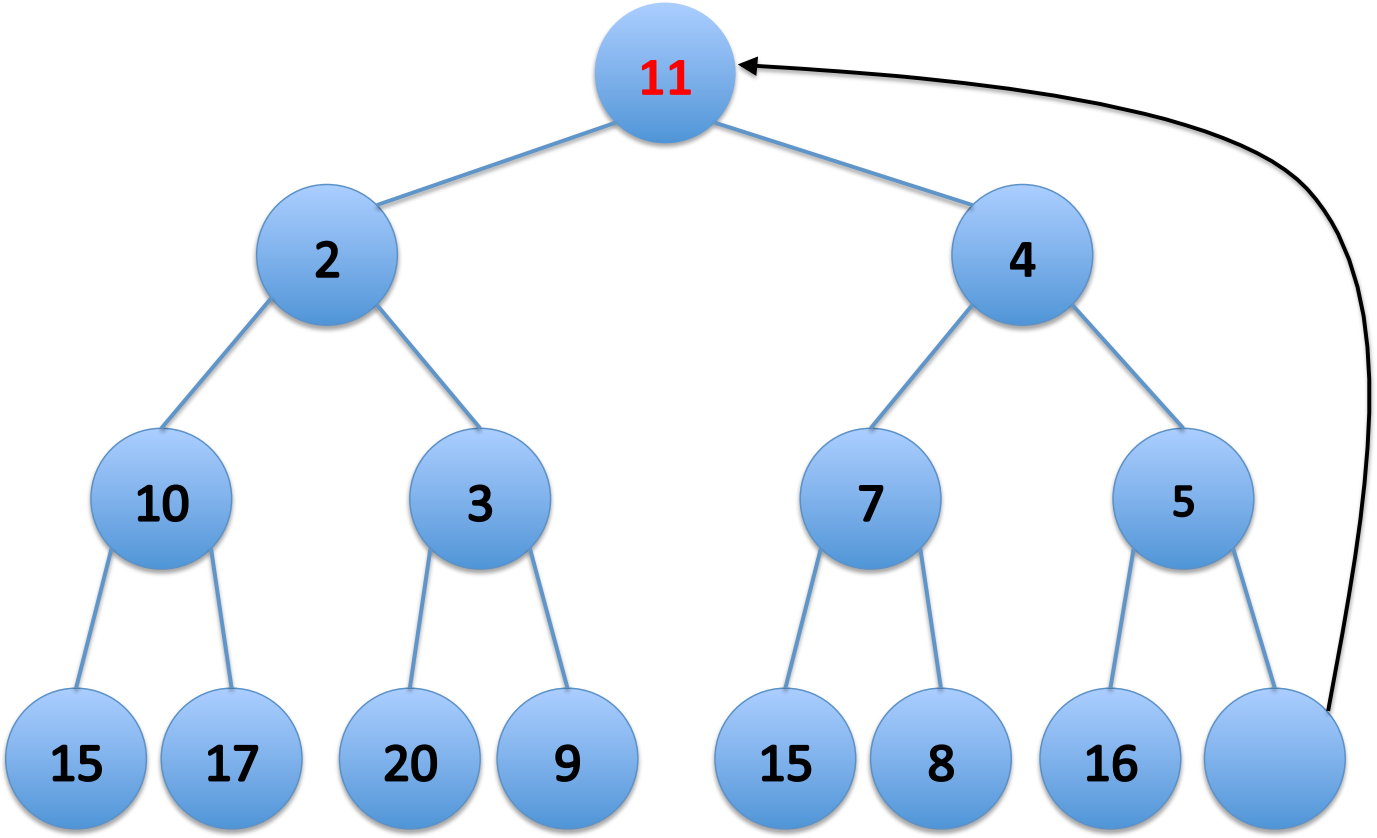
	1	2	4	10	3	7	5	15	17	20	9	15	8	16	11
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: deleteMin



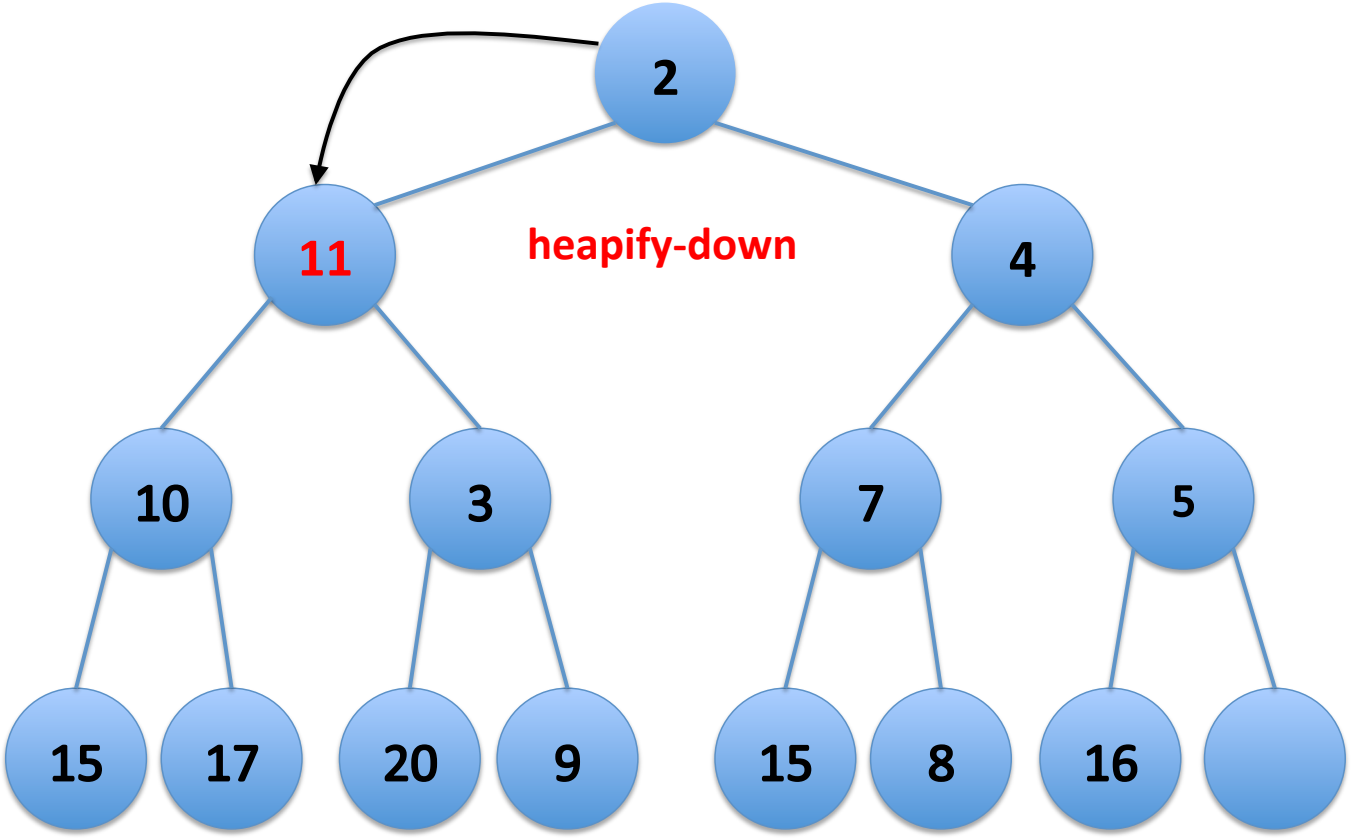
	1	2	4	10	3	7	5	15	17	20	9	15	8	16	11
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: deleteMin



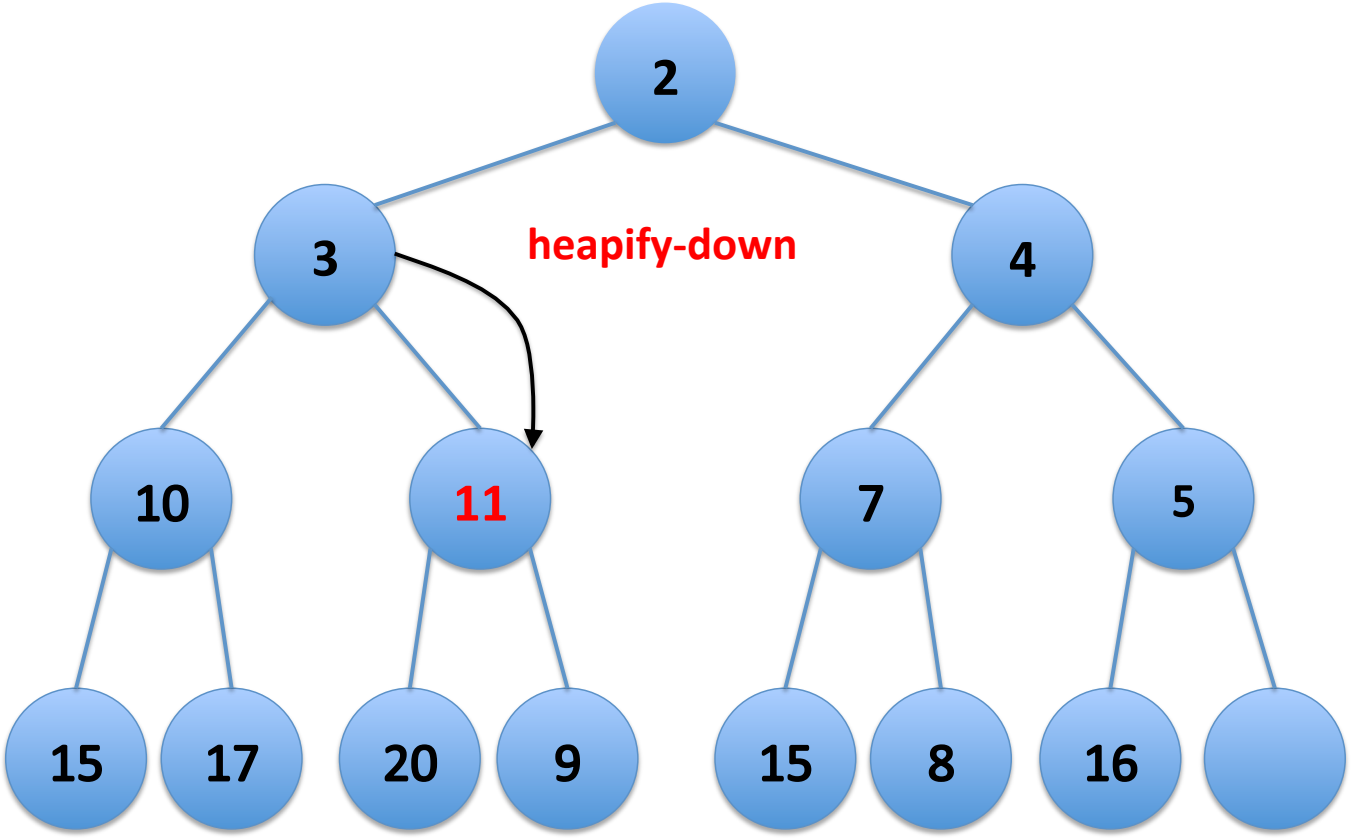
	11	2	4	10	3	7	5	15	17	20	9	15	8	16	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: deleteMin



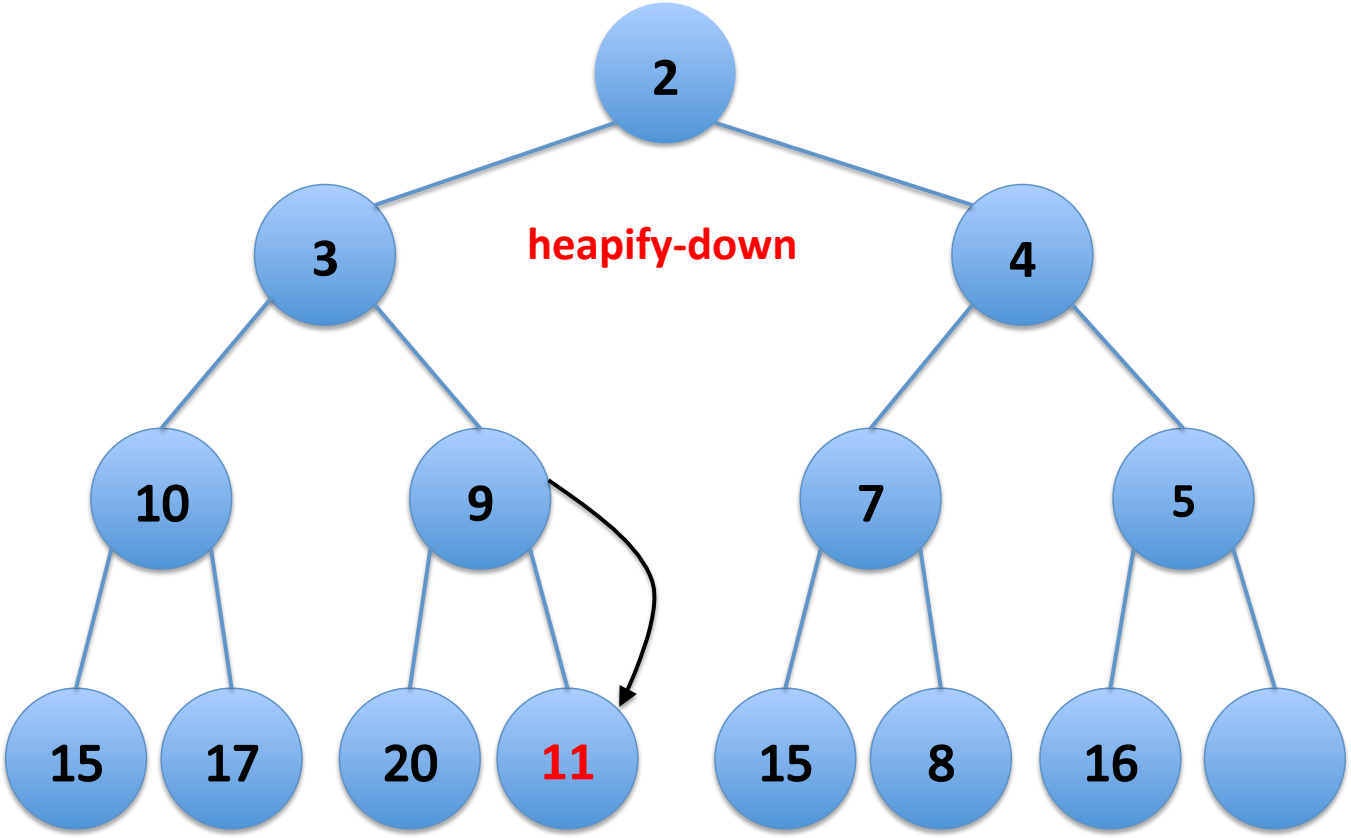
	2	11	4	10	3	7	5	15	17	20	9	15	8	16	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: deleteMin



	2	3	4	10	11	7	5	15	17	20	9	15	8	16	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operation: deleteMin



	2	3	4	10	9	7	5	15	17	20	11	15	8	16	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Operations Complexity

- **insert(H, v)**: $O(\log n)$.
- **findMin(H)**: $O(1)$.
- **deleteMin(H)**: $O(\log n)$.

Priority Queue

The **heap** previously defined can be used to create a **priority queue**.

The class **priority_queue** implements a queue from which elements are read according to their priority:

- **push()** inserts an element into the queue.
- **top()** and **pop()** access and remove the next element. However, the next element is not the first inserted element. Rather, it is the element that has the highest priority:
  - Elements are partially sorted according to their value. As usual, you can provide the sorting criterion as a template parameter. By default, the elements are sorted by using `operator <` in descending order.
  - The next element is always the "highest" element. If more than one "highest" element exists, which element comes next is undefined.

Priority Queue Example

The following example illustrate some simple use of priority queues.

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
  priority_queue<float> q;
  // insert three elements into the priority queue
  q.push(66.6);
  q.push(22.2);
  q.push(44.4);
  // read and print two elements
  cout << q.top() << '␣';
  q.pop();
  cout << q.top() << endl;
  q.pop();
  // insert three more elements
  q.push(11.1);
  q.push(55.5);
  q.push(33.3);
```

Priority Queue Example

```
int main()
{
  ...
  // skip one element
  q.pop();
  // pop and print remaining elements
  while (!q.empty())
  {
    cout << q.top() << '␣';
    q.pop();
  }
  cout << endl;
}
```

The resulting output is:

```
66.6 44.4
33.3 22.2 11.1
```

Exercice 43

Try to solve the problem (`priority_queue`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 1203

Binary Search Tree

Binary Search Tree

A Binary Search Tree is a binary tree in symmetric order.

Binary Tree

A binary tree is either:

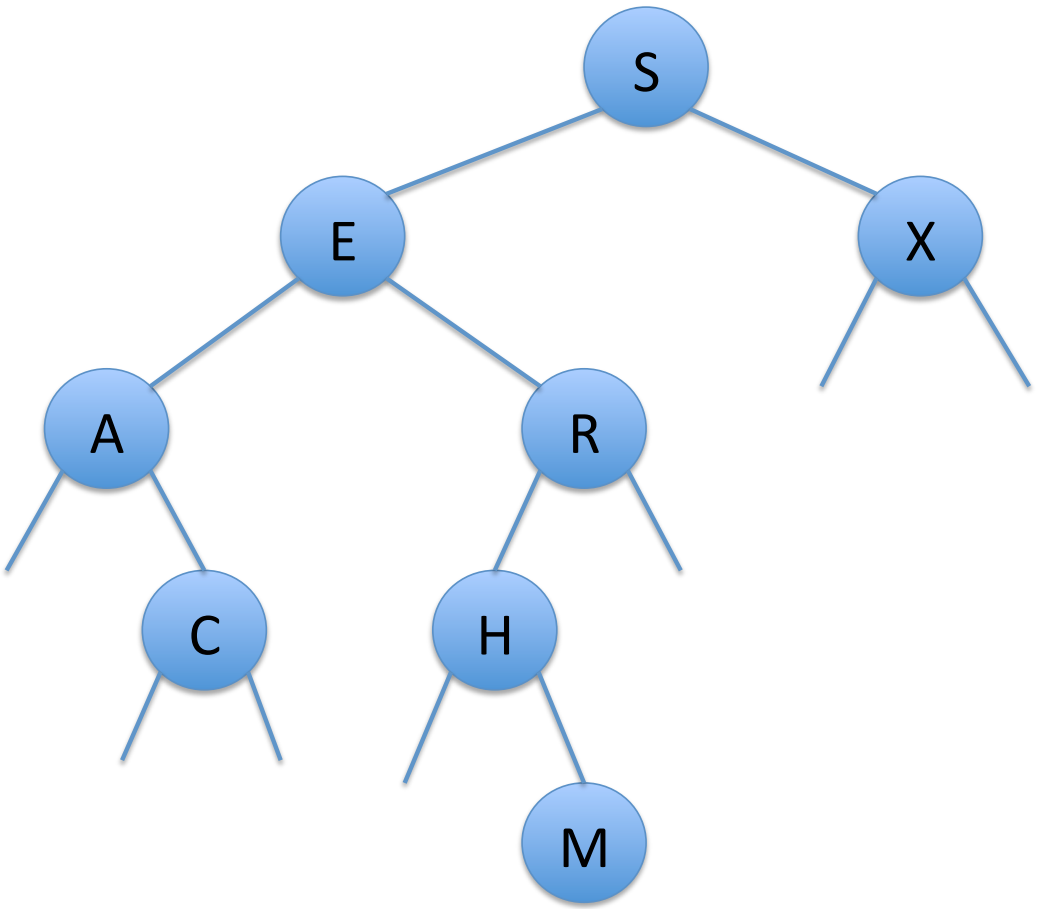
- Empty.
- A node with a left and a right tree.

Symmetric order

Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.

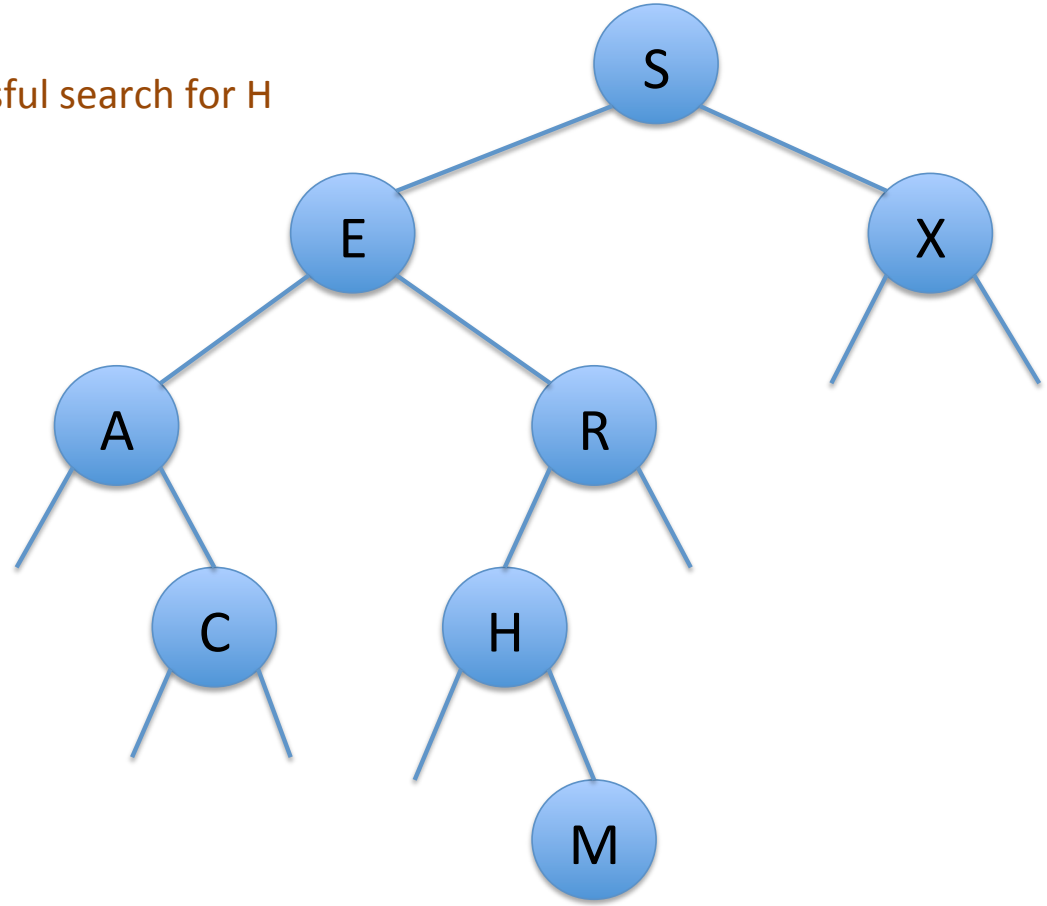
Binary Search Tree Example



Binary Search Tree: Find

Search: If empty, search fail; if less, go left; if greater, go right; If equal, search hit.

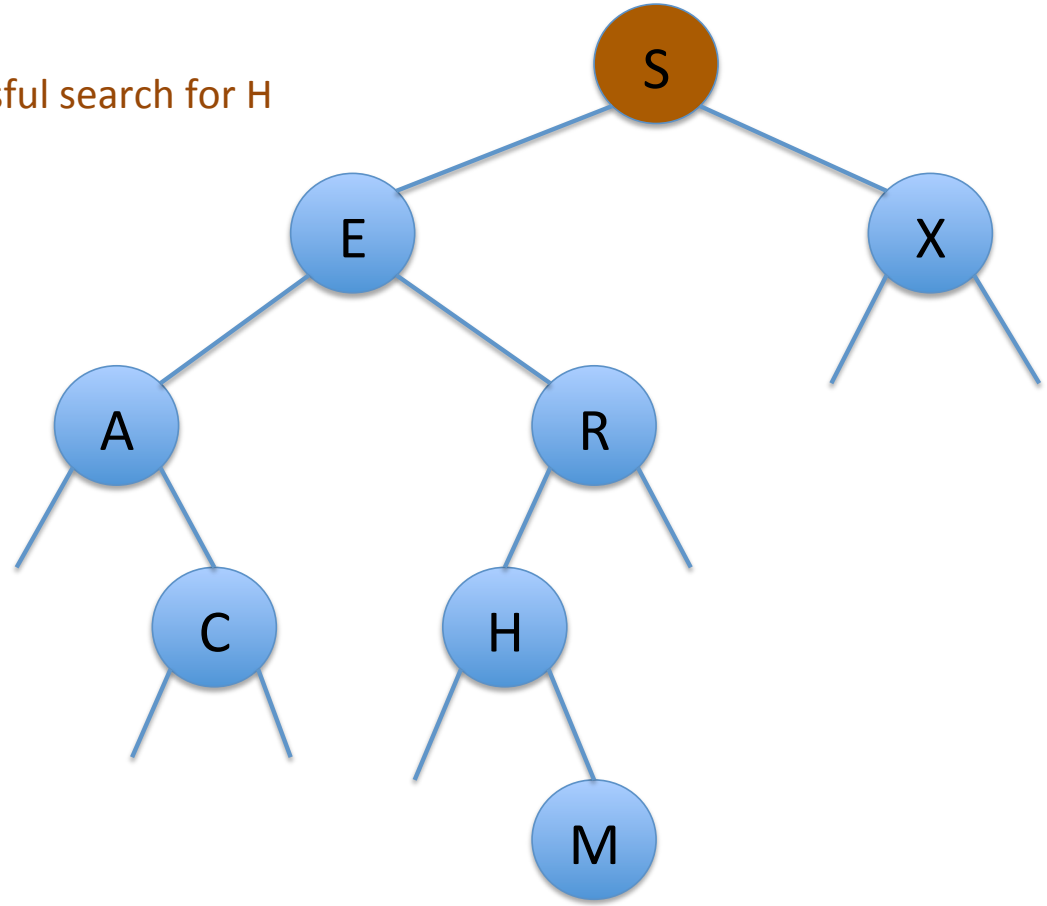
Successful search for H



Binary Search Tree: Find

Search: If empty, search fail; if less, go left; if greater, go right; If equal, search hit.

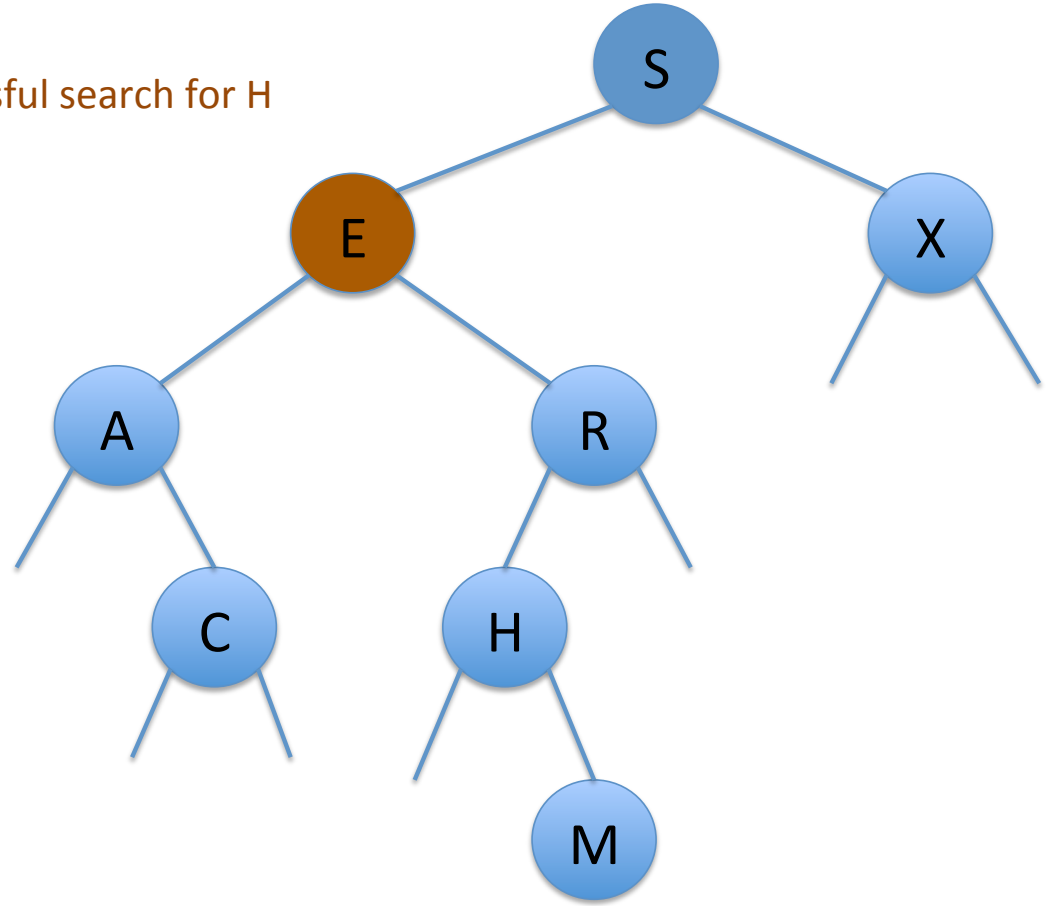
Successful search for H



Binary Search Tree: Find

Search: If empty, search fail; if less, go left; if greater, go right; If equal, search hit.

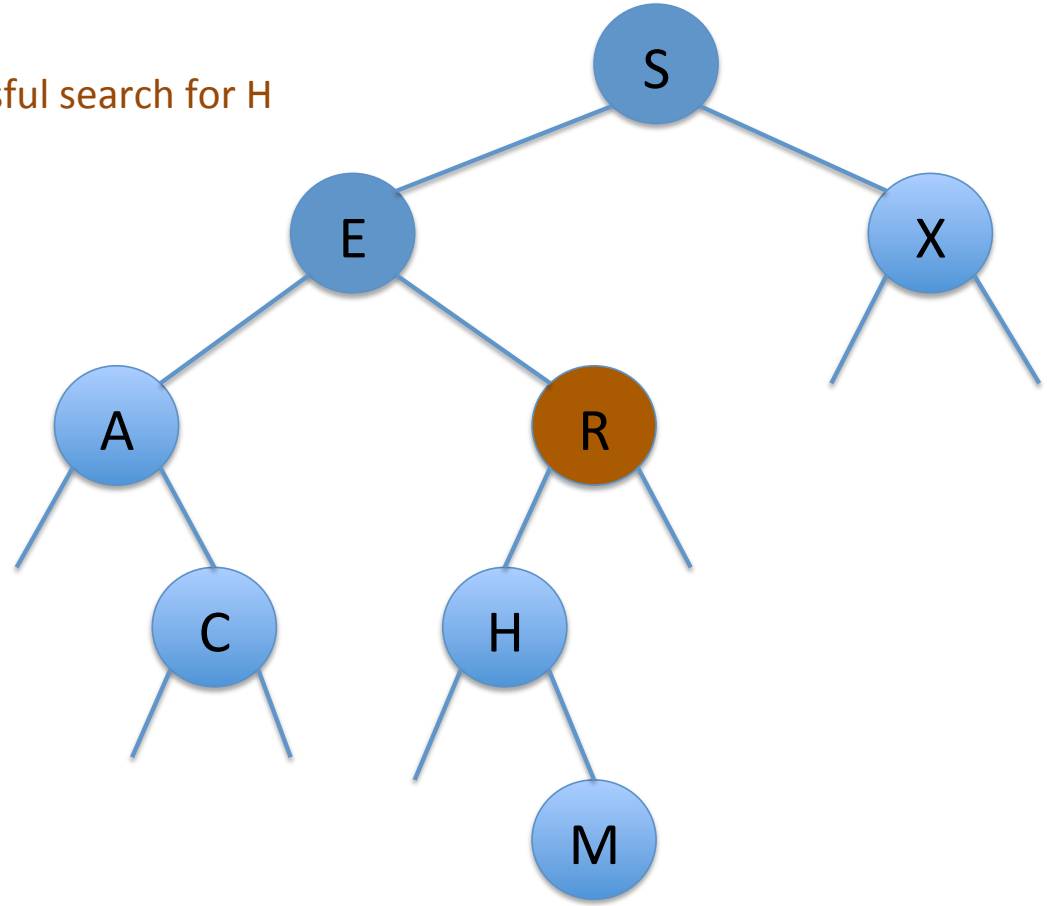
Successful search for H



Binary Search Tree: Find

Search: If empty, search fail; if less, go left; if greater, go right; If equal, search hit.

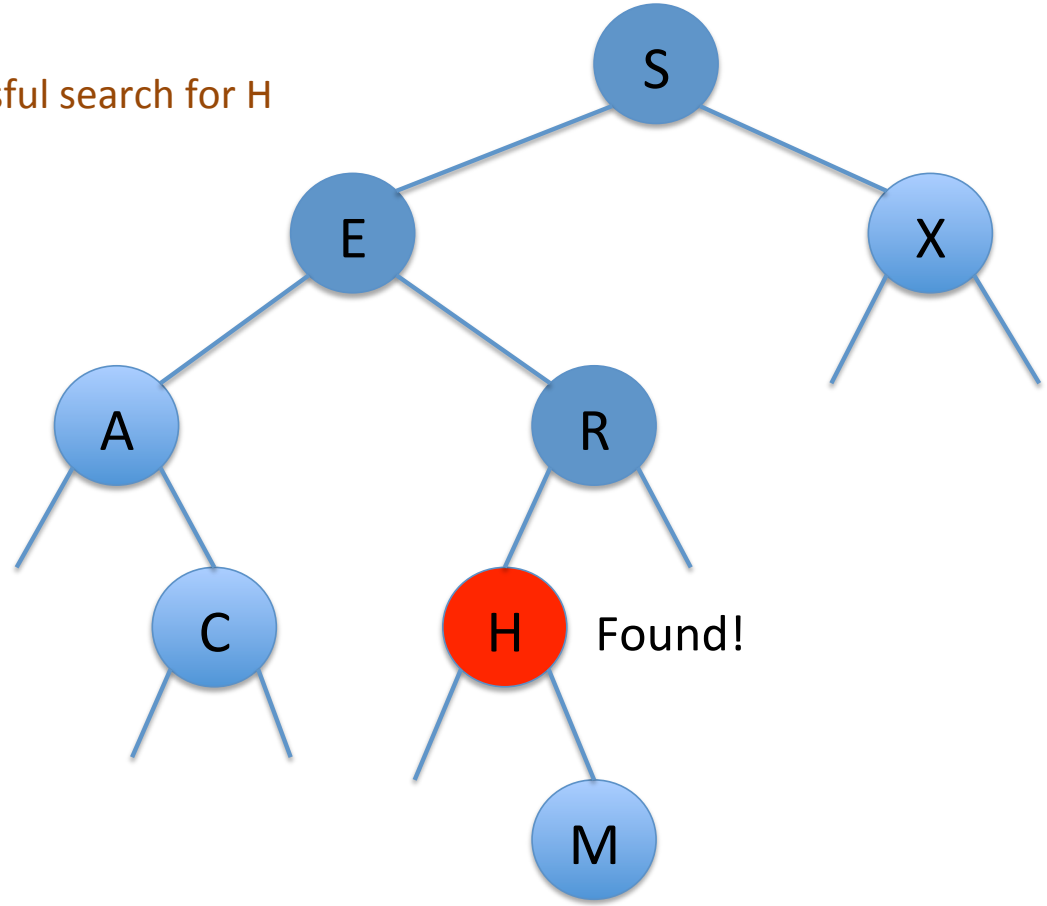
Successful search for H



Binary Search Tree: Find

Search: If empty, search fail; if less, go left; if greater, go right; If equal, search hit.

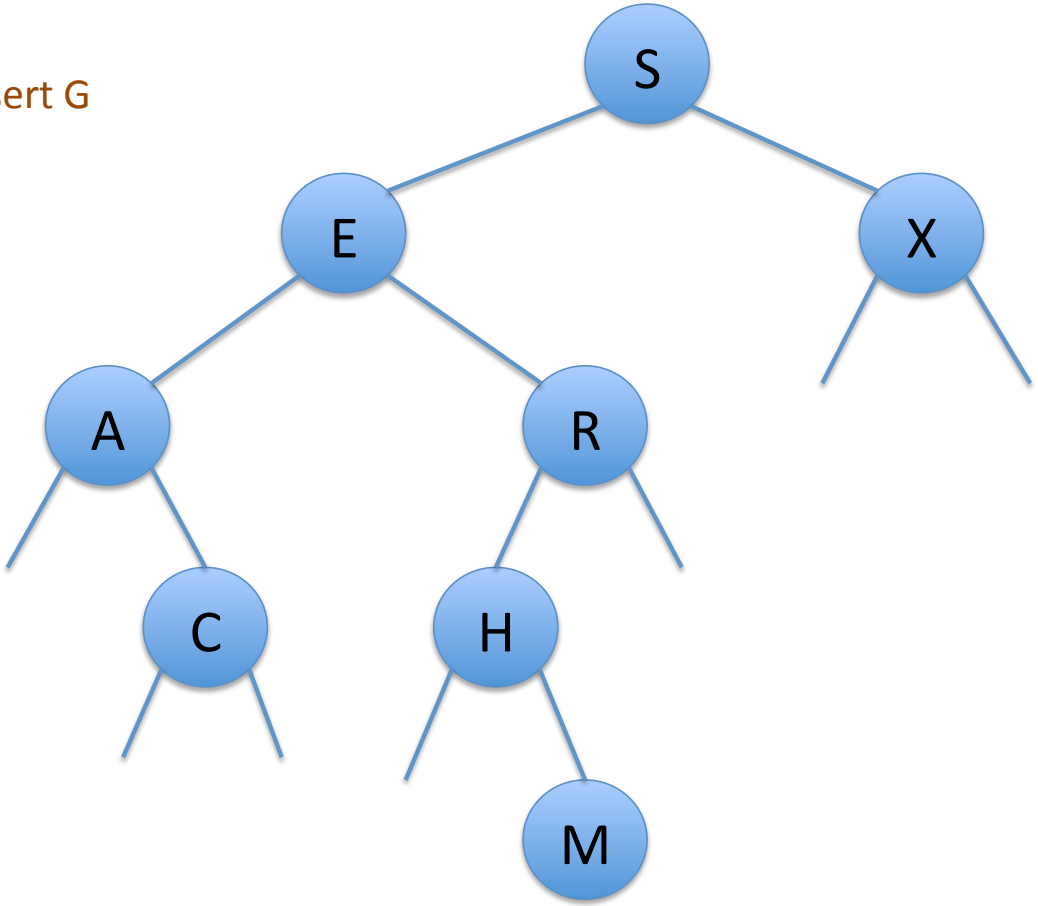
Successful search for H



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

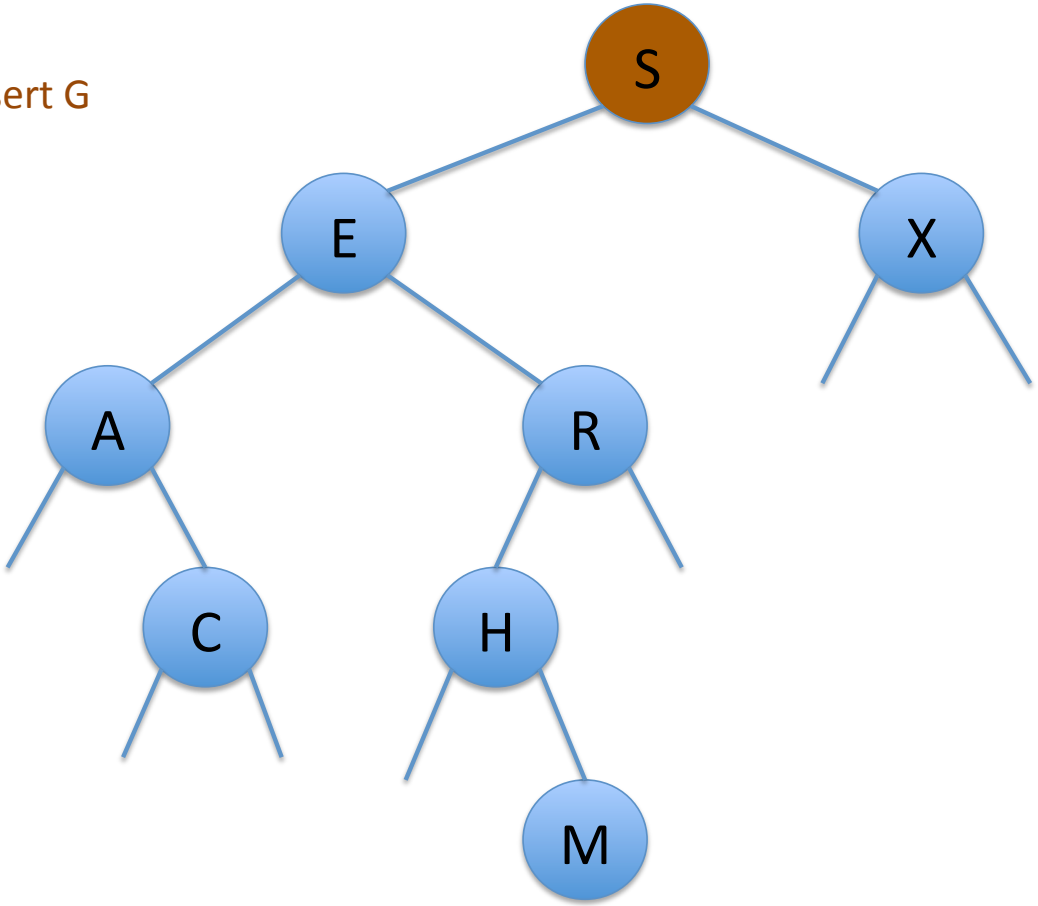
Insert G



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

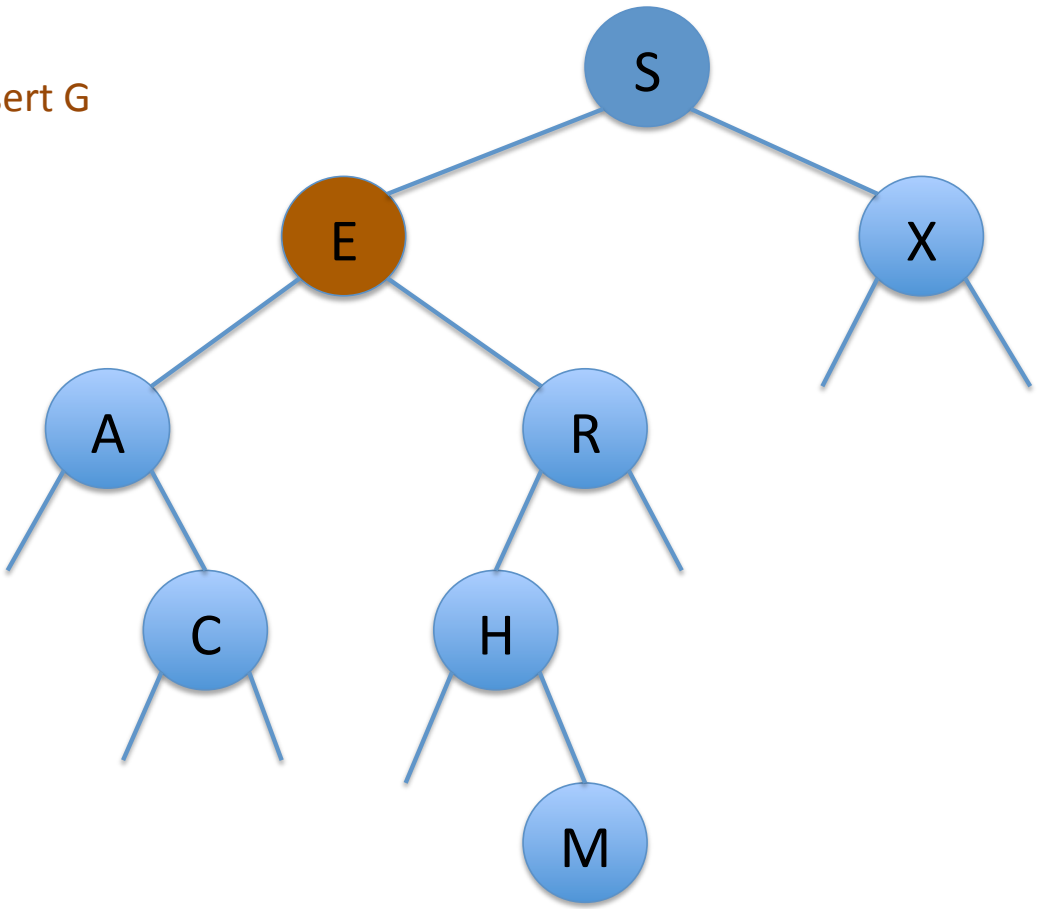
Insert G



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

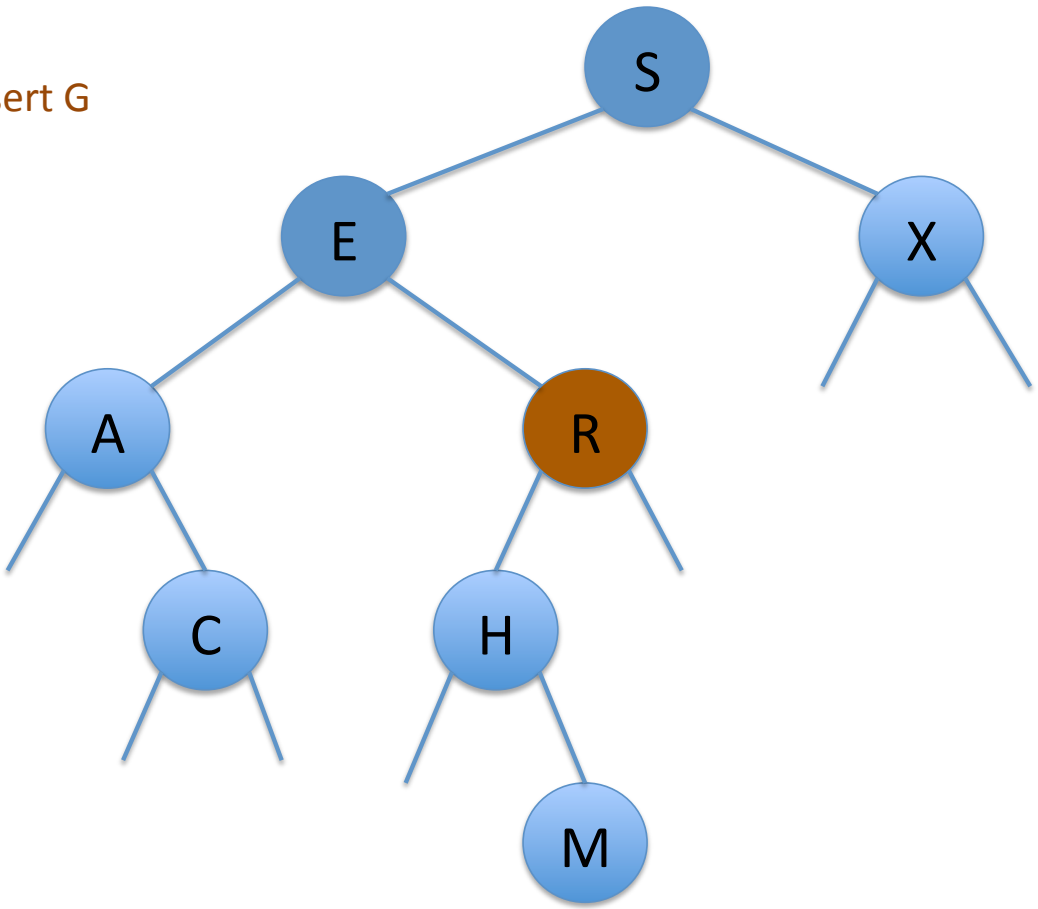
Insert G



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

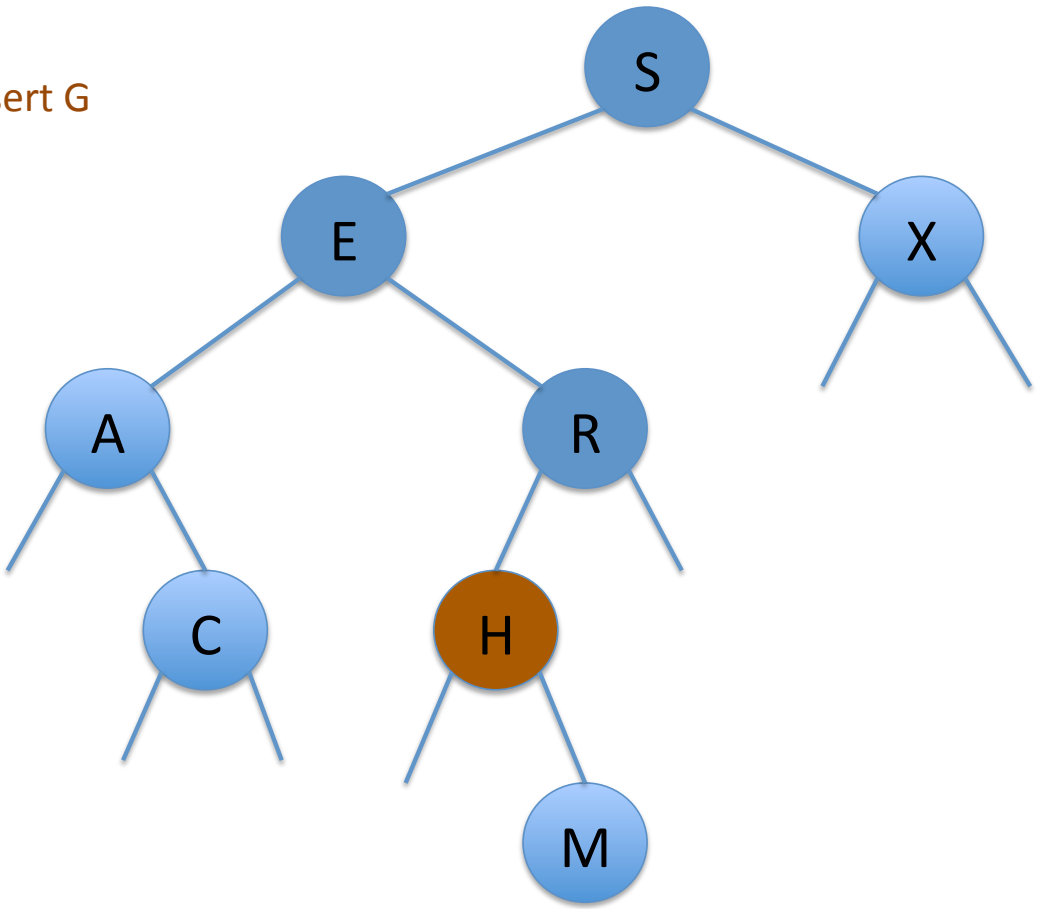
Insert G



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

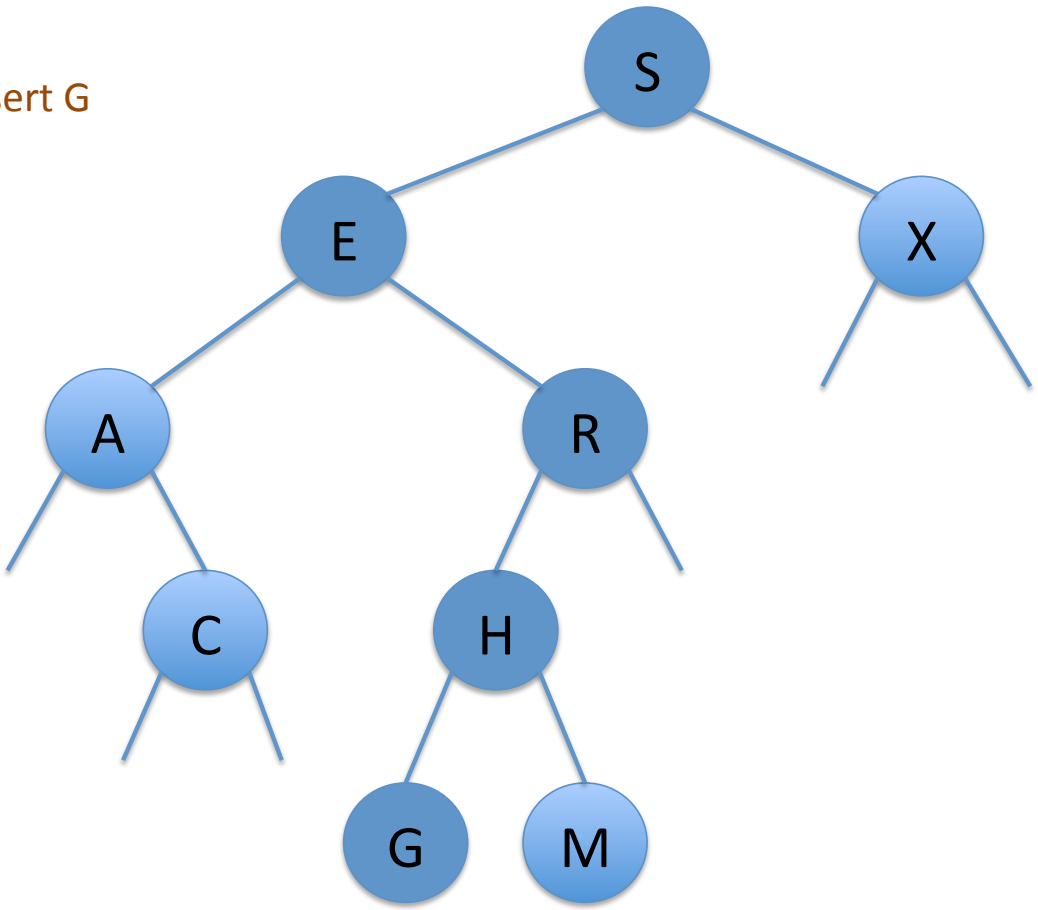
Insert G



Binary Search Tree: Insert

Insert: If empty, insert; if less, go left; if greater, go right.

Insert G



Balanced Binary Search Tree Principle

As we have seen in the preceding slides, in a Binary Search Tree (BST), the complexity of `search` and `insert` are in $O(\log n)$ (n is the number of nodes) if the tree is balanced (a binary tree's height is $\lfloor \log n \rfloor$).

But with the previous `insert` method, the height of the BST depends on the order of the elements inserted.

We give here an overview on how we can insert elements in any order and still have a balanced BST.

Balanced Binary: 2-3 Search Trees

TO DO

Set and Multiset

A Balanced Binary Search Tree (usually a Red-Black tree) is generally used to implement `Set` and `Multiset`.

Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two types of containers is that multisets allow duplicates, whereas sets do not.

If a special sorting criterion is not passed, the elements are compared with **operator** $<$.

Multiset Example

The following example illustrate some simple use of multisets.

```
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;
int main() {
  typedef multiset<int,greater<int> > IntSet;
  IntSet coll1; // empty multiset container
  // insert elements in random order
  coll1.insert(4);
  coll1.insert(3);
  coll1.insert(5);
  coll1.insert(1);
  coll1.insert(6);
  coll1.insert(2);
  coll1.insert(5);
  ...
}
```

Multiset Example

```
int main() {
  ...
  // iterate over all elements and print them
  IntSet::iterator pos;
  for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
      cout << *pos << '␣';
  }
  cout << endl;
  // insert 4 again and process return value
  IntSet::iterator ipos = coll1.insert(4);
  cout << "4␣inserted␣as␣element␣"
       << distance(coll1.begin(), ipos) + 1 << endl;
  // assign elements to another multiset with ascending order
  multiset<int> coll2(coll1.begin(), coll1.end());
  // print all elements of the copy
  copy (coll2.begin(), coll2.end(),
        ostream_iterator<int>(cout, "␣"));
  cout << endl;
  ...
}
```

Multiset Example

```
int main() {
  ...
  // remove all elements up to element with value 3
  coll2.erase(coll2.begin(), coll2.find(3));
  // remove all elements with value 5
  int num = coll2.erase(5);
  cout << num << "␣element(s)␣removed" << endl;
  // print all elements
  copy (coll2.begin(), coll2.end(),
        ostream_iterator<int>(cout, "␣"));
  cout << endl;
  return 0;
}
```

The resulting output is:

```
6 5 5 4 3 2 1
4 inserted as element 5
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6
```

Exercice 44

Try to solve the problem (`multiset`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 00978

Map and Multimap

A Balanced Binary Search Tree (usually a Red-Black tree) is generally used to implement `Map` and `Multimap`.

Maps and multimaps are containers that manage key/value pairs as elements. These containers sort their elements automatically, according to a certain sorting criterion that is used for the key. The difference between the two is that multimaps allow duplicates, whereas maps do not.

If a special sorting criterion is not passed, the elements are compared with **operator** $<$.

Map Example

The following example illustrate some simple use of maps.

```
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  typedef map<string,float> StringFloatMap;
  StringFloatMap stocks; // create empty container
  // insert some elements
  stocks["BASF"] = 369.50;
  stocks["VW"] = 413.50;
  stocks["Daimler"] = 819.00;
  stocks["BMW"] = 834.00;
  stocks["Siemens"] = 842.20;
  ...
}
```

Map Example

```
int main()
{
  ...
  // print all elements
  StringFloatMap::iterator pos;
  cout << left; // left-adjust values
  for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock:␣" << setw(12) << pos->first
         << "price:␣" << pos->second << endl;
  }
  cout << endl;
  // all prices doubled
  for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    pos->second *= 2;
  }
  ...
}
```

Map Example

```
int main()
{
  ...
  // print all elements
  for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock:␣" << setw(12) << pos->first
         << "price:␣" << pos->second << endl;
  }
  cout << endl;
  // rename key from "VW" to "Volkswagen"
  // - provided only by exchanging element
  stocks["Volkswagen"] = stocks["VW"];
  stocks.erase("VW");
  // print all elements
  for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock:␣" << setw(12) << pos->first
         << "price:␣" << pos->second << endl;
  }
  return 0;
}
```

Map Example

The resulting output is:

```
stock: BASF        price: 369.5
stock: BMW         price: 834
stock: Daimler     price: 819
stock: Siemens     price: 842.2
stock: VW          price: 413.5

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: VW          price: 827

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: Volkswagen  price: 827
```

Exercice 45

Try to solve the problem (`map`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11572

Hash Table

TO DO

Unordered_map

TO DO

Graph

The **graph** is a pervasive structure which appears in many computer science problems. A graph $G = (V, E)$ is composed of a set of vertices (V) and edges (E; storing connectivity information between vertices in V). We will explore many important graph problems and algorithms in another part of this course.

We frequently represent a graph by:

- An adjacency matrix.
- Or, an adjacency list.

Graph: Adjacency Matrix

We can represent a graph by an adjacency matrix (see figure on next slide), usually in the form of a 2D array:

```
int adj_mat[V][V];
// or
vector<vector<int>> adj_mat;
```

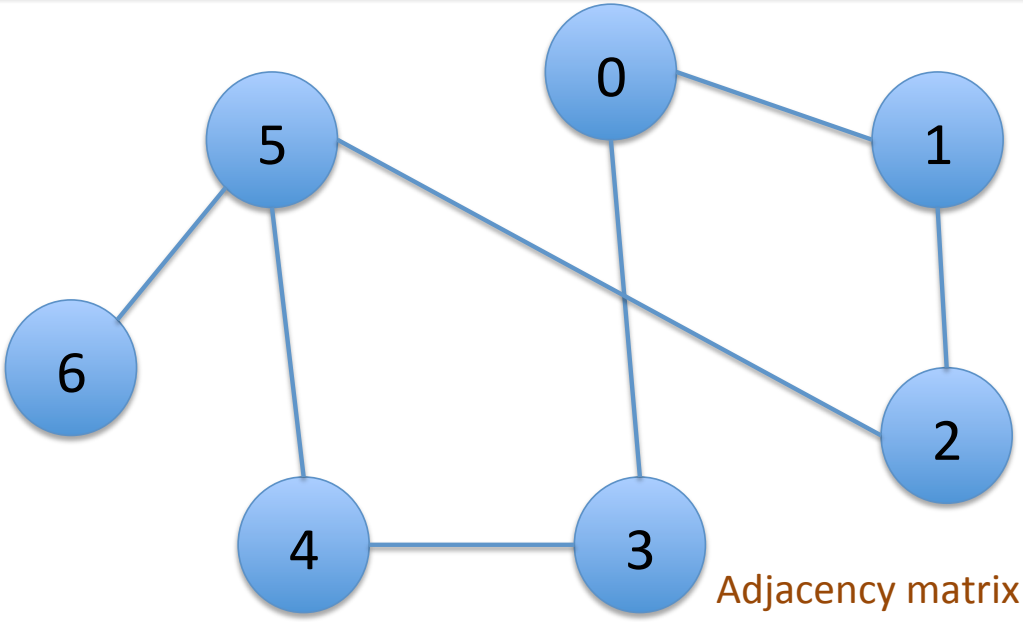
This representation has $O(V^2)$ space complexity.

For a weighted graph, set `adj_mat[i][j] = weight(i, j)`.

This representation **can be used for small dense graph if connectivity between two vertices is frequently required**. However, it is **not recommended for large sparse graph** as it would require too much space.

Another drawback of adjacency matrix is that it also takes $O(V)$ time to enumerate the list of neighbors of a vertex v (an operation common to many graph algorithms).

Graph Illustration



Adjacency list representation

0	1	3	
1	0	2	
2	1	5	
3	0	4	
4	3	5	
5	2	4	6
6	5		

Adjacency matrix representation

	0	**1**	**2**	**3**	**4**	**5**	**6**
0		X		X			
1	X		X				
2		X				X	
3	X				X		
4						X	
5			X		X		X
6						X	

Graph: Adjacency List

We can represent a graph by an adjacency matrix (see figure on previous slide), usually in the form of a vector of vector of pair:

```
vector<vector<pair<int, int>> adj_list;
```

For example, `adj_list[2]` represents the neighbors of vertex 2. Each pair contains two pieces of information, the index of the neighboring vertex and the weight of the edge.

The space complexity of adjacency list is $O(V + E)$. As E is usually much smaller than $O(V^2)$, adjacency lists are often more space-efficient than adjacency matrices.

With adajcency lists, we can enumerate the neighbors of a vertex v efficiently. If v has k neighbors, the enumeration will require $O(k)$ time.

Graph: Adjacency List Implementation

```
#include <vector>
#include <string>
#include <sstream>
#include <list>

class graph
{
public:
  typedef std::list<std::pair<int, float>> NEIGHBORS;
  typedef std::vector<NEIGHBORS> ADJ_LIST;

private:
  const int nb_vertices;
  int nb_edges;
  ADJ_LIST adj;
  bool directed;
  ...
```

Graph Implementation

```
  ...
public:
  graph(int n = 1000, bool directed = false):nb_vertices(n),nb_edges(0),
                                              directed(directed)
  {
    adj.assign(n, NEIGHBORS());
  }
  int vertices() {
    return nb_vertices;
  }
  int edges() {
    return nb_edges;
  }
  // degree for undirected graph and out-degree for directed graph
  int degree(int i) {
    return adj[i].size();
  }
  ...
```

Graph Implementation

```
...
void add_edge(int i, int j, float weight = 1)
{
  nb_edges++;
  adj[i].push_back(std::make_pair(j, weight));
  if (!directed) adj[j].push_back(std::make_pair(i, weight));
}
const NEIGHBORS& neighbors(int i)
{
  return adj[i];
}
bool connected(int i, int j)
{
  for (const auto& edge : adj[i])
    {
      if (edge.first == j) return true;
    }
  return false;
}
...
```

Graph Implementation

```
  ...
  std::string to_string()
  {
    std::stringstream res;
    res << nb_vertices << "␣vertices,␣" << nb_edges << "␣edges\n";
    for (int i = 0; i < nb_vertices; i++)
      {
        res << i << ":␣";
        for (const auto& edge : adj[i])
        {
          res << "(" << edge.first << ",␣w:" << edge.second << ")␣";
        }
        res << "\n";
      }
    return res.str();
  }
};
```

Exercice 46

Try to solve the problem (`oriented graph`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11991

Union Find

The **Union Find Disjoint Sets** is a data structure to model a collection of *disjoint sets* with the ability to efficiently – $\approx O(1)$ for practical number of sets – determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint set into a larger set.

For example, such data structure can be used to solve the problem of finding connected components in an undirected graph, or to test if an item has already been treated.

Union Find Principle

Data structure:

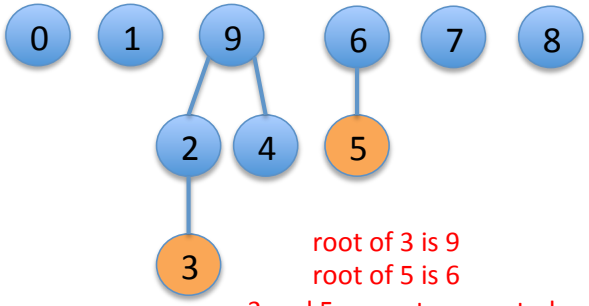
Integer array id[] of size N.

Interpretation: id[i] is parent of i.

Root of i is id[id[id[...id[i]...]]].

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

Connected(3, 5) = false.



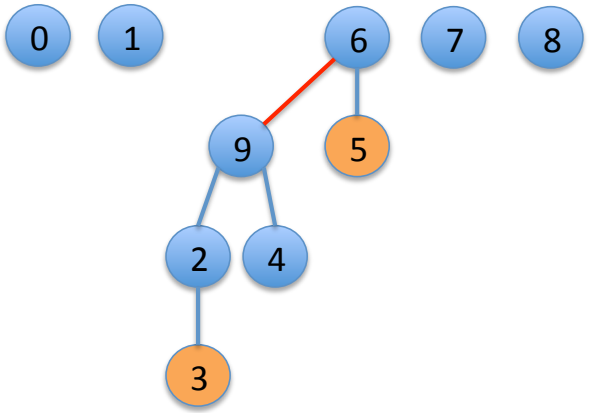
root of 3 is 9
root of 5 is 6
3 and 5 are not connected

Connected: Check if i and j have the same root.

Union: To merge components containing i and j set the id of i's root to the id of j's root.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	6

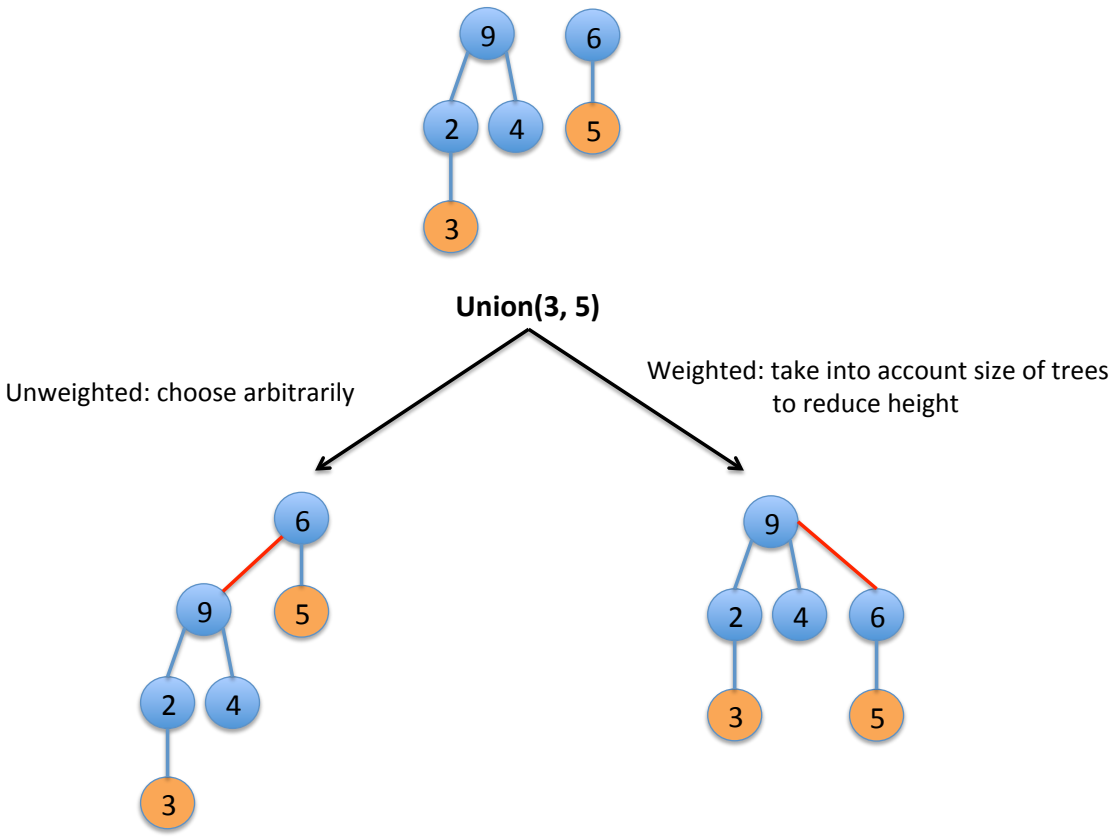
Union(3, 5).



Weighted Union Find

- Modify *union find* to avoid tall trees.
- Keep track of size of each tree (number of nodes).
- Balance by linking root of smaller tree to root of larger tree.

For example:



Weighted Union Find Complexity

Find: takes time proportional to depth of i.

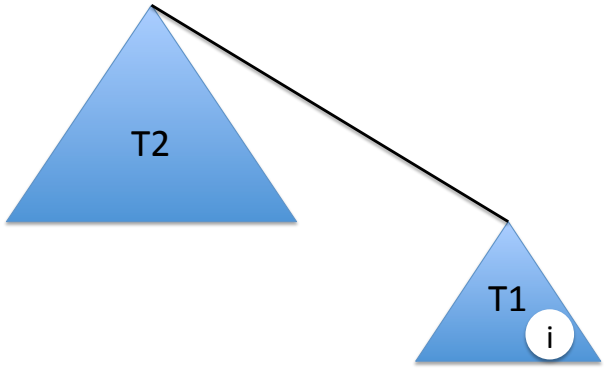
Union: takes constant time, given roots (uses find to get roots).

Proposition

For an array of size N, the depth of any node i is at most $\log N$.

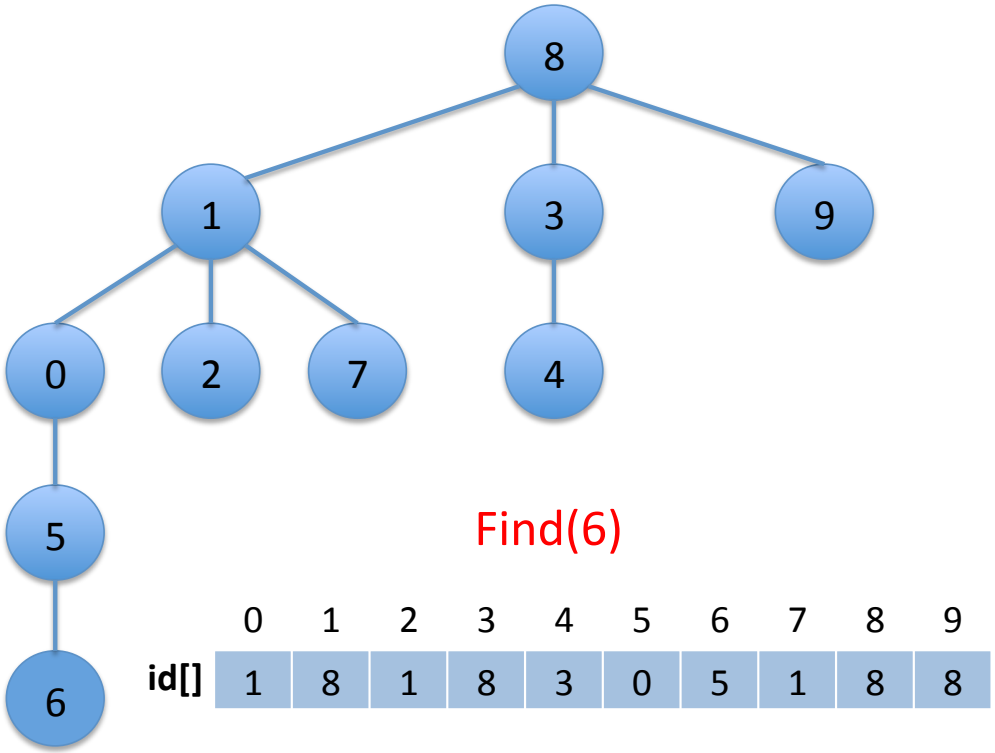
Proof. When does depth of i increase?
Increases by 1 when tree T_1 containing i is merged into another tree T_2:

- The size of the tree containing i at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing i can double at most $\log N$ times.



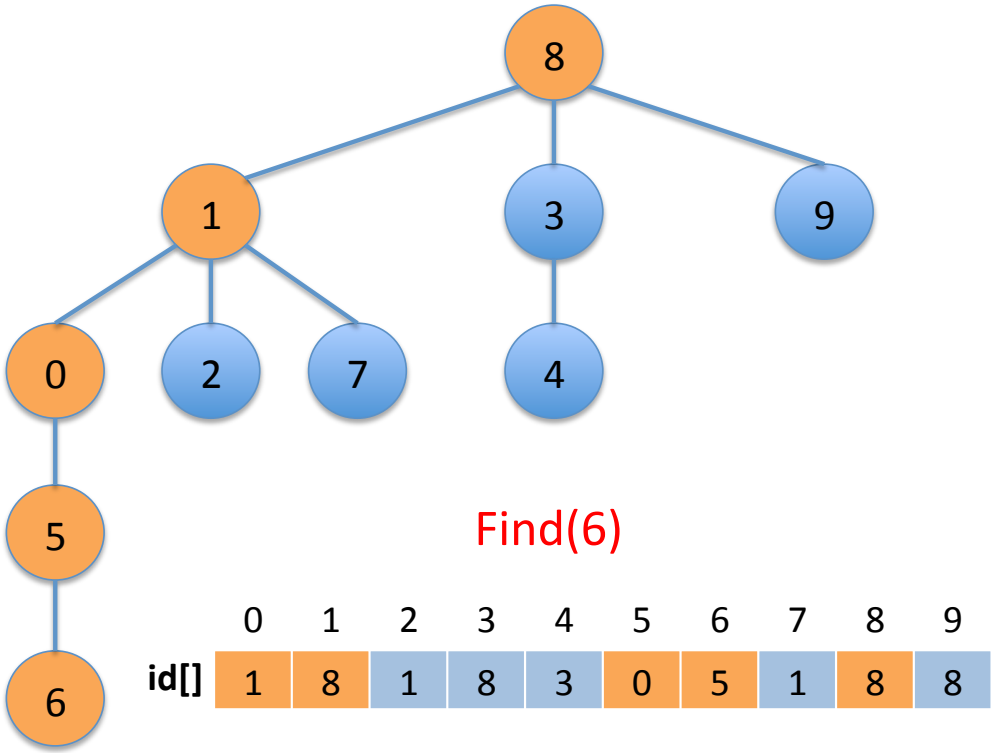
Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.



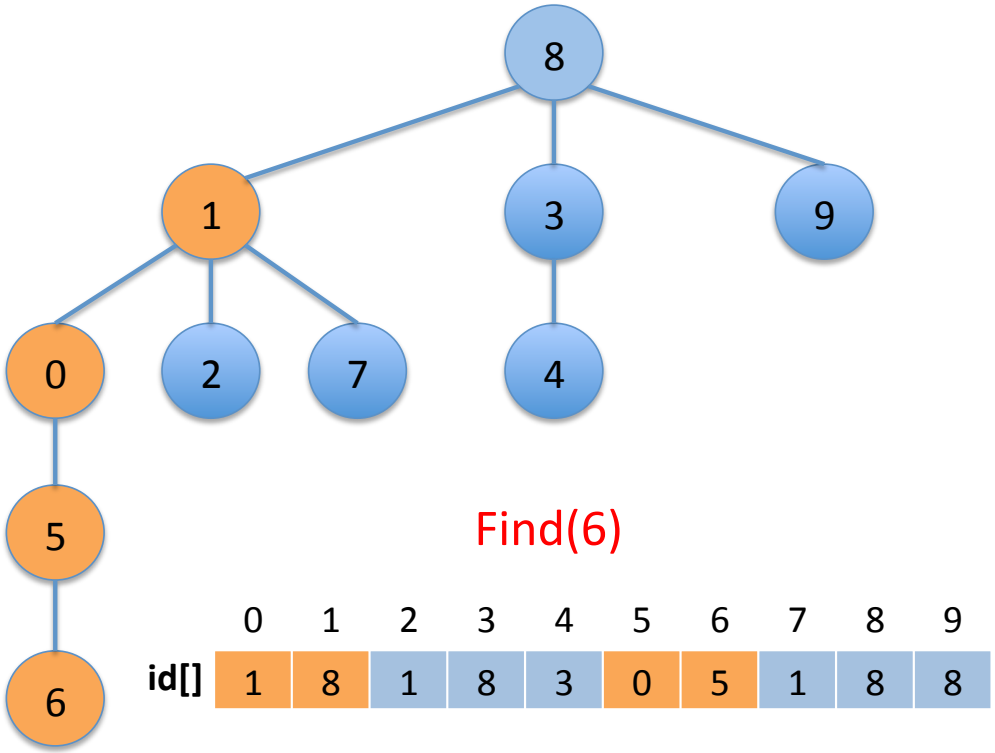
Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.



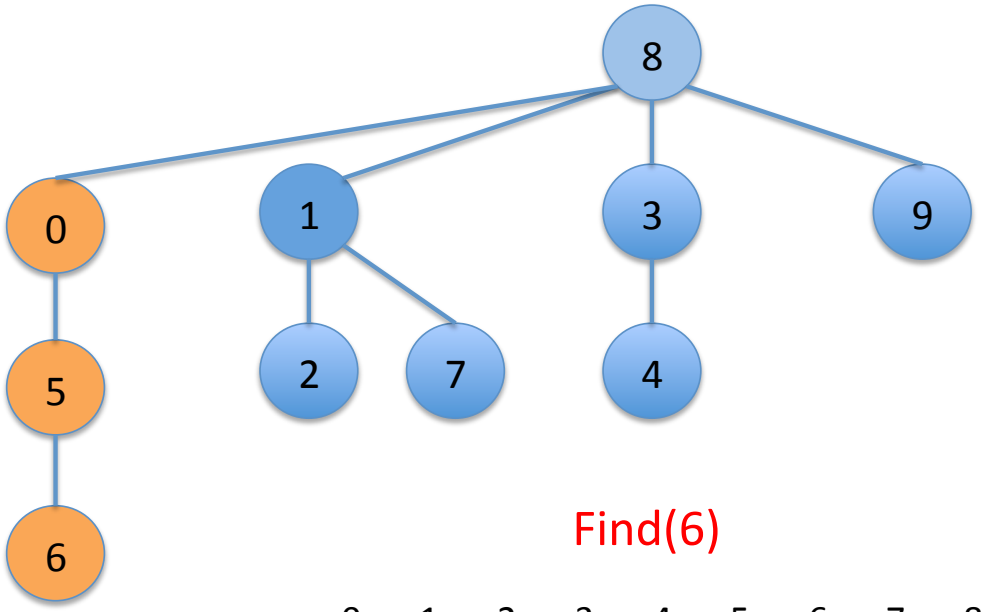
Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.



Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.

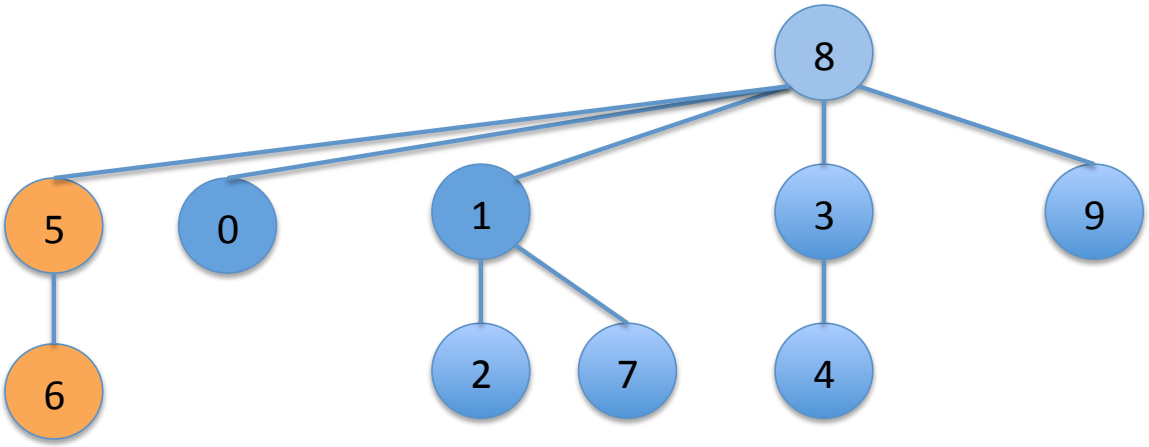


Find(6)

	0	1	2	3	4	5	6	7	8	9
id[]	8	8	1	8	3	0	5	1	8	8

Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.

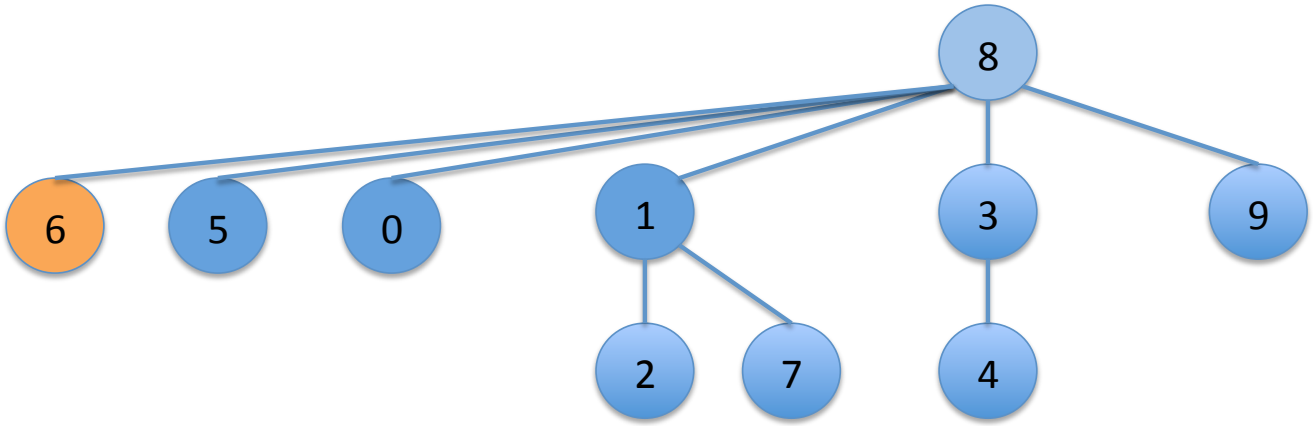


Find(6)

	0	1	2	3	4	5	6	7	8	9
id[]	8	8	1	8	3	8	5	1	8	8

Weighted Union Find with Path Compression

Quick union with path compression: just after computing the root of i, set the id of each examined node to point to that root.



Find(6)

	0	1	2	3	4	5	6	7	8	9
id[]	8	8	1	8	3	8	8	1	8	8

Weighted Union Find with Path Compression Complexity

Proposition. [Hopcroft-Ulman, Tarjan]

Starting from an empty data structure, any sequence of M union-find operations on N objects makes $\leq c(N + M \log^* N)$ array accesses.

Where $\log^*$ is the iterate logarithm function defined by:

$$\log^* x = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^*(\log x) & \text{if } x > 1 \end{cases}$$

N	$\log^* N$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

Union Find Implementation

```
#include <vector>
class union_find
{
private:
  std::vector<int> id; // id[i] = parent of i
  std::vector<int> sz;//sz[i] = number of objects in subtree rooted at i
  int count; // number of components
public:
  union_find(int N)
  {
    count = N;
    id.assign(N, 1);
    sz.assign(N, 0);
    for (int i = 0; i < N; i++)
    {
      id[i] = i;
      sz[i] = 1;
    }
  }
  ...
```

Union Find Implementation

```
...
int nb_components()
{
  return count;
}

int size_set(int i)
{
  return sz[find_set(i)];
}

int find_set(int i)
{
  return (id[i] == i) ? i : (id[i] = find_set(id[i]));
}
...
```

Union Find Implementation

```
  ...
  bool connected(int i, int j)
  {
    return find_set(i) == find_set(j);
  }

  void union_set(int p, int q)
  {
    int i = find_set(p);
    int j = find_set(q);
    if (i == j) return;
    // make smaller root point to larger one
    if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else               { id[j] = i; sz[i] += sz[j]; }
    count--;
  }
};
```

Exercice 47

Try to solve the problems (`union find`):

- UVa Online Judge (http://uva.onlinejudge.org) problem number 793
- UVa Online Judge (http://uva.onlinejudge.org) problem number 10583

Segment Tree

TO DO

Exercice 48

Try to solve the problem (`segment tree with lazy updates`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11402

Problem Solving Tools

TO DO

Design of Algorithms by Induction

TO DO

Invariants

TO DO

Reduction

TO DO

Complete Search

The **complete search** technique, also known as **brute force**, is a method for solving a problem by traversing the entire (or part of the) search space to obtain the require solution.

During the search, we can prune part of the search space if we have determined that these parts have no possibility of containing the required solution.

As we have discussed in the part about testing, it is often simple to come up with a correct brute force solution to a problem. But the complexity is high so the dimension of the problem must be small.

Complete Search

For some problems, the complete search approach is mandatory, for example if we want to list all the $N!$ permutations or all the subsets of a set of size N, we have to list all the elements of the search space.

For other challenging problems, running complete search on small instances can help us to understand its structure through patterns in the output that can be exploited to design a faster algorithm.

We present complete search using:

- A recursive approach known as **Recursive Backtracking**.
- An iterative approach.

For each of these cases, we show how to use pruning to reduce the exploration.

Recursive Backtracking

We will represent our solution as a vector $(a_1, a_2, \ldots, a_n)$, where each element a_i is selected from a finite set S_i. For example, such vector can represent:

- An arrangement where a_i contains the i-th element of the permutation.
- A subset, where a_i is true if the i-th element of the set is in the subset.
- A sequence of moves in a game, ...

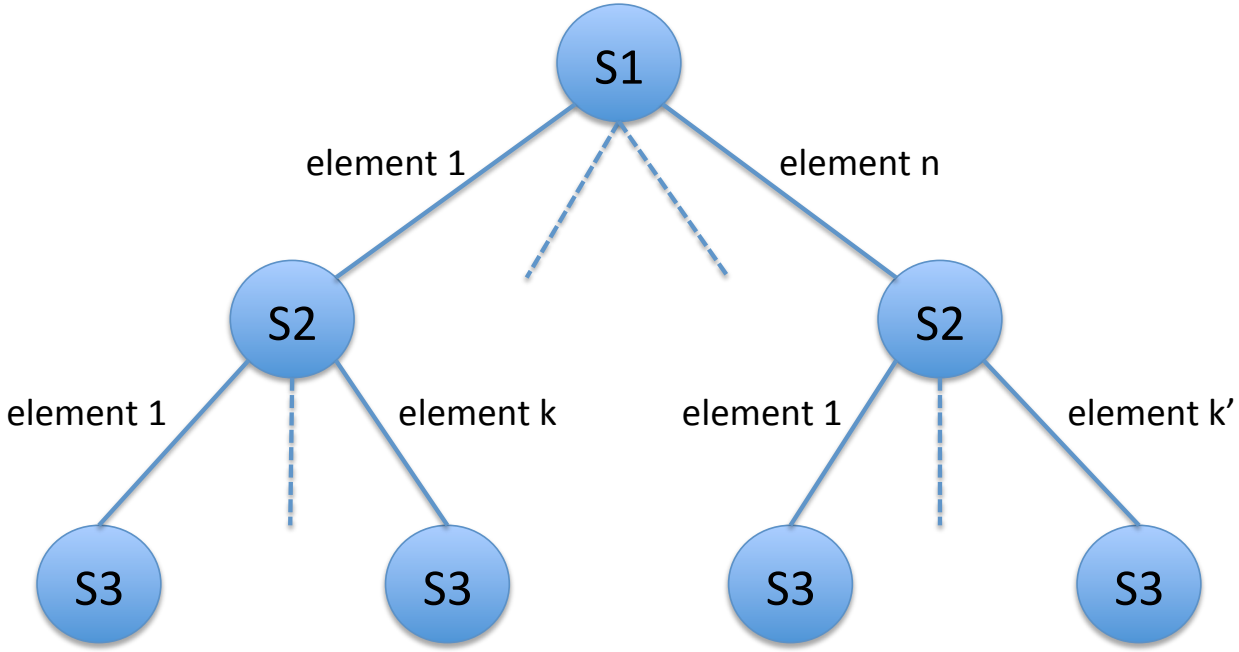
At each step in the backtracking algorithm, we try to extend a given partial solution $a = (a_1, a_2, \ldots, a_k)$ by adding another element at the end. The next slide gives the pseudo-code for recursive backtracking.

Recursive Backtracking Pseudo-Code

```
Backtrack((a_1, a_2, ..., a_k), k)
  If (a_1, a_2, ..., a_k) is a solution then
    process it
  Else
    compute S_{k+1}
    While S_{k+1} ≠ ∅ do
      a_{k+1} ← an element in S_{k+1}
      S_{k+1} ← S_{k+1} \ {a_{k+1}}
      Backtrack((a_1, a_2, ..., a_k, a_{k+1}), k + 1)
    Endwhile
  Endif
```

Recursive Backtracking State Space

The state space can be viewed as a tree that is explored in a depth first search manner by the `Backtrack` algorithm described previously:



Recursive Backtracking Implementation

```
template <typename T>
class backtracking
{
  domain<T>& dom;
public:
  backtracking(domain<T>& dom) : dom(dom)
  {
  }
  void execute()
  {
    backtrack(0);
  }
  ...
};
```

Recursive Backtracking Implementation

```
  ...
  void backtrack(int k)
  {
    if (dom.is_a_solution(k)) { dom.process_solution(k); }
    else
      {
        k++;
        std::vector<T> possibilities;
        dom.next(possibilities, k);
        for (int i = 0; i < possibilities.size(); ++i) {
          dom.set(k, possibilities[i]);
          dom.make_move(k);
          backtrack(k);
          dom.unmake_move(k);
          if (dom.stop()) return;
        }
      }
  }
};
```

Recursive Backtracking Implementation

```
template <typename T>
struct domain
{
  std::vector<T> candidate;
  domain(int size) : candidate(size + 1) {}
  virtual bool is_a_solution(int k) = 0;
  virtual void process_solution(int k) = 0;
  virtual void next(std::vector<T>& possibilities, int k) = 0;
  virtual void make_move(int k) {}
  virtual void unmake_move(int k) {}
  virtual bool stop()
  {
    return false;
  }
  void set(int i, const T& elem)
  {
    candidate[i] = elem;
  }
};
```

Recursive Backtracking Example: Subsets

Problem: List all the subsets of $\{1, 2, \ldots, N\}$ for a given N.

```
class subsets : public domain<bool>
{
private:
  int N;
public:
  subsets(int N) : domain(N) { this->N = N; }
  bool is_a_solution(int k) { return k == N; }
  void process_solution(int k) {
    printf("{");
    for (int i = 1; i <= k; ++i) {
      if (candidate[i]) printf("␣%d", i);
    }
    printf("␣}\n");
  }
  void next(std::vector<bool>& possibilities, int k) {
    possibilities.push_back(true);
    possibilities.push_back(false);
  }
};
```

Exercice 49

Write the `domain class` to print all $N!$ permutations of the integers: $1, 2, \ldots, N$

Recursive Backtracking and Pruning

We consider the $0-1$ knapsack problem (or subset sum problem) to illustrate pruning in recursive backtracking.

Problem: Given n items, each with its own value V_i and weight W_i, and a maximum knapsack capacity C, compute the maximum value of the items that we can carry, if we can either ignore or take a particular item.

For example: $n = 4$, $C = 16$,

	item 1	item 2	item 3	item 4
V	40	30	50	10
W	2	5	10	5

- If we select item 2 and 3, we cannot take any other item and we obtain a total value of 80 which is not optimal.
- If we select item 1 and 3, we have a total weight of 12 and a value of 90 which is the best we can do.

Recursive Backtracking and Pruning

TO DO

Recursive Backtracking and Pruning: Implementation

TO DO

Exercice 50

Try to solve the problem (`recursive backtracking`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 750

Iterative Complete Search

TO DO

Exercice 51

Try to solve the problem (`for loops`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11565

Exercice 52

Try to solve the problem (`extreme version of the preceding problem`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11571

Binary Search

TO DO

Exercice 53

Try to solve the problem (`bissection + simulation`):

UVa Online Judge (http://uva.onlinejudge.org) problem number 11402

Greedy

TO DO

Divide and Conquer

TO DO

Dynamic Programming

TO DO

Exercice 54

Try to solve the problem (DP):

UVa Online Judge (http://uva.onlinejudge.org) problem number 357

Specific Domains

TO DO

Graph

TO DO

Mathematics

TO DO

String Processing

TO DO

Computational Geometry

TO DO

Simplex

TO DO

Harder Problems

TO DO