1 Introduction

> You, my young friend, are going to learn to program the algorithms of this book, and then go on to win programming contests, sparkle during your job interviews, and finally roll up your sleeves, get to work, and greatly improve the gross national product!

Mistakenly, computer scientists are still often considered the magicians of modern times. Computers have slowly crept into our businesses, our homes and our machines, and have become important enablers in the functioning of our world. However, there are many that use these devices without really mastering them, and hence, they do not fully enjoy their benefits. Knowing how to program provides the ability to fully exploit their potential to solve problems in an efficient manner. Algorithms and programming techniques have become a necessary background for many professions. Their mastery allows the development of creative and efficient computer-based solutions to problems encountered every day.

This text presents a variety of algorithmic techniques to solve a number of classic problems. It describes practical situations where these problems arise, and presents simple implementations written in the programming language Python. Correctly implementing an algorithm is not always easy: there are numerous traps to avoid and techniques to apply to guarantee the announced running times. The examples in the text are embellished with explanations of important implementation details which must be respected.

For the last several decades, programming competitions have sprung up at every level all over the world, in order to promote a broad culture of algorithms. The problems proposed in these contests are often variants of classic algorithmic problems, presented as frustrating enigmas that will never let you give up until you solve them!

1.1 Programming Competitions

In a programming competition, the candidates must solve several problems in a fixed time. The problems are often variants of classic problems, such as those addressed in this book, dressed up with a short story. The inputs to the problems are called *instances*. An instance can be, for example, the adjacency matrix of a graph for a shortest path problem. In general, a small example of an instance is provided, along with its solution. The source code of a solution can be uploaded to a server via

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

a web interface, where it is compiled and tested against instances hidden from the public. For some problems the code is called for each instance, whereas for others the input begins with an integer indicating the number of instances occurring in the input. In the latter case, the program must then loop over each instance, solve it and display the results. A submission is accepted if it gives correct results in a limited time, on the order of a few seconds.

Figure 1.1 The logo of the ICPC nicely shows the steps in the resolution of a problem. A helium balloon is presented to the team for each problem solved.

To give here a list of all the programming competitions and training sites is quite impossible, and such a list would quickly become obsolete. Nevertheless, we will review some of the most important ones.

ICPC The oldest of these competitions was founded by the *Association for Computing Machinery* in 1977 and supported by them up until 2017. This contest, known as the ICPC, for *International Collegiate Programming Contest*, is organised in the form of a tournament. The starting point is one of the regional competitions, such as the *South-West European Regional Contest* (SWERC), where the two best teams qualify for the worldwide final. The particularity of this contest is that each three-person team has only a single computer at their disposal. They have only 5 hours to solve a maximum number of problems among the 10 proposed. The first ranking criterion is the number of submitted solutions accepted (i.e. tested successfully against a set of unknown instances). The next criterion is the sum over the submitted problems of the time between the start of the contest and the moment of the accepted submission. For each erroneous submission, a penalty of 20 minutes is added.

There are several competing theories on what the ideal composition of a team is. In general, a good programmer and someone familiar with algorithms is required, along with a specialist in different domains such as graph theory, dynamic programming, etc. And, of course, the team members need to get along together, even in stressful situations!

For the contest, each team can bring 25 pages of reference code printed in an 8-point font. They can also access the online documentation of the Java API and the C++ standard library.

Google Code Jam In contrast with the ICPC contest, which is limited to students up to a Master's level, the Google Code Jam is open to everyone. This more recent annual competition is for individual contestants. Each problem comes in

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

general with a deck of small instances whose resolution wins a few points, and a set of enormous instances for which it is truly important to find a solution with the appropriate algorithmic complexity. The contestants are informed of the acceptance of their solution for the large instances only at the end of the contest. However, its true strong point is the possibility to access the solutions submitted by all of the participants, which is extremely instructive.

The competition *Facebook Hacker Cup* is of a similar nature.

Prologin The French association Prologin organises each year a competition targeted at students up to twenty years old. Their capability to solve algorithmic problems is put to test in three stages: an online selection, then regional competitions and concluding with a national final. The final is atypically an endurance test of 36 hours, during which the participants are confronted with a problem in Artificial Intelligence. Each candidate must program a "champion" to play a game whose rules are defined by the organisers. At the end of the day, the champions are thrown in the ring against each other in a tournament to determine the final winner.

The website `https://prologin.org` includes complete archives of past problems, with the ability to submit algorithms online to test the solutions.

France-IOI Each year, the organisation France-IOI prepares junior and senior high school students for the International Olympiad in Informatics. Since 2011, they have organised the 'Castor Informatique' competition, addressed at students from Grade 4 to Grade 12 (675,000 participants in 2018). Their website `http://france-ioi.org` hosts a large number of algorithmic problems (more than 1,000).

Numerous programming contests organised with the goal of selecting candidates for job offers also exist. The web site `www.topcoder.com`, for example, also includes tutorials on algorithms, often very well written.

For training, we particularly recommend `https://codeforces.com`, a well-respected web site in the community of programming competitions, as it proposes clear and well-prepared problems.

1.1.1 Training Sites

A number of websites propose problems taken from the annals of competitions, with the possibility to test solutions as a training exercise. This is notably the case for Google Code Jam and Prologin (in French). The collections of the annals of the ICPC contests can be found in various locations.

Traditional online judges The following sites contain, among others, many problems derived from the ICPC contests.

uva `http://uva.onlinejudge.org`

icpcarchive `http://icpcarchive.ecs.baylor.edu`, `http://livearchive.onlinejudge.org`

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

Chinese online judges Several training sites now exist in China. They tend to have a purer and more refined interface than the traditional judges. Nevertheless, sporadic failures have been observed.

poj `http://poj.org`
tju `http://acm.tju.edu.cn` (Shut down since 2017)
zju `http://acm.zju.edu.cn`

Modern online judges Sphere Online Judge `http://spoj.com` and Kattis `http://open.kattis.com` have the advantage of accepting the submission of solutions in a variety of languages, including Python.

spoj `http://spoj.com`
kattis `http://open.kattis.com`
zju `http://acm.zju.edu.cn`

Other sites

codechef `http://codechef.com`
codility `http://codility.com`
gcj `http://code.google.com/codejam`
prologin `http://prologin.org`
slpc `http://cs.stanford.edu/group/acm`

Throughout this text, problems are proposed at the end of each section in relation to the topic presented. They are accompanied with their identifiers to a judge site; for example [spoj:CMPLS] refers to the problem '*Complete the Sequence!*' at the URL `www.spoj.com/problems/CMPLS/`. The site `http://tryalgo.org` contains links to all of these problems. The reader thus has the possibility to put into practice the algorithms described in this book, testing an implementation against an online judge.

The languages used for programming competitions are principally C++ and Java. The SPOJ judge also accepts Python, while the Google Code Jam contest accepts many of the most common languages. To compensate for the differences in execution speed due to the choice of language, the online judges generally adapt the time limit to the language used. However, this adaptation is not always done carefully, and it is sometimes difficult to have a solution in Python accepted, even if it is correctly written. We hope that this situation will be improved in the years to come. Also, certain judges work with an ancient version of Java, in which such useful classes as `Scanner` are not available.

1.1.2 Responses of the Judges

When some code for a problem is submitted to an online judge, it is evaluated via a set of private tests and a particularly succinct response is returned. The principal response codes are the following:

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

Accepted Your program provides the correct output in the allotted time. Congratulations!

Presentation Error Your program is almost accepted, but the output contains extraneous or missing blanks or end-of-lines. This message occurs rarely.

Compilation Error The compilation of your program generates errors. Often, clicking on this message will provide the nature of the error. Be sure to compare the version of the compiler used by the judge with your own.

Wrong Answer Re-read the problem statement, a detail must have been overlooked. Are you sure to have tested all the limit cases? Might you have left debugging statements in your code?

Time Limit Exceeded You have probably not implemented the most efficient algorithm for this problem, or perhaps have an infinite loop somewhere. Test your loop invariants to ensure loop termination. Generate a large input instance and test locally the performance of your code.

Runtime Error In general, this could be a division by zero, an access beyond the limits of an array, or a `pop()` on an empty stack. However, other situations can also generate this message, such as the use of `assert` in Java, which is often not accepted.

The taciturn behaviour of the judges nevertheless allows certain information to be gleaned from the instances. Here is a trick that was used during an ICPC / SWERC contest. In a problem concerning graphs, the statement indicated that the input consisted of connected graphs. One of the teams doubted this, and wrote a connectivity test. In the positive case, the program entered into an infinite loop, while in the negative case, it caused a division by zero. The error code generated by the judge (Time Limit Exceeded ou Runtime Error) allowed the team to detect that certain graphs in the input were not connected.

1.2 Python in a Few Words

The programming language Python was chosen for this book, for its readability and ease of use. In September 2017, Python was identified by the website `https://stackoverflow.com` as the programming language with the greatest growth in high-income countries, in terms of the number of questions seen on the website, notably thanks to the popularity of machine learning.[1] Python is also the language retained for such important projects as the formal calculation system SageMath, whose critical portions are nonetheless implemented in more efficient languages such as C++ or C.

Here are a few details on this language. This chapter is a short introduction to Python and does not claim to be exhaustive or very formal. For the neophyte reader we recommend the site `python.org`, which contains a high-quality introduction as well as exceptional documentation. A reader already familiar with Python can profit

[1] `https://stackoverflow.blog/2017/09/06/incredible-growth-python/`

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

enormously by studying the programs of David Eppstein, which are very elegant and highly readable. Search for the keywords `Eppstein PADS`.

Python is an interpreted language. Variable types do not have to be declared, they are simply inferred at the time of assignment. There are neither keywords `begin/end` nor brackets to group instructions, for example in the blocks of a function or a loop. The organisation in blocks is simply based on indentation! A typical error, difficult to identify, is an erroneous indentation due to spaces used in some lines and tabs in others.

Basic Data Types

In Python, essentially four basic data types exist: Booleans, integers, floating-point numbers and character strings. In contrast with most other programming languages, the integers are not limited to a fixed number of bits (typically 64), but use an arbitrary precision representation. The functions—more precisely the *constructors*: `bool, int, float, str`—allow the conversion of an object to one of these basic types. For example, to access the digits of a specific integer given its decimal representation, it can be first transformed into a string, and then the characters of the string can be accessed. However, in contrast with languages such as C, it is not possible to directly modify a character of a string: strings are *immutable*. It is first necessary to convert to a list representation of the characters; see below.

Data Structures

The principal complex data structures are dictionaries, sets, lists and n-tuples. These structures are called *containers*, as they contain several objects in a structured manner. Once again, there are functions `dict`, `set`, `list` and `tuple` that allow the conversion of an object into one of these structures. For example, for a string `s`, the function `list(s)` returns a list `L` composed of the characters of the string. We could then, for example, replace certain elements of the list `L` and then recreate a string by concatenating the elements of `L` with the expression `''.join(L)`. Here, the empty string could be replaced by a separator: for example, `'-'.join(['A','B','C'])` returns the string `"A-B-C"`.

1.2.1 Manipulation of Lists, n-tuples, Dictionaries

Note that lists in Python are not linked lists of cells, each with a pointer to its successor in the list, as is the case in many other languages. Instead, lists are arrays of elements that can be accessed and modified using their index into the array. A list is written by enumerating its elements between square brackets `[` and `]`, with the elements separated by commas.

Lists The indices of a list start with 0. The last element can also be accessed with the index -1, the second last with -2 and so on. Here are some examples of operations to extract elements or sublists of a list. This mechanism is known as *slicing*, and is also available for strings.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

The following expressions have a complexity linear in the length of L, with the exception of the first, which is in constant time.

`L[i]`	the `i`th element of `L`
`L[i:j]`	the list of elements with indices starting at `i` and up to (but not including) `j`
`L[:j]`	the first `j` elements
`L[i:]`	all the elements from the `i`th onwards
`L[-3:]`	the last three elements of `L`
`L[i:j:k]`	elements from the `i`th up to (but not including) the `j`th, taking only every `k`th element
`L[::2]`	the elements of `L` with even indices
`L[::-1]`	a reverse copy of `L`

The most important methods of a list for our usage are listed below. Their complexity is expressed in terms of n, the length of the list `L`. A function has *constant amortised complexity* if, for several successive calls to it, the average complexity is constant, even if some of these calls take a time linear in n.

`len(L)`	returns the number of elements of the list `L`	$O(1)$
`sorted(L)`	returns a sorted copy of the list `L`	$O(n \log n)$
`L.sort()`	sorts `L` in place	$O(n \log n)$
`L.count(c)`	the number of occurrences of `c` in `L`	$O(n)$
`c in L`	is the element `c` found in `L`?	$O(n)$
`L.append(c)`	append `c` to the end of `L`	amortised $O(1)$
`L.pop()`	extracts and returns the last element of `L`	amortised $O(1)$

Thus a list has all the functionality of a stack, defined in Section 1.5.1 on page 20.

***n*-tuple** An n-tuple behaves much like a list, with a difference in the usage of parentheses to describe it, as in `(1, 2)` or `(left, 'X', right)`. A 1-tuple, composed of only one element, requires the usage of a comma, as in `(2,)`. A 0-tuple, empty, can be written as `()`, or as `tuple()`, no doubt more readable. The main difference with lists is that n-tuples are immutable, just like strings. This is why an n-tuple can serve as a key in a dictionary.

Dictionaries Dictionaries are used to associate objects with values, for example the words of a text with their frequency. A dictionary is constructed as comma-separated key:value pairs between curly brackets, such as `{'the': 4, 'bread': 1, 'is': 6}`, where the keys and values are separated by a colon. An empty dictionary is obtained with `{}`. A membership test of a key `x` in a dictionary `dic` is written `x in dic`. Behind a dictionary there is a hash table, hence the complexity of the expressions `x in dic`, `dic[x]`, `dic[x] = 42` is in practice constant time, even if the worst-case complexity is linear, a case obtained in the improbable event of all keys having the same hash value.

If the keys of a dictionary are all the integers between 0 and $n-1$, the use of a list is much more efficient.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

A loop in Python is written either with the keyword `for` or with `while`. The notation for the loop `for` is `for x in S:`, where the variable `x` successively takes on the values of the container `S`, or of the keys of `S` in the case of a dictionary. In contrast, `while L:` will loop as long as the list `L` is non-empty. Here, an implicit conversion of a list to a Boolean is made, with the convention that only the empty list converts to `False`.

At times, it is necessary to handle at the same time the values of a list along with their positions (indices) within the list. This can be implemented as follows:

```
for index in range(len(L)):
    value = L[index]
    # ... handling of index and value
```

The function `enumerate` allows a more compact expression of this loop:

```
for index, value in enumerate(L):
    # ... handling of index and value
```

For a dictionary, the following loop iterates simultaneously over the keys and values:

```
for key, value in dic.items():
    # ... handling of key and value
```

1.2.2 Specificities: List and Dictionary Comprehension, Iterators

List and Dictionary Comprehension

The language Python provides a syntax close to the notation used in mathematics for certain objects. To describe the list of squares from 0 to n^2, it is possible to use a *list comprehension*:

```
>>> n = 5
>>> squared_numbers = [x ** 2 for x in range(n + 1)]
>>> squared_numbers
[0, 1, 4, 9, 16, 25]
```

which corresponds to the set $\{x^2 | x = 0, \ldots, n\}$ in mathematics.

This is particularly useful to initialise a list of length n:

```
>>> t = [0 for _ in range(n)]
>>> t
[0, 0, 0, 0, 0]
```

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

or to initialise counters for the letters in a string:

```
>>> my_string = "cowboy bebop"
>>> nb_occurrences = {letter: 0 for letter in my_string}
>>> nb_occurrences
{'c': 0, 'o': 0, 'w': 0, 'b': 0, 'y': 0, ' ': 0, 'e': 0, 'p': 0}
```

The second line, a *dictionary comprehension*, is equivalent to the following:

```
nb_occurrences = {}
for letter in my_string:
    nb_occurrences[letter] = 0
```

Ranges and Other Iterators

To loop over ranges of integers, the code `for i in range(n):` can be used to run over the integers from 0 to $n - 1$. Several variants exist:

`range(k, n)`	from k to $n - 1$
`range(k, n, 2)`	from k to $n - 1$ two by two
`range(n - 1, -1, -1)`	from $n - 1$ to 0 (-1 excluded) in decreasing order.

In early versions of Python, `range` returned a list. Nowadays, for efficiency, it returns an object known as an *iterator*, which produces integers one by one, if and when the `for` loop claims a value. Any function can serve as an iterator, as long as it can produce elements at different moments of its execution using the keyword `yield`. For example, the following function iterates over all pairs of elements of a given list:.

```
def all_pairs(L):
    n = len(L)
    for i in range(n):
        for j in range(i + 1, n):
            yield (L[i], L[j])
```

1.2.3 Useful Modules and Packages

Modules

Certain Python objects must be imported from a module or a package with the command `import`. A package is a collection of modules. Two methods can be used; the second avoids potential naming collisions between the methods in different modules:

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
>>> from math import sqrt
>>> sqrt(4)
2
>>> import math
>>> math.sqrt(4)
2
```

math This module contains mathematical functions and constants such as `log`, `sqrt`, `pi`, etc. Python operates on integers with arbitrary precision, thus there is no limit on their size. As a consequence, there is no integer equivalent to represent $-\infty$ or $+\infty$. For floating point numbers on the other hand, `float('-inf')` and `float('inf')` can be used. Beginning with Python 3.5, `math.inf` (or `from math import inf`) is equivalent to `float('inf')`.

fractions This module exports the class `Fraction`, which allows computations with fractions without the loss of precision of floating point calculations. For example, if `f` is an instance of the class Fraction, then `str(f)` returns a string similar to the form `“3/2”`, expressing `f` as an irreducible fraction.

bisect Provides binary (dichotomous) search functions on a sorted list.

heapq Provides functions to manipulate a list as a heap, thus allowing an element to be added or the smallest element removed in time logarithmic in the size of the heap; see Section 1.5.4 on page 22.

string This module provides, for example, the function `ascii_lowercase`, which returns its argument converted to lowercase characters. Note that the strings themselves already provide numerous useful methods, such as `strip`, which removes whitespace from the beginning and end of a string and returns the result, `lower`, which converts all the characters to lowercase, and especially `split`, which detects the substrings separated by spaces (or by another separator passed as an argument). For example, `“12/OCT/2018”.split(“/”)` returns `[“12”, “OCT”, “2018”]`.

Packages

One of the strengths of Python is the existence of a large variety of code packages. Some are part of the standard installation of the language, while others must be imported with the shell command `pip`. They are indexed and documented at the web site `pypi.org`. Here is a non-exhaustive list of very useful packages.

tryalgo All the code of the present book is available in a package called `tryalgo` and can be imported in the following manner: `pip install tryalgo`.

```
>>> import tryalgo
>>> help(tryalgo)         # for the list of modules
>>> help(tryalgo.arithm)  # for a particular module
```

collections To simplify life, the class `from collections import Counter` can be used. For an object `c` of this class, the expression `c[x]` will return 0 if `x` is not

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

a key in `c`. Only modification of the value associated with `x` will create an entry in `c`, such as, for example, when executing the instruction `c[x] += 1`. This is thus slightly more practical than a dictionary, as is illustrated below.

```
>>> c = {}                 # dictionary
>>> c['a'] += 1            # the key does not exist
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
>>> c['a'] = 1
>>> c['a'] += 1            # now it does
>>> c
{'a': 2}
>>> from collections import Counter
>>> c = Counter()
>>> c['a'] += 1            # the key does not exist, so it is created
Counter({'a': 1})
>>> c = Counter('cowboy bebop')
Counter({'o': 3, 'b': 3, 'c': 1, 'w': 1, 'y': 1, ' ': 1, 'e': 1, 'p': 1})
```

The `collections` package also provides the class `deque`, for *double-ended queue*, see Section 1.5.3 on page 21. With this structure, elements can be added or removed either from the left (head) or from the right (tail) of the queue. This helps implement Dijkstra's algorithm in the case where the edges have only weights 0 or 1, see Section 8.2 on page 126.

This package also provides the class `defaultdict`, which is a dictionary that assigns default values to keys that are yet in the dictionary, hence a generalisation of the class `Counter`.

```
>>> from collections import defaultdict
>>> g = defaultdict(list)
>>> g['paris'].append('marseille')  # 'paris' key is created on the fly
>>> g['paris'].append('lyon')
>>> g
defaultdict(<class 'list'>, {'paris': ['marseille', 'lyon']})
>>> g['paris']  # behaves like a dict
['marseille', 'lyon']
```

See also Section 1.3 on page 13 for an example of reading a graph given as input.

numpy This package provides general tools for numerical calculations involving manipulations of large matrices. For example, `numpy.linalg.solve` solves a linear system, while `numpy.fft.fft` calculates a (fast) discrete Fourier transform.

While writing the code of this book, we have followed the norm PEP8, which provides precise recommendations on the usage of blanks, the choice of names for variables, etc. We advise the readers to also follow these indications, using, for example, the tool `pycodestyle` to validate the structure of their code.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

1.2.4 Interpreters `Python`, `PyPy`, and `PyPy3`

We have chosen to implement our algorithms in Python 3, which is already over 12 years old,[2] while ensuring backwards compatibility with Python 2. The principal changes affecting the code appearing in this text concern `print` and division between integers. In Python 3, `5 / 2` is equal to `2.5`, whereas it gives `2` in Python 2. The integer division operator `//` gives the same result for both versions. As for `print`, in Python 2 it is a keyword, whereas in Python 3 it is a function, and hence requires the parameters to be enclosed by parentheses.

The interpreter of reference is CPython, and its executable is just called `python`. According to your installation, the interpreter of Python 3 could be called `python` or `python3`. Another much more efficient interpreter is PyPy, whose executable is called `pypy` in version 2 and `pypy3` in version 3. It implements a subset of the Python language, called RPython, with quite minimal restrictions, which essentially allow the inference of the type of a variable by an analysis of the source code. The inconvenience is that `pypy` is still under development and certain modules are not yet available. But it can save your life during a contest with time limits!

1.2.5 Frequent Errors

Copy

An error often made by beginners in Python concerns the copying of lists. In the following example, the list B is in fact just a reference to A. Thus a modification of $B[0]$ results also in a modification of $A[0]$.

```
A = [1, 2, 3]
B = A  # Beware! Both variables refer to the same object
```

For B to be a distinct copy of A, the following syntax should be used:

```
A = [1, 2, 3]
B = A[:]  # B becomes a distinct copy of A
```

The notation [:] can be used to make a copy of a list. It is also possible to make a copy of all but the first element, $A[1:]$, or all but the last element, $A[:-1]$, or even in reverse order $A[::-1]$. For example, the following code creates a matrix M, all of whose rows are the same, and the modification of $M[0][0]$ modifies the whole of the first column of M.

```
M = [[0] * 10] * 10  # Do not write this!
```

[2] Python 3.0 final was released on 3 December, 2008.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

A square matrix can be correctly initialised using one of the following expressions:

```
M1 = [[0] * 10 for _ in range(10)]
M2 = [[0 for j in range(10)] for i in range(10)]
```

The module numpy permits easy manipulations of matrices; however, we have chosen not to profit from it in this text, in order to have generic code that is easy to translate to Java or C++.

Ranges

Another typical error concerns the use of the function range. For example, the following code processes the elements of a list A between the indices 0 and 9 inclusive, in order.

```
for i in range(0, 10):  # 0 included, 10 excluded
    process(A[i])
```

To process the elements in descending order, it is not sufficient to just swap the arguments. In fact, range(10, 0, -1)—the third argument indicates the step—is the list of elements with indices 10 (included) to 0 (excluded). Thus the loop must be written as:

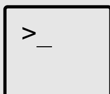
```
for i in range(9, -1, -1):  # 9 included, -1 excluded
    process(A[i])
```

1.3 Input-Output

1.3.1 Read the Standard Input

For most problems posed by programming contests, the input data are read from *standard input*, and the responses displayed on *standard output*. For example, if the input file is called test.in, and your program is prog.py, the contents of the input file can be directed to your program with the following command, launched from a command window:

```
python prog.py < test.in
```



In general, under Mac OS X, a command window can be obtained by typing *Command-Space Terminal*, and under Windows, via *Start* → *Run* → *cmd.*
If you are running Linux, the keyboard shortcut is generally *Alt-F2*, but that you probably already knew…

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

If you wish to save the output of your program to a file called `test.out`, type:

```
python prog.py < test.in > test.out
```

A little hint: if you want to display the output at the same time as it is being written to a file `test.out`, use the following (the command `tee` is not present by default in Windows):

```
python prog.py < test.in | tee test.out
```

The inputs can be read line by line via the command `input()`, which returns the next input line in the form of a string, excluding the end-of-line characters.[3] The module `sys` contains a similar function `stdin.readline()`, which does not suppress the end-of-line characters, but according to our experience has the advantage of being four times as fast!

If the input line is meant to contain an integer, we can convert the string with the function `int` (if it is a floating point number, then we must use `float` instead). In the case of a line containing several integers separated by spaces, the string can first be cut into different substrings using `split()`; these can then be converted into integers with the method `map`. For example, in the case of two integers `height` and `width` to be read on the same line, separated by a space, the following command suffices:

```
import sys

height, width = map(int, sys.stdin.readline().split())
```

If your program exhibits performance problems while reading the inputs, our experience shows that a factor of two can be gained by reading the whole of the inputs with a single system call. The following code fragment assumes that the inputs are made up of only integers, eventually on multiple lines. The parameter 0 in the function `os.read` means that the read is from standard input, and the constant `M` must be an upper bound on the size of the file. For example, if the file contains 10^7 integers, each between 0 and 10^9, then as each integer is written with at most 10 characters and there are at most 2 characters separating the integers (`\n` and `\r`), we can choose $M = 12 \cdot 10^7$.

[3] According to the operating system, the end-of-line is indicated by the characters `\r`, `\n`, or both, but this is not important when reading with `input()`. Note that in Python 2 the behaviour of `input()` is different, so it is necessary to use the equivalent function `raw_input()`.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
import os

instance = list(map(int, os.read(0, M).split()))
```

Example – Read a Graph on Input

If the inputs are given in the form:

```
3
paris tokyo 9471
paris new-york 5545
new-york singapore 15344
```

where 3 is the number of edges of a graph and each edge is represented by `<departure> <arrival> <distance>`, then the following code, using `defaultdict` to initialise the new keys in an empty dictionary, allows the construction of the graph:

```
from collections import defaultdict

nb_edges = int(input())

g = defaultdict(dict)
for _ in range(nb_edges):
    u, v, weight = input().split()
    g[u][v] = int(weight)
    # g[v][u] = int(weight)  # For an undirected graph
```

Example—read three matrices A,B,C and test if $AB = C$

In this example, the inputs are in the following form: the first line contains a single integer n. It is followed by $3n$ lines each containing n integers separated by spaces, coding the values contained in three $n \times n$ matrices A,B,C, given row by row. The goal is to test if the product A times B is equal to the matrix C. A direct approach by matrix multiplication would have a complexity $O(n^3)$. However, a probabilistic solution exists in $O(n^2)$, which consists in randomly choosing a vector x and testing whether $A(Bx) = Cx$. This is the *Freivalds test* (1979). What is the probability that the algorithm outputs equality even if $AB \neq C$? Whenever the computations are made modulo d, the probability of error is at most $1/d$. This error probability can be made arbitrarily small by repeating the test several times. The following code has an error probability bounded above by 10^{-6}.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
from random import randint
from sys import stdin



def readint():
    return int(stdin.readline())


def readarray(typ):
    return list(map(typ, stdin.readline().split()))
def readmatrix(n):
    M = []
    for _ in range(n):
        row = readarray(int)
        assert len(row) == n
        M.append(row)
    return M


def mult(M, v):
    n = len(M)
    return [sum(M[i][j] * v[j] for j in range(n)) for i in range(n)]


def freivalds(A, B, C):
    n = len(A)
    x = [randint(0, 1000000) for j in range(n)]
    return mult(A, mult(B, x)) == mult(C, x)


if __name__ == "__main__":
    n = readint()
    A = readmatrix(n)
    B = readmatrix(n)
    C = readmatrix(n)
    print(freivalds(A, B, C))
```

Note the test on the variable `__name__`. This test is evaluated as True if the file containing this code is called directly, and as False if the file is included with the `import` keyword.

Problem

Enormous Input Test [spoj:INTEST]

1.3.2 Output Format

The outputs of your program are displayed with the command `print`, which produces a new line with the values of its arguments. The generation of end-of-line characters can be suppressed by passing `end=''` as an argument.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

To display numbers with fixed precision and length, there are at least two possibilities in Python. First of all, there is the operator `%` that works like the function `printf` in the language C. The syntax is `s % a`, where `s` is a format string, a character string including typed display indicators beginning with `%`, and where `a` consists of one or more arguments that will replace the display indicators in the format string.

```
>>> i_test = 1
>>> answer = 1.2142
>>> print("Case #%d: %.2f gigawatts!!!" % (i_test, answer))
Case #1: 1.21 gigawatts!!!
```

The letter `d` after the `%` indicates that the first argument should be interpreted as an integer and inserted in place of the `%d` in the format string. Similarly, the letter `f` is used for floats and `s` for strings. A percentage can be displayed by indicating `%%` in the format string. Between the character `%` and the letter indicating the type, further numerical indications can be given. For example, `%.2f` indicates that up to two digits should be displayed after the decimal point.

Another possibility is to use the method `format` of a string, which follows the syntax of the language C#. This method provides more formatting possibilities and is in general easier to manipulate.

```
>>> print("Case #{}: {:.2f} gigawatts!!!".format(i_test, answer))
Case #1: 1.21 gigawatts!!!
```

Finally, beginning with Python 3.6, *f-strings*, or formatted string literals, exist.

```
>>> print(f"Case #{testCase}: {answer:.2f} gigawatts!!!")
Case #1: 1.21 gigawatts!!!
```

In this case, the floating point precision itself can be a variable, and the formatting is embedded with each argument.

```
>>> precision = 2
>>> print(f"Case #{testCase}: {answer:.{precision}f} gigawatts!!!")
Case #1: 1.21 gigawatts!!!
```

1.4 Complexity

To write an efficient program, it is first necessary to find an algorithm of appropriate complexity. This complexity is expressed as a function of the size of the inputs. In order to easily compare complexities, the notation of Landau symbols is used.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

Landau Symbols

The complexity of an algorithm is, for example, said to be $O(n^2)$ if the execution time can be bounded above by a quadratic function in n, where n represents the size or some parameter of the input. More precisely, for two functions f, g we denote $f \in O(g)$ if positive constants n_0, c exist, such that for every $n \geq n_0$, $f(n) \leq c \cdot g(n)$. By an abuse of notation, we also write $f = O(g)$. This notation allows us to ignore the multiplicative and additive constants in a function f and brings out the magnitude and form of the dependence on a parameter.

Similarly, if for constants $n_0, c > 0$ we have $f(n) \geq c \cdot g(n)$ for every $n \geq n_0$, then we write $f \in \Omega(g)$. If $f \in O(g)$ and $f \in \Omega(g)$, then we write $f \in \Theta(g)$, indicating that f and g have the same order of magnitude of complexity. Finally, if $f \in O(g)$ but not $g \in O(f)$, then we write $f \in o(g)$

Complexity Classes

If the complexity of an algorithm is $O(n^c)$ for some constant c, it is said to be *polynomial* in n. A problem for which a polynomial algorithm exists is said to be *polynomial*, and the class of such problems bears the name P. Unhappily, not all problems are polynomial, and numerous problems exist for which no polynomial algorithm has been found to this day.

One such problem is k-SAT: Given n Boolean variables and m clauses each containing k literals (a variable or its negation), is it possible to assign to each variable a Boolean value in such a manner that each clause contains at least one literal with the value `True` (SAT is the version of this problem without a restriction on the number of variables in a clause)? The particularity of each of these problems is that a potential solution (assignment to each of the variables) satisfying all the constraints can be verified in polynomial time by evaluating all the clauses: they are in the class NP (for Non-deterministic Polynomial). We can easily solve 1-SAT in polynomial time, hence 1-SAT is in P. 2-SAT is also in P; this is the subject of Section 6.9 on page 110. However, from 3-SAT onwards, the answer is not known. We only know that solving 3-SAT is at least as difficult as solving SAT.

It turns out that $\mathsf{P} \subseteq \mathsf{NP}$—intuitively, if we can construct a solution in polynomial time, then we can also verify a solution in polynomial time. It is believed that $\mathsf{P} \neq \mathsf{NP}$, but this conjecture remains unproven to this day. In the meantime, researchers have linked NP problems among themselves using *reductions*, which transform in polynomial time an algorithm for a problem A into an algorithm for a problem B. Hence, if A is in P, then B is also in P: A is 'at least as difficult' as B.

The problems that are at least as difficult as SAT constitute the class of problems NP-hard, and among these we distinguish the NP-complete problems, which are defined as those being at the same time NP-hard and in NP. Solve any one of these in polynomial time and you will have solved them all, and will be gratified by eternal recognition, accompanied by a million dollars.[4] At present, to solve these problems in an acceptable time, it is necessary to restrict them to instances with properties

[4] www.claymath.org/millennium-problems/p-vs-np-problem

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

that can aid the resolution (planarity of a graph, for example), permit the program to answer correctly with only a constant probability or produce a solution only close to an optimal solution.

Happily, most of the problems encountered in programming contests are polynomial.

If these questions of complexity classes interest you, we recommend Christos H. Papadimitriou's book on computational complexity (2003).

For programming contests in particular, the programs must give a response within a few seconds, which gives current processors the time to execute on the order of tens or hundreds of millions of operations. The following table gives a rough idea of the acceptable complexities for a response time of a second. These numbers are to be taken with caution and, of course, depend on the language used,[5] the machine that will execute the code and the type of operation (integer or floating point calculations, calls to mathematical functions, etc.).

size of input	acceptable complexity
1000000	$O(n)$
100000	$O(n \log n)$
1000	$O(n^2)$

The reader is invited to conduct experiments with simple programs to test the time necessary to execute n integer multiplications for different values of n. We insist here on the fact that the constants lurking behind the Landau symbols can be very large, and at times, in practice, an algorithm with greater asymptotic complexity may be preferred. An example is the multiplication of two $n \times n$ matrices. The naive method costs $O(n^3)$ operations, whereas a recursive procedure discovered by Strassen (1969) costs only $O(n^{2.81})$. However, the hidden multiplicative constant is so large that for the size of matrices that you might need to manipulate, the naive method without doubt will be more efficient.

In Python, adding an element to a list takes constant time, as does access to an element with a given index. The creation of a sublist with $L[i : j]$ requires time $O(\max\{1, j - i\})$. Dictionaries are represented by hash tables, and the insertion of a key-value pair can be done in constant time, see Section 1.5.2 on page 21. However, this time constant is non-negligible, hence if the keys of the dictionary are the integers from 0 to $n - 1$, it is preferable to use a list.

Amortised Complexity

For certain data structures, the notion of amortised time complexity is used. For example, a list in Python is represented internally by an array, equipped with a variable *length* storing the size. When a new element is added to the array with the method `append`, it is stored in the array at the index indicated by the variable `length`, and then

[5] Roughly consider Java twice as slow and Python four times as slow as C++.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

the variable `length` is incremented. If the capacity of the array is no longer sufficient, a new array twice as large is allocated, and the contents of the original array are copied over. Hence, for n successive calls to `append` on a list initially empty, the time of each call is usually constant, and it is occasionally linear in the current size of the list. However, the sum of the time of these calls is $O(n)$, which, once spread out over each of the calls, gives an amortised time $O(1)$.

1.5 Abstract Types and Essential Data Structures

We will now tackle a theme that is at the heart of the notion of efficient programming: the data structures underlying our programs to solve problems.

An *abstract type* is a description of the possible values that a set of objects can take on, the operations that can be performed on these objects and the behaviour of these operations. An abstract type can thus be seen as a *specification*.

A *data structure* is a concrete way to organise the data in order to treat them efficiently, respecting the clauses in the specification. Thus, we can implement an abstract type by one or more different data structures and determine the complexity in time and memory of each operation. Thereby, based on the frequency of the operations that need to be executed, we will prefer one or another of the implementations of an abstract type to solve a given problem.

To program well, a mastery of the data structures offered by a language and its associated standard library is essential. In this section, we review the most useful data structures for programming competitions.

1.5.1 Stacks

A **stack** (see Figure 1.2) is an object that keeps track of a set of elements and provides the following operations: test if the stack is empty, add an element to the top of the stack *(push)*, access the element at the top of the stack and remove an element *(pop)*. Python lists can serve as stacks. An element is pushed with the method `append(element)` and popped with the method `pop()`.

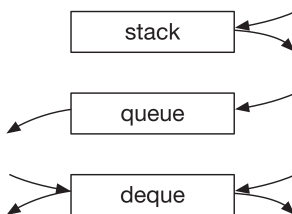


Figure 1.2 The three principal sequential access data structures provided by Python.

If a list is used as a Boolean, for example as the condition of an `if` or `while` instruction, it takes on the value `True` if and only if it is non-empty. In fact, this is

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

the case for all objects implementing the method `__len__`; for these, `if x:` behaves exactly like `if len(x) > 0:`. All of these operations execute in constant time.

1.5.2 Dictionaries

A **dictionary** allows the association of keys to values, in the same manner that an array associates indices to values. The internal workings are based on a hash table, which uses a hash function to associate the elements with indices in an array, and implements a mechanism of collision management in the case where different elements are sent to the same index. In the best case, the read and write operations on a dictionary execute in constant time, but in the worst case they execute in linear time if it is necessary to run through a linear number of keys to handle the collisions. In practice, this degenerate case arrives only rarely, and in this book, we generally assume that accesses to dictionary entries take constant time. If the keys are of the form $0, 1, \ldots, n-1$, it is, of course, always preferable for performance reasons to use a simple array.

Problems
Encryption [spoj:CENCRY]
Phone List [spoj:PHONELST]
A concrete simulation [spoj:ACS]

1.5.3 Queues

A **queue** is similar to a stack, with the difference that elements are added to the end of the queue (enqueued) and are removed from the front of the queue (dequeued). A queue is also known as a FIFO queue (*first in, first out*, like a waiting line), whereas a stack is called LIFO (*last in, first out*, like a pile of plates).

In the Python standard library, there are two classes implementing a queue. The first, `Queue`, is a synchronised implementation, meaning that several processes can access it simultaneously. As the programs of this book do not exploit parallelism, the use of this class is not recommended, as it entails a slowdown because of the use of semaphores for the synchronisation. The second class is `deque` (for *double-ended queue*). In addition to the methods `append(element)` and `popleft()`, which, respectively, add to the end of the queue and remove from the head of the queue, `deque` offers the methods `appendleft(element)` and `pop()`, which add to the head of the queue and remove from the end of the queue. We can thus speak of a **queue with two ends**. This more sophisticated structure will be found useful in Section 8.2 on page 126, where it is used to find the shortest path in a graph the edges of which have weights 0 or 1.

We recommend the use of `deque`—and in general, the use of the data structures provided by the standard library of the language—but as an example we illustrate here how to implement a queue using two stacks. One stack corresponds to the head of the queue for extraction and the other corresponds to the tail for insertion. Once the head

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

stack is empty, it is swapped with the tail stack (reversed). The operator `__len__` uses `len(q)` to recover the number of elements in the queue `q`, and then `if q` can be used to test if `q` is non-empty, happily in constant time.

```
class OurQueue:
    def __init__(self):
        self.in_stack = []        # tail
        self.out_stack = []       # head

    def __len__(self):
        return len(self.in_stack) + len(self.out_stack)

    def push(self, obj):
        self.in_stack.append(obj)
    def pop(self):
        if not self.out_stack:    # head is empty
            # Note that the in_stack is assigned to the out_stack
            #  in reverse order. This is because the in_stack stores
            #  elements from oldest to newest whereas the out_stack
            #  needs to pop elements from newest to oldest
            self.out_stack = self.in_stack[::-1]
            self.in_stack = []
        return self.out_stack.pop()
```

1.5.4 Priority Queues and Heaps

A **priority queue** is an abstract type allowing elements to be added and an element with minimal key to be removed. It turns out to be very useful, for example, in sorting an array (heapsort), in the construction of a Huffman code (see Section 10.1 on page 172) and in the search for the shortest path between two nodes in a graph (see Dijkstra's algorithm, Section 8.3 on page 127). It is typically implemented with a heap, which is a data structure in the form of a tree.

Perfect and Quasi-Perfect Binary Trees

We begin by examining a very specific type of data structure: the quasi-perfect binary tree. For more information on these trees, see Chapter 10 on page 171, dedicated to them.

A binary tree is either empty or is made up of a node with two children or subtrees, left and right. The node at the top of the tree is the root, while the nodes with two empty children, at the bottom of the tree, are the leaves. A binary tree is *perfect* if all its leaves are at the same distance from the root. It is *quasi-perfect* if all its leaves are on, at most, two levels, the second level from the bottom is completely full and all the leaves on the bottom level are grouped to the left. Such a tree can conveniently be represented by an array, see Figure 1.3. The element at index 0 of this array is ignored. The root is found at index 1, and the two children of a node i are at $2i$ and $2i + 1$. Simple index manipulations allow us to ascend or descend in the tree.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

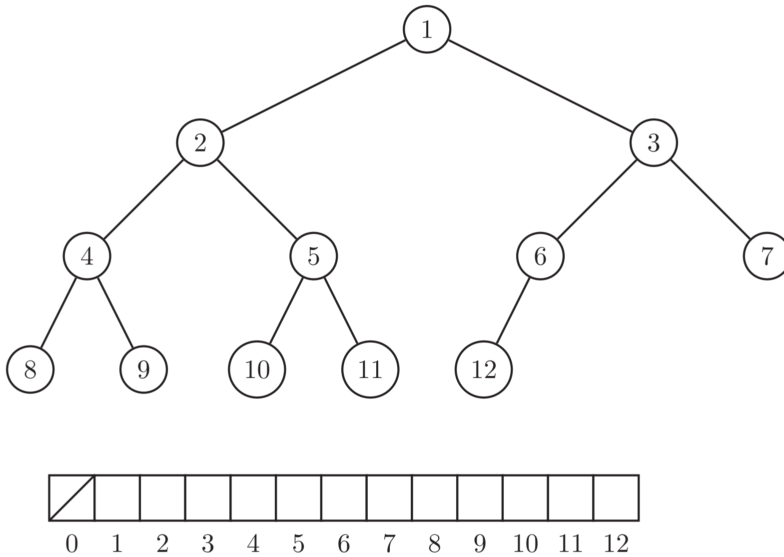


Figure 1.3 A quasi-perfect binary tree and its representation by an array.

Priority Queues and Heaps

A *heap*, also known as a tournament tree, is a tree verifying the heap property: each node has a key smaller than each of its children, for a certain order relation. The root of a heap is thus the element with the smallest key. A variant known as a *max-heap* exists, wherein each node has a key greater that each of its children.

In general, we are interested in binary heaps, which are quasi-perfect binary tournament trees. This data structure allows the extraction of the smallest element and the insertion of a new element with a logarithmic cost, which is advantageous. The objects in question can be from an arbitrary set of elements equipped with a total order relation. A heap also allows an update of the priority of an element, which is a useful operation for Dijkstra's algorithm (see Section 8.1 on page 125) when a shorter path has been discovered towards a summit.

In Python, heaps are implemented by the module `heapq`. This module provides a function to transform an array A into a heap (`heapify(A)`), which results in an array representing a quasi-perfect tree as described in the preceding section, except that the element with index 0 does not remain empty and instead contains the root. The module also allows the insertion of a new element (`heappush(heap,element)`) and the extraction of the minimal element (`heappop(heap)`).

However, this module does not permit the value of an element in the heap to be changed, an operation useful for Dijkstra's algorithm to improve the time complexity. We thus propose the following implementation, which is more complete.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

Implementation details

The structure contains an array `heap`, storing the heap itself, as well as a dictionary `rank`, allowing the determination of the index of an element stored in the heap. The principal operations are `push` and `pop`. A new element is inserted with `push`: it is added as the last leaf in the heap, and then the heap is reorganised to respect the heap order. The minimal element is extracted with `pop`: the root is replaced by the last leaf in the heap, and then the heap is reorganised to respect the heap order, see Figure 1.4.

The operator `__len__` returns the number of elements in the heap. The existence of this operator permits the inner workings of Python to implicitly convert a heap to a Boolean and to perform such conditional tests as, for example, `while h`, which loops while the heap `h` is non-empty.

The average complexity of the operations on our heap is $O(\log n)$; however, the worst-case complexity is $O(n)$, due to the use of the dictionary (`rank`).

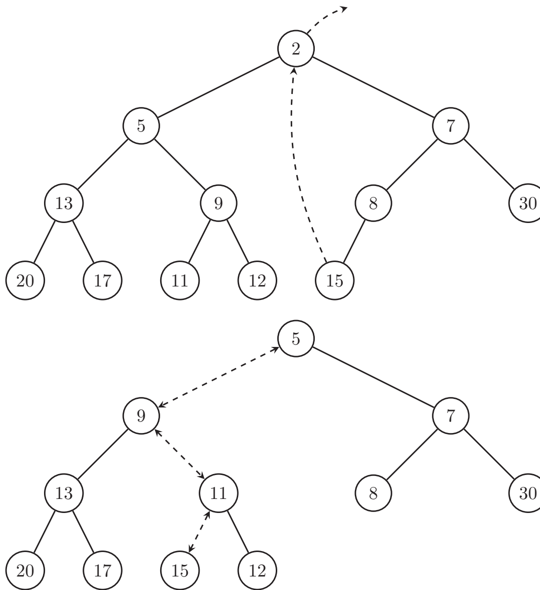


Figure 1.4 The operation `pop` removes and returns the value 2 of the heap and replaces it by the last leaf 15. Then the operation `down` performs a series of exchanges to place 15 in a position respecting the heap order.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
class OurHeap:
    def __init__(self, items):
        self.heap = [None]  # index 0 will be ignored
        self.rank = {}
        for x in items:
            self.push(x)

    def __len__(self):
        return len(self.heap) - 1

    def push(self, x):
        assert x not in self.rank
        i = len(self.heap)
        self.heap.append(x)     # add a new leaf
        self.rank[x] = i
        self.up(i)              # maintain heap order

    def pop(self):
        root = self.heap[1]
        del self.rank[root]
        x = self.heap.pop()     # remove last leaf
        if self:                # if heap is not empty
            self.heap[1] = x     # move the last leaf
            self.rank[x] = 1     # to the root
            self.down(1)         # maintain heap order
        return root
```

The reorganisation is done with the operations `up(i)` and `down(i)`, which are called whenever an element with index i is too small with respect to its parent (for `up`) or too large for its children (for `down`). Hence, `up` effects a series of exchanges of a node with its parents, climbing up the tree until the heap order is respected. The action of `down` is similar, for an exchange between a node and its child with the smallest value.

Finally, the method `update` permits the value of a heap element to be changed. It then calls `up` or `down` to preserve the heap order. It is this method that requires the introduction of the dictionary `rank`.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
    def up(self, i):
        x = self.heap[i]
        while i > 1 and x < self.heap[i // 2]:
            self.heap[i] = self.heap[i // 2]
            self.rank[self.heap[i // 2]] = i
            i //= 2
        self.heap[i] = x       # insertion index found
        self.rank[x] = i


    def down(self, i):
        x = self.heap[i]
        n = len(self.heap)
        while True:
            left = 2 * i       # climb down the tree
            right = left + 1
            if (right < n and self.heap[right] < x and
                    self.heap[right] < self.heap[left]):
                self.heap[i] = self.heap[right]
                self.rank[self.heap[right]] = i   # move right child up
                i = right
            elif left < n and self.heap[left] < x:
                self.heap[i] = self.heap[left]
                self.rank[self.heap[left]] = i    # move left child up
                i = left
            else:
                self.heap[i] = x   # insertion index found
                self.rank[x] = i
                return


    def update(self, old, new):
        i = self.rank[old]     # change value at index i
        del self.rank[old]
        self.heap[i] = new
        self.rank[new] = i
        if old < new:          # maintain heap order
            self.down(i)
        else:
            self.up(i)
```

1.5.5 Union-Find

Definition

Union-find is a data structure used to store a partition of a universe V and that allows the following operations, also known as *requests* in the context of dynamic data structures:

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

- `find(v)` returns a canonical element of the set containing v. To test if u and v are in the same set, it suffices to compare `find(u)` with `find(v)`.
- `union(u,v)` combines the set containing u with that containing v.

Application

Our principal application of this structure is to determine the connected components of a graph, see Section 6.5 on page 94. Every addition of an edge will correspond to a call to `union`, while `find` is used to test if two vertices are in the same component. Union-find is also used in Kruskal's algorithm to determine a minimal spanning tree of a weighted graph, see Section 10.4 on page 179.

Data Structures with Quasi-Constant Time per Request

We organise the elements of a set in an oriented tree towards a canonical element, see Figure 1.5. Each element `v` has a reference `parent[v]` towards an element higher in the tree. The root `v`—the canonical element of the set—is indicated by a special value in `parent[v]`: we can choose 0, −1, or `v` itself, as long as we are consistent. We also store the size of the set in an array `size[v]`, where `v` is the canonical element. There are two ideas in this data structure:

1. When traversing an element on the way to the root, we take advantage of the opportunity to compress the path, i.e. we link directly to the root that the elements encountered.
2. During a union, we hang the tree of smaller rank beneath the root of the tree of larger rank. The *rank* of a tree corresponds to the depth it would have if none of the paths were compressed.

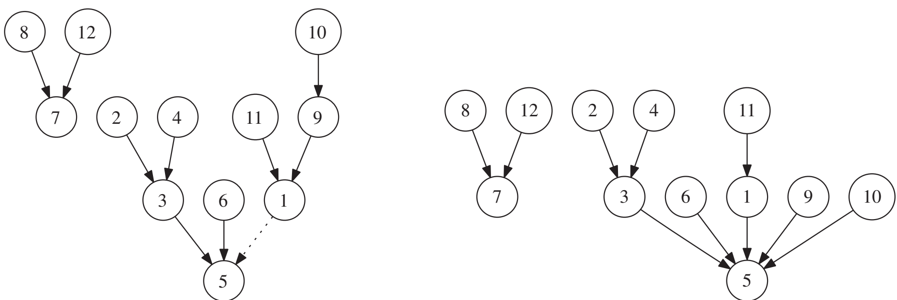


Figure 1.5 On the left: The structure union-find for a graph with two components {7, 8, 12} and {2, 3, 4, 5, 6, 9, 10, 11}. On the right: after a call to `find(10)`, all the vertices on the path to~the root now point directly to the root 5. This accelerates future calls to `find` for these vertices.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

This gives us:

```
class UnionFind:
    def __init__(self, n):
        self.up_bound = list(range(n))
        self.rank = [0] * n

    def find(self, x_index):
        if self.up_bound[x_index] == x_index:
            return x_index
        self.up_bound[x_index] = self.find(self.up_bound[x_index])
        return self.up_bound[x_index]

    def union(self, x_index, y_index):
        repr_x = self.find(x_index)
        repr_y = self.find(y_index)
        if repr_x == repr_y:       # already in the same component
            return False
        if self.rank[repr_x] == self.rank[repr_y]:
            self.rank[repr_x] += 1
            self.up_bound[repr_y] = repr_x
        elif self.rank[repr_x] > self.rank[repr_y]:
            self.up_bound[repr_y] = repr_x
        else:
            self.up_bound[repr_x] = repr_y
        return True
```

It can be proved that any arbitrary sequence of m operations `union` or `find` on a universe of size n costs a time $O((m+n)\alpha(n))$, where α is the inverse of Ackermann's function, which in practice can be considered as the constant 4.

Problem
Havannah [gcj:2013round3B]

1.6 Techniques

1.6.1 Comparison

In Python, a comparison between n-tuples is based on lexicographical order. This allows us, for example, to find, at the same time, the largest value in an array along with the corresponding index, taking the largest index in the case of equality.

```
max((tab[i], i) for i,tab_i in enumerate(tab))
```

The most common element of an array can be found by using a dictionary to count the number of occurrences of each element, and then using the above technique to select the element with the largest count, or the first element in lexicographical order if there are several candidates with the maximal count. Our implementation has a complexity of $O(nk)$ on average, but $O(n^2k)$ in the worst case because of the

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

use of a dictionary, where n is the number of words given and k is the maximal length of a word.

```
def majority(L):
    count = {}
    for word in L:
        if word in count:
            count[word] += 1
        else:
            count[word] = 1
    # Using min() like this gives the first word with
    #   maximal count "for free"
    val_1st_max, arg_1st_max = min((-count[word], word) for word in count)
    return arg_1st_max
```

1.6.2 Sorting

An array of n elements in Python can be sorted in time $O(n \log n)$. We distinguish two kinds of sorts:

- a sort with the method `sort()` modifies the list in question (it is said to sort 'in place') ;
- a sort with the function `sorted()` returns a sorted copy of the list in question.

Given a list L of n distinct integers, we would like to determine two integers in L with minimal difference. This problem can be solved by sorting L, and then running through L to select the pair with the smallest difference. Note the use of minimum over a collection of pairs of integers, compared in lexicographical order. Hence, `valmin` will contain the smallest distance between two successive elements of L and `argmin` will contain the corresponding index.

```
def closest_values(L):
    assert len(L) >= 2
    L.sort()
    valmin, argmin = min((L[i] - L[i - 1], i) for i in range(1, len(L)))
    return L[argmin - 1], L[argmin]
```

Sorting n elements requires $\Omega(n \log n)$ comparisons between the elements in the worst case. To be convinced, consider an input of n distinct integers. An algorithm must select one among $n!$ possible orders. Each comparison returns one of two values (greater or smaller) and thus divides the search space in two. Finally, it requires at most $\lceil \log_2(n!) \rceil$ comparisons to identify a particular order, giving the lower bound $\Omega(\log(n!)) = \Omega(n \log n)$.

Variants

In certain cases, an array of n integers can be sorted in time $O(n)$, for example when they are all found between 0 and cn for some constant c. In this case, we can simply

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

scan the input and store the number of occurrences of each element in an array `count` of size cn. We then scan `count` by increasing index, and write the values from 0 to cn to the output array, repeating them as often as necessary. This technique is known as *counting sort*; a similar variant is *pigeonhole sort*.

Problems
Spelling Lists [spoj:MIB]
Yodaness Level [spoj:YODANESS]

1.6.3 Sweep Line

Numerous problems in geometry can be solved with the *sweep line* technique, including many problems concerning intervals, i.e. one-dimensional geometric objects. The idea is to sweep over the different input elements from left to right and perform a specific processing for each element encountered.

Example—Intersection of Intervals
Given n intervals $[\ell_i, r_i)$ for $i = 0, \ldots, n-1$, we wish to find a value x included in a maximum number of intervals. Here is a solution in time $O(n \log n)$. We sort the endpoints, and then sweep them from left to right with an imaginary pointer `x`. A counter `c` keeps track of the number of intervals whose beginning has been seen, but not yet the end, hence it counts the number of intervals containing x.

Note that the order of processing of the elements of `B` guarantees that the right endpoints of the intervals are handled before the left endpoints of the intervals, which is necessary when dealing with intervals that are half-open to the right.

```
def max_interval_intersec(S):
    B = ([(left,  +1) for left, right in S] +
         [(right, -1) for left, right in S])
    B.sort()
    c = 0
    best = (c, None)
    for x, d in B:
        c += d
        if best[0] < c:
            best = (c, x)
    return best
```

Problem
Back to the future [prologin:demi2012]

1.6.4 Greedy Algorithms

We illustrate here an important algorithmic technique: informally, a greedy algorithm produces a solution step by step, at each step making a choice to maximise a

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

local criterion. A formal notion exists that is related to combinatorial structures known as *matroids*, and it can be used to prove the optimality or the non-optimality of greedy algorithms. We do not tackle this here, but refer to Kozen (1992) for a very good introduction.

For some problems, an optimal solution can be produced step by step, making a decision that is in a certain sense *locally optimal*. Such a proof is, in general, based on an *exchange argument*, showing that any optimal solution can be transformed into the solution produced by the algorithm without modifying the cost. The reader is invited to systematically at least sketch such a proof, as intuition can often be misleading and the problems that can be solved by a greedy algorithm are in fact quite rare.

An example is making change with coins, see Section 11.2 on page 184. Suppose you work in a store, and dispose of an infinity of coins with a finite number of distinct values. You must make change for a client for a specific sum and you wish to do it with as few coins as possible. In most countries, the values of the coins are of the form $x \cdot 10^i$ with $x \in \{1,2,5\}$ and $i \in \mathbb{N}$, and in this case we can in a greedy manner make change by repeatedly selecting the most valuable coin that is not larger than the remaining sum to return. However, in general, this approach does not work, for example with coins with values 1, 4 and 5, and with a total of 8 to return. The greedy approach generates a solution with 4 coins, i.e. $5+1+1+1$, whereas the optimal is composed of only 2 coins: $4+4$.

Example – Minimal Scalar Product

We introduce the technique with the aid of a simple example. Given two vectors x and y of n non-negative integers, find a permutation π of $\{1,\ldots,n\}$ such that the sum $\sum_i x_i y_{\pi(i)}$ is minimal.

Application

Suppose there are n tasks to be assigned to n workers in a *bijective* manner, i.e. where each task must be assigned to a different worker. Each task has a duration in hours and each worker has a price per hour. The goal is to find the assignment that minimises the total amount to be paid.

Algorithm in $O(n \log n)$

Since applying the same permutation to both x and y preserves the optimal solution, without loss of generality we can suppose that x is sorted in increasing order. We claim that a solution exists that multiplies x_0 by a maximal element y_j. Fix a permutation π with $\pi(0)=i$ and $\pi(k)=j$ for an index k and $y_i < y_j$. We observe that $x_0 y_i + x_k y_j$ is greater or equal to $x_0 y_j + x_k y_i$, which implies that without increasing the cost, π can be transformed such that x_0 is multiplied by y_j. The observation is justified by the following computation, which requires x_0, x_k to be non-negative:

$$x_0 \le x_k$$
$$x_0(y_j - y_i) \le x_k(y_j - y_i)$$

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

$$x_0 y_j - x_0 y_i \leq x_k y_j - x_k y_i$$
$$x_0 y_j + x_k y_i \leq x_0 y_i + x_k y_j.$$

By repeating the argument on the vector x with x_0 removed and on y with y_j removed, we find that the product is minimal when $i \mapsto y_{\pi(i)}$ is decreasing.

```
def min_scalar_prod(x, y):
    x1 = sorted(x)  # make copies to preserve the input arguments
    y1 = sorted(y)
    return sum(x1[i] * y1[-i - 1] for i in range(len(x1)))
```

Problems
Minimum Scalar Product [gcj:2008round1A]
Minimal Coverage [timus:1303]

1.6.5 Dynamic Programming

Dynamic programming is one of the methods that must be part of your 'Swiss army knife' as a programmer. The idea is to store a few precious bits of information so as to not have to recompute them while solving a problem (a sort of a cache), and then to reflect on how to astutely combine these bits. More formally, we decompose a problem into subproblems, and we construct the optimal solution of the principal problem out of the solutions to the subproblems.

Typically, these problems are defined over recursive structures. For example, in a rooted tree, the descendants of the root are themselves roots of subtrees. Similarly, an interval of indices in an array is the union of two shorter intervals. The reader is invited to master the dynamic programming techniques presented in this text, as numerous problems in programming competitions are variants of these classic problems.

A classic example is the computation of the nth Fibonacci number, defined by the following recurrence:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(i) = F(i-1) + F(i-2).$$

This number gives, for example, the number of possibilities to climb an n-step stairway by taking one or two steps at a time. A naive implementation of F by a recursive function is extremely inefficient, since for the same parameter i, $F(i)$ is computed several times, see Figure 1.6.

```
def fibo_naive(n):
    if n <= 1:
        return n
    return fibo_naive(n - 1) + fibo_naive(n - 2)
```

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

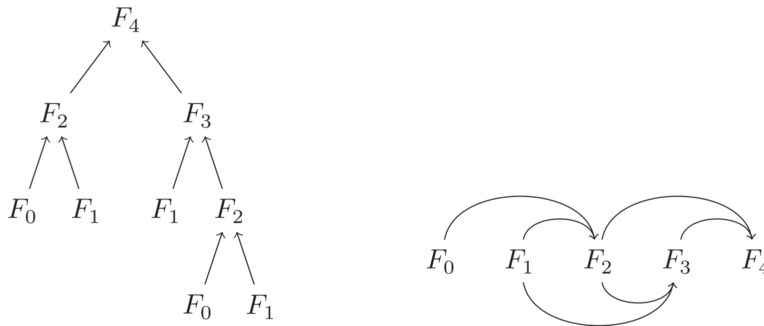


Figure 1.6 On the left, the exhaustive exploration tree of the recursive computation of the Fibonacci number F_4. On the right, the acyclic oriented graph of the value dependencies of dynamic programming, with many fewer nodes.

Try this function on a not-very-large value, $n = 32$, and you will see that the code can already take more than a second, even on a powerful machine; the computation time increases exponentially with n.

A solution by dynamic programming simply consists of storing in an array of size $n+1$ the values of $F(0)$ to $F(n)$, and filling it in increasing order of the indices. Hence, at the moment we compute $F(i)$, the values $F(i-1)$ and $F(i-2)$ will already have been computed; they are the last two values entered into the array.

```
def fibo_dp(n):
    mem = [0, 1]
    for i in range(2, n + 1):
        mem.append(mem[-2] + mem[-1])
    return mem[-1]
```

However, the above code uses $n+1$ memory locations, whereas 2 are sufficient, those corresponding to the last two Fibonacci numbers computed.

```
def fibo_dp_mem(n):
    mem = [0, 1]
    for i in range(2, n + 1):
        mem[i % 2] = mem[0] + mem[1]
    return mem[n % 2]
```

In this way, the memory complexity is reduced from linear in n (hence, it is exponential in the input size $\log n$) to $O(\log n)$; hence, it is linear in the size of the data. Indeed, the Fibonacci number n can be expressed using $O(n)$ bits.

Now that you have a complete example of the principle of dynamic programming in mind, we can reveal to you a trick using the *decorator* `@lru_cache(maxsize=None)` present in the standard library `functools`. This decorator performs a *memoisation*,[6] i.e. it stores the successive calls of a function with their arguments in a

[6] To be more precise, LRU stands for 'Least Recently Used cache', but indicating 'maxsize=None' deactivates the cache mechanism, since the cache is then non-bounded, unless for physical considerations.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

dictionary, to avoid the combinatorial explosion of the calls. In other words, adding `@lru_cache(maxsize=None)` transforms any old naive recursive implementation into a clever example of dynamic programming. Isn't life beautiful?

```
from functools import lru_cache


@lru_cache(maxsize=None)
def fibo_naive(n):
    if n <= 1:
        return n
    return fibo_naive(n - 1) + fibo_naive(n - 2)
```

This code is as fast as the implementation using dynamic programming, thanks to the memoisation provided by the decorator `@lru_cache`.

Problem
Fibonacci Sum [spoj:FIBOSUM]

1.6.6 Encoding of Sets by Integers

A technique to efficiently handle sets of integers between 0 and k for an integer k on the order of 63 consists of encoding them as integers.[7] More precisely, we use the binary decomposition of an integer as the characteristic vector of the subset to be represented. The encodings and operations are summarised in the following table:

value	encoding	explanation
$\{\}$	0	empty set
$\{i\}$	$1 << i$	this notation represents 2^i
$\{0,1,\ldots,n-1\}$	$(1 << n) - 1$	$2^n - 1 = 2^0 + 2^1 + \cdots + 2^{n-1}$
$A \cup B$	$A \mid B$	the operator $\mid$ is the binary *or*
$A \cap B$	$A \;\&\; B$	the operator & is the binary *and*
$(A \backslash B) \cup (B \backslash A)$	$A \,\hat{}\, B$	the operator ˆ is the *exclusive or*
$A \subseteq B$	$A \;\&\; B == A$	test of inclusion
$i \in A$	$(1 << i) \& \; A$	test of membership
$\{\min A\}$	$-A \;\&\; A$	this expression equals 0 if A is empty

The justification of this last expression is shown in Figure 1.7. It is useful, for example, to count in a loop the cardinality of a set. There is no equivalent expression to obtain the maximum of a set.

To illustrate this encoding technique, we apply it to a classic problem.

[7] The limit comes from the fact that integers in Python are, in general, coded in a machine word, which today is usually 63 bits plus a sign bit, for a total of 64 bits.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

We wish to obtain the minimum of the set $\{3,5,6\}$ encoded by an integer. This integer is $2^3 + 2^5 + 2^6 = 104$.

64+32+8=104	01101000
complement of 104	10010111
-104	10011000
-104&104 = 8	00001000

The result is 2^3 encoding the singleton $\{3\}$.

Figure 1.7 Example of the extraction of the minimum of a set.

Example – Fair Partition into Three Portions

Definition

Given n integers $x_0, \ldots, x_{n-1}$, we wish to partition them into three sets with the same sum.

Naive Algorithm in Time $O(2^{2n})$

This algorithm uses exhaustive search. This consists of enumerating all the disjoint subsets $A, B \subseteq \{0,\ldots,n-1\}$ and comparing the three values $f(A), f(B), f(C)$ with $C = \{0,\ldots,n-1\} \setminus A \setminus B$ and $f(S) = \sum_{i \in S} x_i$. The implementation skips explicit computation of the set C by verifying that $f(A) = f(B)$ and $3f(A) = f(\{0,\ldots,n-1\})$.

```
def three_partition(x):
    f = [0] * (1 << len(x))
    for i, _ in enumerate(x):
        for S in range(1 << i):
            f[S | (1 << i)] = f[S] + x[i]
    for A in range(1 << len(x)):
        for B in range(1 << len(x)):
            if A & B == 0 and f[A] == f[B] and 3 * f[A] == f[-1]:
                return (A, B, ((1 << len(x)) - 1) ^ A ^ B)
    return None
```

For another utilisation of this technique, see the resolution for numbers in the TV contest ‘Des Chiffres et des Lettres’ (1965), presented as ‘Countdown’ (1982) in the UK, Section 15.6 on page 243.

1.6.7 Binary (Dichotomic) Search

Definition

Let f be a Boolean function—in $\{0,1\}$—with

$$f(0) \le \ldots \le f(n-1) = 1.$$

We wish to find the smallest integer k such that $f(k) = 1$.

Algorithm in Time $O(\log n)$

The solution is sought in an interval $[\ell, h]$, initially $\ell = 0, h = n-1$. Then f is tested in the middle of the interval at $m = \lfloor (\ell + h)/2 \rfloor$. As a function of the result, the search

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

space is reduced either to $[\ell, m]$ or to $[m+1, h]$. Note that because of the rounding below in the computation of m, the second interval is never empty, nor the first for that matter. The search terminates after $\lceil \log_2(n) \rceil$ iterations, when the search interval is reduced to a singleton.

```
def discrete_binary_search(tab, lo, hi):
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if tab[mid]:
            hi = mid
        else:
            lo = mid + 1
    return lo
```

Libraries

Binary search is provided in the standard module `bisect`, so that in many cases you will not have to write it yourself. Consider the case of a sorted array `tab` of n elements, where we want to find the insertion point of a new element x. The function `bisect_left(tab, x, 0, n)` returns the first index i such that `tab[i]` $\geq x$.

Continuous Domain

This technique can equally be applied when the domain of f is continuous and we seek the smallest value x_0 with $f(x) = 1$ for every $x \geq x_0$. The complexity will then depend on the precision required for x_0.

```
def continuous_binary_search(f, lo, hi, gap=1e-4):
    while hi - lo > gap:
        mid = (lo + hi) / 2.0
        if f(mid):
            hi = mid
        else:
            lo = mid
    return lo
```

Search without a Known Upper Bound

Let f be a monotonic Boolean function with $f(0) = 0$ and the guarantee that an integer n exists such that $f(n) = 1$. The smallest integer n_0 with $f(n_0) = 1$ can be found in time $O(\log n_0)$, even in the absence of an upper bound for n_0. Initially, we set $n = 1$, and then repeatedly double it while $f(n) = 0$. Once a value n is found with $f(n) = 1$, we proceed with the standard binary search.

Ternary (Trichotomic) Search

Let f be a function on $\{0, \ldots, n-1\}$, increasing and then decreasing, for which we seek to find the maximal value. In this case, it makes sense to divide the search interval $[l, h]$ not in two but in three portions $[l, a], [a+1, b], [b+1, h]$. By comparing the values of $f(a)$ with $f(b)$, we can decide which of the intervals $[l, b], [a+1, h]$ contains a

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

maximal value. The number of iterations necessary is again logarithmic, on the order of $\log_{3/2} n$.

Search in an Interval $[0,2^k)$

In the case of a search space where the size n is a power of 2, it is possible to slightly improve the usual binary search, by using only bit manipulation operators, such as binary shifts or *exclusive or*. We begin with the index of the last element of the array, which in binary is written as a sequence of ones of length k. For each of the bits, we test whether its replacement by 0 results in an index i such that `tab[i]` remains true. In this case, we update the upper endpoint of the search interval.

```
def optimized_binary_search(tab, logsize):
    hi = (1 << logsize) - 1
    intervalsize = (1 << logsize) >> 1
    while intervalsize > 0:
        if tab[hi ^ intervalsize]:
            hi ^= intervalsize
        intervalsize >>= 1
    return hi
```

Invert a Function

Let f be a continuous and strictly monotonic function. Thus there exists an inverse function f^{-1}, again monotonic. Imagine that f^{-1} is much easier to compute than f, in this case we can use it to compute $f(x)$ for a given value of x. Indeed, it suffices to find the smallest value y such that $f^{-1}(y) \geq x$.

Example—Filling Tasks

Suppose there are n tanks in the form of rectangular blocks at different heights, interconnected by a network of pipes. We pour into this system a volume V of liquid, and wish to know the resulting height of the liquid in the tanks.

```
level = continuous_binary_search(lambda level: volume(level) >= V, 0, hi)
```

Problems

Fill the cisterns [spoj:CISTFILL]
Egg Drop [gcj:eggdrop]

1.7 Advice

Here is a bit of common sense advice to help you rapidly solve algorithmic problems and produce a working program. Be organised and systematic. For this, it is important to not be carried away with the desire to start hacking before all the details are clear. It is easy to leap into the implementation of a method that can never work for a reason

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

that would not have escaped you if you had taken a bit more time to ponder before starting to pound on the keyboard.

When possible, in your program, separate the input of the instance from the computation of the solution. Be nice to yourself and systematically add to the comments the name of the problem, if possible with its URL, and specify the complexity of your algorithm. You will appreciate this effort when you revisit your code at a later date. Above all, stay coherent in your code and reuse the terms given in the problem statement to highlight the correspondence. There is little worse than having to debug a program whose variable names are not meaningful.

Carefully Read the Problem Statement

What complexity is acceptable? Pay attention to the limits stated for the instances. Analyse your algorithm before implementing it.

Understand the example given. Carefully read the typical example furnished with the problem statement. If the input describes a graph, draw its picture. Once you have an idea of a solution, of course verify it on the example or examples provided.

What guarantees on the inputs? Do not deduce guarantees from the examples. Do not suppose anything. If it is not stated that the graph is not empty, there will probably be an instance with an empty graph. If it is not said that a character string does not contain a space, there will probably be an instance with such a string. Etc, etc.

What type of number should you choose? Integer or floating point? Can the numbers be negative? If you are coding in C++ or Java, determine a bound on the largest intermediate value in your calculations, to decide between the use of 16-, 32- or 64-bit integers.

Which problem is easy? For a competition involving several problems, start with a rapid overview to determine the type of each problem (greedy algorithm, implicit graph, dynamic programming, etc.) and estimate its level of difficulty. Concentrate then on the easiest problems in priority. During a team competition, distribute the problems according to the domains of expertise of each participant. Monitor the progress of the other teams to identify the problems that are easy to solve.

Plan Your Work

Understand the example. Draw pictures. Discover the links with known problems. How can you exploit the particularities of the instances?

When possible, use existing modules. Master the classics: binary search, sorting, dictionaries.

Use the same variable names as in the problem statement. Preferably, use names that are short but meaningful. Avoid variables such as `O` or `l`, which are too easily confused with numbers.

Initialise your variables. Make sure that the variables are reinitialised before the processing of each new instance. Treading on the remnants of the preceding

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

iteration is a classic error. Take, for example, a program solving a graph problem whose input consists of the number of instances, followed by each instance, beginning with two integers: the number of vertices n and the number of edges m. These are followed by two arrays of integers A, B of size m, encoding the endpoints of the edges for each instance. Imagine that the program encodes the graph in the form of an adjacency list G and for every $i = 0, \dots, m-1$, adds $B[i]$ to $G[A[i]]$ and $A[i]$ to $G[B[i]]$. If the lists are not emptied before the beginning of each input of an instance, the edges accumulate to form a superposition of all the graphs.

Debug

Make all your mistakes *now* to have the proper reflexes later.

Generate test sets for the limit cases (*Wrong Answer*) and for large instances (*Time Limit Exceeded* or *Runtime Error*).

Explain the algorithm to a team-mate and add appropriate comments to the program. You must be capable of explaining each and every line.

Simplify your implementation. Regroup all similar bits of code. Never go wild with copy-and-paste (note from the translators—one of the most observed sources of buggy programs from novice programmers!).

Take a step back by passing to another problem, and come back later for a fresh look.

Compare the environment of your local machine with that of the server where your code will be tested.

1.8 A Problem: 'Frosting on the Cake'

See [icpcarchive:8154].

Iskander the Baker is decorating a huge cake by covering the rectangular surface of the cake with frosting. For this purpose, he mixes frosting sugar with lemon juice and beetle juice, in order to produce three kinds of frosting: yellow, pink and white. These colours are identified by the numbers 0 for yellow, 1 for pink and 2 for white.

To obtain a nice pattern, he partitions the cake surface into vertical stripes of width $A_1, A_2, \dots, A_n$ centimeters, and horizontal stripes of height $B_1, B_2, \dots, B_n$ centimeters, for some positive integer n. These stripes split the cake surface into $n \times n$ rectangles. The intersection of vertical stripe i and horizontal stripe j has colour number $(i + j) \bmod 3$ for all $1 \le i, j \le n$, see Figure 1.8. To prepare the frosting, Iskander wants to know the total surface in square centimeters to be coloured for each of the three colours, and asks for your help.

Input

The input consists of the following integers:

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

- on the first line: the integer n,
- on the second line: the values of $A_1, \ldots, A_n$, n integers separated by single spaces,
- on the third line: the values of $B_1, \ldots, B_n$, n integers separated by single spaces.

Limits

The input satisfies $3 \leq n \leq 100\,000$ and $1 \leq A_1, \ldots, A_n, B_1, \ldots, B_n \leq 10\,000$.

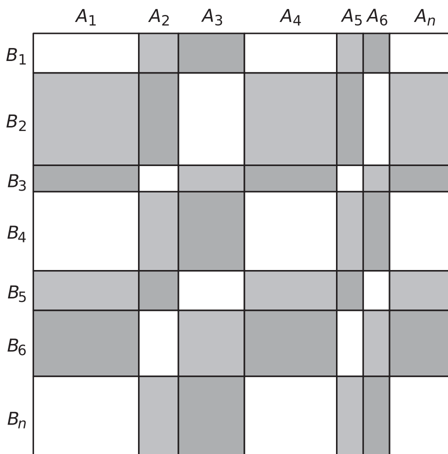


Figure 1.8 An instance of the problem 'Frosting on the Cake'

Output

The output should consist of three integers separated by single spaces, representing the total area for each colour 0, 1 and 2.

Solution

When you see the upper bound on n, you want to find a solution that runs in time $O(n)$ or possibly $O(n \log n)$. This rules out the naive solution which loops over all n^2 grid cells and accumulates their areas in variables corresponding to each colour. But with a little thought about the problem, we can make the following observation. Permuting columns or rows preserves the total area of each colour. Hence, we can reduce to the $n = 3$ case, by simply summing the values of each colour class A_{3k}, A_{3k+1} and A_{3k+2}. Then the answer per colour class is just the sum of the areas of three rectangles.

Hence, a solution could be as short as the following one. The tricky part relies in not mixing up the colours.

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press

```
def read_ints(): return [int(x) for x in input().split()]


def cat(l): return tuple(sum(l[n::3]) for n in [1, 2, 0])


input()  # n
A = cat(read_ints())
B = cat(read_ints())
print("{} {} {}".format(B[2] * A[0] + B[0] * A[2] + B[1] * A[1],
                        B[2] * A[1] + B[0] * A[0] + B[1] * A[2],
                        B[2] * A[2] + B[0] * A[1] + B[1] * A[0]))
```

https://doi.org/10.1017/9781108591928.002 Published online by Cambridge University Press