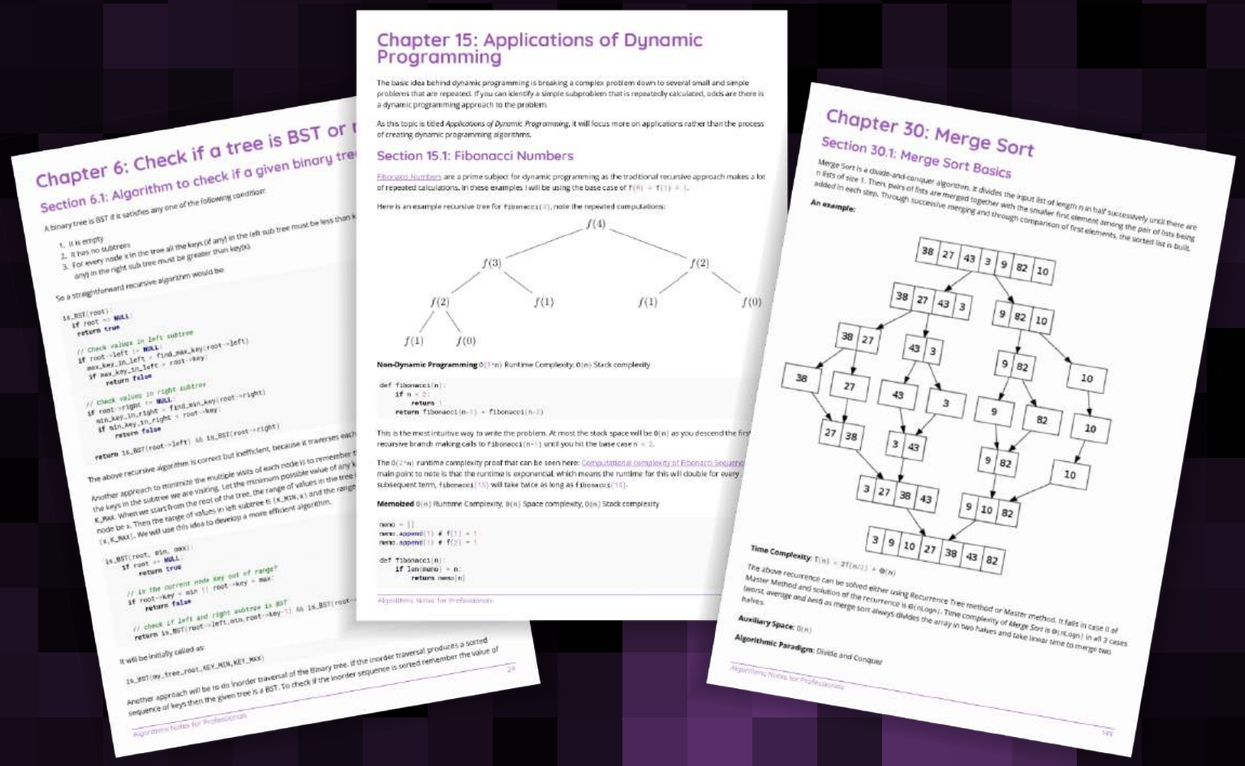
算法

专业人员笔记

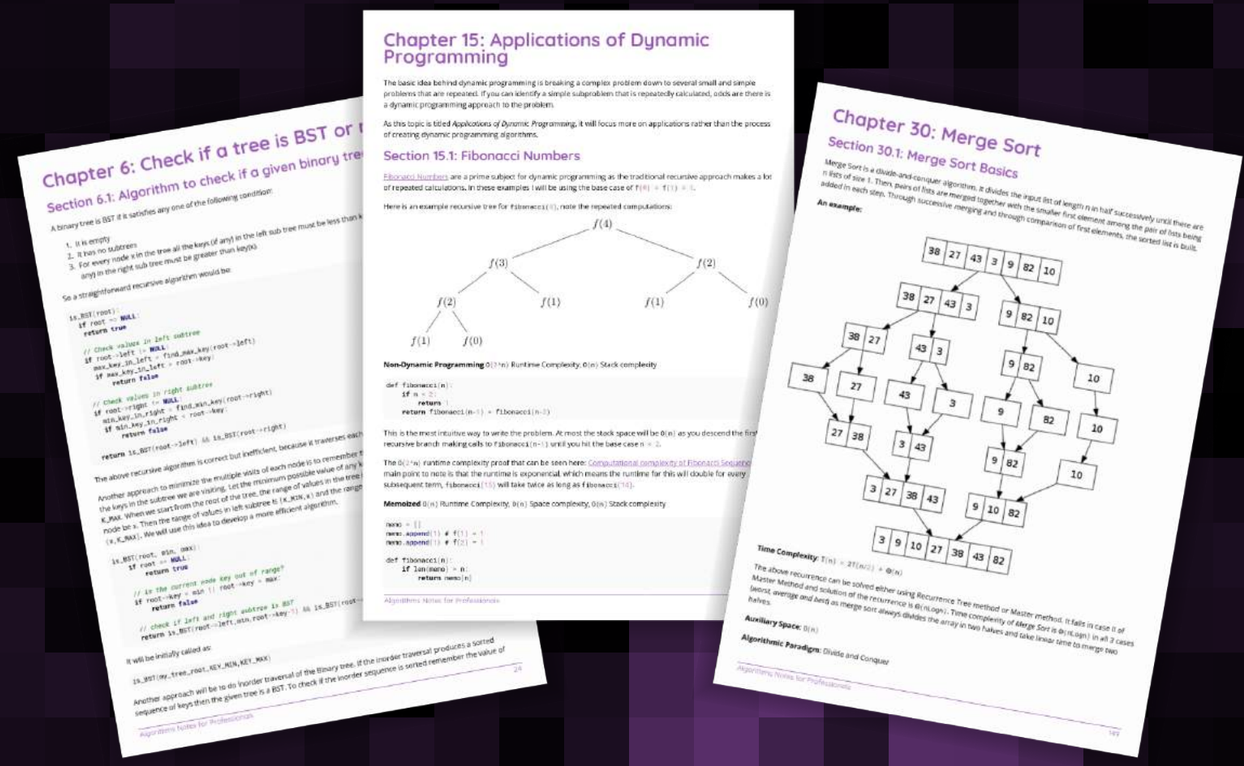
Algorithms

Notes for Professionals



200多页

专业提示和技巧



200+ pages

of professional hints and tricks

目录

Contents

关于

欢迎随意免费分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/AlgorithmsBook

本 *专业人士算法笔记* 一书汇编自Stack Overflow Documentation，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费教材，旨在教育用途，与官方算法组织或公司及Stack Overflow无关。所有商标及注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/AlgorithmsBook

This *Algorithms Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Algorithms group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：算法入门

第1.1节：一个示例算法问题

算法问题通过描述它必须处理的完整实例集合以及在这些实例之一上运行后的输出来指定。问题与问题实例之间的区别是根本性的。被称为排序的算法问题定义如下：[Skiena:2008:ADM:1410219]

- 问题：排序
- 输入：一个包含n个键的序列，a_1，a_2，...，a_n。
- 输出：输入序列的重新排序，使得 a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n

排序的一个*实例*可能是一个字符串数组，例如{Haskell，Emacs}，或者是一个数字序列，例如{ 154，245，1337 }。

第1.2节：Swift中简单Fizz Buzz算法入门

对于刚接触Swift编程的读者以及来自Python或Java等不同编程背景的读者，本文应该非常有帮助。在这篇文章中，我们将讨论一个实现Swift算法的简单解决方案。

Fizz Buzz

你可能见过Fizz Buzz写作Fizz Buzz、FizzBuzz或Fizz-Buzz；它们指的都是同一件事。这个“东西”就是今天讨论的主要话题。首先，什么是FizzBuzz？

这是面试中常见的一个问题。

想象一个从1到10的数字序列。

```
1 2 3 4 5 6 7 8 9 10
```

Fizz和Buzz分别指的是3和5的倍数。换句话说，如果一个数字能被3整除，就用fizz代替；如果一个数字能被5整除，就用buzz代替。如果一个数字同时是3和5的倍数，则用“fizz buzz”代替。实质上，它模拟了著名的儿童游戏“fizz buzz”。

要解决这个问题，打开Xcode创建一个新的playground，并初始化如下数组：

```
// 例如
let number  = [1,2,3,4,5]
// 这里3是fizz，5是buzz
```

要找出所有的fizz和buzz，我们必须遍历数组，检查哪些数字是fizz，哪些是buzz。为此，创建一个for循环来遍历我们初始化的数组：

```
for num in number {
  // 这里是主体和计算部分
}
```

之后，我们可以简单地使用 Swift 中的“if else”条件和取模运算符 % 来定位 fizz 和 buzz

Chapter 1: Getting started with algorithms

Section 1.1: A sample algorithmic problem

An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. The algorithmic *problem* known as *sorting* is defined as follows: [Skiena:2008:ADM:1410219]

- Problem: Sorting
- Input: A sequence of *n* keys, a_1, a_2, ..., a_n.
- Output: The reordering of the input sequence such that a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n

An *instance* of sorting might be an array of strings, such as { Haskell, Emacs } or a sequence of numbers such as { 154, 245, 1337 }.

Section 1.2: Getting Started with Simple Fizz Buzz Algorithm in Swift

For those of you that are new to programming in Swift and those of you coming from different programming bases, such as Python or Java, this article should be quite helpful. In this post, we will discuss a simple solution for implementing swift algorithms.

Fizz Buzz

You may have seen Fizz Buzz written as Fizz Buzz, FizzBuzz, or Fizz-Buzz; they're all referring to the same thing. That "thing" is the main topic of discussion today. First, what is FizzBuzz?

This is a common question that comes up in job interviews.

Imagine a series of a number from 1 to 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz and Buzz refer to any number that's a multiple of 3 and 5 respectively. In other words, if a number is divisible by 3, it is substituted with fizz; if a number is divisible by 5, it is substituted with buzz. If a number is simultaneously a multiple of 3 AND 5, the number is replaced with "fizz buzz." In essence, it emulates the famous children game "fizz buzz".

To work on this problem, open up Xcode to create a new playground and initialize an array like below:

```
// for example
let number  = [1,2,3,4,5]
// here 3 is fizz and 5 is buzz
```

To find all the fizz and buzz, we must iterate through the array and check which numbers are fizz and which are buzz. To do this, create a for loop to iterate through the array we have initialised:

```
for num in number {
  // Body and calculation goes here
}
```

After this, we can simply use the "if else" condition and module operator in swift ie - % to locate the fizz and buzz

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
print(num)
  }
}
```

太好了！你可以去 Xcode playground 的调试控制台查看输出。你会发现“fizz”已经在你的数组中被筛选出来了。

对于 Buzz 部分，我们将使用相同的技巧。让我们在浏览文章之前先试一试—完成后你可以对照本文检查你的结果。

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
print(num)
  }
}
```

检查输出！

这相当简单—你将数字除以3，输出fizz，将数字除以5，输出buzz。现在，增加数组中的数字

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

我们将数字范围从1-10增加到1-15，以演示“fizz buzz”的概念。由于15
是3和5的公倍数，该数字应被替换为“fizz buzz”。自己试试并检查答案吧！

这是解决方案：

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
print(num)
  }
}
```

等等.....还没结束！该算法的全部目的是正确地定制运行时。想象一下，如果范围从1-15增加到1-100。编译器将检查每个数字以确定它是否能被3或5整除。然后它会再次遍历数字，检查数字是否能被3和5整除。代码本质上必须遍历数组中的每个数字两次—先按3运行数字，然后再按5运行。为了加快速度，我们可以直接告诉代码用15来除数字。

这是最终代码：

```
for num in number {
```



```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
    print(num)
  }
}
```

Great! You can go to the debug console in Xcode playground to see the output. You will find that the "fizzes" have been sorted out in your array.

For the Buzz part, we will use the same technique. Let's give it a try before scrolling through the article — you can check your results against this article once you've finished doing this.

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Check the output!

It's rather straight forward — you divided the number by 3, fizz and divided the number by 5, buzz. Now, increase the numbers in the array

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

We increased the range of numbers from 1-10 to 1-15 in order to demonstrate the concept of a "fizz buzz." Since 15 is a multiple of both 3 and 5, the number should be replaced with "fizz buzz." Try for yourself and check the answer!

Here is the solution:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Wait...it's not over though! The whole purpose of the algorithm is to customize the runtime correctly. Imagine if the range increases from 1-15 to 1-100. The compiler will check each number to determine whether it is divisible by 3 or 5. It would then run through the numbers again to check if the numbers are divisible by 3 and 5. The code would essentially have to run through each number in the array twice — it would have to runs the numbers by 3 first and then run it by 5. To speed up the process, we can simply tell our code to divide the numbers by 15 directly.

Here is the final code:

```
for num in number {
```

```
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
print(num)
  }
}
```

就这么简单，你可以使用任何你选择的语言开始编程

享受编码的乐趣


```
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

As Simple as that, you can use any language of your choice and get started

Enjoy Coding

第二章：算法复杂度

第2.1节：大Θ符号

与仅表示某算法运行时间上界的大O符号不同，大Θ符号是一个紧界；既是上界也是下界。紧界更精确，但计算起来也更困难。

大Θ符号是对称的：f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))直观理解，f(x) = Θ(g(x))

意味着函数f(x)和g(x)的图像增长速率相同，或者说对于足够大的x值，图像"表现"相似。

大Θ符号的完整数学表达式如下：
Θ(f(x)) = {g: N0 -> R 且 c1, c2, n0 > 0，满足对于所有 n > n0 有 c1 < abs(g(n) / f(n))，其中 abs 表示绝对值 }

举个例子

如果输入为 n 的算法需要 42n^2 + 25n + 4 次操作完成，我们说它是 O(n^2)，但它也是 O(n^3) 和 O(n^100)。然而，它是 Θ(n^2)，但不是 Θ(n^3)、Θ(n^4) 等。一个是 Θ(f(n)) 的算法也是 O(f(n))，但反之不成立！

正式的数学定义

Θ(g(x)) 是一组函数。

Θ(g(x)) = {f(x) 使得存在正常数 c1、c2、N 满足 0 <= c1*g(x) <= f(x) <= c2*g(x) 对所有 x > N}

因为 Θ(g(x)) 是一个集合，我们可以写作 f(x) ∈ Θ(g(x)) 来表示 f(x) 是 Θ(g(x)) 的成员。相反，我们通常写作 f(x) = Θ(g(x)) 来表达相同的意思——这是常用的写法。

每当 Θ(g(x)) 出现在公式中时，我们将其解释为代表某个匿名函数，我们不关心其具体名称。例如方程 T(n) = T(n/2) + Θ(n)，意味着 T(n) = T(n/2) + f(n)，其中 f(n) 是集合 Θ(n) 中的一个函数。

设 f 和 g 是定义在实数某子集上的两个函数。当且仅当存在正常数 K 和 L 以及实数 x0，使得以下不等式成立时，我们写作 f(x) = Θ(g(x)) 当 x ->无穷大：

K|g(x)| <= f(x) <= L|g(x)| 对所有 x >= x0。

该定义等价于：

f(x) = O(g(x)) 且 f(x) = Ω(g(x))

一种使用极限的方法

如果 limit(x->infinity) f(x)/g(x) = c ∈ (0,∞)，即极限存在且为正，则 f(x) = Θ(g(x))

常见复杂度类

名称	符号说明	n = 10	n = 100
常数	Θ(1)	1	1
对数	Θ(log(n))	3	7
线性	Θ(n)	10	100

Chapter 2: Algorithm Complexity

Section 2.1: Big-Theta notation

Unlike Big-O notation, which represents only upper bound of the running time for some algorithm, Big-Theta is a tight bound; both upper and lower bound. Tight bound is more precise, but also more difficult to compute.

The Big-Theta notation is symmetric: f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))

An intuitive way to grasp it is that f(x) = Θ(g(x)) means that the graphs of f(x) and g(x) grow in the same rate, or that the graphs 'behave' similarly for big enough values of x.

The full mathematical expression of the Big-Theta notation is as follows:
Θ(f(x)) = {g: N0 -> R and c1, c2, n0 > 0, where c1 < abs(g(n) / f(n)), for every n > n0 and abs is the absolute value }

An example

If the algorithm for the input n takes 42n^2 + 25n + 4 operations to finish, we say that is 0(n^2), but is also 0(n^3) and 0(n^100). However, it is Θ(n^2) and it is not Θ(n^3), Θ(n^4) etc. Algorithm that is Θ(f(n)) is also 0(f(n)), but not vice versa!

Formal mathematical definition

Θ(g(x)) is a set of functions.

Θ(g(x)) = {f(x) such that there exist positive constants c1, c2, N such that 0 <= c1*g(x) <= f(x) <= c2*g(x) **for** all x > N}

Because Θ(g(x)) is a set, we could write f(x) ∈ Θ(g(x)) to indicate that f(x) is a member of Θ(g(x)). Instead, we will usually write f(x) = Θ(g(x)) to express the same notion - that's the common way.

Whenever Θ(g(x)) appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example the equation T(n) = T(n/2) + Θ(n), means T(n) = T(n/2) + f(n) where f(n) is a function in the set Θ(n).

Let f and g be two functions defined on some subset of the real numbers. We write f(x) = Θ(g(x)) as x->infinity if and only if there are positive constants K and L and a real number x0 such that holds:

K|g(x)| <= f(x) <= L|g(x)| for all x >= x0.

The definition is equal to:

f(x) = O(g(x)) and f(x) = Ω(g(x))

A method that uses limits

if limit(x->infinity) f(x)/g(x) = c ∈ (0,∞) i.e. the limit exists and it's positive, then f(x) = Θ(g(x))

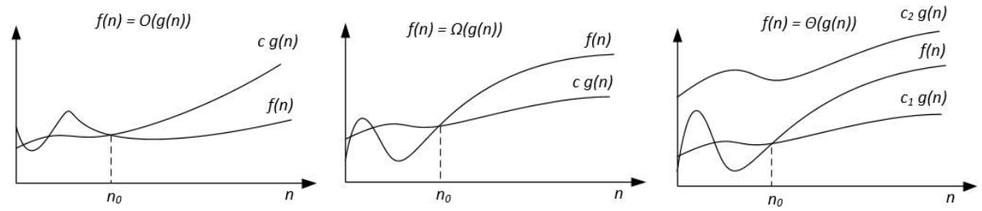
Common Complexity Classes

Name	Notation	n = 10	n = 100
Constant	Θ(1)	1	1
Logarithmic	Θ(log(n))	3	7
Linear	Θ(n)	10	100

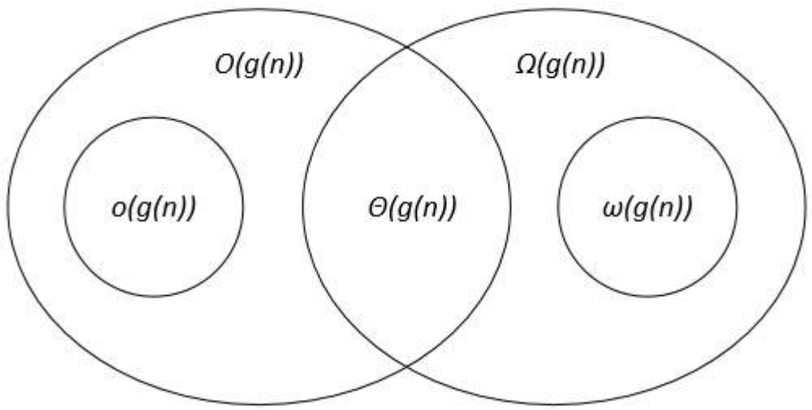
线性对数Θ(n*log(n))		30	700
二次方	Θ(n^2)	100	10 000
指数Θ(2^n)		1 024	1.267650e+ 30
阶乘	Θ(n!)	3 628 800	9.332622e+157

第2.2节：渐近符号的比较

设f(n)和g(n)为定义在正实数集合上的两个函数，c，c1，c2，n0为正实数常数。

符号说明	f(n) = O(g(n))	f(n) = Ω(g(n))	f(n) = Θ(g(n))	f(n) = o(g(n))	f(n) = ω(g(n))
形式定义	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq f(n) \leq c\ g(n)$	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c\ g(n) \leq f(n)$	$\exists\ c1, c2 > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq f(n) < c\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c\ g(n) < f(n)$
f、g的渐近比较与实数a、b之间的类比	$a \leq b$	$a \geq b$	$a = b$	$a < b$	$a > b$
示例	$7n + 10 = O(n^2 + n - 9)$	$n^3 - 34 = \Omega(10n^2 - 7n + 1)$	$1/2\ n^2 - 7n = \Theta(n^2)$	$5n^2 = o(n^3)$	$7n^2 = \omega(n)$
图形解释					

渐近符号可以用韦恩图表示如下：



链接

托马斯·H·科尔门，查尔斯·E·莱瑟森，罗纳德·L·里维斯特，克利福德·斯坦。《算法导论》。

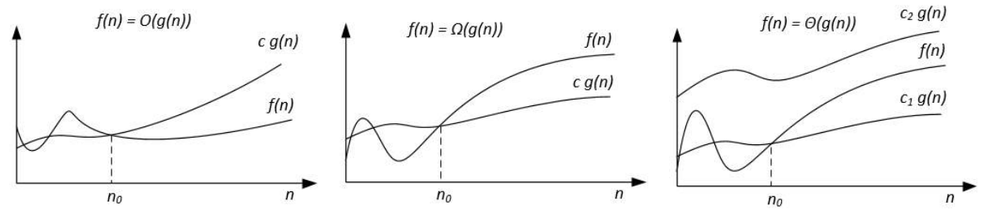
第2.3节：大欧米伽符号

Ω符号用于渐近下界。

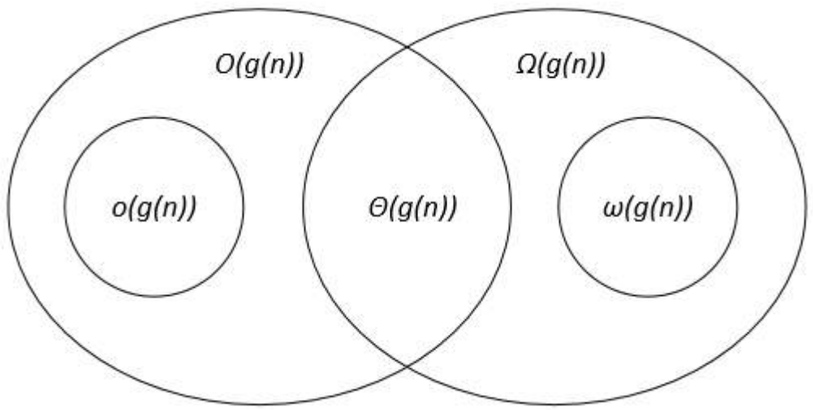
Linearithmic	Θ(n*log(n))	30	700
Quadratic	Θ(n^2)	100	10 000
Exponential	Θ(2^n)	1 024	1.267650e+ 30
Factorial	Θ(n!)	3 628 800	9.332622e+157

Section 2.2: Comparison of the asymptotic notations

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers, `c`, `c1`, `c2`, `n0` are positive real constants.

Notation	f(n) = O(g(n))	f(n) = Ω(g(n))	f(n) = Θ(g(n))	f(n) = o(g(n))	f(n) = ω(g(n))
Formal definition	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq f(n) \leq c\ g(n)$	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c\ g(n) \leq f(n)$	$\exists\ c1, c2 > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq f(n) < c\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c\ g(n) < f(n)$
Analogy between the asymptotic comparison of f, g and real numbers a, b	$a \leq b$	$a \geq b$	$a = b$	$a < b$	$a > b$
Example	$7n + 10 = O(n^2 + n - 9)$	$n^3 - 34 = \Omega(10n^2 - 7n + 1)$	$1/2\ n^2 - 7n = \Theta(n^2)$	$5n^2 = o(n^3)$	$7n^2 = \omega(n)$
Graphic interpretation					

The asymptotic notations can be represented on a Venn diagram as follows:



Links

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

Section 2.3: Big-Omega Notation

Ω-notation is used for asymptotic lower bound.

形式定义

设f(n)和g(n)是定义在正实数集合上的两个函数。如果存在正的常数c和 n_0，使得：

`0 ≤ c g(n) ≤ f(n)` 对于所有 `n ≥ n_0`成立。

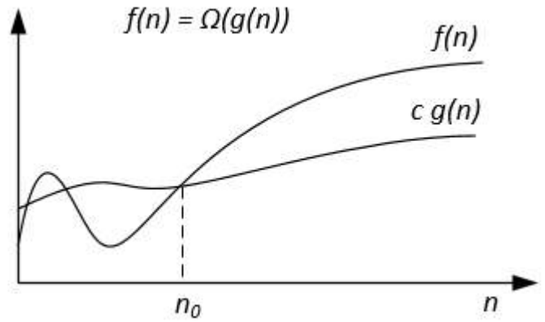
注释

`f(n) = Ω(g(n))`表示f(n)的渐进增长速度不低于g(n)。我们也可以在算法分析不足以说明Θ(g(n))或/和O(g(n))时，使用Ω(g(n))来描述。

由符号定义可得定理：

对于任意两个函数f(n)和g(n)，当且仅当`f(n) = O(g(n))`且`f(n) = Ω(g(n))`时，有`f(n) = Θ(g(n))`成立。

图示上，Ω符号可以表示为：



例如，设`f(n) = 3n^2 + 5n - 4`，则`f(n) = Ω(n^2)`。同样`f(n) = Ω(n)`，甚至`f(n) = Ω(1)`也是正确的。

另一个完美匹配算法的例子：如果顶点数为奇数，则输出“无完美匹配”，否则尝试所有可能的匹配。

我们本想说该算法需要指数时间，但实际上你无法使用通常的 Ω 定义证明一个 Ω(n^2) 下界，因为该算法在 n 为奇数时运行线性时间。我们应该改为定义
f(n)=Ω(g(n))，表示存在某个常数 c>0，使得对于无限多个 n，有 f(n)≥ c g(n)。这在上界和下界之间建立了良好的对应关系：f(n)=Ω(g(n)) 当且仅当 f(n) != o(g(n))。

参考文献

正式定义和定理取自书籍《托马斯·H·科尔曼、查尔斯·E·莱瑟森、罗纳德·L·里维斯特、克利福德·斯坦。《算法导论》》。

Formal definition

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers. We write `f(n) = Ω(g(n))` if there are positive constants c and `n0` such that:

`0 ≤ c g(n) ≤ f(n) for all n ≥ n0`.

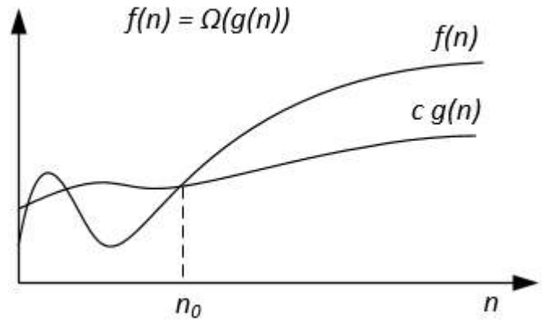
Notes

`f(n) = Ω(g(n))` means that `f(n)` grows asymptotically no slower than `g(n)`. Also we can say about `Ω(g(n))` when algorithm analysis is not enough for statement about `Θ(g(n))` or / and `O(g(n))`.

From the definitions of notations follows the theorem:

For two any functions `f(n)` and `g(n)` we have `f(n) = Θ(g(n))` if and only if `f(n) = O(g(n))` and `f(n) = Ω(g(n))`.

Graphically Ω-notation may be represented as follows:



For example lets we have `f(n) = 3n^2 + 5n - 4`. Then `f(n) = Ω(n^2)`. It is also correct `f(n) = Ω(n)`, or even `f(n) = Ω(1)`.

Another example to solve perfect matching algorithm : If the number of vertices is odd then output "No Perfect Matching" otherwise try all possible matchings.

We would like to say the algorithm requires exponential time but in fact you cannot prove a `Ω(n^2)` lower bound using the usual definition of Ω since the algorithm runs in linear time for n odd. We should instead define `f(n)=Ω(g(n))` by saying for some constant `c>0`, `f(n)≥ c g(n)` for infinitely many n. This gives a nice correspondence between upper and lower bounds: `f(n)=Ω(g(n))` iff `f(n) != o(g(n))`.

References

Formal definition and theorem are taken from the book "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms".

第3章：大O符号

定义

大O符号本质上是一种数学符号，用于比较函数的收敛速度。
设 `n -> f(n)` 和 `n -> g(n)` 是定义在自然数上的函数。若且唯若当 `n` 趋近于无穷大时，`f(n)/g(n)` 有界，我们称 `f = O(g)`。换句话说，`f = O(g)` 当且仅当存在常数 `A`，使得对所有 `n`，有 `f(n)/g(n) <= A`。

实际上，大O符号在数学中的应用范围更广，但为简化起见，我将其限定为算法复杂度分析中使用的范围：定义在自然数上的函数，且函数值非零，且 `n` 趋向无穷大的情况。

这意味着什么？

我们以`f(n) = 100n^2 + 10n + 1` 和 `g(n) = n^2` 为例。很明显，当 `n` 趋近于无穷大时，这两个函数都趋向于无穷大。但有时仅知道极限是不够的，我们还想知道函数趋近极限的速度。像大O符号这样的概念有助于通过收敛速度来比较和分类函数。

让我们通过应用定义来判断 `f = O(g)` 是否成立。我们有 `f(n)/g(n) = 100 + 10/n + 1/n^2`。由于 `10/n` 在 `n=1` 时为 10 并且是递减的，且 `1/n^2` 在 `n=1` 时为 1 并且也是递减的，我们有 `f(n)/g(n) <= 100 + 10 + 1 = 111`。定义得到了满足，因为我们找到了 `f(n)/g(n)` 的上界（111），因此 `f = O(g)`（我们说 `f` 是 `n^2` 的大 O）。

这意味着 `f` 以大致与 `g` 相同的速度趋向无穷大。现在这听起来可能有些奇怪，因为我们发现 `f` 最多比 `g` 大 111 倍，换句话说，当 `g` 增长 1 时，`f` 最多增长 111。看起来增长 111 倍并不是“近似相同的速度”。实际上，大 O 符号并不是一个非常精确的函数收敛速度分类方法，这就是为什么在数学中，当我们需要精确估计速度时，会使用“等价关系”。但为了将算法分为大致的速度类别，大 O 已经足够。我们不需要区分那些增长速度固定倍数不同的函数，只需要区分那些增长速度无限快于彼此的函数。

例如，如果我们取 `h(n) = n^2*log(n)`，我们看到 `h(n)/g(n) = log(n)`，随着 `n` 趋向无穷大，`log(n)` 也趋向无穷大，因此 `h` 不是 `O(n^2)`，因为 `h` 的增长速度无限快于 `n^2`。

现在我需要做一个补充说明：你可能注意到如果 `f = O(g)` 且 `g = O(h)`，那么 `f = O(h)`。例如在我们的例子中，我们有 `f = O(n^3)`，且 `f = O(n^4)`……在算法复杂度分析中，我们经常说 `f =O(g)` 来表示 `f = O(g)` 且 `g = O(f)`，这可以理解为“`g` 是 `f` 的最小大 O”。在数学中，我们称这样的函数是彼此的大 Theta。

它是如何使用的？

在比较算法性能时，我们关注算法执行的操作次数。这称为时间复杂度。在这个模型中，我们认为每个基本操作（加法、乘法、比较、赋值等）都需要固定时间，并计算这些操作的次数。我们通常可以将这个次数表示为输入大小的函数，记为 `n`。遗憾的是，这个次数通常随着 `n` 增长趋向无穷大（如果不是，我们说算法是 `O(1)`）。我们用大 O 将算法分为大致的速度类别：当我们说“一个 `O(n^2)` 算法”时，意思是它执行的操作次数作为 `n` 的函数是 `O(n^2)`。这意味着我们的算法大致与一个执行操作次数等于输入大小平方的算法一样快，或更快。“或更快”部分是因为我用了大 O 而不是大 Theta，但通常人们说大 O 时是指大 Theta。

Chapter 3: Big-O Notation

Definition

The Big-O notation is at its heart a mathematical notation, used to compare the rate of convergence of functions. Let `n -> f(n)` and `n -> g(n)` be functions defined over the natural numbers. Then we say that `f = O(g)` if and only if `f(n)/g(n)` is bounded when n approaches infinity. In other words, `f = O(g)` if and only if there exists a constant A, such that for all n, `f(n)/g(n) <= A`.

Actually the scope of the Big-O notation is a bit wider in mathematics but for simplicity I have narrowed it to what is used in algorithm complexity analysis : functions defined on the naturals, that have non-zero values, and the case of n growing to infinity.

What does it mean ?

Let's take the case of `f(n) = 100n^2 + 10n + 1` and `g(n) = n^2`. It is quite clear that both of these functions tend to infinity as n tends to infinity. But sometimes knowing the limit is not enough, and we also want to know the *speed* at which the functions approach their limit. Notions like Big-O help compare and classify functions by their speed of convergence.

Let's find out if `f = O(g)` by applying the definition. We have `f(n)/g(n) = 100 + 10/n + 1/n^2`. Since `10/n` is 10 when n is 1 and is decreasing, and since `1/n^2` is 1 when n is 1 and is also decreasing, we have`f(n)/g(n) <= 100 + 10 + 1 = 111`. The definition is satisfied because we have found a bound of `f(n)/g(n)` (111) and so `f = O(g)` (we say that f is a Big-O of `n^2`).

This means that f tends to infinity at approximately the same speed as g. Now this may seem like a strange thing to say, because what we have found is that f is at most 111 times bigger than g, or in other words when g grows by 1, f grows by at most 111. It may seem that growing 111 times faster is not "approximately the same speed". And indeed the Big-O notation is not a very precise way to classify function convergence speed, which is why in mathematics we use the equivalence relationship when we want a precise estimation of speed. But for the purposes of separating algorithms in large speed classes, Big-O is enough. We don't need to separate functions that grow a fixed number of times faster than each other, but only functions that grow *infinitely* faster than each other. For instance if we take `h(n) = n^2*log(n)`, we see that `h(n)/g(n) = log(n)` which tends to infinity with n so h is *not* O(n^2), because h grows *infinitely* faster than n^2.

Now I need to make a side note : you might have noticed that if `f = O(g)` and `g = O(h)`, then `f = O(h)`. For instance in our case, we have `f = O(n^3)`, and `f = O(n^4)`... In algorithm complexity analysis, we frequently say `f = O(g)` to mean that `f = O(g)` *and* `g = O(f)`, which can be understood as "g is the smallest Big-O for f". In mathematics we say that such functions are Big-Thetas of each other.

How is it used ?

When comparing algorithm performance, we are interested in the number of operations that an algorithm performs. This is called *time complexity*. In this model, we consider that each basic operation (addition, multiplication, comparison, assignment, etc.) takes a fixed amount of time, and we count the number of such operations. We can usually express this number as a function of the size of the input, which we call n. And sadly, this number usually grows to infinity with n (if it doesn't, we say that the algorithm is O(1)). We separate our algorithms in big speed classes defined by Big-O : when we speak about a "O(n^2) algorithm", we mean that the number of operations it performs, expressed as a function of n, is a O(n^2). This says that our algorithm is approximately as fast as an algorithm that would do a number of operations equal to the square of the size of its input, *or faster*. The "or faster" part is there because I used Big-O instead of Big-Theta, but usually people will say Big-O to mean Big-Theta.

在计算操作次数时，我们通常考虑最坏情况：例如，如果有一个最多运行 n 次的循环，且每次循环包含 5 个操作，那么我们计算的操作次数是 5n。也可以考虑平均情况复杂度。

快速提示：一个快速算法是执行操作次数少的算法，因此如果操作次数增长得更快，那么算法就更慢：O(n) 比 O(n^2) 更好。

我们有时也关注算法的空间复杂度。为此，我们考虑算法占用的内存字节数作为输入大小的函数，并以同样的方式使用大 O。

第3.1节：一个简单的循环

以下函数用于查找数组中的最大元素：

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

输入大小是数组的大小，在代码中我称之为len。

让我们计算操作数。

```
int max = INT_MIN;
size_t i = 0;
```

这两个赋值只执行一次，所以是2次操作。循环中执行的操作有：

```
if (max < array[i])
i++;
max = array[i]
```

由于循环中有3个操作，且循环执行了n次，我们将已有的2个操作加上3n，得到3n + 2。因此我们的函数需要3n + 2个操作来找到最大值（其复杂度为3n + 2）。这是一个多项式，其中增长最快的项是n的因子，所以复杂度是O(n)。

你可能已经注意到"操作"这个概念定义得不够明确。例如，我说if (max < array[i])算作一个操作，但根据不同的架构，这条语句可能会编译成三条指令：一次内存读取，一次比较和一次跳转。我还将所有操作视为相同，尽管例如内存操作会比其他操作慢，并且它们的性能会因缓存效应等原因大幅波动。我也完全忽略了返回语句、函数调用时创建栈帧等因素。最终这对复杂度分析无关紧要，因为无论我如何计数操作，只会改变n的系数和常数项，结果仍然是O(n)。

复杂度显示算法随输入规模的增长情况，但这并不是性能的唯一方面！

第3.2节：嵌套循环

下面的函数通过取每个元素，然后遍历整个数组来检查数组中是否有重复元素

When counting operations, we usually consider the worst case: for instance if we have a loop that can run at most n times and that contains 5 operations, the number of operations we count is 5n. It is also possible to consider the average case complexity.

Quick note : a fast algorithm is one that performs few operations, so if the number of operations grows to infinity *faster*, then the algorithm is *slower*: O(n) is better than O(n^2).

We are also sometimes interested in the *space complexity* of our algorithm. For this we consider the number of bytes in memory occupied by the algorithm as a function of the size of the input, and use Big-O the same way.

Section 3.1: A Simple Loop

The following function finds the maximal element in an array:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

The input size is the size of the array, which I called `len` in the code.

Let's count the operations.

```
int max = INT_MIN;
size_t i = 0;
```

These two assignments are done only once, so that's 2 operations. The operations that are looped are:

```
if (max < array[i])
i++;
max = array[i]
```

Since there are 3 operations in the loop, and the loop is done n times, we add `3n` to our already existing 2 operations to get `3n + 2`. So our function takes `3n + 2` operations to find the max (its complexity is `3n + 2`). This is a polynomial where the fastest growing term is a factor of n, so it is O(n).

You probably have noticed that "operation" is not very well defined. For instance I said that `if (max < array[i])` was one operation, but depending on the architecture this statement can compile to for instance three instructions : one memory read, one comparison and one branch. I have also considered all operations as the same, even though for instance the memory operations will be slower than the others, and their performance will vary wildly due for instance to cache effects. I also have completely ignored the return statement, the fact that a frame will be created for the function, etc. In the end it doesn't matter to complexity analysis, because whatever way I choose to count operations, it will only change the coefficient of the n factor and the constant, so the result will still be O(n). Complexity shows how the algorithm scales with the size of the input, but it isn't the only aspect of performance!

Section 3.2: A Nested Loop

The following function checks if an array has any duplicates by taking each element, then iterating over the whole array to see if the element is there

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

内层循环在每次迭代中执行的操作数量与 n 成正比且为常数。外层循环也执行一些常数操作，并运行内层循环 n 次。外层循环本身运行 n 次。因此，内层循环中的操作运行了 n^2 次，外层循环中的操作运行了 n 次，对 i 的赋值操作执行了一次。因此，复杂度大致为 an^2 + bn + c，由于最高项是 n^2，故大O表示法为 O(n^2)。

正如你可能注意到的，我们可以通过避免多次进行相同的比较来改进算法。
我们可以从内层循环的 i + 1 开始，因为它之前的所有元素都已经与所有数组元素进行了比较，包括索引为 i + 1 的元素。这使得我们可以去掉 i == j 的检查。

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

显然，这个第二个版本执行的操作更少，因此效率更高。这如何转化为大O表示法呢？现在内层循环体运行的次数是 1 + 2 + ... + n - 1 = n(n-1)/2 次。这仍然是一个二次多项式，因此复杂度仍然是 O(n^2)。我们显然降低了复杂度，因为我们大致将操作次数减少了一半，但根据大O定义，我们仍处于同一复杂度 class。要将复杂度降低到更低的级别，我们需要将操作次数除以一个随 n 趋于无穷大的量。

第3.3节：O(log n) 类型的算法

假设我们有一个大小为 n 的问题。现在，对于算法的每一步（我们需要编写），原始问题的规模变为之前的一半（n/2）。

因此，每一步，我们的问题规模减半。

步骤	问题
1	n/2
2	n/4
3	n/8
4	n/16

当问题空间被缩减（即完全解决）时，无法再进一步缩减（n变为1）
退出检查条件后。


```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

The inner loop performs at each iteration a number of operations that is constant with `n`. The outer loop also does a few constant operations, and runs the inner loop `n` times. The outer loop itself is run `n` times. So the operations inside the inner loop are run `n^2` times, the operations in the outer loop are run `n` times, and the assignment to `i` is done one time. Thus, the complexity will be something like `an^2 + bn + c`, and since the highest term is `n^2`, the O notation is `O(n^2)`.

As you may have noticed, we can improve the algorithm by avoiding doing the same comparisons multiple times. We can start from `i + 1` in the inner loop, because all elements before it will already have been checked against all array elements, including the one at index `i + 1`. This allows us to drop the `i == j` check.

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Obviously, this second version does less operations and so is more efficient. How does that translate to Big-O notation? Well, now the inner loop body is run `1 + 2 + ... + n - 1 = n(n-1)/2` times. This is *still* a polynomial of the second degree, and so is still only `O(n^2)`. We have clearly lowered the complexity, since we roughly divided by 2 the number of operations that we are doing, but we are still in the same complexity *class* as defined by Big-O. In order to lower the complexity to a lower class we would need to divide the number of operations by something that *tends to infinity* with `n`.

Section 3.3: O(log n) types of Algorithms

Let's say we have a problem of size n. Now for each step of our algorithm(which we need write), our original problem becomes half of its previous size(n/2).

So at each step, our problem becomes half.

Step	Problem
1	n/2
2	n/4
3	n/8
4	n/16

When the problem space is reduced(i.e solved completely), it cannot be reduced any further(n becomes equal to 1) after exiting check condition.

1. 假设在第k步或操作次数为：

   问题规模= 1

2. 但我们知道在第k步，我们的问题规模应该是：

   问题规模= n/2^k

3. 根据1和2：

   n/2k = 1 或

   n = 2k

4. 对两边取对数

   loge n = k loge2

   或者

   k = loge n / loge 2

5. **使用公式 logx m / logx n = logn m**

   k = log2 n

   k = log2 n

   或者简写为 k = log n

> 现在我们知道算法最多运行到 log n，因此时间复杂度为 O(log n)

下面是一个非常简单的代码示例来支持上述内容：

```
for(int i=1; i<=n; i=i*2)
{
    // 执行某些操作
}
```

所以现在如果有人问你，n 是 256 时，这个循环（或任何将问题规模减半的算法）会运行多少步，你可以很容易地计算出来。

k = log2 256

k = log2 2 8 (=> logaa = 1)

k = 8

另一个类似情况的非常好的例子是二分查找算法。

1. Let's say at kth step or number of operations:

   problem-size = 1

2. But we know at kth step, our problem-size should be:

   problem-size = n/2k

3. From 1 and 2:

   n/2k = 1 or

   n = 2k

4. Take log on both sides

   loge n = k loge2

   or

   k = loge n / loge 2

5. Using formula **logx m / logx n = logn m**

   k = log2 n

   k = log2 n

   or simply k = log n

> Now we know that our algorithm can run maximum up to log n, hence time complexity comes as O(log n)

A very simple example in code to support above text is :

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

So now if some one asks you if n is 256 how many steps that loop(or any other algorithm that cuts down it's problem size into half) will run you can very easily calculate.

k = log2 256

k = log2 2 8 (=> logaa = 1)

k = 8

Another very good example for similar case is **Binary Search Algorithm**.

```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return -1;// 未成功的结果
}
```

第3.4节：一个O(log n)的例子

介绍

考虑以下问题：

`L` 是一个包含 `n` 个有符号整数的排序列表（`n` 足够大），例如 `[-5, -2, -1, 0, 1, 2, 4]`（这里，`n` 的值为7）。如果已知 `L` 中包含整数0，如何找到0的索引？

朴素方法

首先想到的是逐个读取索引直到找到0。最坏情况下，操作次数为 `n`，因此复杂度是O(n)。

对于较小的 `n` 值，这种方法效果不错，但有没有更高效的方法？

二分法

考虑以下算法（Python3）：

```
a = 0
b = n-1
while True:
h = (a+b)//2 ## // 是整数除法，所以 h 是整数
  if L[h] == 0:
    return h
elif L[h] > 0:
b = h
  elif L[h] < 0:
a = h
```

`a` 和 `b` 是包含 0 的索引。每次进入循环时，我们使用 `a` 和 `b` 之间的一个索引，并用它来缩小搜索范围。

在最坏情况下，我们必须等待 `a` 和 `b` 相等。但这需要多少次操作？不是 `n`，因为每次进入循环时，我们将 `a` 和 `b` 之间的距离大约减半。复杂度是 O(log n)。

说明

注意：当我们写“log”时，指的是二进制对数，即以 2 为底的对数（我们写作“log_2”）。由于 O(log_2 n) = O(log n)（你可以自己算算），我们将使用“log”代替“log_2”。


```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return -1;// Unsuccessful result
}
```

Section 3.4: An O(log n) example

Introduction

Consider the following problem:

`L` is a sorted list containing `n` signed integers (`n` being big enough), for example `[-5, -2, -1, 0, 1, 2, 4]` (here, `n` has a value of 7). If `L` is known to contain the integer 0, how can you find the index of 0 ?

Naïve approach

The first thing that comes to mind is to just read every index until 0 is found. In the worst case, the number of operations is `n`, so the complexity is O(n).

This works fine for small values of `n`, but is there a more efficient way ?

Dichotomy

Consider the following algorithm (Python3):

```
a = 0
b = n-1
while True:
  h = (a+b)//2 ## // is the integer division, so h is an integer
  if L[h] == 0:
    return h
  elif L[h] > 0:
    b = h
  elif L[h] < 0:
    a = h
```

`a` and `b` are the indexes between which 0 is to be found. Each time we enter the loop, we use an index between `a` and `b` and use it to narrow the area to be searched.

In the worst case, we have to wait until `a` and `b` are equal. But how many operations does that take? Not `n`, because each time we enter the loop, we divide the distance between `a` and `b` by about two. Rather, the complexity is O(log n).

Explanation

Note: When we write "log", we mean the binary logarithm, or log base 2 (which we will write "log_2"). As O(log_2 n) = O(log n) (you can do the math) we will use "log" instead of "log_2".

我们设操作次数为 x：我们知道 1 = n / (2^x)。

所以 2^x = n，则 x = log n。

结论

当面对连续的除法（无论是除以二还是除以任何数字）时，记住其复杂度是对数级的。

Let's call x the number of operations: we know that 1 = n / (2^x).

So 2^x = n, then x = log n

Conclusion

When faced with successive divisions (be it by two or by any number), remember that the complexity is logarithmic.

第4章：树

第4.1节：典型的多叉树表示

通常我们将多叉树（每个节点可能有无限多个子节点）表示为二叉树（每个节点恰好有两个子节点），“下一个”子节点被视为兄弟节点。注意，如果树本身是二叉树，这种表示会产生额外的节点。

然后我们遍历兄弟节点并递归访问子节点。由于大多数树相对较浅——有很多子节点但层级较少，这使得代码效率较高。人类家谱是个例外（祖先层级很多，每层子节点较少）。

如有必要，可以保留回指针以允许向上遍历树。这些指针更难维护。

请注意，通常会有一个函数在根节点调用，另一个递归函数带有额外参数，在本例中为树的深度。

```
结构体节点
   {
结构体节点 *下一个;
      结构体节点 *子节点;
       std::string 数据;
   }

   void 打印树_递归(结构体节点 *节点, int 深度)
   {
       int i;

       while(节点)
       {
            if(节点->子节点)
            {
                for(i=0;i<深度*3;i++)
           printf(" ");                pri
ntf("{"):
打印树_递归(节点->子节点, 深度 +1);
                for(i=0;i<深度*3;i++)
           printf(" ");                pri
ntf("{"):

                for(i=0;i<深度*3;i++)
printf(" ");                printf("%s
", 节点->数据.c_str());                节点 = 节点->下一个;


            }
       }
   }

   void 打印树(节点 *根节点)
   {
printree_r(root, 0);
   }
```

第4.2节：介绍

树是更一般的节点-边图数据结构的一个子类型。

Chapter 4: Trees

Section 4.1: Typical anary tree representation

Typically we represent an anary tree (one with potentially unlimited children per node) as a binary tree, (one with exactly two children per node). The "next" child is regarded as a sibling. Note that if a tree is binary, this representation creates extra nodes.

We then iterate over the siblings and recurse down the children. As most trees are relatively shallow - lots of children but only a few levels of hierarchy, this gives rise to efficient code. Note human genealogies are an exception (lots of levels of ancestors, only a few children per level).

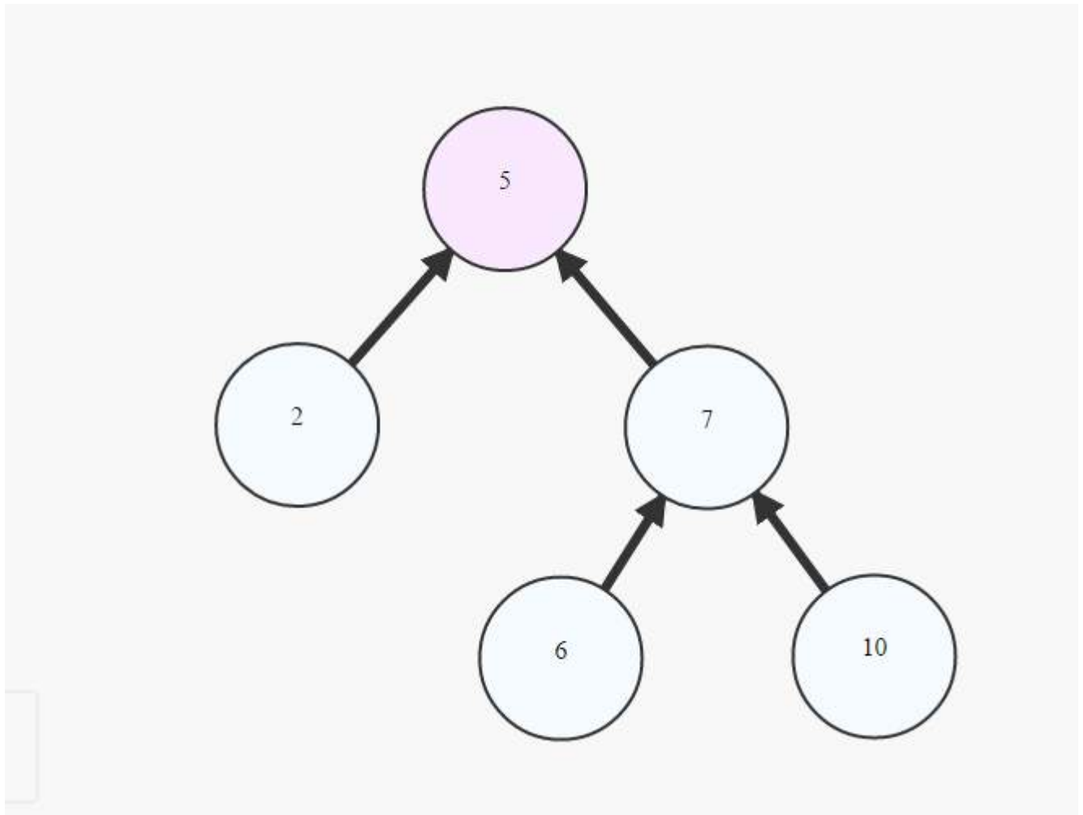
If necessary back pointers can be kept to allow the tree to be ascended. These are more difficult to maintain.

Note that it is typical to have one function to call on the root and a recursive function with extra parameters, in this case tree depth.

```
struct node
{
   struct node *next;
   struct node *child;
   std::string data;
}

void printtree_r(struct node *node, int depth)
{
   int i;

   while(node)
   {
       if(node->child)
       {
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):
          printtree_r(node->child, depth +1);
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):

          for(i=0;i<depth*3;i++)
             printf(" ");
           printf("%s\n", node->data.c_str());

           node = node->next;
       }
   }
}

void printtree(node *root)
{
   printree_r(root, 0);
}
```

Section 4.2: Introduction

Trees are a sub-type of the more general node-edge graph data structure.



要成为一棵树，图必须满足两个要求：

- 它是无环的。 它不包含环（或“回路”）。
- **它是连通的。** 对于图中的任意节点，每个节点都是可达的。所有节点通过图中的一条路径可达。

树数据结构在计算机科学中非常常见。树被用来建模许多不同的
算法数据结构，如普通二叉树、红黑树、B树、AB树、23树、堆和

字典树（tries）。

通常将树称为有根树，方法是：

```
选择1个单元称为“根”
将“根”绘制在顶部
根据每个单元格与根节点的距离，为图中的每个单元格创建较低的层级-距离越大，单元格越低（如上例）
```

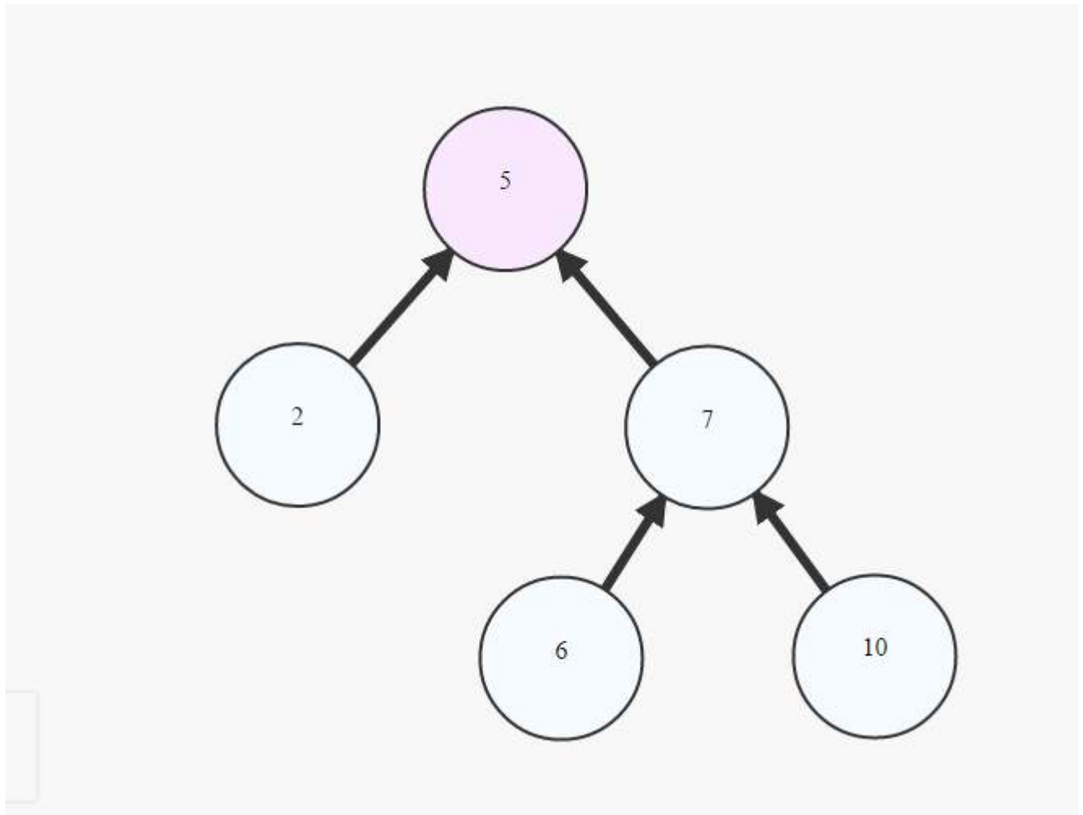
树的常用符号：T

第4.3节：检查两个二叉树是否相同

1.例如，如果输入是：

示例：1

a)



To be a tree, a graph must satisfy two requirements:

- **It is acyclic.** It contains no cycles (or "loops").
- **It is connected.** For any given node in the graph, every node is reachable. All nodes are reachable through one path in the graph.

The tree data structure is quite common within computer science. Trees are used to model many different algorithmic data structures, such as ordinary binary trees, red-black trees, B-trees, AB-trees, 23-trees, Heap, and tries.

it is common to refer to a Tree as a `Rooted Tree` by:

```
choosing 1 cell to be called `Root`
painting the `Root` at the top
creating lower layer for each cell in the graph depending on their distance from the root -the
bigger the distance, the lower the cells (example above)
```

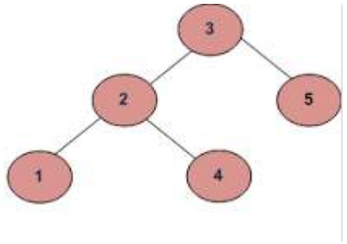
common symbol for trees: `T`

Section 4.3: To check if two Binary trees are same or not

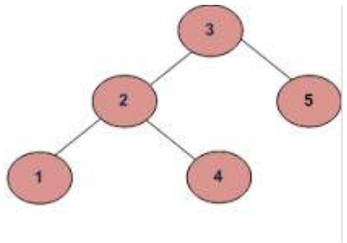
1. For example if the inputs are:

Example:1

a)



b)

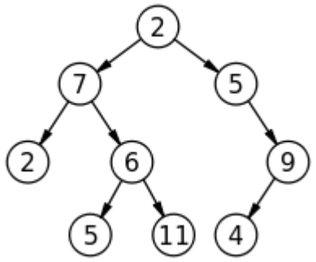


输出应为真。

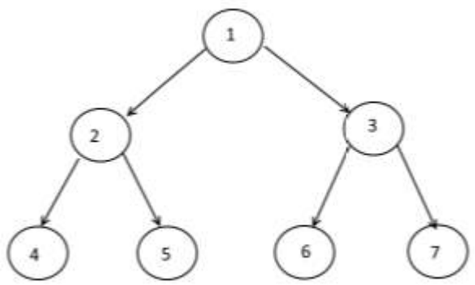
示例：2

如果输入是：

a)



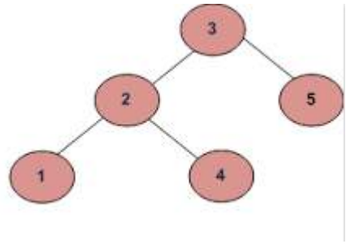
b)



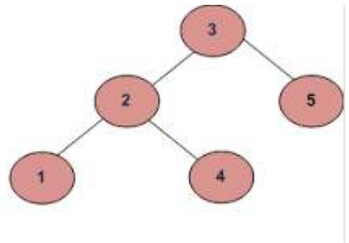
输出应为假。

相应的伪代码：

```
boolean sameTree(node root1, node root2){
```

b)

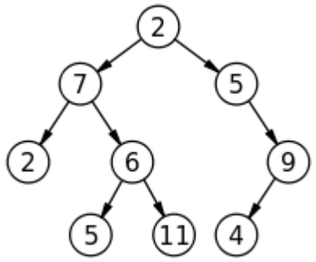


Output should be true.

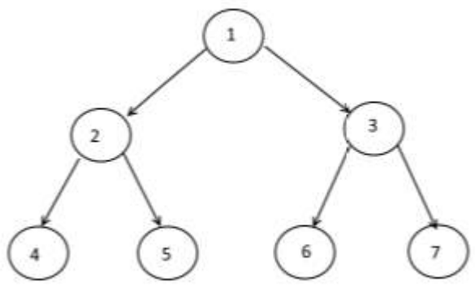
Example:2

If the inputs are:

a)



b)



Output should be false.

Pseudo code for the same:

```
boolean sameTree(node root1, node root2){
```

```
if(root1 == NULL && root2 == NULL)
return true;

if(root1 == NULL || root2 == NULL)
return false;

if(root1->data == root2->data
     && sameTree(root1->left,root2->left)
        && sameTree(root1->right, root2->right))
return true;

}
```



```
if(root1 == NULL && root2 == NULL)
return true;

if(root1 == NULL || root2 == NULL)
return false;

if(root1->data == root2->data
     && sameTree(root1->left,root2->left)
        && sameTree(root1->right, root2->right))
return true;

}
```

第5章：二叉搜索树

二叉树是每个节点最多有两个子节点的树。二叉搜索树（BST）是一种二叉树，其元素按特殊顺序排列。在每个BST中，左子树中的所有值（即键）都小于右子树中的值。

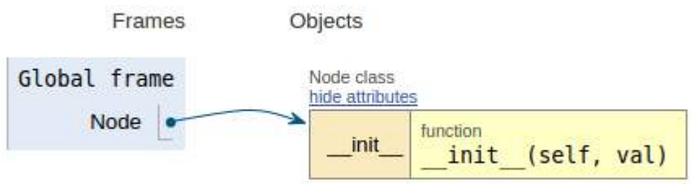
第5.1节：二叉搜索树 - 插入（Python）

这是一个使用Python实现的二叉搜索树插入的简单示例。

下面给出了一个示例：

在代码片段之后，每张图片展示了执行的可视化，这使得更容易理解这段代码的工作原理。

```
class Node:
def __init__(self, val):
self.l_child = None
        self.r_child = None
        self.data = val
```




```
def insert(root, node):
    if root is None:
root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
root.l_child = node
            else:
insert(root.l_child, node)
        else:
            if root.r_child is None:
```


Chapter 5: Binary Search Trees

Binary tree is a tree that each node in it has maximum of two children. Binary search tree (BST) is a binary tree which its elements positioned in special order. In each BST all values(i.e key) in left sub tree are less than values in right sub tree.

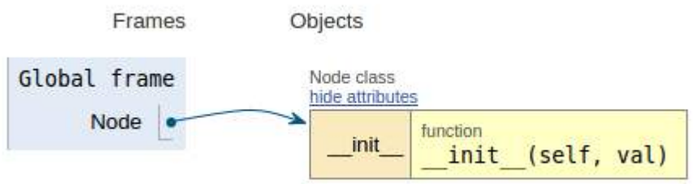
Section 5.1: Binary Search Tree - Insertion (Python)

This is a simple implementation of Binary Search Tree Insertion using Python.

An example is shown below:

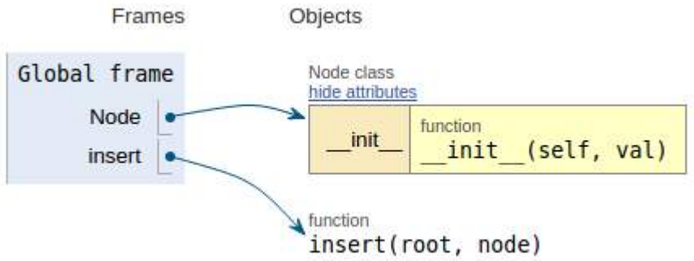
Following the code snippet each image shows the execution visualization which makes it easier to visualize how this code works.

```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```

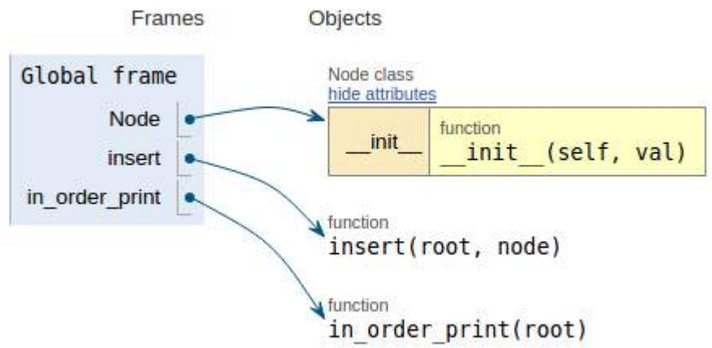



```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
```

```
root.r_child = node
                else:
insert(root.r_child, node)
```



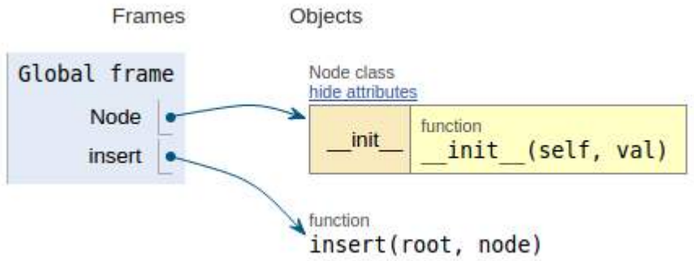

```
def in_order_print(root):
    if not root:
        返回
in_order_print(root.l_child)
    print root.data
in_order_print(root.r_child)
```



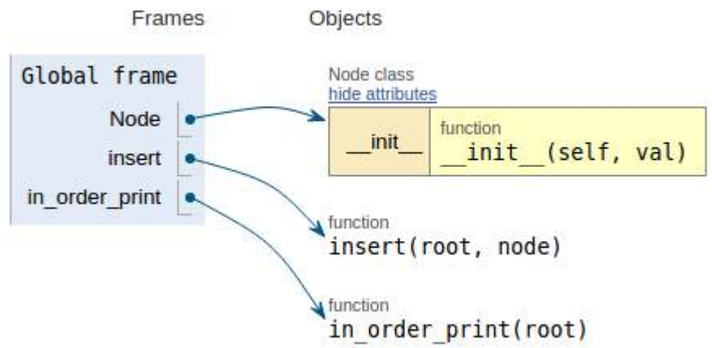

```
def pre_order_print(root):
    if not root:
        return
print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



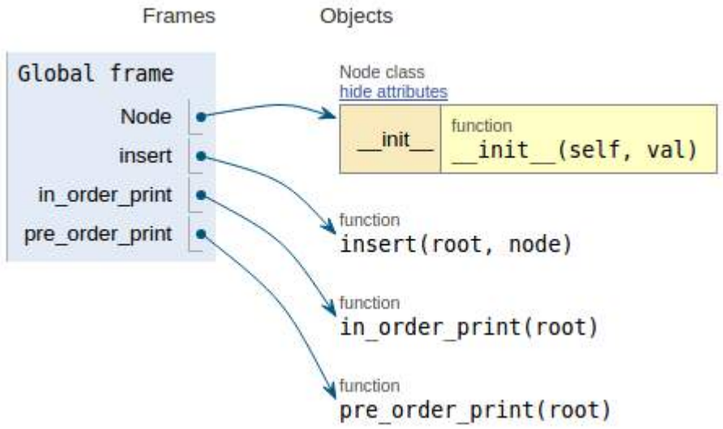
```
                root.r_child = node
            else:
                insert(root.r_child, node)
```




```
def in_order_print(root):
    if not root:
        return
    in_order_print(root.l_child)
    print root.data
    in_order_print(root.r_child)
```



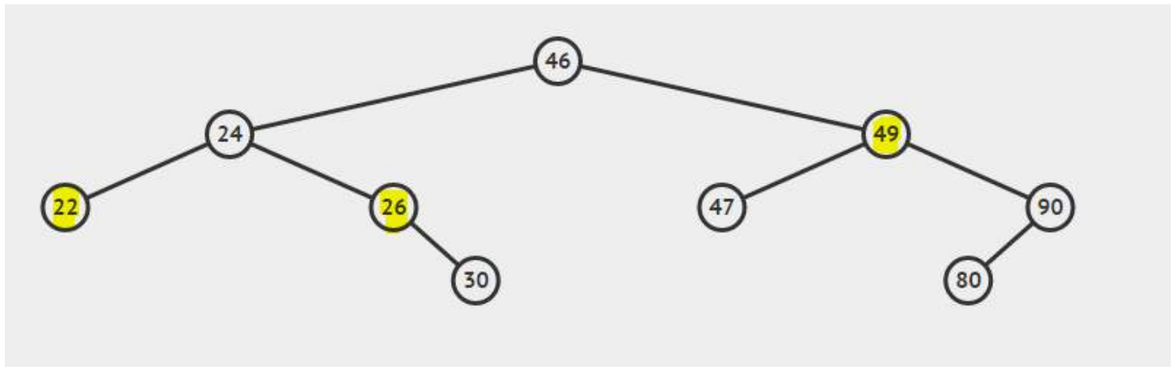

```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



第5.2节：二叉搜索树 - 删除（C++）

在开始删除之前，我想先介绍一下什么是二叉搜索树（BST）。BST中的每个节点最多有两个子节点（左子节点和右子节点）。节点的左子树的键值小于或等于其父节点的键值。节点的右子树的键值大于其父节点的键值。

在保持二叉搜索树性质的前提下删除树中的节点。



删除节点时需要考虑三种情况。

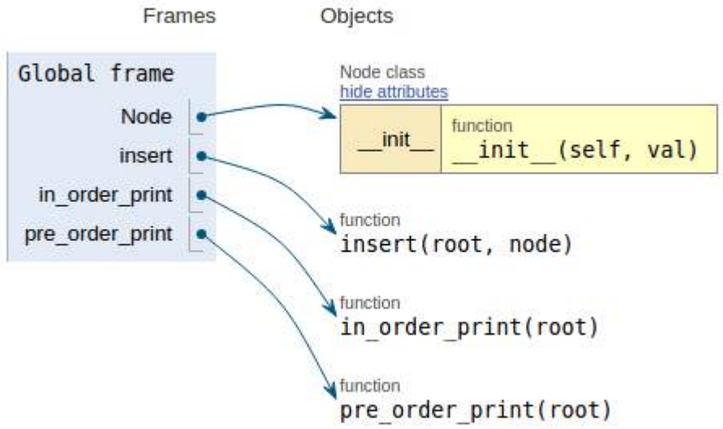
- 情况1：要删除的节点是叶子节点。（值为22的节点）。
- 案例2：要删除的节点有一个子节点。（值为26的节点）。
- 案例3：要删除的节点有两个子节点。（值为49的节点）。

案例说明：

1. 当要删除的节点是叶子节点时，只需删除该节点并向其父节点传递nullptr。
2. 当要删除的节点只有一个子节点时，将子节点的值复制到该节点的值中，然后删除子节点（转换为案例1）

3. 当要删除的节点有两个子节点时，可以将其右子树中的最小值复制到该节点，然后从该节点的右子树中删除该最小值（转换为案例2）

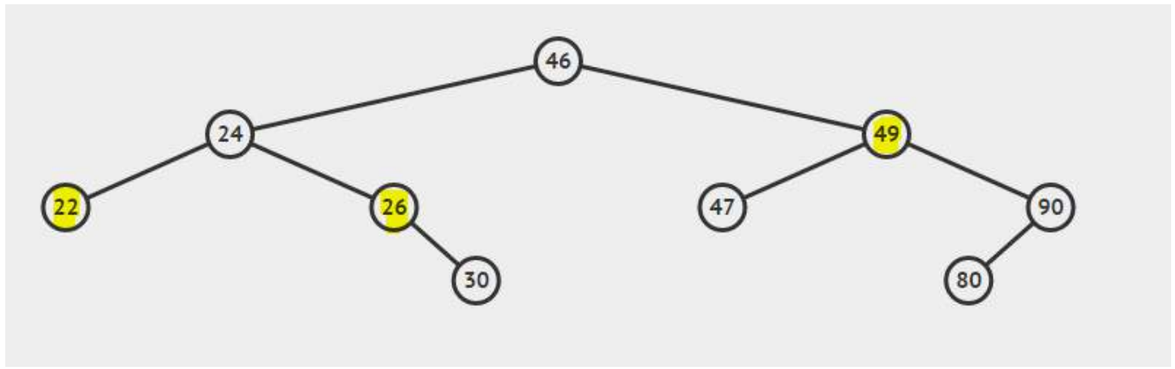
注意：右子树中的最小值最多只有一个子节点，且如果有子节点，则为右子节点；如果有左子节点，则说明它不是最小值，或者不符合二叉搜索树（BST）性质。



Section 5.2: Binary Search Tree - Deletion(C++)

Before starting with deletion I just want to put some lights on what is a Binary search tree(BST), Each node in a BST can have maximum of two nodes(left and right child).The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key.

Deleting a node in a tree while maintaining its **Binary search tree property.**



There are three cases to be considered while deleting a node.

- Case 1: Node to be deleted is the leaf node.(Node with value 22).
- Case 2: Node to be deleted has one child.(Node with value 26).
- Case 3: Node to be deleted has both children.(Node with value 49).

Explanation of cases:

1. When the node to be deleted is a leaf node then simply delete the node and pass `nullptr` to its parent node.
2. When a node to be deleted is having only one child then copy the child value to the node value and delete the child **(Converted to case 1)**
3. When a node to be delete is having two childs then the minimum from its right sub tree can be copied to the node and then the minimum value can be deleted from the node's right subtree **(Converted to Case 2)**

Note: The minimum in the right sub tree can have a maximum of one child and that too right child if it's having the left child that means it's not the minimum value or it's not following BST property.

树中节点的结构及删除的代码：

```
struct node
{
    int data;
节点 *左子节点, *右子节点;
};

节点* 删除节点(节点 *根节点, 整数 数据)
{
  如果(根节点 == 空指针) 返回 根节点;
  否则如果(数据 < 根节点->数据) 根节点->左子节点  = 删除节点(根节点->左子节点, 数据);
  否则如果(数据 > 根节点->数据) 根节点->右子节点 = 删除节点(根节点->右子节点, 数据);

  否则
  {
    如果(根节点->左子节点 == 空指针 && 根节点->右子节点 == 空指针) // 情况1
    {
释放(根节点);
     根节点 = 空指针;
    }
    否则如果(根节点->左子节点 == 空指针)       // 情况2
    {
节点* 临时节点 = 根节点;
     根节点= 根节点->右子节点;
     释放(临时节点);
    }
    否则如果(根节点->右子节点 == 空指针)      // 情况2
    {
node* temp = root;
       root = root->left;
       free(temp);
    }
    else                                // 情况 3
    {
node* temp = root->right;

       while(temp->left != nullptr) temp = temp->left;

root->data = temp->data;
       root->right = delete_node(root->right, temp->data);
    }
  }
  return root;
}
```

上述代码的时间复杂度是 O(h)，其中 h 是树的高度。

第5.3节：二叉搜索树中的最近公共祖先

考虑以下二叉搜索树：

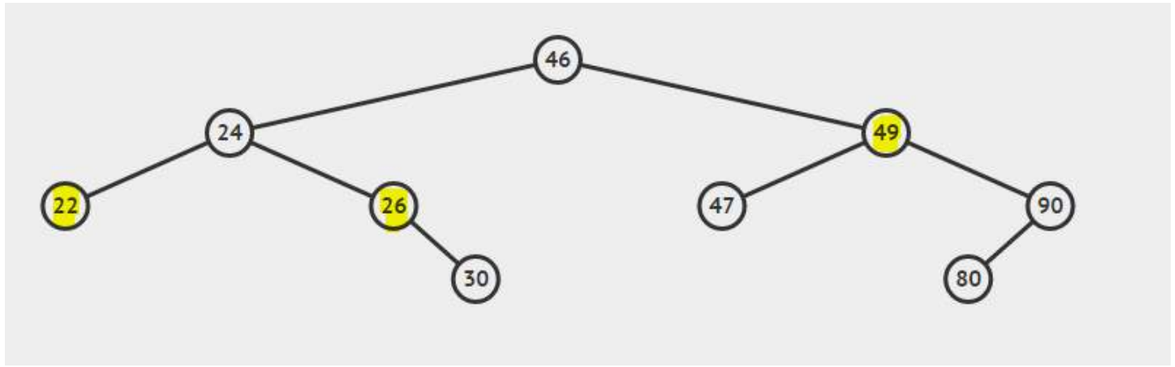
The structure of a node in a tree and the code for Deletion:

```
struct node
{
    int data;
    node *left, *right;
};

node* delete_node(node *root, int data)
{
  if(root == nullptr) return root;
  else if(data < root->data) root->left  = delete_node(root->left, data);
  else if(data > root->data) root->right = delete_node(root->right, data);

  else
  {
    if(root->left == nullptr && root->right == nullptr) // Case 1
    {
      free(root);
      root = nullptr;
    }
    else if(root->left == nullptr)       // Case 2
    {
       node* temp = root;
       root= root->right;
       free(temp);
    }
    else if(root->right == nullptr)      // Case 2
    {
       node* temp = root;
       root = root->left;
       free(temp);
    }
    else                                 // Case 3
    {
       node* temp = root->right;

       while(temp->left != nullptr) temp = temp->left;

       root->data = temp->data;
       root->right = delete_node(root->right, temp->data);
    }
  }
  return root;
}
```

Time complexity of above code is O(*h*), where *h* is the height of the tree.

Section 5.3: Lowest common ancestor in a BST

Consider the BST:



22 和 26 的最近公共祖先是 24

26 和 49 的最近公共祖先是 46

22 和 24 的最近公共祖先是 24

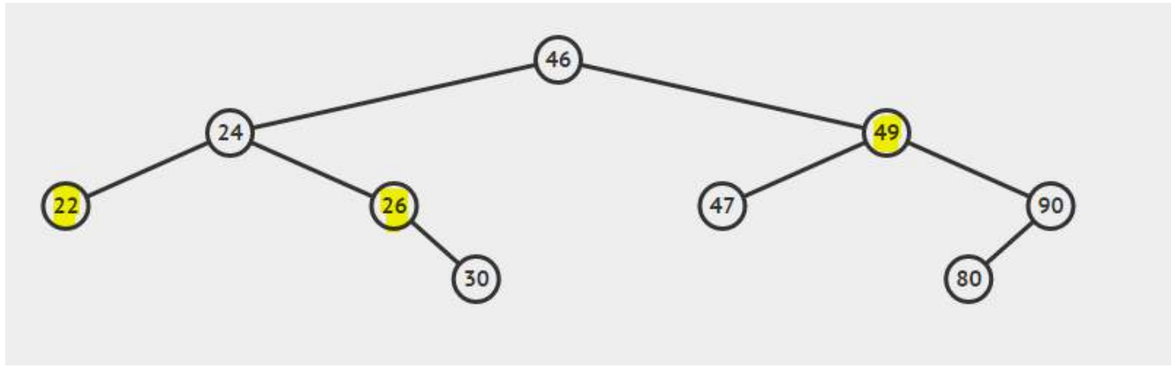
可以利用二叉搜索树的性质来查找节点的最近公共祖先

伪代码：

```
lowestCommonAncestor(root,node1, node2){

if(root == NULL)
return NULL;



else if(node1->data == root->data || node2->data== root->data)
    return root;

    else if((node1->data <= root->data && node2->data > root->data)
              || (node2->data <= root->data && node1->data > root->data)){

             return root;
    }

    else if(root->data > max(node1->data,node2->data)){
            return lowestCommonAncestor(root->left, node1, node2);
    }

        else {
            return lowestCommonAncestor(root->right, node1, node2);
      }
        }
```

第5.4节：二叉搜索树 - Python

```
class Node(object):
def __init__(self, val):
self.l_child = None
        self.r_child = None
        self.val = val


class 二叉搜索树(object):
def 插入(self, root, node):
```

Lowest common ancestor of 22 and 26 is 24

Lowest common ancestor of 26 and 49 is 46

Lowest common ancestor of 22 and 24 is 24

Binary search tree property can be used for finding nodes lowest ancestor

Psuedo code:

```
lowestCommonAncestor(root,node1, node2){

if(root == NULL)
return NULL;



else if(node1->data == root->data || node2->data== root->data)
    return root;

    else if((node1->data <= root->data && node2->data > root->data)
              || (node2->data <= root->data && node1->data > root->data)){

             return root;
    }

    else if(root->data > max(node1->data,node2->data)){
            return lowestCommonAncestor(root->left, node1, node2);
    }

        else {
            return lowestCommonAncestor(root->right, node1, node2);
      }
        }
```

Section 5.4: Binary Search Tree - Python

```
class Node(object):
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.val = val


class BinarySearchTree(object):
    def insert(self, root, node):
```

```
        if root is None:
            return node

        if root.val < node.val:
root.r_child = self.插入(root.r_child, node)
        else:
root.l_child = self.插入(root.l_child, node)

        return root


def 中序遍历(self, root):
        if not root:
            返回 无
        否则:
self.in_order_place(root.l_child)
            print root.val
self.in_order_place(root.r_child)

    def pre_order_place(self, root):
        if not root:
            返回 无
        否则:
print root.val
            self.pre_order_place(root.l_child)
            self.pre_order_place(root.r_child)

    def post_order_place(self, root):
        if not root:
            返回 无
        否则:
self.post_order_place(root.l_child)
            self.post_order_place(root.r_child)
            print root.val
```

""" 创建不同的节点并插入数据 """

```
r = Node(3)
node = BinarySearchTree()
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8]

for nd in nodeList:
node.insert(r, Node(nd))

print "------中序遍历 ---------"
print (node.in_order_place(r))
print "------先序遍历 ---------"
print (node.pre_order_place(r))
print "------后序遍历 ---------"
print (node.post_order_place(r))
```



```
        if root is None:
            return node

        if root.val < node.val:
            root.r_child = self.insert(root.r_child, node)
        else:
            root.l_child = self.insert(root.l_child, node)

        return root


    def in_order_place(self, root):
        if not root:
            return None
        else:
            self.in_order_place(root.l_child)
            print root.val
            self.in_order_place(root.r_child)

    def pre_order_place(self, root):
        if not root:
            return None
        else:
            print root.val
            self.pre_order_place(root.l_child)
            self.pre_order_place(root.r_child)

    def post_order_place(self, root):
        if not root:
            return None
        else:
            self.post_order_place(root.l_child)
            self.post_order_place(root.r_child)
            print root.val
```

""" Create different node and insert data into it"""

```
r = Node(3)
node = BinarySearchTree()
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8]

for nd in nodeList:
    node.insert(r, Node(nd))

print "------In order ---------"
print (node.in_order_place(r))
print "------Pre order ---------"
print (node.pre_order_place(r))
print "------Post order ---------"
print (node.post_order_place(r))
```

第6章：检查一棵树是否为二叉搜索树

第6.1节：检查给定二叉树是否为二叉搜索树的算法

如果满足以下任一条件，则二叉树是二叉搜索树（BST）：

1. 它是空的
2. 它没有子树
3. 对树中的每个节点x，左子树中的所有键（如果有）必须小于key(x)，右子树中的所有键（如果有）必须大于key(x)。

因此，一个直接的递归算法如下：

```
is_BST(root)：
   如果 root == NULL：
      返回 true

   // 检查左子树中的值
   if root->left != NULL:
max_key_in_left = find_max_key(root->left)
      if max_key_in_left > root->key:
            return false

   // 检查右子树中的值
   if root->right != NULL:
min_key_in_right = find_min_key(root->right)
      if min_key_in_right < root->key:
            return false

   return is_BST(root->left) && is_BST(root->right)
```

上述递归算法是正确的，但效率低，因为它多次遍历每个节点。

另一种减少多次访问每个节点的方法是记住我们访问的子树中键的最小和最大可能值。设任何键的最小可能值为K_MIN，最大值为
K_MAX。当我们从树的根节点开始时，树中值的范围是[K_MIN,K_MAX]。设根节点的键为 x。则左子树的值范围是[K_MIN,x)，右子树的值范围是
(x,K_MAX]。我们将利用这个思想来开发一个更高效的算法。

```
is_BST(root, min, max)：
      如果 root == NULL：
            返回 true

      // 当前节点的键是否超出范围？
      if root->key < min || root->key > max：
            return false

      // 检查左子树和右子树是否为BST
      return is_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)
```

初始调用方式为：

```
is_BST(my_tree_root,KEY_MIN,KEY_MAX)
```

另一种方法是对二叉树进行中序遍历。如果中序遍历产生的键序列是有序的，那么给定的树就是BST。为了检查中序序列是否有序，需要记住

Chapter 6: Check if a tree is BST or not

Section 6.1: Algorithm to check if a given binary tree is BST

A binary tree is BST if it satisfies any one of the following condition:

1. It is empty
2. It has no subtrees
3. For every node x in the tree all the keys (if any) in the left sub tree must be less than key(x) and all the keys (if any) in the right sub tree must be greater than key(x).

So a straightforward recursive algorithm would be:

```
is_BST(root):
  if root == NULL:
   return true

  // Check values in left subtree
  if root->left != NULL:
    max_key_in_left = find_max_key(root->left)
    if max_key_in_left > root->key:
        return false

  // Check values in right subtree
  if root->right != NULL:
    min_key_in_right = find_min_key(root->right)
    if min_key_in_right < root->key:
        return false

  return is_BST(root->left) && is_BST(root->right)
```

The above recursive algorithm is correct but inefficient, because it traverses each node mutiple times.

Another approach to minimize the multiple visits of each node is to remember the min and max possible values of the keys in the subtree we are visiting. Let the minimum possible value of any key be `K_MIN` and maximum value be `K_MAX`. When we start from the root of the tree, the range of values in the tree is `[K_MIN,K_MAX]`. Let the key of root node be x. Then the range of values in left subtree is `[K_MIN,x)` and the range of values in right subtree is `(x,K_MAX]`. We will use this idea to develop a more efficient algorithm.

```
is_BST(root, min, max):
    if root == NULL:
        return true

    // is the current node key out of range?
    if root->key < min || root->key > max:
        return false

    // check if left and right subtree is BST
    return is_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)
```

It will be initially called as:

```
is_BST(my_tree_root,KEY_MIN,KEY_MAX)
```

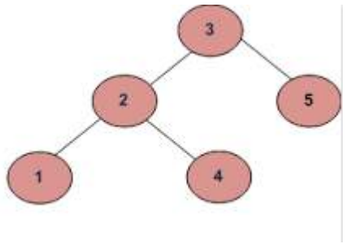
Another approach will be to do inorder traversal of the Binary tree. If the inorder traversal produces a sorted sequence of keys then the given tree is a BST. To check if the inorder sequence is sorted remember the value of

先前访问的节点并将其与当前节点进行比较。

第6.2节：判断给定的输入树是否满足二叉搜索树性质

例如

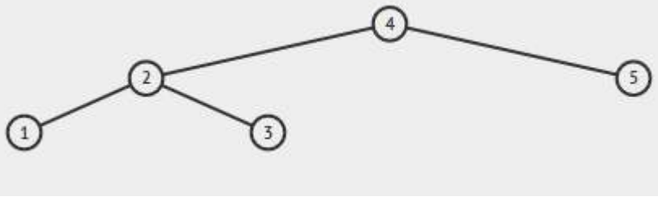
如果输入是：



输出应为假：

因为左子树中的4大于根节点的值（3）

如果输入是：



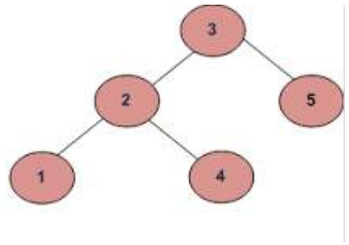
输出应为真

previously visited node and compare it against the current node.

Section 6.2: If a given input tree follows Binary search tree property or not

For example

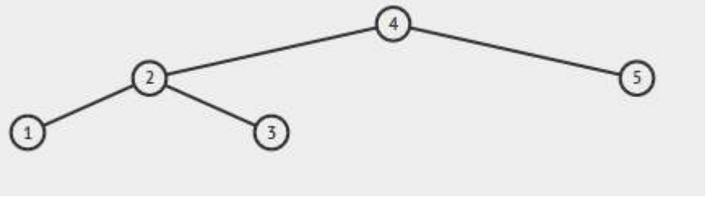
if the input is:



Output should be false:

As 4 in the left sub-tree is greater than the root value(3)

If the input is:



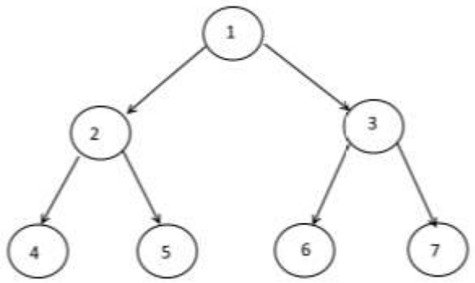
Output should be true

第7章：二叉树遍历

以某种特定顺序访问二叉树的节点称为遍历。

第7.1节：层序遍历 - 实现

例如，如果给定的树是：



层序遍历将会是

1 2 3 4 5 6 7

按层打印节点数据。

代码：

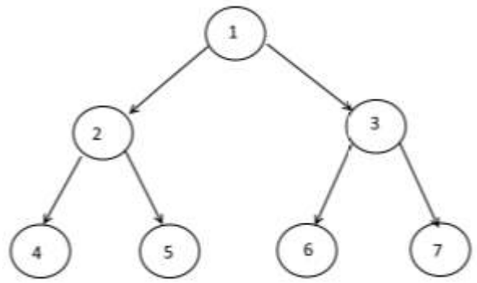
```
#include<iostream>
#include<queue>
#include<malloc.h>

using namespace std;

struct node{

    int data;
    node *left;
    node *right;
};

void levelOrder(struct node *root){

        if(root == NULL)     return;

queue<node *> Q;
Q.push(root);

        while(!Q.empty()){
struct     node* curr = Q.front();
            cout<< curr->data <<" ";
            if(curr->left != NULL) Q.push(curr-> left);
                if(curr->right != NULL) Q.push(curr-> right);

Q.pop();


        }
```


Chapter 7: Binary Tree traversals

Visiting a node of a binary tree in some particular order is called traversals.

Section 7.1: Level Order traversal - Implementation

For example if the given tree is:



Level order traversal will be

1 2 3 4 5 6 7

Printing node data level by level.

Code:

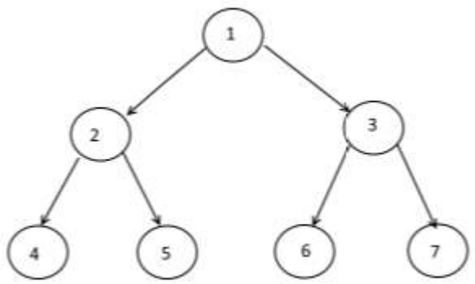
```
#include<iostream>
#include<queue>
#include<malloc.h>

using namespace std;

struct node{

    int data;
    node *left;
    node *right;
};

void levelOrder(struct node *root){

        if(root == NULL)    return;

        queue<node *> Q;
        Q.push(root);

        while(!Q.empty()){
        struct    node* curr = Q.front();
            cout<< curr->data <<" ";
            if(curr->left != NULL) Q.push(curr-> left);
                if(curr->right != NULL) Q.push(curr-> right);

                Q.pop();


        }
```

```
}
struct node* newNode(int data)
{
struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
  node->left = NULL;
    node->right = NULL;

     return(node);
}

int main(){

struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left  = newNode(6);
    root->right->right = newNode(7);


printf("二叉树的层序遍历是 ");      levelOrder(root);


    return 0;


}
```

队列数据结构用于实现上述目标。

第7.2节：二叉树的先序、中序和后序遍历

考虑以下二叉树：



先序遍历（根节点） 是先遍历节点，然后遍历该节点的左子树，最后遍历该节点的右子树。

因此，上述树的先序遍历结果为：

1 2 4 5 3 6 7

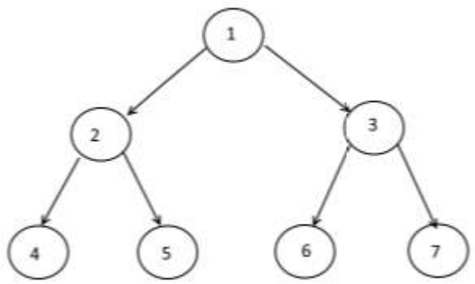
中序遍历（根节点） 是先遍历节点的左子树，然后遍历节点，最后遍历节点的右子树。


```
}
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main(){

    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left  = newNode(6);
    root->right->right = newNode(7);


    printf("Level Order traversal of binary tree is \n");
    levelOrder(root);

    return 0;


}
```

Queue data structure is used to achieve the above objective.

Section 7.2: Pre-order, Inorder and Post Order traversal of a Binary Tree

Consider the Binary Tree:



Pre-order traversal(root) is traversing the node then left sub-tree of the node and then the right sub-tree of the node.

So the pre-order traversal of above tree will be:

1 2 4 5 3 6 7

In-order traversal(root) is traversing the left sub-tree of the node then the node and then right sub-tree of the

节点。

因此，上述树的中序遍历结果为：

4 2 5 1 6 3 7

后序遍历（根节点） 是先遍历节点的左子树，然后遍历右子树，最后遍历该节点。

因此，上述树的后序遍历结果为：

4 5 2 6 7 3 1

node.

So the in-order traversal of above tree will be:

4 2 5 1 6 3 7

Post-order traversal(root) is traversing the left sub-tree of the node then the right sub-tree and then the node.

So the post-order traversal of above tree will be:

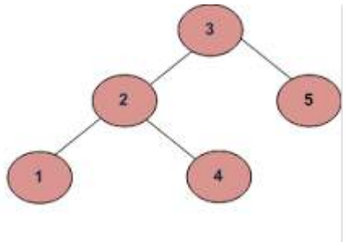
4 5 2 6 7 3 1

第8章：二叉树的最近公共祖先

两个节点n1和n2的最近公共祖先定义为树中最低的一个节点，该节点同时拥有n1和n2作为其子孙。

第8.1节：寻找最近公共祖先

考虑如下树：



值为1和4的节点的最近公共祖先是2

值为1和5的节点的最近公共祖先是3

值为2和4的节点的最近公共祖先是4

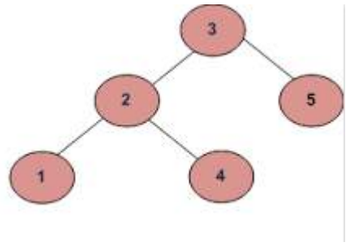
值为1和2的节点的最近公共祖先是2

Chapter 8: Lowest common ancestor of a Binary Tree

Lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in the tree that has both n1 and n2 as descendants.

Section 8.1: Finding lowest common ancestor

Consider the tree:



Lowest common ancestor of nodes with value 1 and 4 is 2

Lowest common ancestor of nodes with value 1 and 5 is 3

Lowest common ancestor of nodes with value 2 and 4 is 4

Lowest common ancestor of nodes with value 1 and 2 is 2

第9章：图

图是由一些点和连接这些点的线组成的集合，这些线连接的是点的某个（可能为空的）子集。图中的点称为图顶点、“节点”或简称“点”。类似地，连接图顶点的线称为图边、“弧”或“线”。

图 G 可以定义为一对 (V,E)，其中 V 是顶点集合，E 是顶点之间的边集合，满足 E ⊆ {(u,v) | u, v ∈ V}。

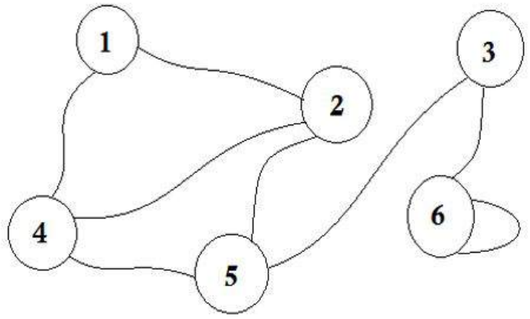
第9.1节：存储图（邻接矩阵）

存储图有两种常见方法：

- 邻接矩阵
- 邻接表

邻接矩阵是一种用于表示有限图的方阵。矩阵的元素表示图中顶点对是否相邻。

相邻的意思是“紧挨着或毗邻某物”，或者在某物旁边。例如，你的邻居就是与你相邻的人。在图论中，如果我们可以从节点 A 走到节点 B，则可以说节点 B 与节点 A 相邻。
现在我们将学习如何通过邻接矩阵存储哪些节点与哪些节点相邻。这意味着，我们将表示哪些节点之间共享边。这里的矩阵指的是二维数组。



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

这里你可以看到图旁边的一个表格，这就是我们的邻接矩阵。这里 Matrix[i][j] = 1 表示节点 i 和节点 j 之间有一条边。如果没有边，我们就简单地将 Matrix[i][j] = 0。

这些边可以带权重，比如它可以表示两个城市之间的距离。然后我们将在矩阵中填入该值。
Matrix[i][j] 而不是放置1。

上述描述的图是双向或无向的，这意味着，如果我们可以从节点1到达节点2，那么我们也可以从节点2到达节点1。如果图是有向的，那么图的一侧会有箭头标志。即便如此，我们仍然可以使用邻接矩阵来表示它。

Chapter 9: Graph

A graph is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are called graph vertices, "nodes" or simply "points." Similarly, the lines connecting the vertices of a graph are called graph edges, "arcs" or "lines."

A graph G can be defined as a pair (V,E), where V is a set of vertices, and E is a set of edges between the vertices E ⊆ {(u,v) | u, v ∈ V}.

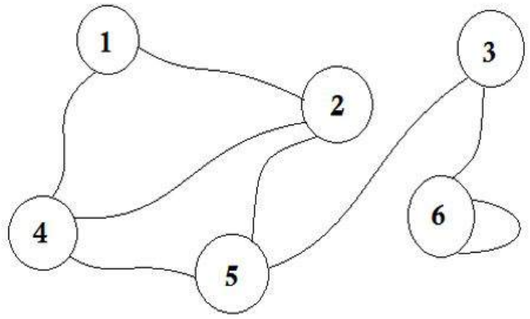
Section 9.1: Storing Graphs (Adjacency Matrix)

To store a graph, two methods are common:

- Adjacency Matrix
- Adjacency List

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

Adjacent means 'next to or adjoining something else' or to be beside something. For example, your neighbors are adjacent to you. In graph theory, if we can go to **node B** from **node A**, we can say that **node B** is adjacent to **node A**. Now we will learn about how to store which nodes are adjacent to which one via Adjacency Matrix. This means, we will represent which nodes share edge between them. Here matrix means 2D array.

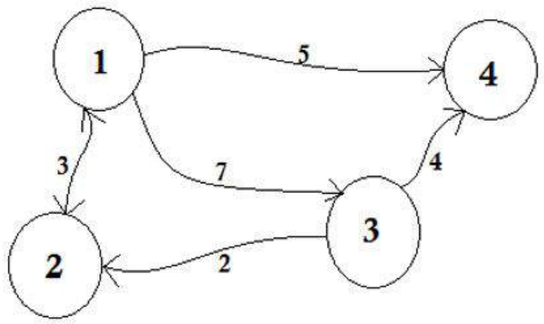


Node	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

Here you can see a table beside the graph, this is our adjacency matrix. Here **Matrix[i][j] = 1** represents there is an edge between **i** and **j**. If there's no edge, we simply put **Matrix[i][j] = 0**.

These edges can be weighted, like it can represent the distance between two cities. Then we'll put the value in **Matrix[i][j]** instead of putting 1.

The graph described above is *Bidirectional* or *Undirected*, that means, if we can go to **node 1** from **node 2**, we can also go to **node 2** from **node 1**. If the graph was *Directed*, then there would've been arrow sign on one side of the graph. Even then, we could represent it using adjacency matrix.



Node	1	2	3	4
1	Inf	3	7	5
2	3	Inf	Inf	Inf
3	Inf	2	Inf	4
4	Inf	Inf	Inf	Inf

我们用无穷大来表示不共享边的节点。需要注意的是，如果图是无向的，矩阵将变成对称的。

创建矩阵的伪代码：

```
过程 邻接矩阵(N):     // N表示节点数
矩阵[N][N]
对于 i 从 1 到 N
     对于 j 从 1 到 N
        输入 -> 矩阵[i][j]
    结束循环
结束循环
```

我们也可以用这种常见方式填充矩阵：

```
过程 邻接矩阵(N, E):     // N -> 节点数
矩阵[N][E]                          // E -> 边数
对于 i 从 1 到 E
输入 -> n1, n2, cost
    矩阵[n1][n2] = cost
    矩阵[n2][n1] = cost
结束循环
```

对于有向图，我们可以去掉Matrix[n2][n1] = cost这一行。

使用邻接矩阵的缺点：

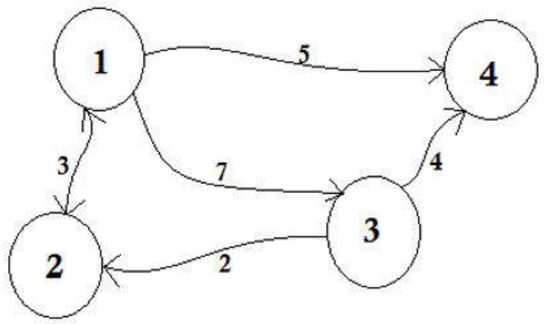
内存是一个大问题。无论有多少条边，我们总是需要一个大小为N * N的矩阵，其中N是节点数。如果有10000个节点，矩阵大小将是4 * 10000 * 10000，大约381兆字节。如果考虑边很少的图，这将是巨大的内存浪费。

假设我们想找出从节点**u**可以到达哪些节点。我们需要检查**u**的整行，这会花费大量时间。

唯一的好处是，我们可以很容易地使用邻接

使用上述伪代码实现的Java代码：

```
import java.util.Scanner;

public class Represent_Graph_Adjacency_Matrix
{
    private final int vertices;
```

Node	1	2	3	4
1	Inf	3	7	5
2	3	Inf	Inf	Inf
3	Inf	2	Inf	4
4	Inf	Inf	Inf	Inf

We represent the nodes that don't share edge by *infinity*. One thing to be noticed is that, if the graph is undirected, the matrix becomes *symmetric*.

The pseudo-code to create the matrix:

```
Procedure AdjacencyMatrix(N):    //N represents the number of nodes
Matrix[N][N]
for i from 1 to N
    for j from 1 to N
        Take input -> Matrix[i][j]
    endfor
endfor
```

We can also populate the Matrix using this common way:

```
Procedure AdjacencyMatrix(N, E):    // N -> number of nodes
Matrix[N][E]                        // E -> number of edges
for i from 1 to E
    input -> n1, n2, cost
    Matrix[n1][n2] = cost
    Matrix[n2][n1] = cost
endfor
```

For directed graphs, we can remove **Matrix[n2][n1] = cost** line.

The drawbacks of using Adjacency Matrix:

Memory is a huge problem. No matter how many edges are there, we will always need N * N sized matrix where N is the number of nodes. If there are 10000 nodes, the matrix size will be 4 * 10000 * 10000 around 381 megabytes. This is a huge waste of memory if we consider graphs that have a few edges.

Suppose we want to find out to which node we can go from a node **u**. We'll need to check the whole row of **u**, which costs a lot of time.

The only benefit is that, we can easily find the connection between **u-v** nodes, and their cost using Adjacency Matrix.

Java code implemented using above pseudo-code:

```
import java.util.Scanner;

public class Represent_Graph_Adjacency_Matrix
{
    private final int vertices;
```

```
    private int[][] 邻接矩阵;

    public 表示图的邻接矩阵(int 顶点数)
    {
顶点 = v;
邻接矩阵 = new int[顶点 + 1][顶点 + 1];
    }

    public void 创建边(int 到, int 从, int 边)
    {
        try
        {
邻接矩阵[到][从] = 边;
        }
        catch (ArrayIndexOutOfBoundsException 索引)
        {
            System.out.println("顶点不存在");
        }
    }

    public int 获取边(int 到, int 从)
    {
        try
        {
            return 邻接矩阵[到][从];
        }
        catch (ArrayIndexOutOfBoundsException 索引)
        {
            System.out.println("顶点不存在");
        }
        return -1;
    }

    public static void main(String args[])
    {
        int v, e, count = 1, to = 0, from = 0;
        Scanner sc = new Scanner(System.in);
        Represent_Graph_Adjacency_Matrix graph;
        try
        {
            System.out.println("请输入顶点数: ");
          v = sc.nextInt();
            System.out.println("请输入边数: ");
          e = sc.nextInt();

graph = new Represent_Graph_Adjacency_Matrix(v);

            System.out.println("请输入边: <to> <from>");
            while (count <= e)
            {
to = sc.nextInt();
                from = sc.nextInt();

                graph.makeEdge(to, from, 1);
                count++;
            }

            System.out.println("给定图的邻接矩阵为: ");
            System.out.print("   ");
            for (int i = 1; i <= v; i++)
                System.out.print(i + " ");
            System.out.println();
```



```
    private int[][] adjacency_matrix;

    public Represent_Graph_Adjacency_Matrix(int v)
    {
        vertices = v;
        adjacency_matrix = new int[vertices + 1][vertices + 1];
    }

    public void makeEdge(int to, int from, int edge)
    {
        try
        {
            adjacency_matrix[to][from] = edge;
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
    }

    public int getEdge(int to, int from)
    {
        try
        {
            return adjacency_matrix[to][from];
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
        return -1;
    }

    public static void main(String args[])
    {
        int v, e, count = 1, to = 0, from = 0;
        Scanner sc = new Scanner(System.in);
        Represent_Graph_Adjacency_Matrix graph;
        try
        {
            System.out.println("Enter the number of vertices: ");
            v = sc.nextInt();
            System.out.println("Enter the number of edges: ");
            e = sc.nextInt();

            graph = new Represent_Graph_Adjacency_Matrix(v);

            System.out.println("Enter the edges: <to> <from>");
            while (count <= e)
            {
                to = sc.nextInt();
                from = sc.nextInt();

                graph.makeEdge(to, from, 1);
                count++;
            }

            System.out.println("The adjacency matrix for the given graph is: ");
            System.out.print("   ");
            for (int i = 1; i <= v; i++)
                System.out.print(i + " ");
            System.out.println();
```

```
            for (int i = 1; i <= v; i++)
            {
                System.out.print(i + " ");
                for (int j = 1; j <= v; j++)
                    System.out.print(graph.getEdge(i, j) + " ");
                System.out.println();
            }

        }
        catch (Exception E)
        {
            System.out.println("Somthing went wrong");
        }

sc.close();
    }
}
```

运行代码：保存文件并使用 `javac Represent_Graph_Adjacency_Matrix.java` 编译

示例：

```
$ java Represent_Graph_Adjacency_Matrix
请输入顶点数：
4
请输入边数：
6
请输入边：
1 1
3 4
2 3
1 4
2 4
1 2
给定图的邻接矩阵为：
1 2 3 4
1 1 1 0 1
2 0 0 1 1
3 0 0 0 1
4 0 0 0 0
```

第9.2节：图论导论

图论是研究图的学科，图是用于建模对象之间成对关系的数学结构。

你知道吗，地球上几乎所有的问题都可以转化为道路和城市的问题，并加以解决？图论是在计算机发明之前很久就被发明出来的。莱昂哈德·欧拉写了一篇关于柯尼斯堡七桥问题的论文，这被认为是图论的第一篇论文。从那时起，人们逐渐意识到，如果我们能将任何问题转化为这个城市-道路问题，就能通过图论轻松解决它。

图论有许多应用。其中最常见的应用之一是寻找两个城市之间的最短距离。我们都知道，为了访问你的电脑，这个网页必须经过服务器上的许多路由器。
图论帮助找出需要经过的路由器。在战争期间，哪些街道需要被轰炸以切断首都与其他地区的联系，也可以通过图论找出。


```
            for (int i = 1; i <= v; i++)
            {
                System.out.print(i + " ");
                for (int j = 1; j <= v; j++)
                    System.out.print(graph.getEdge(i, j) + " ");
                System.out.println();
            }

        }
        catch (Exception E)
        {
            System.out.println("Somthing went wrong");
        }

        sc.close();
    }
}
```

Running the code: Save the file and compile using `javac Represent_Graph_Adjacency_Matrix.java`

Example:

```
$ java Represent_Graph_Adjacency_Matrix
Enter the number of vertices:
4
Enter the number of edges:
6
Enter the edges:
1 1
3 4
2 3
1 4
2 4
1 2
The adjacency matrix for the given graph is:
1 2 3 4
1 1 1 0 1
2 0 0 1 1
3 0 0 0 1
4 0 0 0 0
```

Section 9.2: Introduction To Graph Theory

Graph Theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.

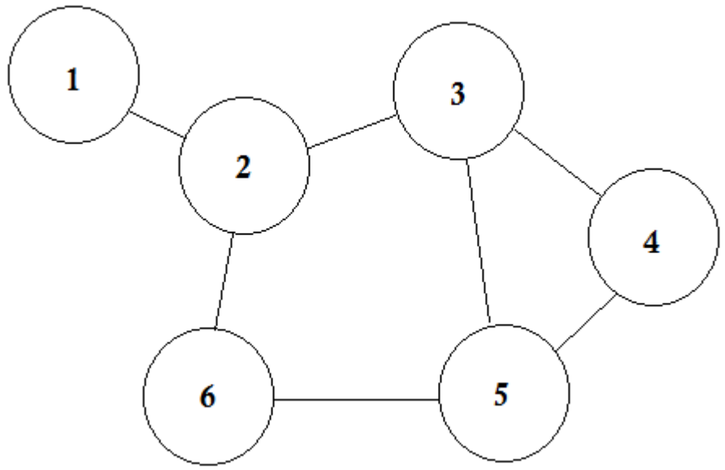
Did you know, almost all the problems of planet Earth can be converted into problems of Roads and Cities, and solved? Graph Theory was invented many years ago, even before the invention of computer. Leonhard Euler wrote a paper on the Seven Bridges of Königsberg which is regarded as the first paper of Graph Theory. Since then, people have come to realize that if we can convert any problem to this City-Road problem, we can solve it easily by Graph Theory.

Graph Theory has many applications.One of the most common application is to find the shortest distance between one city to another. We all know that to reach your PC, this web-page had to travel many routers from the server. Graph Theory helps it to find out the routers that needed to be crossed. During war, which street needs to be bombarded to disconnect the capital city from others, that too can be found out using Graph Theory.

让我们先学习一些图论的基本定义。

图：

假设我们有6个城市。我们将它们标记为1、2、3、4、5、6。现在我们连接那些彼此之间有道路的城市。



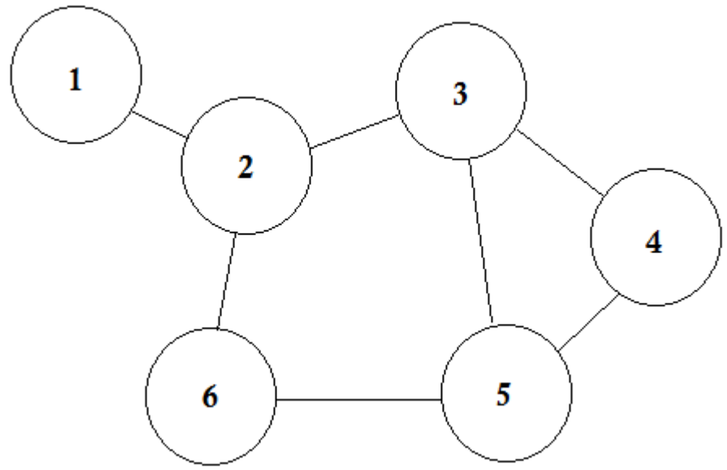
这是一个简单的图，显示了一些城市以及连接它们的道路。在图论中，我们称每个城市为节点或顶点，道路称为边。图就是这些节点和边的连接。

一个节点可以代表很多东西。在某些图中，节点代表城市，有些代表机场，有些代表棋盘上的一个格子。边表示节点之间的关系。这个关系可以是从一个机场到另一个机场的时间，骑士从一个格子移动到其他所有格子的走法等。

Let us first learn some basic definitions on Graph Theory.

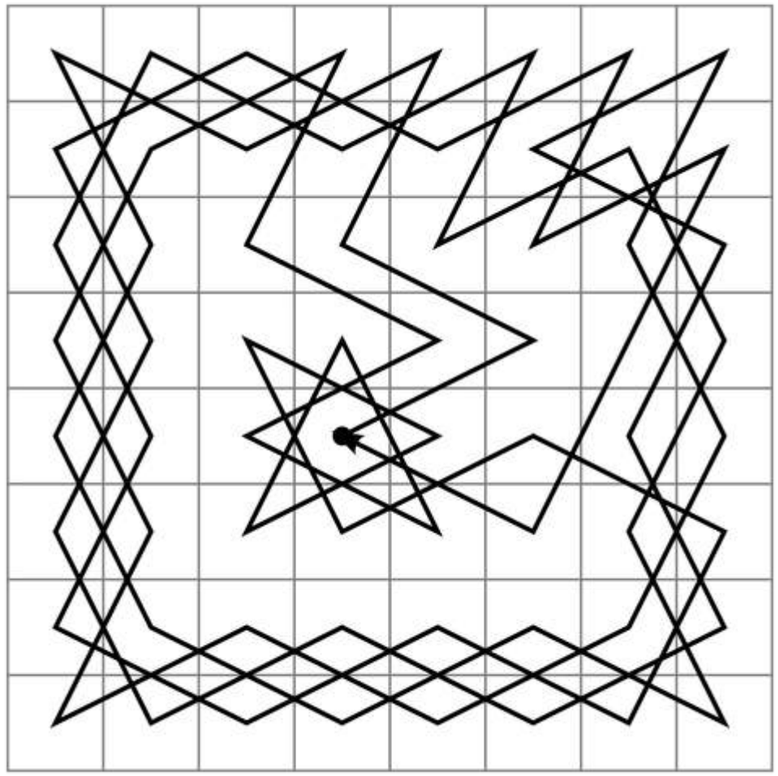
Graph:

Let's say, we have 6 cities. We mark them as 1, 2, 3, 4, 5, 6. Now we connect the cities that have roads between each other.



This is a simple graph where some cities are shown with the roads that are connecting them. In Graph Theory, we call each of these cities **Node** or **Vertex** and the roads are called **Edge.** Graph is simply a connection of these nodes and edges.

A **node** can represent a lot of things. In some graphs, nodes represent cities, some represent airports, some represent a square in a chessboard. **Edge** represents the relation between each nodes. That relation can be the time to go from one airport to another, the moves of a knight from one square to all the other squares etc.



棋盘上骑士的路径

棋盘上骑士的路径

简单来说，节点代表任何对象，边代表两个对象之间的关系。

邻接节点：

如果节点A与节点B共享一条边，那么B被认为是A的邻接节点。换句话说，如果两个节点直接相连，它们被称为邻接节点。一个节点可以有多个邻接节点。

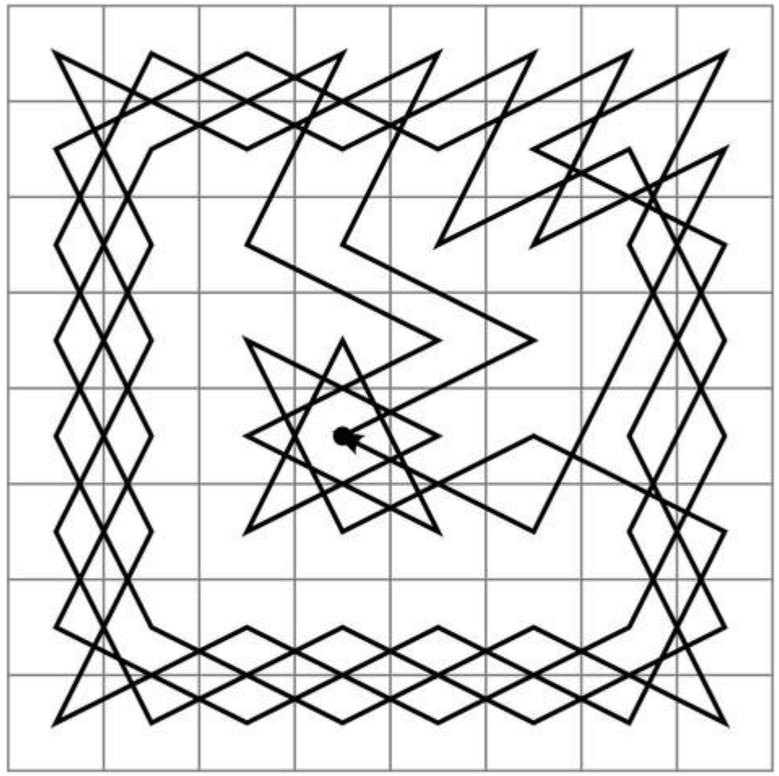
有向图和无向图：

在有向图中，边的一侧带有方向标志，这意味着边是单向的。另一方面，无向图的边在两侧都有方向标志，这意味着它们是双向的。
通常无向图的边两侧没有方向标志。

假设有一个正在举行的聚会。聚会中的人用节点表示，如果两个人握手，则这两个人之间有一条边。这个图是无向的，因为任何人A与人B握手当且仅当B也与A握手。相反，如果从人A到另一人B的边表示A钦佩B，那么这个图是有向的，因为钦佩不一定是相互的。前一种图称为无向图，边称为无向边，而后一种图称为有向图，边称为有向边。

带权图和无权图：

带权图是指每条边都分配了一个数字（权重）的图。根据具体问题，这些权重可能表示成本、长度或容量等。



Path of Knight in a Chessboard

Path of Knight in a Chessboard

In simple words, a **Node** represents any object and **Edge** represents the relation between two objects.

Adjacent Node:

If a node **A** shares an edge with node **B**, then **B** is considered to be adjacent to **A**. In other words, if two nodes are directly connected, they are called adjacent nodes. One node can have multiple adjacent nodes.

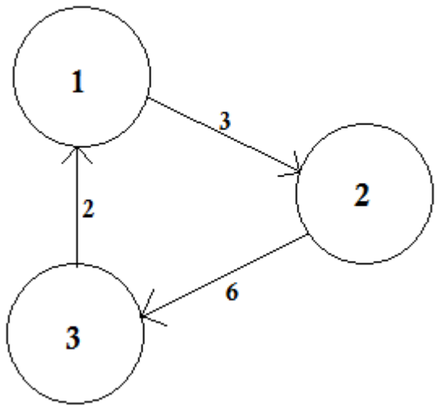
Directed and Undirected Graph:

In directed graphs, the edges have direction signs on one side, that means the edges are *Unidirectional*. On the other hand, the edges of undirected graphs have direction signs on both sides, that means they are *Bidirectional*. Usually undirected graphs are represented with no signs on the either sides of the edges.

Let's assume there is a party going on. The people in the party are represented by nodes and there is an edge between two people if they shake hands. Then this graph is undirected because any person **A** shake hands with person **B** if and only if **B** also shakes hands with **A**. In contrast, if the edges from a person **A** to another person **B** corresponds to **A**'s admiring **B**, then this graph is directed, because admiration is not necessarily reciprocated. The former type of graph is called an *undirected graph* and the edges are called *undirected edges* while the latter type of graph is called a *directed graph* and the edges are called *directed edges.*

Weighted and Unweighted Graph:

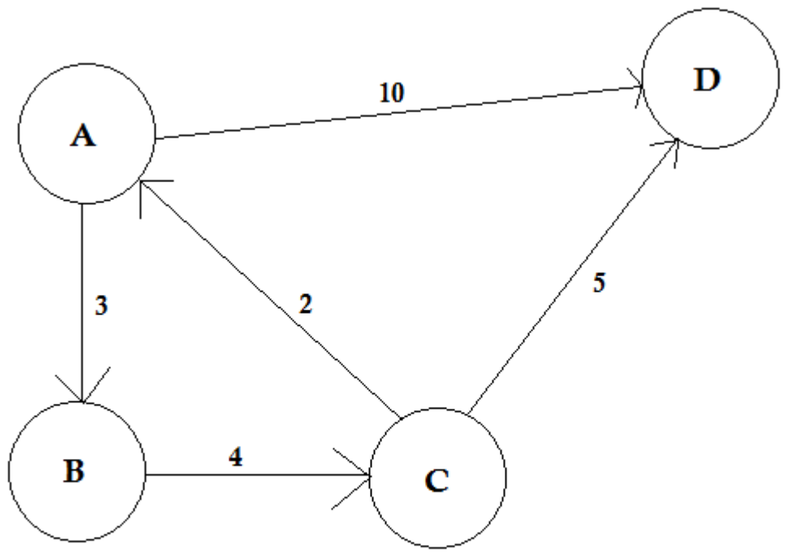
A weighted graph is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand.



无权图则相反。我们假设所有边的权重相同（通常为1）。

路径：

路径表示从一个节点到另一个节点的方式。它由一系列边组成。可能存在多条路径

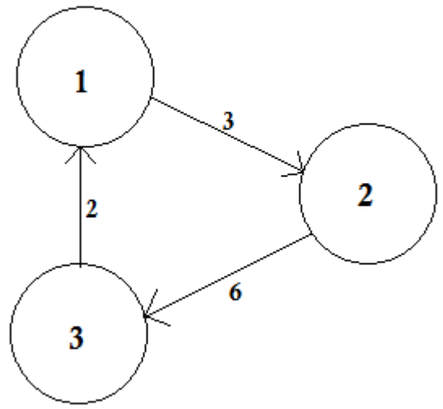


两个节点之间的路径。

在上面的例子中，从A到D有两条路径。A->B，B->C，C->D是一条路径。这条路径的代价是3 + 4 +2 = 9。还有另一条路径A->D。这条路径的代价是10。代价最低的路径称为最短路径。

度数：

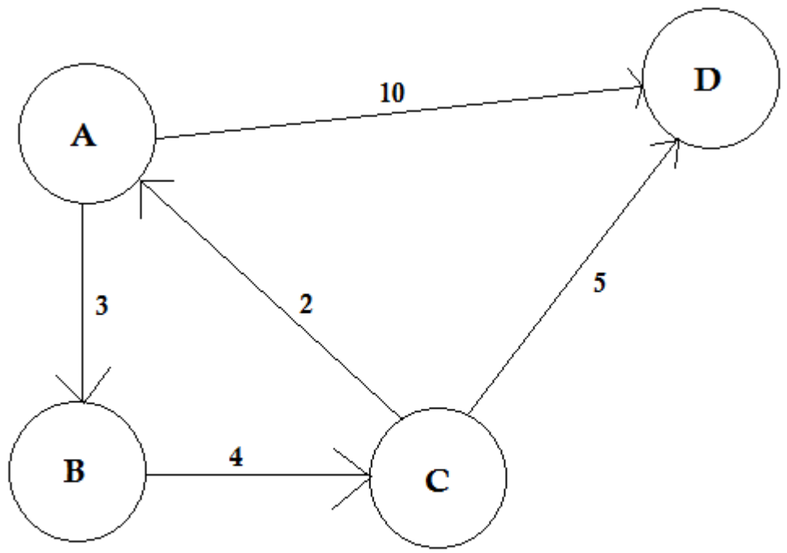
顶点的度数是与其相连的边的数量。如果有任何边两端都连接到该顶点（环），则计为两次。



An unweighted graph is simply the opposite. We assume that, the weight of all the edges are same (presumably 1).

Path:

A path represents a way of going from one node to another. It consists of sequence of edges. There can be multiple



paths between two nodes.

In the example above, there are two paths from **A** to **D**. **A->B, B->C, C->D** is one path. The cost of this path is **3 + 4 + 2 = 9**. Again, there's another path **A->D**. The cost of this path is **10**. The path that costs the lowest is called *shortest path*.

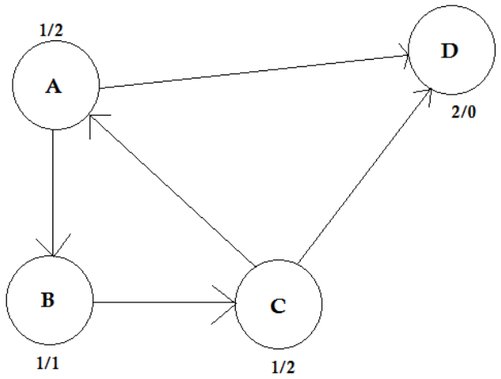
Degree:

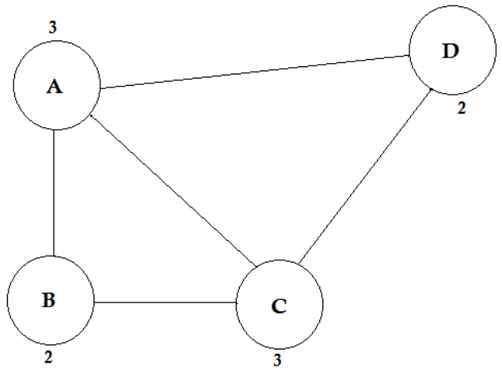
The degree of a vertex is the number of edges that are connected to it. If there's any edge that connects to the vertex at both ends (a loop) is counted twice.

在有向图中，节点有两种度数：

- 入度：指向该节点的边的数量。
- 出度：从该节点指向其他节点的边的数量。

对于无向图，则统称为度数。





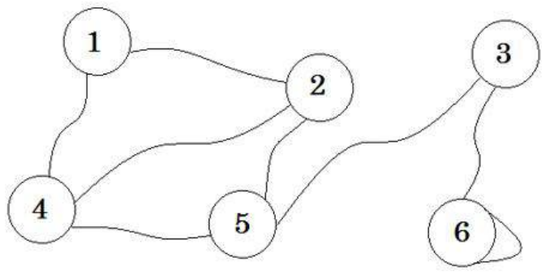
与图论相关的一些算法

- Bellman–Ford算法
- Dijkstra算法
- Ford–Fulkerson算法
- Kruskal算法
- 最近邻算法
- Prim算法
- 深度优先搜索
- 广度优先搜索

第9.3节：存储图（邻接表）

邻接表是一组无序列表，用于表示有限图。每个列表描述图中一个顶点的邻居集合。它占用的内存较少。

让我们看一个图及其邻接矩阵：



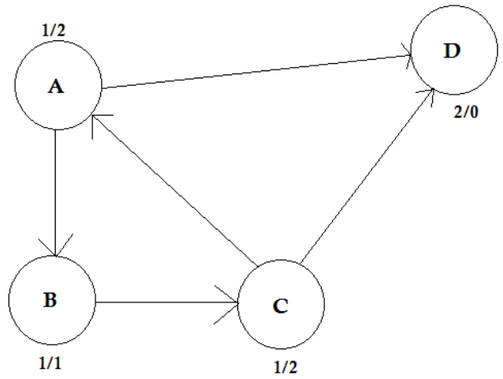
Node	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

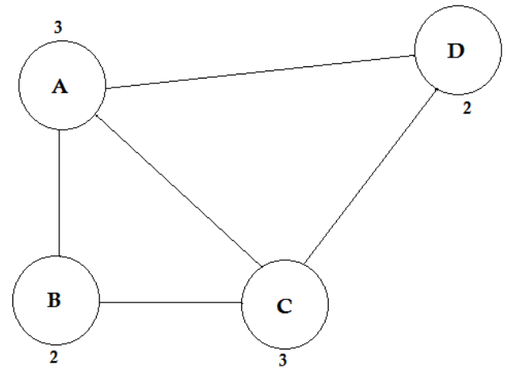
现在我们用这些值创建一个列表。

In directed graphs, the nodes have two types of degrees:

- In-degree: The number of edges that point to the node.
- Out-degree: The number of edges that point from the node to other nodes.

For undirected graphs, they are simply called degree.





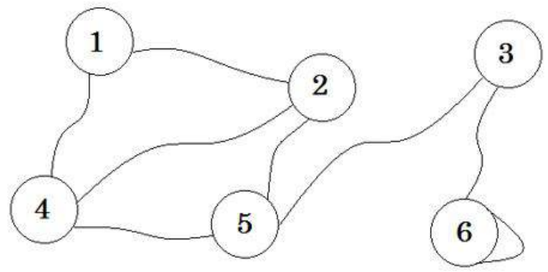
Some Algorithms Related to Graph Theory

- Bellman–Ford algorithm
- Dijkstra's algorithm
- Ford–Fulkerson algorithm
- Kruskal's algorithm
- Nearest neighbour algorithm
- Prim's algorithm
- Depth-first search
- Breadth-first search

Section 9.3: Storing Graphs (Adjacency List)

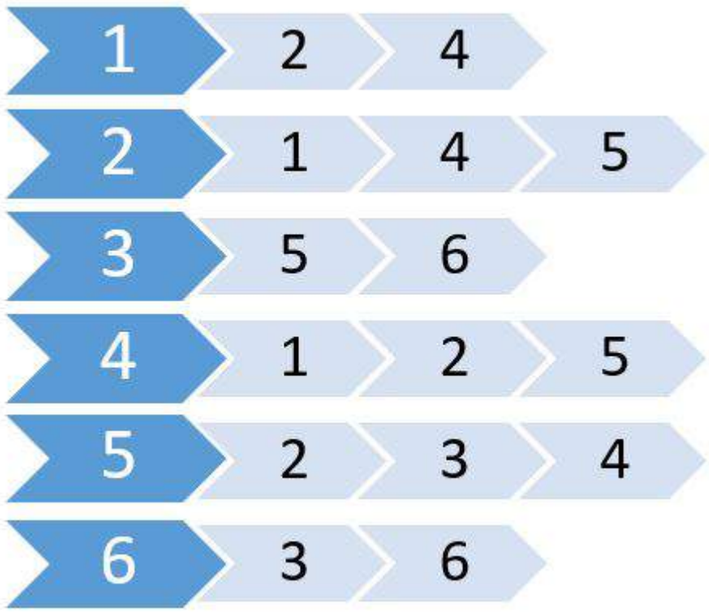
Adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in a graph. It takes less memory to store graphs.

Let's see a graph, and its adjacency matrix:



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

Now we create a list using these values.



这称为邻接表。它显示了哪些节点与哪些节点相连。我们可以使用二维数组存储这些信息，但这将占用与邻接矩阵相同的内存。相反，我们将使用动态分配的内存来存储它。

许多语言支持Vector或List，我们可以用它们来存储邻接表。对于这些，我们不需要指定List的大小，只需指定最大节点数。

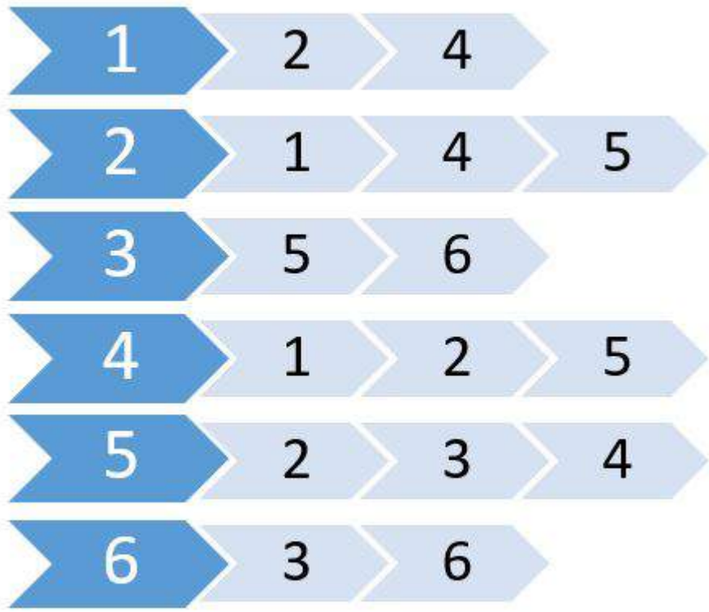
伪代码如下：

```
过程 邻接表-List(maxN, E):        // maxN表示最大节点数
edge[maxN] = Vector()                        // E表示边的数量
对于 i 从 1 到 E
输入 -> x, y                          // 这里 x, y 表示 x 和 y 之间存在一条边
    edge[x].push(y)
edge[y].push(x)
结束 for
返回 edge
```

由于这是一个无向图，如果存在从 x 到 y 的边，也存在从 y 到 x 的边。如果是有向图，我们会省略第二条边。对于带权图，我们还需要存储权值。我们将创建另一个vector 或 list，命名为 cost[] 来存储这些权值。伪代码如下：

```
过程 邻接表-列表(maxN, E):
edge[maxN] = Vector()
cost[maxN] = Vector()
对于 i 从 1 到 E
    输入 -> x, y, w
    edge[x].push(y)
    cost[x].push(w)
结束 for
返回 edge, cost
```

通过这个，我们可以轻松找出与任意节点相连的节点总数，以及这些节点具体是哪些。



This is called adjacency list. It shows which nodes are connected to which nodes. We can store this information using a 2D array. But will cost us the same memory as Adjacency Matrix. Instead we are going to use dynamically allocated memory to store this one.

Many languages support **Vector** or **List** which we can use to store adjacency list. For these, we don't need to specify the size of the **List**. We only need to specify the maximum number of nodes.

The pseudo-code will be:

```
Procedure Adjacency-List(maxN, E):       // maxN denotes the maximum number of nodes
edge[maxN] = Vector()                    // E denotes the number of edges
for i from 1 to E
    input -> x, y                        // Here x, y denotes there is an edge between x, y
    edge[x].push(y)
    edge[y].push(x)
end for
Return edge
```

Since this one is an undirected graph, it there is an edge from **x** to **y**, there is also an edge from **y** to **x**. If it was a directed graph, we'd omit the second one. For weighted graphs, we need to store the cost too. We'll create another **vector** or **list** named **cost[]** to store these. The pseudo-code:

```
Procedure Adjacency-List(maxN, E):
edge[maxN] = Vector()
cost[maxN] = Vector()
for i from 1 to E
    input -> x, y, w
    edge[x].push(y)
    cost[x].push(w)
end for
Return edge, cost
```

From this one, we can easily find out the total number of nodes connected to any node, and what these nodes are.

它比邻接矩阵耗时更少。但如果我们需要判断 u 和 v 之间是否存在边，使用邻接矩阵会更方便。

第9.4节：拓扑排序

拓扑排序，或称拓扑序，是将有向无环图中的顶点排列在一条线上，即在一个列表中，使得所有有向边都从左向右。若图中存在有向环，则无法存在这样的排序，因为你不可能一直向右走却又回到起点。

形式上，设图为G = (V, E)，则其所有顶点的线性排序满足：如果G包含一条从顶点u到顶点v的边(u, v) ∈ E，则u在排序中位于v之前。

需要注意的是，每个有向无环图至少有一个拓扑排序。

已知有算法可以在线性时间内构造任意有向无环图的拓扑排序，举例如下：

1. 调用depth_first_search(G)计算每个顶点v的完成时间v.f
2. 每当一个顶点完成时，将其插入链表的前端
3. 链表中的顶点即为排序结果。

拓扑排序可以在(V + E)时间内完成，因为深度优先搜索算法耗时为(V + E)，且将每个|V|个顶点插入链表前端耗时为Ω(1)（常数时间）。

许多应用使用有向无环图表示事件之间的先后关系。我们使用拓扑排序，以便获得一个顺序，使得每个顶点在其所有后继之前被处理。

图中的顶点可能代表需要执行的任务，边则表示一个任务必须在另一个任务之前完成的约束；拓扑排序即为执行顶点集合V中描述的任务的有效顺序。

问题实例及其解决方案

令一个顶点v表示一个任务(完成时间: int)，即任务(4)表示一个需要4小时完成的任务，边 e表示一个冷却时间(小时: int)，其中冷却时间(3)表示完成任务后需要冷却的时间长度。

令我们的图称为dag（因为它是一个有向无环图），并且它包含5个顶点：

```
A <- dag.add_vertex(任务(4));
B <- dag.add_vertex(任务(5));
C <- dag.add_vertex(任务(3));
D <- dag.add_vertex(任务(2));
E <- dag.add_vertex(任务(7));
```

我们用有向边连接这些顶点，使得图保持无环，

```
// A ---> C ----+
// |      |     |
// v      v     v
// B ---> D --> E
dag.add_edge(A, B, 冷却时间(2));
dag.add_edge(A, C, 冷却时间(2));
dag.add_edge(B, D, 冷却时间(1));
dag.add_edge(C, D, 冷却时间(1));
dag.add_edge(C, E, 冷却时间(1));
dag.add_edge(D, E, 冷却时间(3));
```


It takes less time than Adjacency Matrix. But if we needed to find out if there's an edge between **u** and **v**, it'd have been easier if we kept an adjacency matrix.

Section 9.4: Topological Sort

A topological ordering, or a topological sort, orders the vertices in a directed acyclic graph on a line, i.e. in a list, such that all directed edges go from left to right. Such an ordering cannot exist if the graph contains a directed cycle because there is no way that you can keep going right on a line and still return back to where you started from.

Formally, in a graph `G = (V, E)`, then a linear ordering of all its vertices is such that if `G` contains an edge `(u, v) ∈ E`from vertex `u` to vertex `v` then `u` precedes `v` in the ordering.

It is important to note that each DAG has *at least one* topological sort.

There are known algorithms for constructing a topological ordering of any DAG in linear time, one example is:

1. Call `depth_first_search(G)` to compute finishing times `v.f` for each vertex `v`
2. As each vertex is finished, insert it into the front of a linked list
3. the linked list of vertices, as it is now sorted.

A topological sort can be performed in `□(V + E)` time, since the depth-first search algorithm takes `□(V + E)` time and it takes `Ω(1)` (constant time) to insert each of `|V|` vertices into the front of a linked list.

Many applications use directed acyclic graphs to indicate precedences among events. We use topological sorting so that we get an ordering to process each vertex before any of its successors.

Vertices in a graph may represent tasks to be performed and the edges may represent constraints that one task must be performed before another; a topological ordering is a valid sequence to perform the tasks set of tasks described in `V`.

Problem instance and its solution

Let a vertice `v` describe a `Task(hours_to_complete: int)`, i. e. `Task(4)` describes a `Task` that takes 4 hours to complete, and an edge `e` describe a `Cooldown(hours: int)` such that `Cooldown(3)` describes a duration of time to cool down after a completed task.

Let our graph be called `dag` (since it is a directed acyclic graph), and let it contain 5 vertices:

```
A <- dag.add_vertex(Task(4));
B <- dag.add_vertex(Task(5));
C <- dag.add_vertex(Task(3));
D <- dag.add_vertex(Task(2));
E <- dag.add_vertex(Task(7));
```

where we connect the vertices with directed edges such that the graph is acyclic,

```
// A ---> C ----+
// |      |     |
// v      v     v
// B ---> D --> E
dag.add_edge(A, B, Cooldown(2));
dag.add_edge(A, C, Cooldown(2));
dag.add_edge(B, D, Cooldown(1));
dag.add_edge(C, D, Cooldown(1));
dag.add_edge(C, E, Cooldown(1));
dag.add_edge(D, E, Cooldown(3));
```

那么在A和E之间有三种可能的拓扑排序，

1. A -> B -> D -> E
2. A -> C -> D -> E
3. A -> C -> E

第9.5节：使用深度优先遍历检测有向图中的环

如果在深度优先搜索（DFS）过程中发现了回边，则有向图中存在环。回边是指从某个节点指向其自身或DFS树中某个祖先节点的边。对于不连通的图，我们会得到一个DFS森林，因此必须遍历图中的所有顶点以找到不相交的DFS树。

C++实现：

```
    #include <iostream>
     #include <list>

    using namespace std;

    #define NUM_V   4

    bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
    {
visited[u]=true;
recStack[u]=true;
list<int>::iterator i;
        for(i = graph[u].begin();i!=graph[u].end();++i)
        {
            if(recStack[*i]) //如果顶点 v 在此次深度优先搜索的递归栈中被找到
                return true;
            else if(*i==u) //如果存在从该顶点指向自身的边
                return true;
            else if(!visited[*i])
            {   if(helper(graph, *i, visited, recStack))
                   return true;
            }
        }
recStack[u]=false;
        return false;
    }
    /*
/包装函数对每个未访问的顶点调用辅助函数。辅助函数如果检测到子图（树）中的回边则返回 true，否则返回 false。

*/
    bool isCyclic(list<int> *graph, int V)
    {
      bool visited[V];  //用于跟踪已访问顶点的数组
      bool recStack[V]; //用于跟踪遍历递归栈中顶点的数组。

      for(int i = 0;i<V;i++)
visited[i]=false, recStack[i]=false;  //初始化所有顶点为未访问且未递归


      for(int u = 0; u < V; u++) //迭代检查每个顶点是否已被访问
      {   if(visited[u]==false)
           {  if(helper(graph, u, visited, recStack)) //检查从该顶点的DFS树是否包含环

                return true;
```


then there are three possible topological orderings between `A` and `E`,

1. A -> B -> D -> E
2. A -> C -> D -> E
3. A -> C -> E

Section 9.5: Detecting a cycle in a directed graph using Depth First Traversal

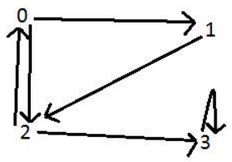
A cycle in a directed graph exists if there's a back edge discovered during a DFS. A back edge is an edge from a node to itself or one of the ancestors in a DFS tree. For a disconnected graph, we get a DFS forest, so you have to iterate through all vertices in the graph to find disjoint DFS trees.

C++ implementation:

```
    #include <iostream>
     #include <list>

    using namespace std;

    #define NUM_V   4

    bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
    {
        visited[u]=true;
        recStack[u]=true;
        list<int>::iterator i;
        for(i = graph[u].begin();i!=graph[u].end();++i)
        {
            if(recStack[*i]) //if vertice v is found in recursion stack of this DFS traversal
                return true;
            else if(*i==u) //if there's an edge from the vertex to itself
                return true;
            else if(!visited[*i])
            {   if(helper(graph, *i, visited, recStack))
                   return true;
            }
        }
        recStack[u]=false;
        return false;
    }
    /*
    /The wrapper function calls helper function on each vertices which have not been visited. Helper
function returns true if it detects a back edge in the subgraph(tree) or false.
*/
    bool isCyclic(list<int> *graph, int V)
    {
      bool visited[V];  //array to track vertices already visited
      bool recStack[V]; //array to track vertices in recursion stack of the traversal.

      for(int i = 0;i<V;i++)
       visited[i]=false, recStack[i]=false;  //initialize all vertices as not visited and not
recursed

      for(int u = 0; u < V; u++) //Iteratively checks if every vertices have been visited
      {   if(visited[u]==false)
           {  if(helper(graph, u, visited, recStack)) //checks if the DFS tree from the vertex
contains a cycle
                return true;
```

```
            }
        }
        return false;
    }
    /*
Driver function
    */
    int main()
    {
list<int>* graph = new list<int>[NUM_V];
        graph[0].push_back(1);
graph[0].push_back(2);
        graph[1].push_back(2);
        graph[2].push_back(0);
        graph[2].push_back(3);
        graph[3].push_back(3);
        bool res = isCyclic(graph, NUM_V);
        cout<<res<<endl;
    }
```

结果：如下所示，图中有三条回边。一条在顶点0和2之间；一条在顶点0、1和2之间；还有一条在顶点3。搜索的时间复杂度是O(V+E)，其中V是顶点数，E是边数。



第9.6节：Thorup算法

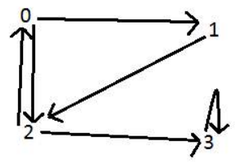
Thorup针对无向图的单源最短路径算法的时间复杂度为O(m)，低于Dijkstra算法。

基本思路如下。（抱歉，我还没尝试实现它，所以可能遗漏一些细节。原论文需要付费，我是从引用它的其他资料中重构的。如果你能验证，请删除这条评论。）

- 有方法可以在O(m)时间内找到生成树（此处不描述）。你需要从最短边“生长”生成树到最长边，生成树在完全生长之前会是一个包含多个连通分量的森林。


```
        }
    }
    return false;
}
/*
Driver function
*/
int main()
{
    list<int>* graph = new list<int>[NUM_V];
    graph[0].push_back(1);
    graph[0].push_back(2);
    graph[1].push_back(2);
    graph[2].push_back(0);
    graph[2].push_back(3);
    graph[3].push_back(3);
    bool res = isCyclic(graph, NUM_V);
    cout<<res<<endl;
}
```

Result: As shown below, there are three back edges in the graph. One between vertex 0 and 2; between vertice 0, 1, and 2; and vertex 3. Time complexity of search is O(V+E) where V is the number of vertices and E is the number of edges.



Section 9.6: Thorup's algorithm

Thorup's algorithm for single source shortest path for undirected graph has the time complexity O(m), lower than Dijkstra.

Basic ideas are the following. (Sorry, I didn't try implementing it yet, so I might miss some minor details. And the original paper is paywalled so I tried to reconstruct it from other sources referencing it. Please remove this comment if you could verify.)

- There are ways to find the spanning tree in O(m) (not described here). You need to "grow" the spanning tree from the shortest edge to the longest, and it would be a forest with several connected components before

完全生长。

- 选择一个整数b（b≥2），只考虑长度限制为b^k的生成森林。合并那些完全相同但k不同的连通分量，并将最小的k称为该连通分量的层级。然后逻辑上将连通分量构造成一棵树。若u是v的最小且完全包含v的不同连通分量，则u是v的父节点。根节点是整个图，叶节点是原图中的单个顶点（层级为负无穷）。该树仍然只有O(n)个节点。

- 维护每个连通分量到源点的距离（类似Dijkstra算法）。包含多个顶点的连通分量的距离是其未扩展子节点的最小距离。将源点的距离设为0，并相应更新祖先节点的距离。

- 考虑以基数 b 计算距离。首次访问第 k 层的节点时，将其子节点放入由该层所有节点共享的桶中（类似桶排序，替代 Dijkstra 算法中的堆），桶的划分依据是距离的第 k 位及更高位数字。每次访问节点时，只考虑其前 b 个桶，访问并移除每个桶，更新当前节点的距离，并使用新的距离将当前节点重新链接到其父节点，等待下一次访问处理后续桶。

- 当访问到叶子节点时，当前距离即为该顶点的最终距离。在原图中展开该节点的所有边，并相应更新距离。

- 重复访问根节点（整个图）直到达到目标节点。

该方法基于这样一个事实：在带有长度限制 l 的生成森林中，两个连通分量之间不存在长度小于 l 的边。因此，从距离 x 开始，你可以只关注一个连通分量，直到达到距离 x + l。你可能会在所有距离更短的顶点都被访问之前访问某些顶点，但这无关紧要，因为已知不会有更短的路径从那些顶点到这里。其他部分类似于桶排序 / MSD 基数排序，当然，这需要 O(m) 的生成树。

fully grown.

- Select an integer b (b>=2) and only consider the spanning forests with length limit b^k. Merge the components which are exactly the same but with different k, and call the minimum k the level of the component. Then logically make components into a tree. u is the parent of v iff u is the smallest component distinct from v that fully contains v. The root is the whole graph and the leaves are single vertices in the original graph (with the level of negative infinity). The tree still has only O(n) nodes.
- Maintain the distance of each component to the source (like in Dijkstra's algorithm). The distance of a component with more than one vertices is the minimum distance of its unexpanded children. Set the distance of the source vertex to 0 and update the ancestors accordingly.
- Consider the distances in base b. When visiting a node in level k the first time, put its children into buckets shared by all nodes of level k (as in bucket sort, replacing the heap in Dijkstra's algorithm) by the digit k and higher of its distance. Each time visiting a node, consider only its first b buckets, visit and remove each of them, update the distance of the current node, and relink the current node to its own parent using the new distance and wait for the next visit for the following buckets.
- When a leaf is visited, the current distance is the final distance of the vertex. Expand all edges from it in the original graph and update the distances accordingly.
- Visit the root node (whole graph) repeatedly until the destination is reached.

It is based on the fact that, there isn't an edge with length less than l between two connected components of the spanning forest with length limitation l, so, starting at distance x, you could focus only on one connected component until you reach the distance x + l. You'll visit some vertices before vertices with shorter distance are all visited, but that doesn't matter because it is known there won't be a shorter path to here from those vertices. Other parts work like the bucket sort / MSD radix sort, and of course, it requires the O(m) spanning tree.

第10章：图遍历

第10.1节：深度优先搜索遍历函数

该函数接受当前节点索引、邻接表（本例中存储为向量的向量）以及用于跟踪节点是否被访问的布尔向量作为参数。

```
void dfs(int node, vector<vector<int>>* graph, vector<bool>* visited) {
    // 检查节点是否已被访问
    if((*visited)[node])
        return;

    // 标记为已访问，避免重复访问同一节点
    (*visited)[node] = true;

    // 在这里执行一些操作
    cout <<node;

    // 以深度优先方式遍历相邻节点
    for(int i = 0; i < (*graph)[node].size(); ++i)
        dfs((*graph)[node][i], graph, visited);
}
```


Chapter 10: Graph Traversals

Section 10.1: Depth First Search traversal function

The function takes the argument of the current node index, adjacency list (stored in vector of vectors in this example), and vector of boolean to keep track of which node has been visited.

```
void dfs(int node, vector<vector<int>>* graph, vector<bool>* visited) {
    // check whether node has been visited before
    if((*visited)[node])
        return;

    // set as visited to avoid visiting the same node twice
    (*visited)[node] = true;

    // perform some action here
    cout << node;

    // traverse to the adjacent nodes in depth-first manner
    for(int i = 0; i < (*graph)[node].size(); ++i)
        dfs((*graph)[node][i], graph, visited);
}
```

第11章：迪杰斯特拉算法

第11.1节：迪杰斯特拉最短路径算法

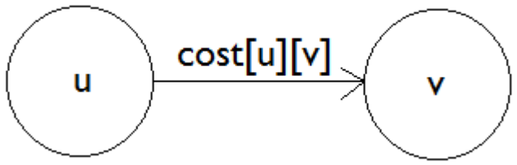
在继续之前，建议先对邻接矩阵和广度优先搜索（BFS）有一个简要了解

迪杰斯特拉算法是单源最短路径算法。它用于寻找图中节点之间的最短路径，图可以表示例如道路网络。该算法由埃兹格·W·迪杰斯特拉（Edsger W.）提出。
迪杰斯特拉于1956年提出，并于三年后发表。

我们可以使用广度优先搜索（BFS）算法找到最短路径。该算法运行良好，但问题是它假设遍历每条路径的代价相同，也就是说每条边的代价相同。迪杰斯特拉算法帮助我们找到在路径代价不相同的情况下的最短路径。

首先我们将看到如何修改BFS来编写迪杰斯特拉算法，然后我们将添加优先队列，使其成为完整的迪杰斯特拉算法。

假设每个节点到源点的距离保存在 d[] 数组中。比如， d[3] 表示从 源点 到 节点3 所需的时间。如果我们不知道距离，就在 d[3] 中存储 无穷大 。同时，设 cost[u][v] 表示边 u-v 的代价。也就是说，从 u 节点到 v 节点需要 cost[u][v] 的时间。



我们需要理解边的松弛。假设从你家，也就是 源点 ，到地点 A 需要 10 分钟，到地点 B 需要 25 分钟。我们有，

```
d[A] = 10
d[B] = 25
```

现在假设从地点 A 到地点 B 需要 7 分钟，这意味着：

```
cost[A][B] = 7
```

那么我们可以从 源点 先到地点 A ，再从地点 A 到地点 B ，
这样总共需要 *10 + 7 = 17* 分钟，而不是 *25* 分钟。所以，

```
d[A] + cost[A][B] < d[B]
```

然后我们更新，

```
d[B] = d[A] + cost[A][B]
```

这就是所谓的松弛。我们从节点 u 到节点 v ，如果 d[u] + cost[u][v] < d[v] ，那么我们就更新 d[v] = d[u] + cost[u][v] 。

在广度优先搜索（BFS）中，我们不需要访问任何节点两次。我们只检查节点是否被访问过。如果未被访问，我们将该节点加入队列，标记为已访问，并将距离加1。在迪杰斯特拉算法中，我们可以将节点加入队列

Chapter 11: Dijkstra's Algorithm

Section 11.1: Dijkstra's Shortest Path Algorithm

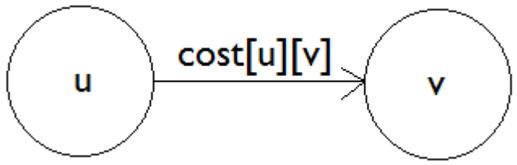
Before proceeding, it is recommended to have a brief idea about Adjacency Matrix and BFS

Dijkstra's algorithm is known as single-source shortest path algorithm. It is used for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by Edsger W. Dijkstra in 1956 and published three years later.

We can find shortest path using Breadth First Search (BFS) searching algorithm. This algorithm works fine, but the problem is, it assumes the cost of traversing each path is same, that means the cost of each edge is same. Dijkstra's algorithm helps us to find the shortest path where the cost of each path is not the same.

At first we will see, how to modify BFS to write Dijkstra's algorithm, then we will add priority queue to make it a complete Dijkstra's algorithm.

Let's say, the distance of each node from the source is kept in **d[]** array. As in, **d[3]** represents that **d[3]** time is taken to reach **node 3** from **source**. If we don't know the distance, we will store *infinity* in **d[3]**. Also, let **cost[u][v]** represent the cost of **u-v**. That means it takes **cost[u][v]** to go from **u** node to **v** node.



We need to understand Edge Relaxation. Let's say, from your house, that is **source**, it takes *10* minutes to go to place **A**. And it takes *25* minutes to go to place **B**. We have,

```
d[A] = 10
d[B] = 25
```

Now let's say it takes *7* minutes to go from place **A** to place **B**, that means:

```
cost[A][B] = 7
```

Then we can go to place **B** from **source** by going to place **A** from **source** and then from place **A**, going to place **B**, which will take *10 + 7 = 17* minutes, instead of *25* minutes. So,

```
d[A] + cost[A][B] < d[B]
```

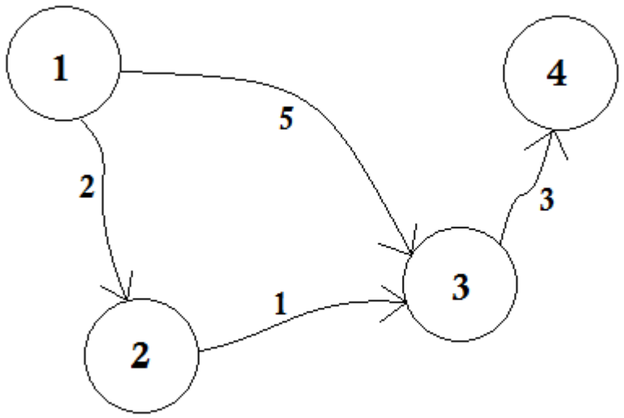
Then we update,

```
d[B] = d[A] + cost[A][B]
```

This is called relaxation. We will go from node **u** to node **v** and if **d[u] + cost[u][v] < d[v]** then we will update **d[v] = d[u] + cost[u][v]**.

In BFS, we didn't need to visit any node twice. We only checked if a node is visited or not. If it was not visited, we pushed the node in queue, marked it as visited and incremented the distance by 1. In Dijkstra, we can push a node

在队列中，而不是用已访问的节点更新它，我们松弛或更新新的边。来看一个例子：



假设，节点1是源点。然后，

```
d[1] = 0
d[2] = d[3] = d[4] = 无穷大 (或一个很大的值)
```

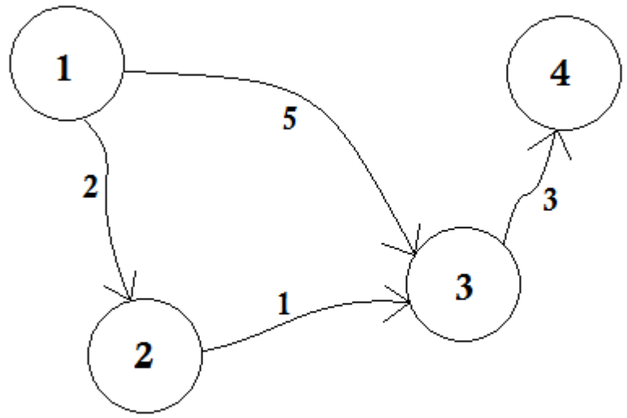
我们将 d[2]，d[3] 和 d[4] 设为无穷大，因为我们还不知道距离。而源点的距离当然是0。现在，我们从源点出发到其他节点，如果能更新它们，就将它们加入队列。
比如说，我们将遍历**边1-2**。因为 **d[1] + 2 < d[2]**，所以将 **d[2] = 2**。同样，我们将遍历**边1-3**这使得 **d[3] = 5**。

我们可以清楚地看到，5不是到达节点3的最短距离。所以像BFS那样只遍历一次节点在这里不起作用。如果我们从节点2通过边2-3到节点3，可以更新 d[3] = d[2] + 1 = 3。所以我们可以看到一个节点可以被多次更新。你问多少次？节点被更新的最大次数是该节点的入度数。

来看访问任意节点多次的伪代码。我们只需修改BFS：

```
过程 BFSmodified(图G, 源点source):
Q = 队列()
distance[] = 无穷大
Q.enqueue(source)
距离[源]=0
当 Q 不为空时
    u <- Q.pop()
    对于图G中从u到v的所有边。adjacentEdges(v) 执行
        如果 distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
        end if
    end for
end while
Return distance
```


in queue and instead of updating it with visited, we *relax* or update the new edge. Let's look at one example:



Let's assume, **Node 1** is the **Source**. Then,

```
d[1] = 0
d[2] = d[3] = d[4] = infinity (or a large value)
```

We set, **d[2], d[3]** and **d[4]** to *infinity* because we don't know the distance yet. And the distance of **source** is of course *0*. Now, we go to other nodes from **source** and if we can update them, then we'll push them in the queue. Say for example, we'll traverse **edge 1-2**. As **d[1] + 2 < d[2]** which will make **d[2] = 2**. Similarly, we'll traverse **edge 1-3** which makes **d[3] = 5**.

We can clearly see that *5* is not the shortest distance we can cross to go to **node 3**. So traversing a node only once, like BFS, doesn't work here. If we go from **node 2** to **node 3** using **edge 2-3**, we can update **d[3] = d[2] + 1 = 3**. So we can see that one node can be updated many times. How many times you ask? The maximum number of times a node can be updated is the number of in-degree of a node.

Let's see the pseudo-code for visiting any node multiple times. We will simply modify BFS:

```
procedure BFSmodified(G, source):
Q = queue()
distance[] = infinity
Q.enqueue(source)
distance[source]=0
while Q is not empty
    u <- Q.pop()
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
        end if
    end for
end while
Return distance
```

这可以用来找到从源节点到所有节点的最短路径。该代码的复杂度不是很好。
原因如下，

在广度优先搜索（BFS）中，当我们从节点 1出发访问所有其他节点时，我们遵循先到先服务的方法。例如，我们在处理节点 2之前，先从源节点访问了**节点 3。如果我们从源节点访问节点 3，则将节点 4更新为5 + 3 = 8。**
当我们再次从**节点 2**更新**节点 3**时，我们需要再次将**节点 4**更新为**3 + 3 = 6**！所以**节点 4**被更新了两次。

Dijkstra提出，与其采用先到先服务的方法，不如先更新最近的节点，这样更新次数会更少。如果我们之前处理了节点2，那么节点 3会先被更新，随后相应地更新节点 4，我们就能轻松得到最短距离！这个想法是从队列中选择距离源节点最近的节点。因此，我们这里将使用优先队列，这样当我们弹出队列时，会带来距离源节点最近的节点u。它是如何做到的？它会检查d[u]的值。

让我们看一下伪代码：

```
过程 dijkstra(图 G, 源节点 source):
Q = 优先队列 priority_queue()
距离 distance[] =无穷大 infinity
Q.enqueue(source)
距离 distance[源节点 source] = 0
当 Q 不为空时
u <- 队列 Q 中距离最小的节点[]
    从队列 Q 中移除 u
     对于图G中从u到v的所有边。adjacentEdges(v) 执行
         如果 distance[u] + cost[u][v] < distance[v]
距离 distance[v] = 距离 distance[u] + 花费 cost[u][v]
Q.入队 enqueue(v)
        结束 if
    end for
end while
Return distance
```

伪代码返回所有其他节点到**source**的距离。如果我们想知道单个节点**v**的距离，当**v**从队列中弹出时，可以直接返回该值。

现在，Dijkstra算法在存在负边时是否有效？如果存在负环，则会发生无限循环，因为每次都会不断降低成本。即使存在负边，Dijkstra算法也无法工作，除非在目标节点弹出后立即返回。但那样就不再是Dijkstra算法了。处理负边/负环需要使用Bellman–Ford算法。

复杂度：

BFS的复杂度是O(log(V+E))，其中V是节点数，E是边数。对于Dijkstra算法，复杂度类似，但优先队列的排序需要O(logV)。因此总复杂度为：O(Vlog(V)+E)下面是一个使用邻接矩阵解决Dijkstra最短路径算法的Java示例

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
```


This can be used to find the shortest path of all node from the source. The complexity of this code is not so good. Here's why,

In BFS, when we go from **node 1** to all other nodes, we follow *first come, first serve* method. For example, we went to **node 3** from **source** before processing **node 2**. If we go to **node 3** from **source**, we update **node 4** as **5 + 3 = 8**. When we again update **node 3** from **node 2**, we need to update **node 4** as **3 + 3 = 6** again! So **node 4** is updated twice.

Dijkstra proposed, instead of going for *First come, first serve* method, if we update the nearest nodes first, then it'll take less updates. If we processed **node 2** before, then **node 3** would have been updated before, and after updating **node 4** accordingly, we'd easily get the shortest distance! The idea is to choose from the queue, the node, that is closest to the **source**. So we will use *Priority Queue* here so that when we pop the queue, it will bring us the closest node **u** from **source**. How will it do that? It'll check the value of **d[u]** with it.

Let's see the pseudo-code:

```
procedure dijkstra(G, source):
Q = priority_queue()
distance[] = infinity
Q.enqueue(source)
distance[source] = 0
while Q is not empty
    u <- nodes in Q with minimum distance[]
    remove u from the Q
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
            Q.enqueue(v)
        end if
    end for
end while
Return distance
```

The pseudo-code returns distance of all other nodes from the **source**. If we want to know distance of a single node **v**, we can simply return the value when **v** is popped from the queue.

Now, does Dijkstra's Algorithm work when there's a negative edge? If there's a negative cycle, then infinity loop will occur, as it will keep reducing the cost every time. Even if there is a negative edge, Dijkstra won't work, unless we return right after the target is popped. But then, it won't be a Dijkstra algorithm. We'll need Bellman–Ford algorithm for processing negative edge/cycle.

Complexity:

The complexity of BFS is **O(log(V+E))** where **V** is the number of nodes and **E** is the number of edges. For Dijkstra, the complexity is similar, but sorting of *Priority Queue* takes **O(logV)**. So the total complexity is: **O(Vlog(V)+E)**

Below is a Java example to solve Dijkstra's Shortest Path Algorithm using Adjacency Matrix

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
```

```
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    void printSolution(int dist[], int n)
    {
        System.out.println("顶点到源点的距离");
        for (int i = 0; i < V; i++)
            System.out.println(i+"   "+dist[i]);
    }

    void dijkstra(int graph[][], int src)
    {

        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

dist[src] = 0;

        for (int count = 0; count < V-1; count++)
        {
            int u = minDistance(dist, sptSet);

sptSet[u] = true;

            for (int v = 0; v < V; v++)

                if (!sptSet[v] && graph[u][v]!=0 &&
                        dist[u] != Integer.MAX_VALUE &&
                        dist[u]+graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

printSolution(dist, V);
    }

    public static void main (String[] args)
    {
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                };
ShortestPath t = new ShortestPath();
```



```
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    void dijkstra(int graph[][], int src)
    {

        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        dist[src] = 0;

        for (int count = 0; count < V-1; count++)
        {
            int u = minDistance(dist, sptSet);

            sptSet[u] = true;

            for (int v = 0; v < V; v++)

                if (!sptSet[v] && graph[u][v]!=0 &&
                        dist[u] != Integer.MAX_VALUE &&
                        dist[u]+graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        printSolution(dist, V);
    }

    public static void main (String[] args)
    {
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                };
        ShortestPath t = new ShortestPath();
```

```
t.dijkstra(graph, 0);
    }
}
```

程序的预期输出是

```
顶点    到源点的距离
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```



```
        t.dijkstra(graph, 0);
    }
}
```

Expected output of the program is

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

第12章：A*路径查找

第12.1节：A*简介

A*（A星）是一种用于寻找从一个节点到另一个节点路径的搜索算法。因此，它可以与广度优先搜索、Dijkstra算法、深度优先搜索或最佳优先搜索进行比较。A*算法因其在图搜索中效率和准确性更优，且不依赖图的预处理而被广泛使用。

A*是最佳优先搜索的一种特化，其中评价函数f以特定方式定义。

f(n) = g(n) + h(n) 是从初始节点到目标节点经过节点 n 的最小代价。

g(n) 是从初始节点到 n 的最小代价。

h(n) 是从 n 到距离 n 最近目标的最小代价。A*是一种启发

式搜索算法，并且只要使用可接受启发式，就总能保证在最短时间内找到最小路径（最小代价路径）。因此，它既是完备的也是最优的。以下动画演示了A*搜索过程——

第12.2节：无障碍迷宫中的A*路径查找

假设我们有以下4乘4的网格：

Chapter 12: A* Pathfinding

Section 12.1: Introduction to A*

A* (A star) is a search algorithm that is used for finding path from one node to another. So it can be compared with Breadth First Search, or Dijkstra's algorithm, or Depth First Search, or Best First Search. A* algorithm is widely used in graph search for being better in efficiency and accuracy, where graph pre-processing is not an option.

A* is a an specialization of Best First Search , in which the function of evaluation *f* is define in a particular way.

f(n) = g(n) + h(n) is the minimum cost since the initial node to the objectives conditioned to go thought node *n*.

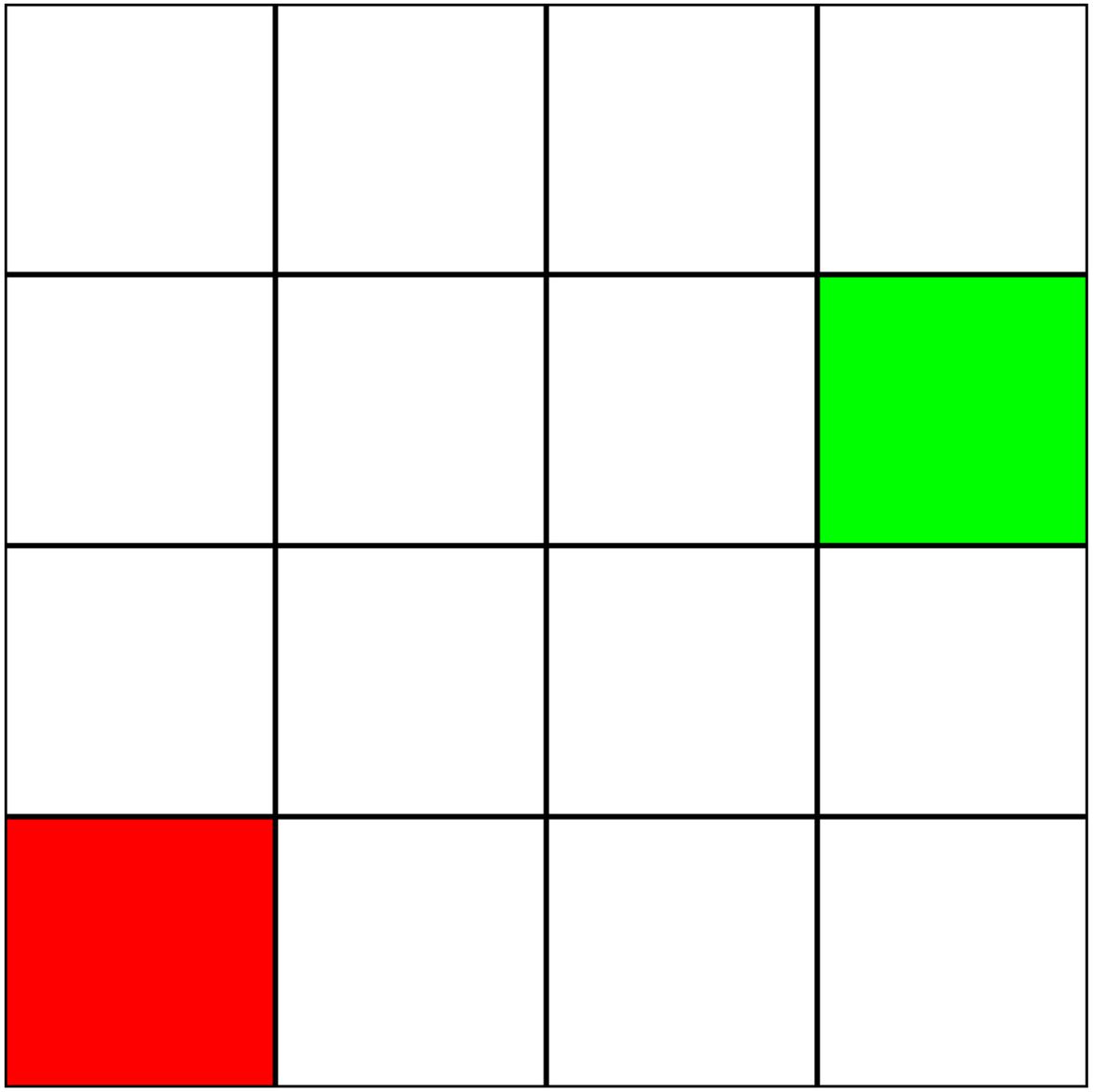
g(n) is the minimum cost from the initial node to *n*.

h(n) is the minimum cost from *n* to the closest objective to *n*

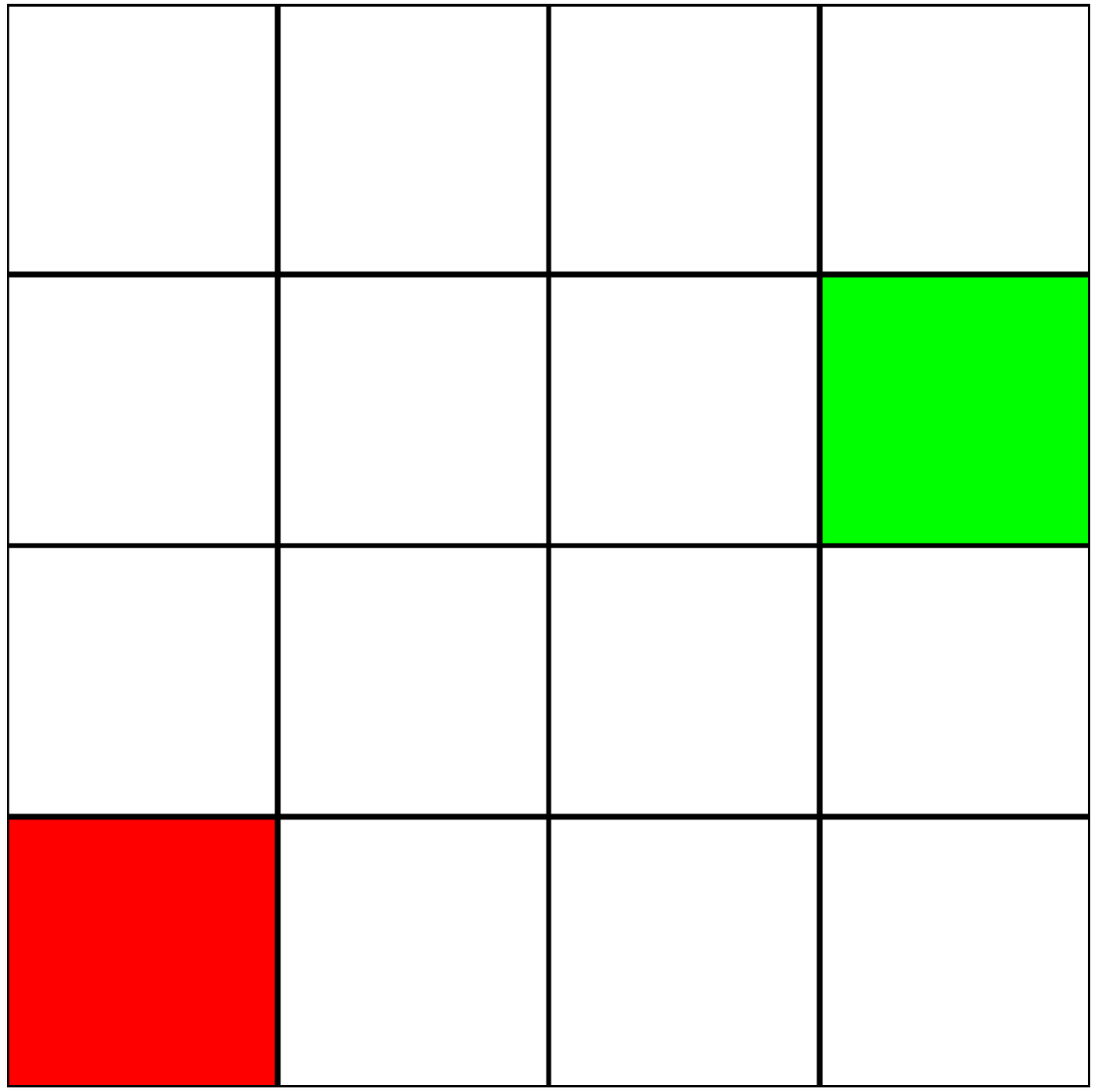
A* is an informed search algorithm and it always guarantees to find the smallest path (path with minimum cost) in the least possible time (if uses admissible heuristic). So it is both *complete* and *optimal*. The following animation demonstrates A* search-

Section 12.2: A* Pathfinding through a maze with no obstacles

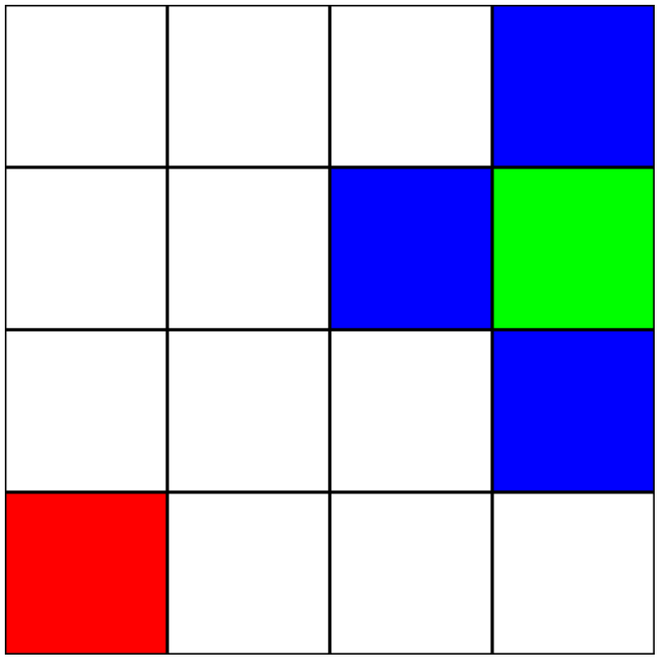
Let's say we have the following 4 by 4 grid:



假设这是一个迷宫。不过这里没有墙壁/障碍物。我们只有一个起点（绿色方块）和一个终点（红色方块）。还假设为了从绿色到红色，我们不能斜着走。所以，从绿色方块开始，看看我们可以移动到哪些方块，并用蓝色标记它们：



Let's assume that this is a *maze*. There are no walls/obstacles, though. We only have a starting point (the green square), and an ending point (the red square). Let's also assume that in order to get from green to red, we cannot move diagonally. So, starting from the green square, let's see which squares we can move to, and highlight them in blue:



为了选择下一个要移动到的方块，我们需要考虑两个启发式指标：

1.“g”值——表示该节点距离绿色方块的距离。
2.“h”值——表示该节点距离红色方块的距离。
3.“f”值——这是“g”值和“h”值的总和。这个最终数值告诉我们应该移动到哪个节点。

为了计算这些启发式指标，我们将使用以下公式：`distance = abs(from.x - to.x) + abs(from.y - to.y)`

这被称为“曼哈顿距离”公式。

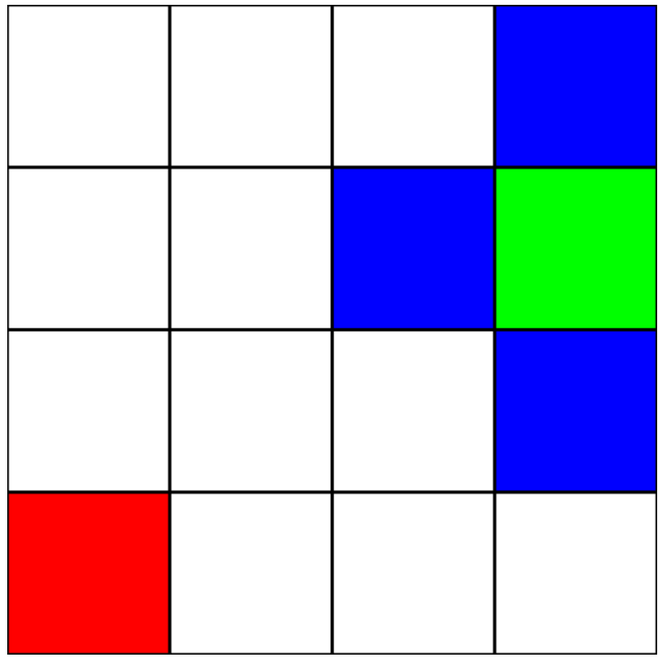
让我们计算绿色方块左边紧邻的蓝色方块的“g”值：`abs(3 - 2) + abs(2 - 2) = 1`

太好了！我们得到了值：1。现在，让我们尝试计算“h”值：`abs(2 - 0) + abs(2 - 0) = 4`

完美。现在，我们来计算“f”值：`1 + 4 = 5`

所以，这个节点的最终值是“5”。

让我们对所有其他蓝色方块做同样的计算。每个方块中心的大数字是“f”值，左上角的数字是“g”值，右上角的数字是“h”值：



In order to choose which square to move to next, we need to take into account 2 heuristics:

1. The "g" value - This is how far away this node is from the green square.
2. The "h" value - This is how far away this node is from the red square.
3. The "f" value - This is the sum of the "g" value and the "h" value. This is the final number which tells us which node to move to.

In order to calculate these heuristics, this is the formula we will use: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

This is known as the "Manhattan Distance" formula.

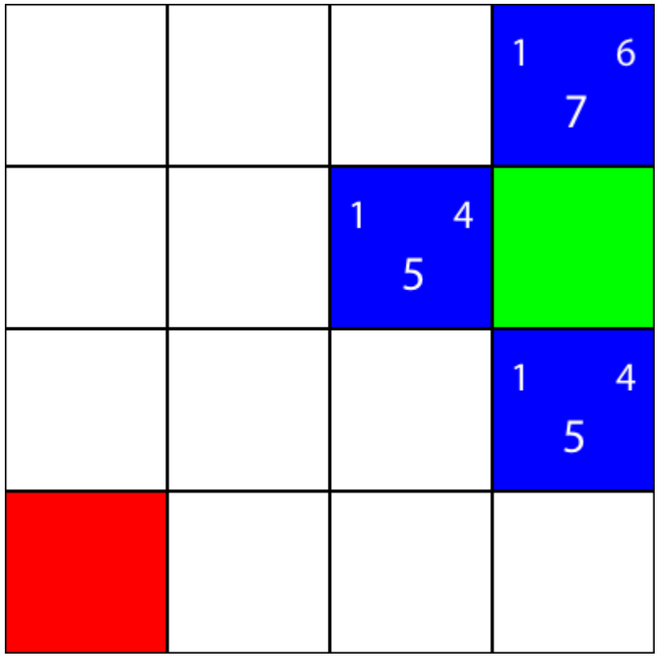
Let's calculate the "g" value for the blue square immediately to the left of the green square: `abs(3 - 2) + abs(2 - 2) = 1`

Great! We've got the value: 1. Now, let's try calculating the "h" value: `abs(2 - 0) + abs(2 - 0) = 4`

Perfect. Now, let's get the "f" value: `1 + 4 = 5`

So, the final value for this node is "5".

Let's do the same for all the other blue squares. The big number in the center of each square is the "f" value, while the number on the top left is the "g" value, and the number on the top right is the "h" value:



我们已经计算了所有蓝色节点的g、h和f值。现在，我们该选择哪个？

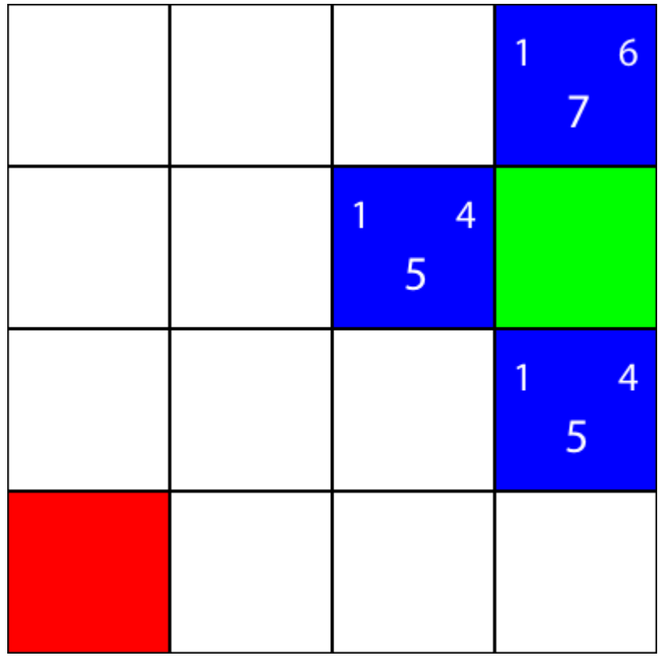
选择f值最低的那个。

不过，在这种情况下，有两个节点的f值相同，都是5。我们该如何选择？

简单来说，可以随机选择一个，或者设定一个优先级。我通常喜欢设定如下优先级：“右 > 上 > 下 > 左”

其中一个f值为5的节点是向“下”移动，另一个是向“左”移动。由于“下”的优先级高于“左”，我们选择向“下”的方块。

我现在将那些我们计算了启发式值但没有移动到的节点标记为橙色，将我们选择的节点标记为青色：



We've calculated the g, h, and f values for all of the blue nodes. Now, which do we pick?

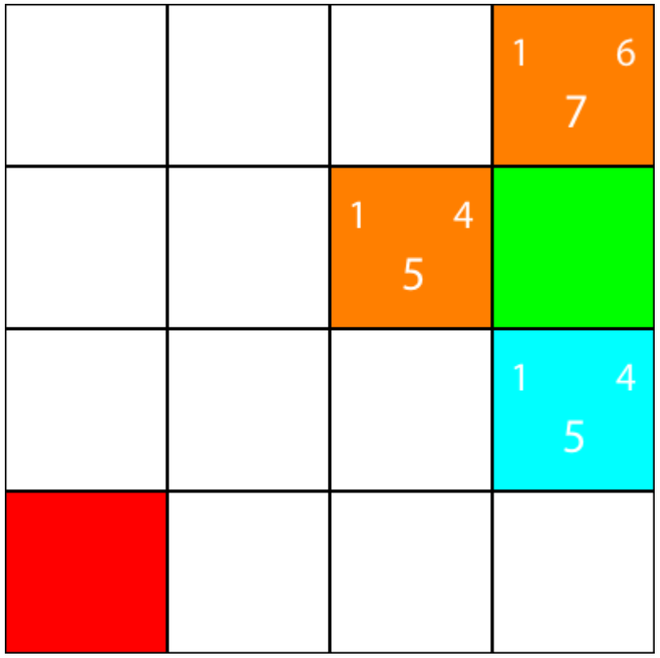
Whichever one has the lowest f value.

However, in this case, we have 2 nodes with the same f value, 5. How do we pick between them?

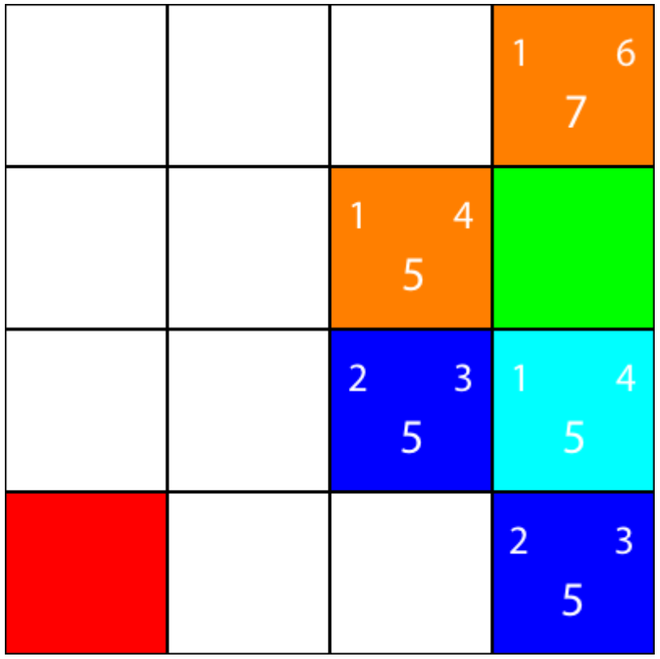
Simply, either choose one at random, or have a priority set. I usually prefer to have a priority like so: "Right > Up > Down > Left"

One of the nodes with the f value of 5 takes us in the "Down" direction, and the other takes us "Left". Since Down is at a higher priority than Left, we choose the square which takes us "Down".

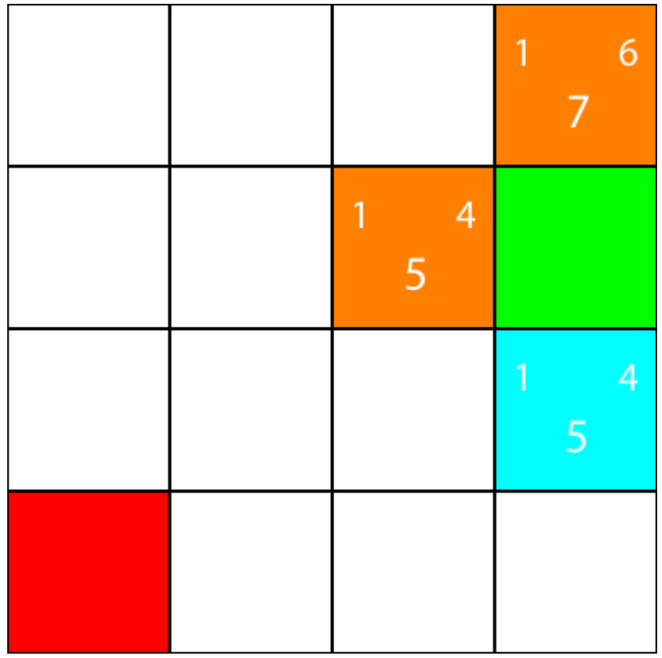
I now mark the nodes which we calculated the heuristics for, but did not move to, as orange, and the node which we chose as cyan:



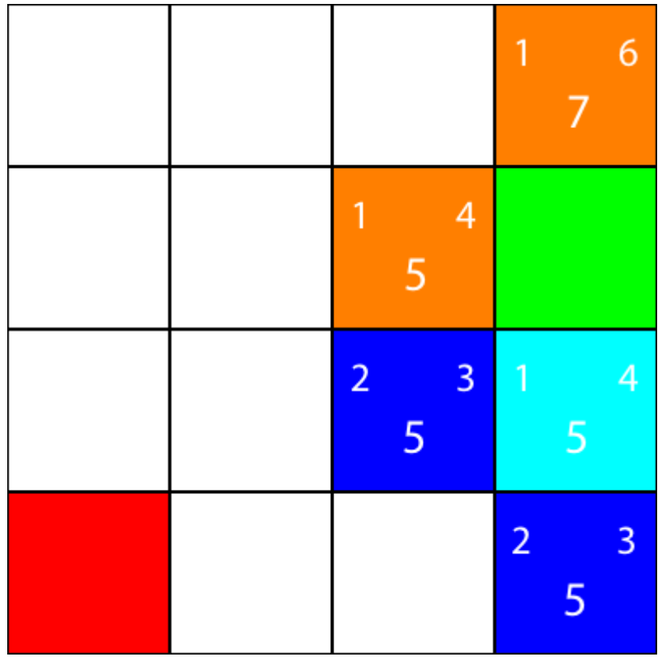
好了，现在让我们计算青色节点周围节点的启发式值：



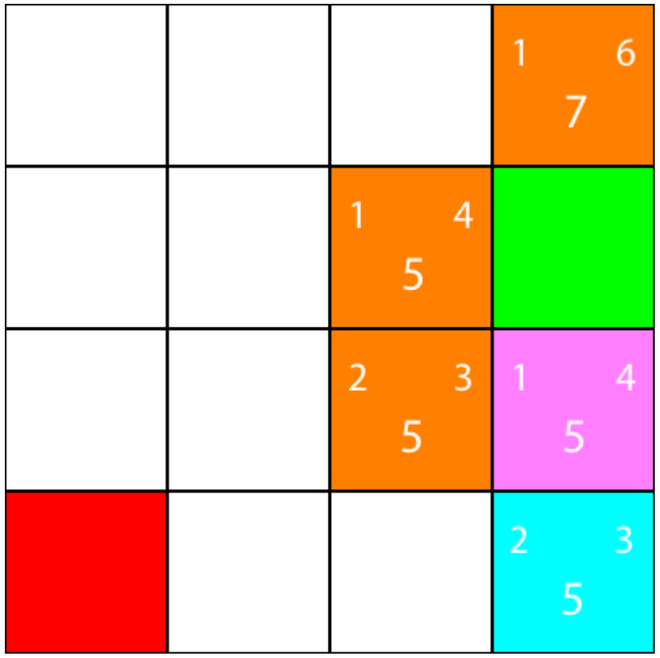
同样，我们选择从青色节点向下的节点，因为所有选项的f值相同：



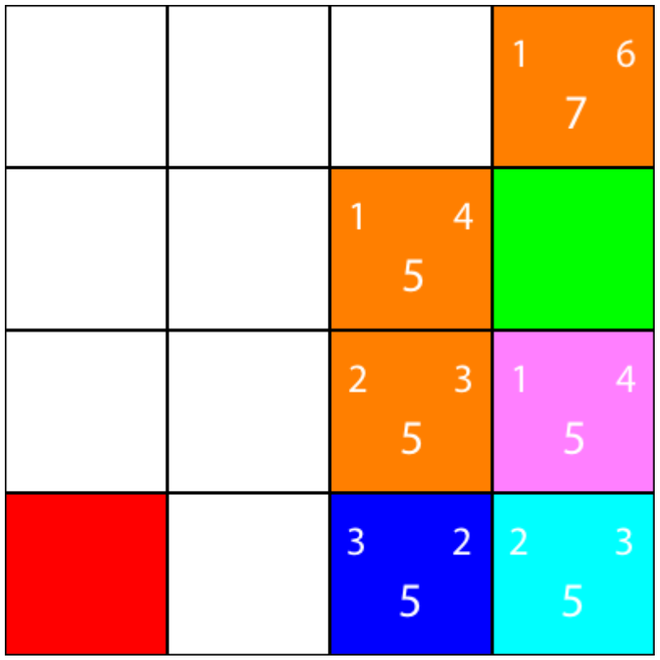
Alright, now let's calculate the same heuristics for the nodes around the cyan node:



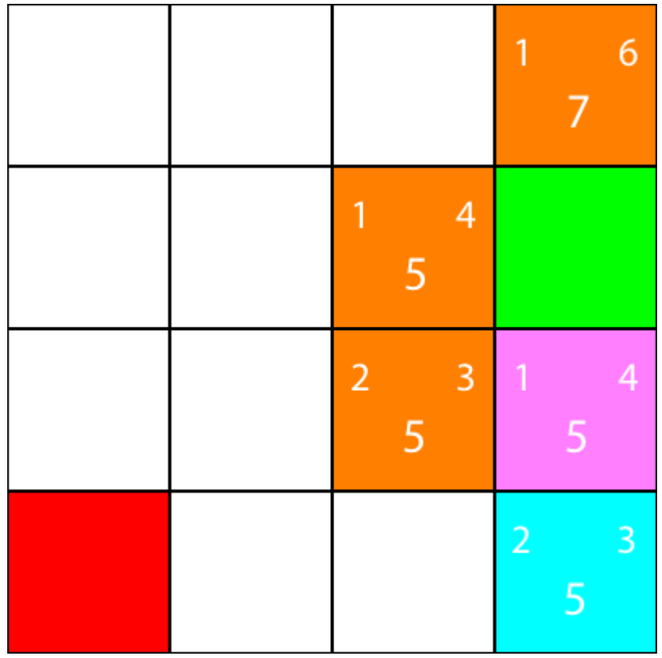
Again, we choose the node going down from the cyan node, as all the options have the same f value:



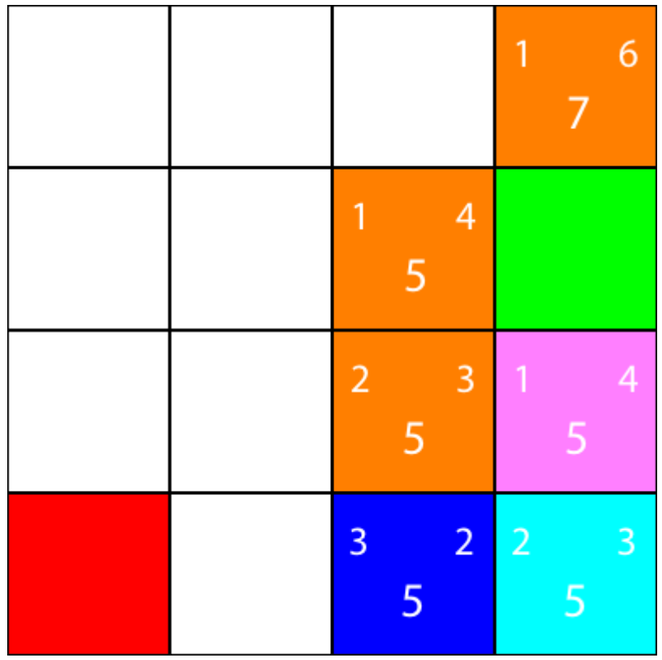
让我们计算青色节点唯一邻居的启发式值：



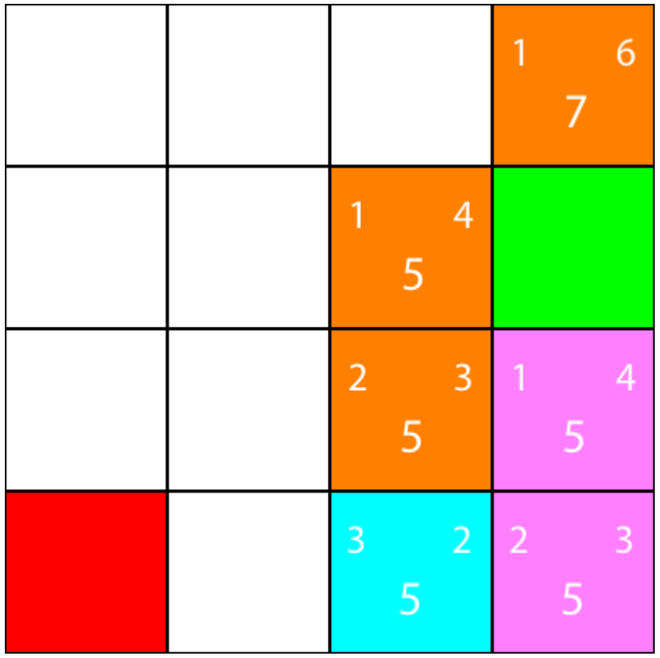
好了，因为我们将遵循之前一直遵循的模式：



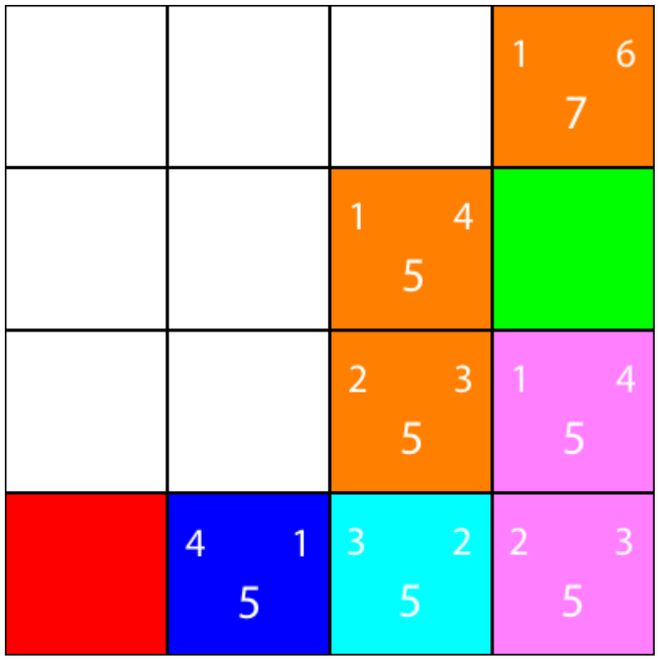
Let's calculate the heuristics for the only neighbour that the cyan node has:



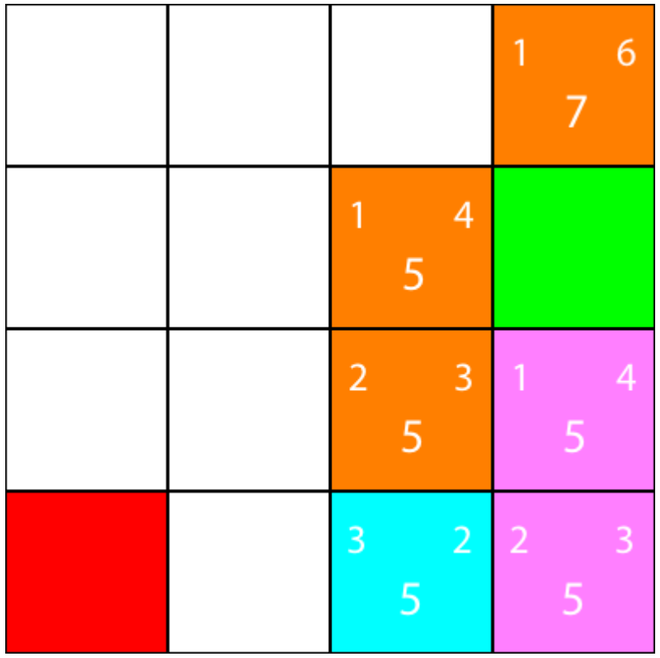
Alright, since we will follow the same pattern we have been following:



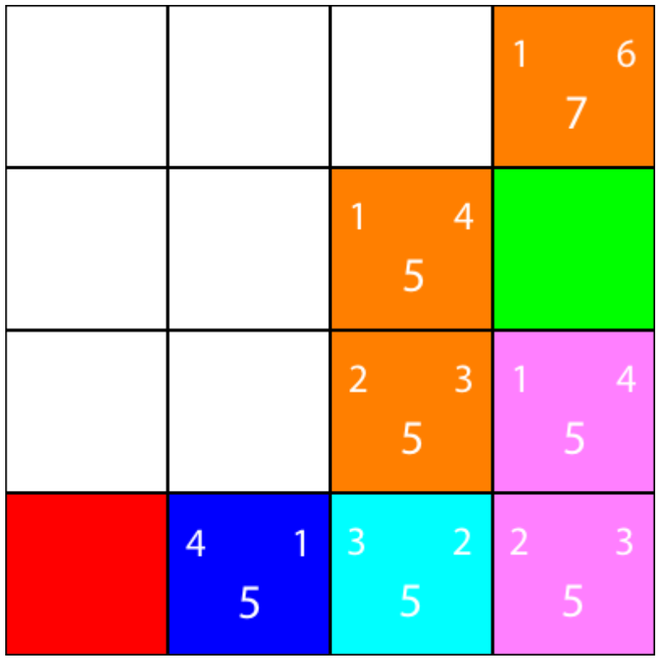
再一次，让我们计算该节点邻居的启发式值：



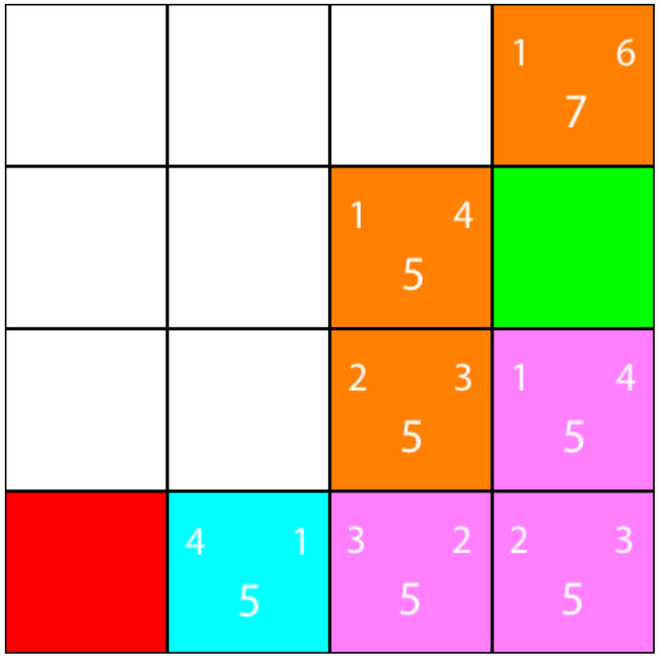
我们过去那里：



Once more, let's calculate the heuristics for the node's neighbour:



Let's move there:

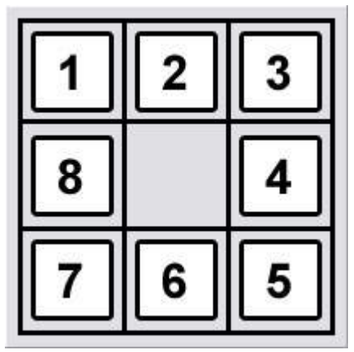


最后，我们可以看到旁边有一个配对方块，所以我们移动过去，完成了。

第12.3节：使用A*算法解决8数码问题

问题定义：

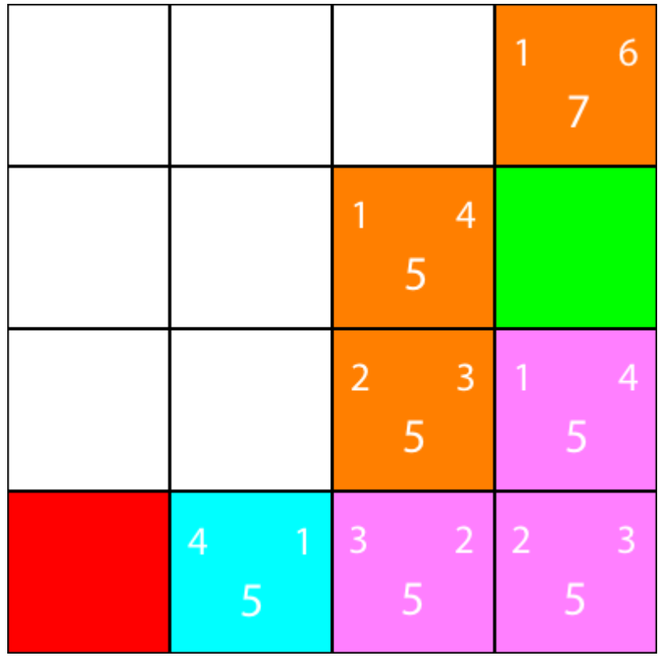
8数码是一种简单的游戏，由一个3×3的网格组成（包含9个方格）。其中一个方格是空的。目标是将周围的方块移动到不同的位置，使数字排列成“目标状态”。



给定8数码游戏的初始状态和要达到的最终状态，找到从初始状态到最终状态的最经济路径。

初始状态：

```
_ 1 3
4 2 5
7 8 6
```

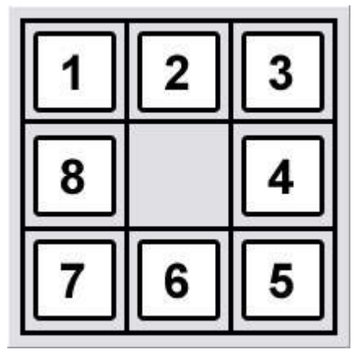



Finally, we can see that we have a *winning square* beside us, so we move there, and we are done.

Section 12.3: Solving 8-puzzle problem using A* algorithm

Problem definition:

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty. The object is to move to squares around into different positions and having the numbers displayed in the "goal state".



Given an initial state of 8-puzzle game and a final state of to be reached, find the most cost-effective path to reach the final state from initial state.

Initial state:

```
_ 1 3
4 2 5
7 8 6
```

> **最终状态：**

```
1 2 3
4 5 6
7 8 _
```

> **假设的启发式函数:**

我们将当前状态与最终状态之间的曼哈顿距离视为该问题陈述的启发式函数。

```
h(n) = | x - p | + | y - q |
其中 x 和 y 是当前状态中的单元格坐标
     p 和 q 是最终状态中的单元格坐标
```

> **总代价函数:**

因此总代价函数 f(n) 表示为，

```
f(n) = g(n) + h(n)，其中 g(n) 是从给定初始状态到达当前状态所需的代价
```

> **示例问题的解答:**

首先我们计算从初始状态到最终状态所需的启发式值。代价函数 g(n) = 0，因为我们处于初始状态

```
h(n) = 8
```

上述数值的获得是因为当前状态中的1比最终状态中的1水平距离多1。同理，2、5、6也是如此。符号_水平距离为2，垂直距离也为2。因此，h(n)的总值为1 + 1 + 1 + 1 + 2 + 2 = 8。总成本函数f(n)等于8 + 0 = 8。

现在，找到了从初始状态可以达到的可能状态，结果是我们可以将_向右或向下移动。

因此，移动后得到的状态为：

```
1_ 3    4 1 3
 4 2 5    _ 2 5
 7 8 6    7 8 6
 (1)      (2)
```

再次使用上述方法计算这些状态的总成本函数，结果分别为6和7。我们选择成本最低的状态，即状态（1）。下一步可能的移动方向是左、右或下。由于之前已经处于左边的状态，我们不会向左移动。因此，我们可以向右或向下移动。

再次找到从状态（1）得到的状态。

```
1 3 _    1 2 3
```


> **Final state**:

```
1 2 3
4 5 6
7 8 _
```

> **Heuristic to be assumed**:

Let us consider the Manhattan distance between the current and final state as the heuristic for this problem statement.

```
h(n) = | x - p | + | y - q |
where x and y are cell co-ordinates in the current state
      p and q are cell co-ordinates in the final state
```

> **Total cost function**:

So the total cost function `f(n)` is given by,

```
f(n) = g(n) + h(n), where g(n) is the cost required to reach the current state from given initial
state
```

> **Solution to example problem**:

First we find the heuristic value required to reach the final state from initial state. The cost function, g(n) = 0, as we are in the initial state

```
h(n) = 8
```

The above value is obtained, as `1` in the current state is 1 horizontal distance away than the `1` in final state. Same goes for `2`, `5`, `6`. `_` is 2 horizontal distance away and 2 vertical distance away. So total value for `h(n)` is 1 + 1 + 1 + 1 + 2 + 2 = 8. Total cost function `f(n)` is equal to 8 + 0 = 8.

Now, the possible states that can be reached from initial state are found and it happens that we can either move `_` to right or downwards.

So states obtained after moving those moves are:

```
1 _ 3    4 1 3
4 2 5    _ 2 5
7 8 6    7 8 6
 (1)      (2)
```

Again the total cost function is computed for these states using the method described above and it turns out to be 6 and 7 respectively. We chose the state with minimum cost which is state (1). The next possible moves can be Left, Right or Down. We won't move Left as we were previously in that state. So, we can move Right or Down.

Again we find the states obtained from (1).

```
1 3 _    1 2 3
```

```
4 2 5    4 _ 5
7 8 6    7 8 6
 (3)      (4)
```

(3) 导致代价函数等于 6，(4) 导致代价函数等于 4。我们还将考虑之前得到的 (2)，其代价函数等于 7。从中选择最小值为 (4)。接下来可能的移动方向是左、右或下。
我们得到状态：

```
 1 2 3    1 2 3    1 2 3
_ 4 5     4 5 _     4 8 5
7 8 6     7 8 6     7 _ 6
 (5)       (6)       (7)
```

我们得到 (5)、(6) 和 (7) 的代价分别为 5、2 和 4。同时，我们有之前的状态 (3) 和 (2)，代价分别为 6 和 7。我们选择代价最小的状态，即 (6)。接下来可能的移动方向是上和下，显然向下将引导我们到达最终状态，使启发函数值等于 0。


```
4 2 5    4 _ 5
7 8 6    7 8 6
 (3)      (4)
```

(3) leads to cost function equal to 6 and (4) leads to 4. Also, we will consider (2) obtained before which has cost function equal to 7. Choosing minimum from them leads to (4). Next possible moves can be Left or Right or Down. We get states:

```
1 2 3    1 2 3    1 2 3
_ 4 5    4 5 _    4 8 5
7 8 6    7 8 6    7 _ 6
 (5)      (6)      (7)
```

We get costs equal to 5, 2 and 4 for (5), (6) and (7) respectively. Also, we have previous states (3) and (2) with 6 and 7 respectively. We chose minimum cost state which is (6). Next possible moves are Up, and Down and clearly Down will lead us to final state leading to heuristic function value equal to 0.

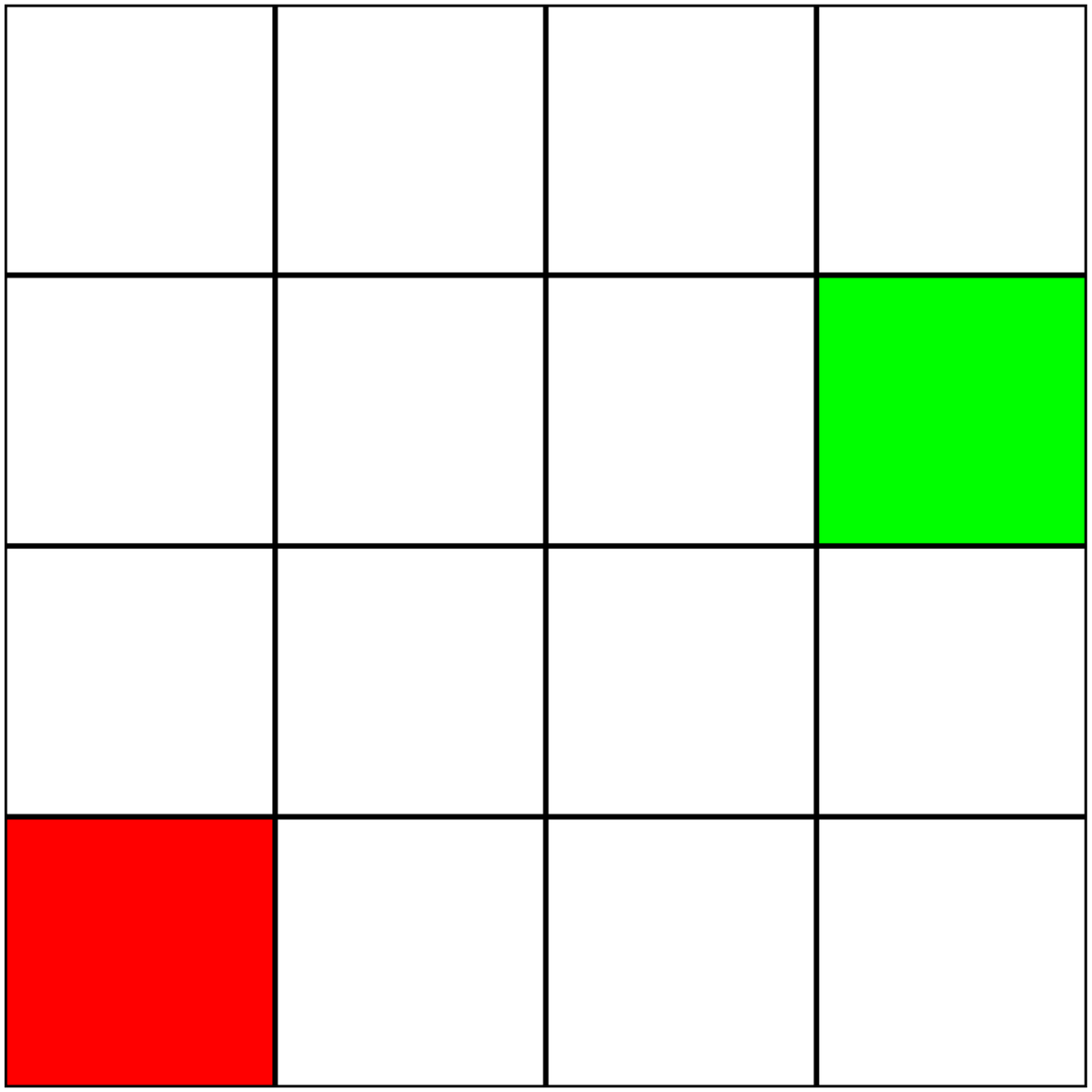
第13章：A* 寻路算法

本章节将重点介绍 A* 寻路算法，它的使用方法以及为何有效。

致未来贡献者的说明：我添加了一个无障碍物的 4x4 网格上的 A* 寻路示例。仍然需要一个带障碍物的示例。

第13.1节：A*寻路的简单示例：一个没有障碍物的迷宫

假设我们有以下4乘4的网格：



假设这是一个迷宫。不过这里没有墙壁/障碍物。我们只有一个起点（绿色方块）和一个终点（红色方块）。还假设为了从绿色到红色，我们不能

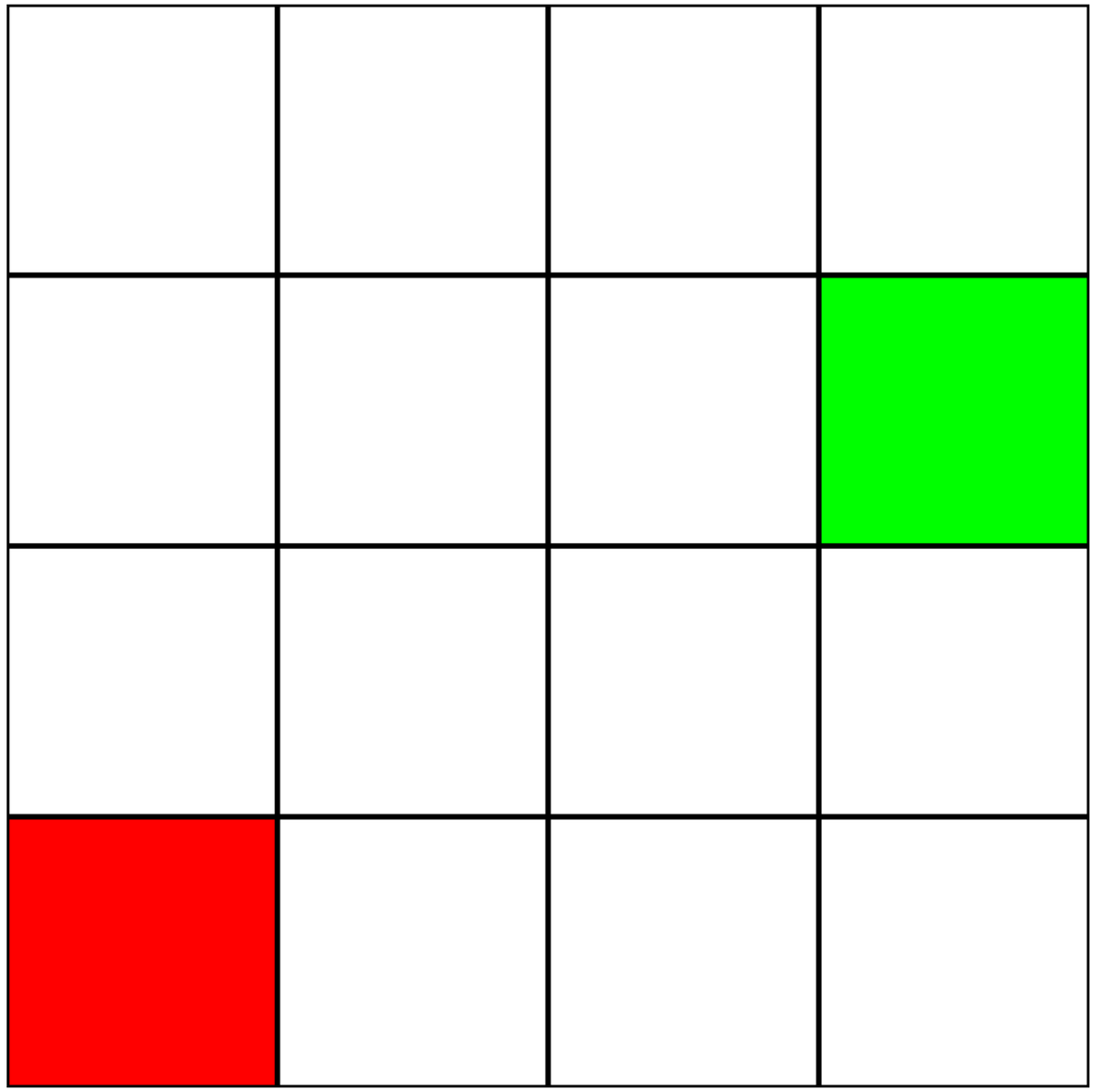
Chapter 13: A* Pathfinding Algorithm

This topic is going to focus on the A* Pathfinding algorithm, how it's used, and why it works.

Note to future contributors: I have added an example for A* Pathfinding without any obstacles, on a 4x4 grid. An example with obstacles is still needed.

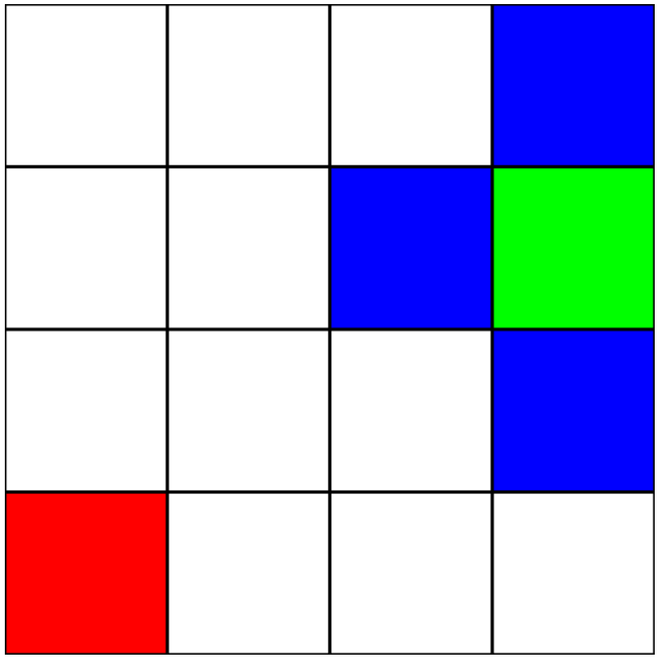
Section 13.1: Simple Example of A* Pathfinding: A maze with no obstacles

Let's say we have the following 4 by 4 grid:



Let's assume that this is a *maze*. There are no walls/obstacles, though. We only have a starting point (the green square), and an ending point (the red square). Let's also assume that in order to get from green to red, we cannot

斜向移动。所以，从绿色方块开始，看看我们可以移动到哪些方块，并用蓝色标出它们：



为了选择下一个要移动到的方块，我们需要考虑两个启发式指标：

1.“g”值——表示该节点距离绿色方块的距离。
2.“h”值——表示该节点距离红色方块的距离。
3.“f”值——这是“g”值和“h”值的总和。这个最终数值告诉我们应该移动到哪个节点。

为了计算这些启发式指标，我们将使用以下公式：`distance = abs(from.x - to.x) + abs(from.y - to.y)`

这被称为“曼哈顿距离”公式。

让我们计算绿色方块左边紧邻的蓝色方块的“g”值：`abs(3 - 2) + abs(2 - 2) = 1`

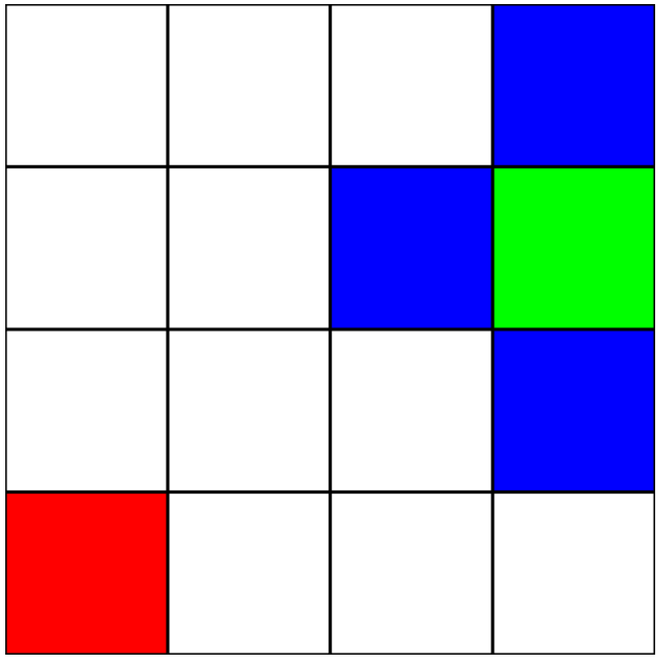
太好了！我们得到了值：1。现在，让我们尝试计算“h”值：`abs(2 - 0) + abs(2 - 0) = 4`

完美。现在，我们来计算“f”值：`1 + 4 = 5`

所以，这个节点的最终值是“5”。

让我们对所有其他蓝色方块做同样的计算。每个方块中心的大数字是“f”值，左上角的数字是“g”值，右上角的数字是“h”值：

move diagonally. So, starting from the green square, let's see which squares we can move to, and highlight them in blue:



In order to choose which square to move to next, we need to take into account 2 heuristics:

1. The "g" value - This is how far away this node is from the green square.
2. The "h" value - This is how far away this node is from the red square.
3. The "f" value - This is the sum of the "g" value and the "h" value. This is the final number which tells us which node to move to.

In order to calculate these heuristics, this is the formula we will use: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

This is known as the "Manhattan Distance" formula.

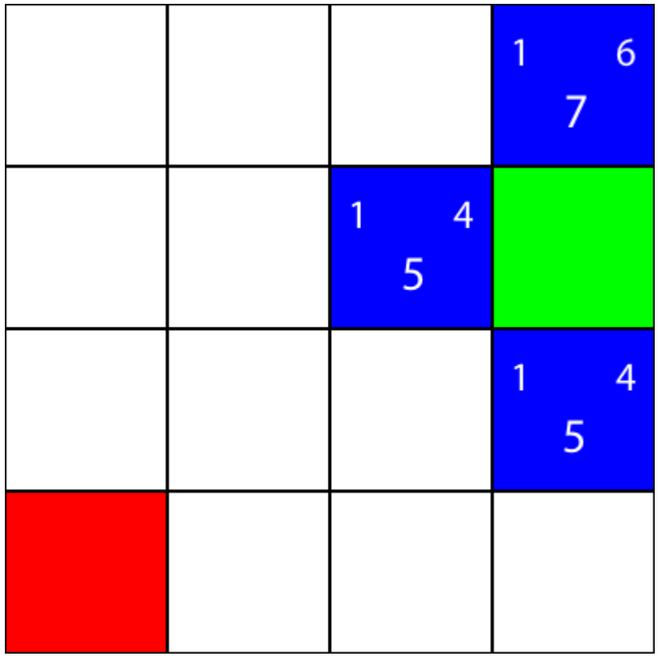
Let's calculate the "g" value for the blue square immediately to the left of the green square: `abs(3 - 2) + abs(2 - 2) = 1`

Great! We've got the value: 1. Now, let's try calculating the "h" value: `abs(2 - 0) + abs(2 - 0) = 4`

Perfect. Now, let's get the "f" value: `1 + 4 = 5`

So, the final value for this node is "5".

Let's do the same for all the other blue squares. The big number in the center of each square is the "f" value, while the number on the top left is the "g" value, and the number on the top right is the "h" value:



我们已经计算了所有蓝色节点的g、h和f值。现在，我们该选择哪个？

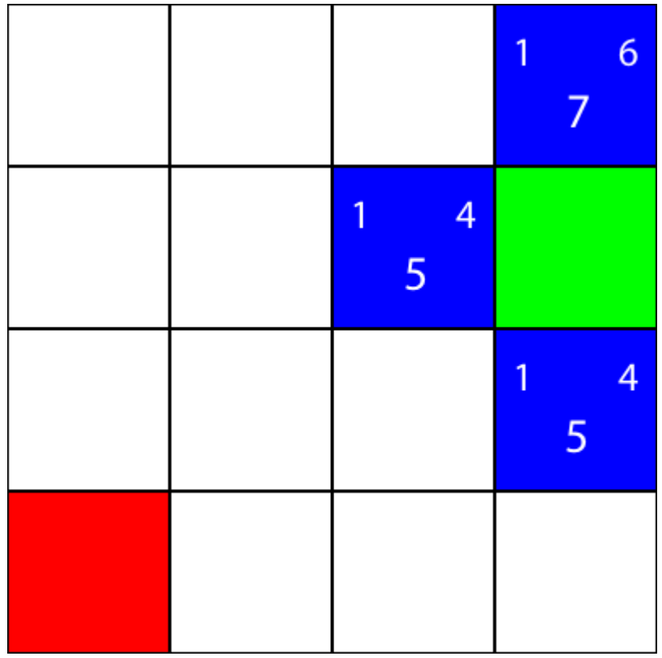
选择f值最低的那个。

不过，在这种情况下，有两个节点的f值相同，都是5。我们该如何选择？

简单来说，可以随机选择一个，或者设定一个优先级。我通常喜欢设定如下优先级：“右 > 上 > 下 > 左”

其中一个f值为5的节点是向“下”移动，另一个是向“左”移动。由于“下”的优先级高于“左”，我们选择向“下”的方块。

我现在将那些我们计算了启发式值但没有移动到的节点标记为橙色，将我们选择的节点标记为青色：



We've calculated the g, h, and f values for all of the blue nodes. Now, which do we pick?

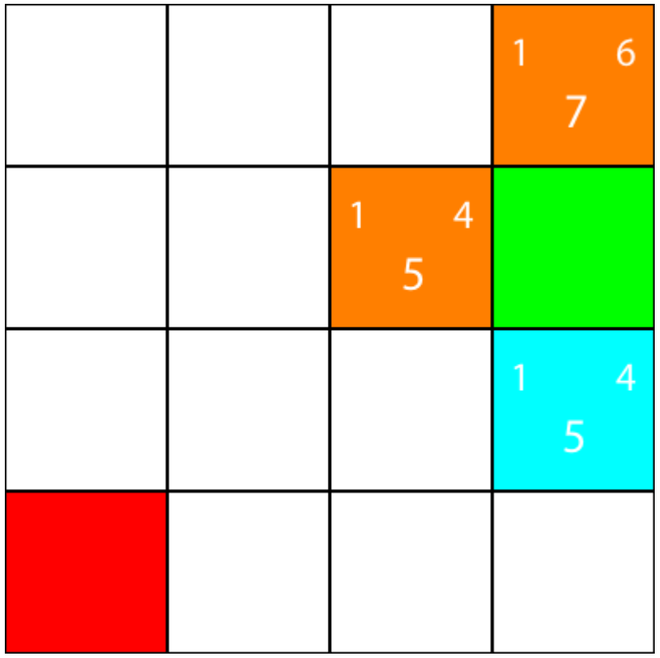
Whichever one has the lowest f value.

However, in this case, we have 2 nodes with the same f value, 5. How do we pick between them?

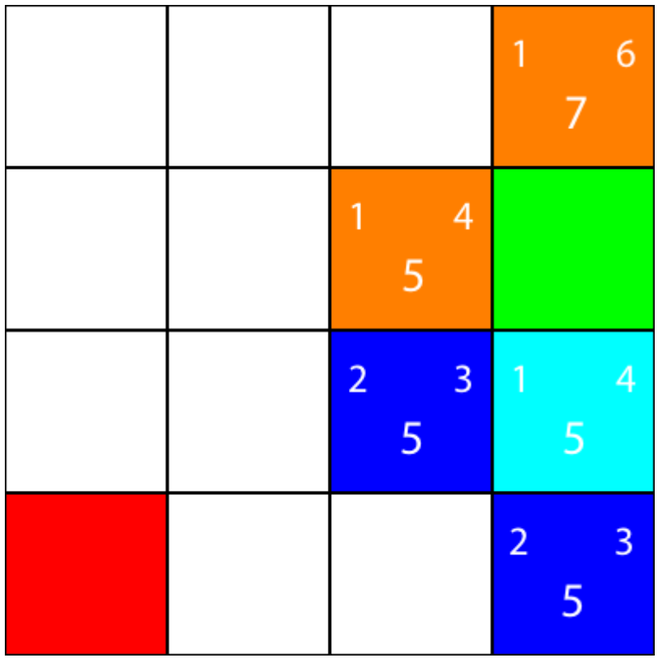
Simply, either choose one at random, or have a priority set. I usually prefer to have a priority like so: "Right > Up > Down > Left"

One of the nodes with the f value of 5 takes us in the "Down" direction, and the other takes us "Left". Since Down is at a higher priority than Left, we choose the square which takes us "Down".

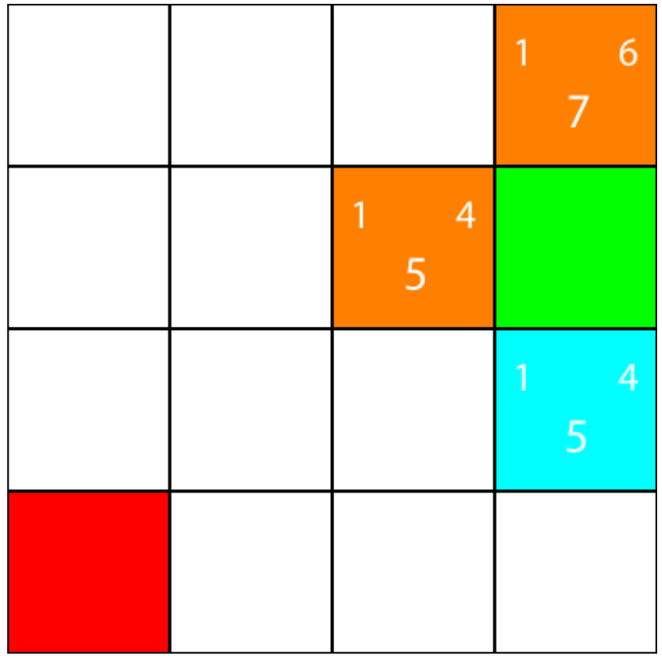
I now mark the nodes which we calculated the heuristics for, but did not move to, as orange, and the node which we chose as cyan:



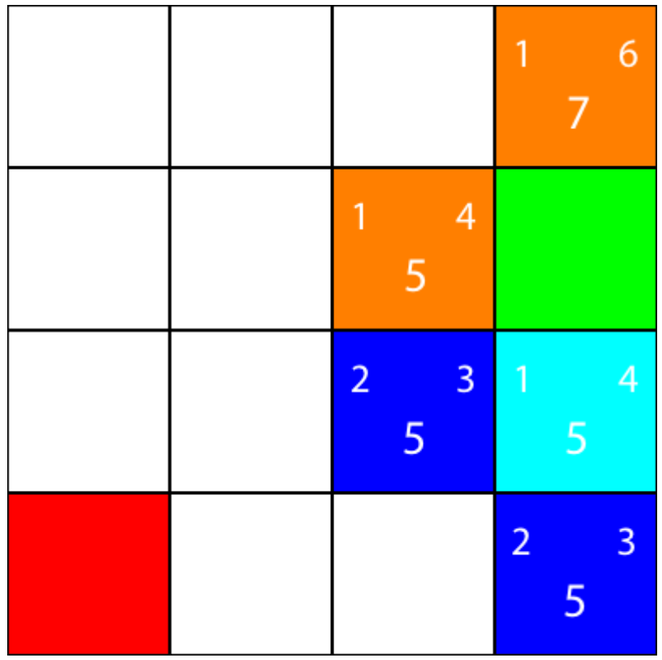
好了，现在让我们计算青色节点周围节点的启发式值：



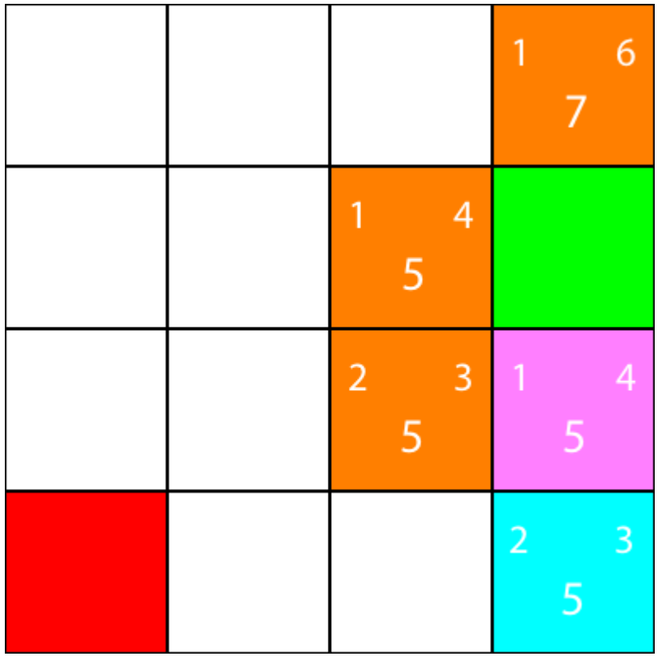
同样，我们选择从青色节点向下的节点，因为所有选项的f值相同：



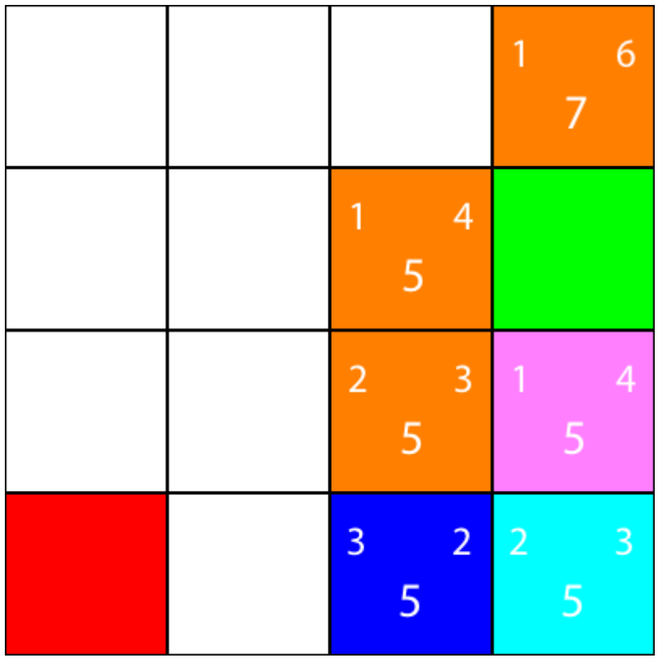
Alright, now let's calculate the same heuristics for the nodes around the cyan node:



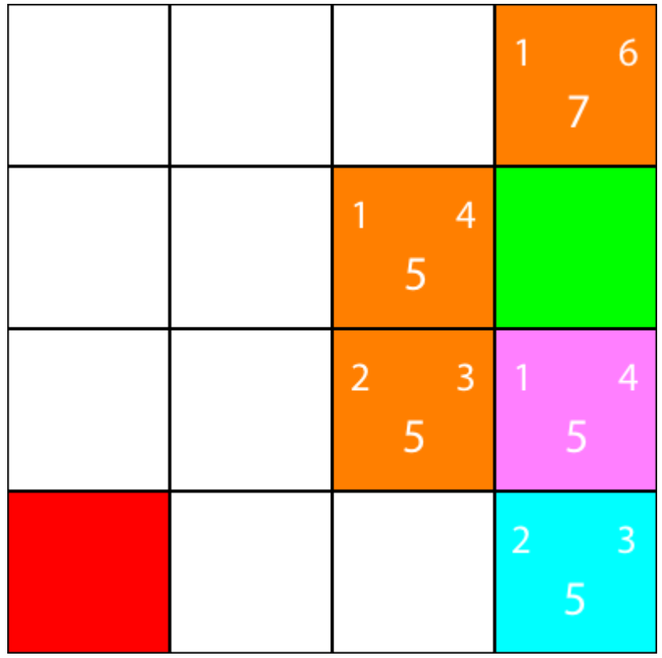
Again, we choose the node going down from the cyan node, as all the options have the same f value:



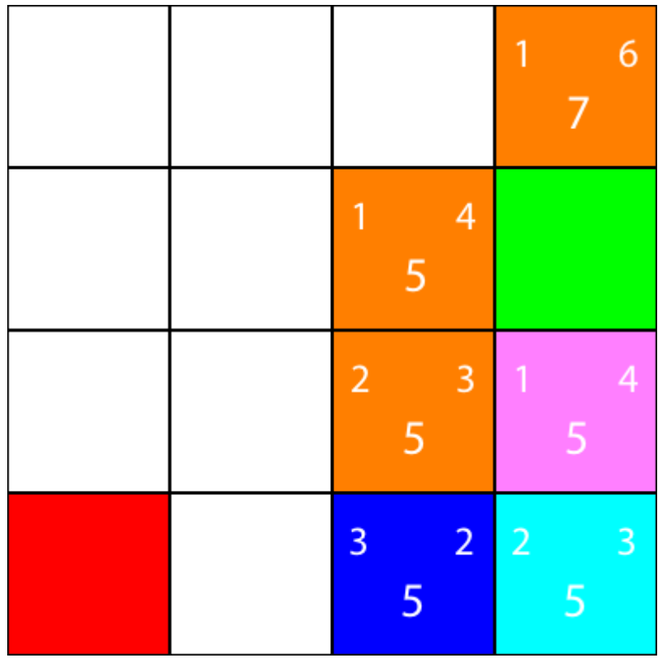
让我们计算青色节点唯一邻居的启发式值：



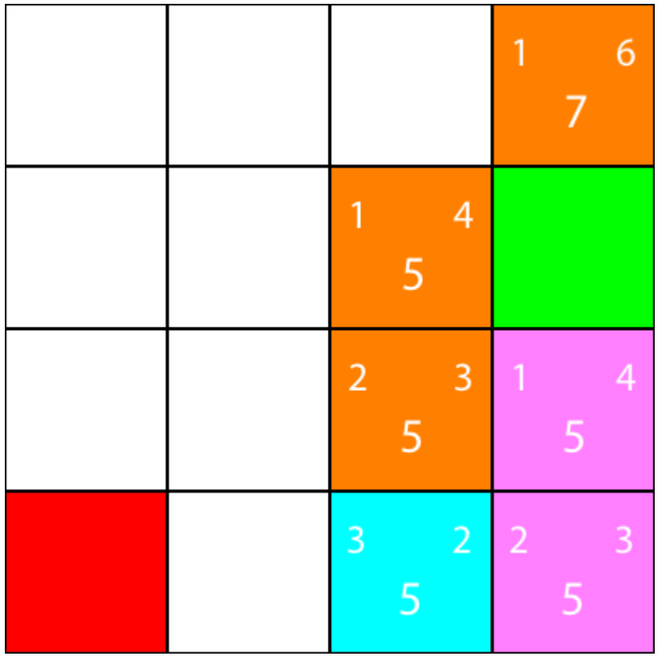
好了，因为我们将遵循之前一直遵循的模式：



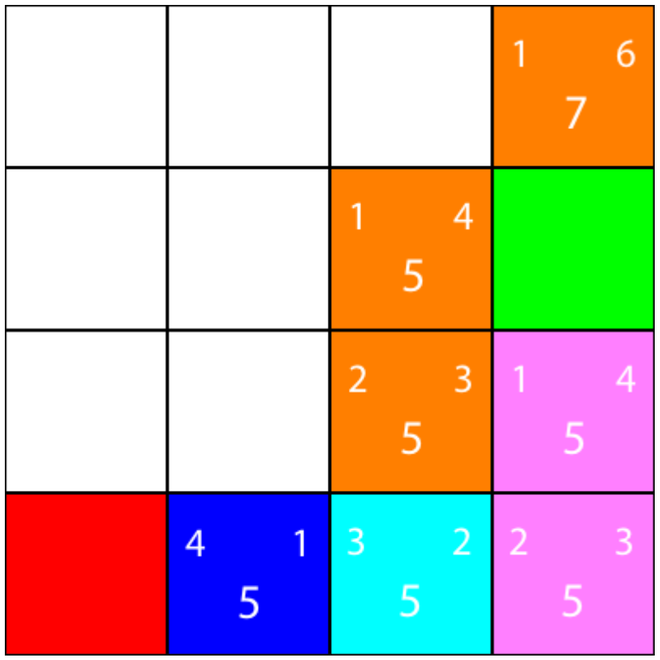
Let's calculate the heuristics for the only neighbour that the cyan node has:



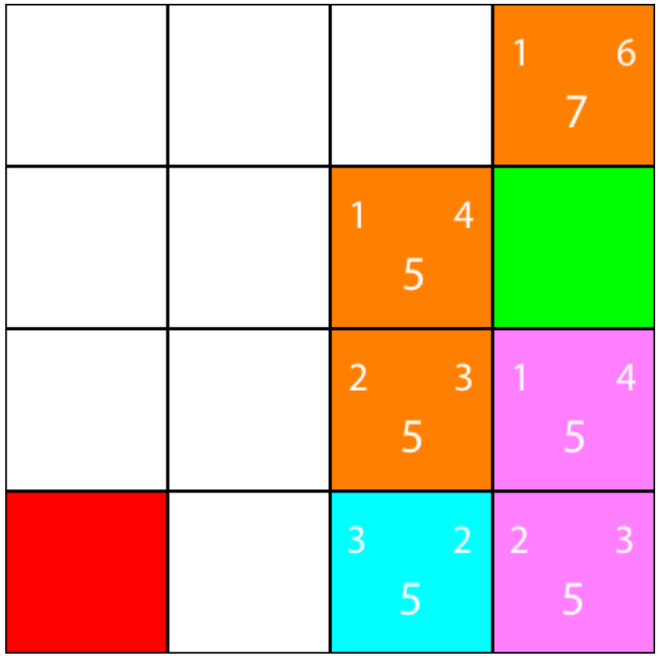
Alright, since we will follow the same pattern we have been following:



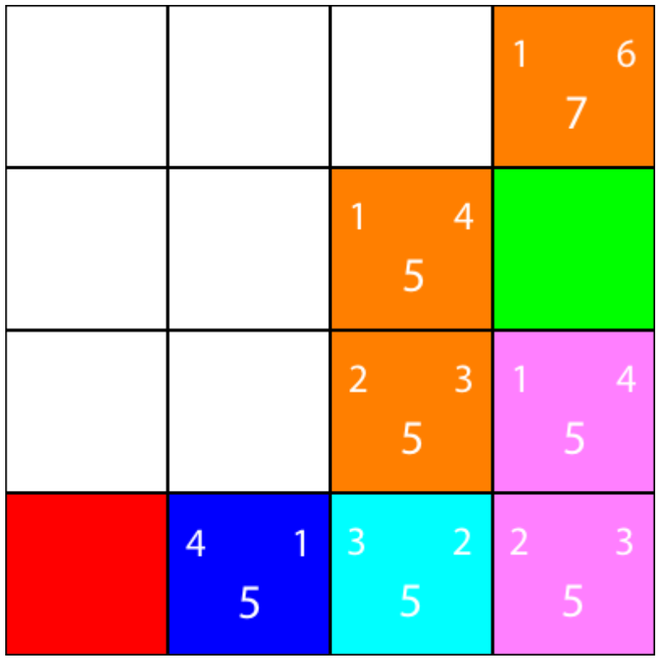
再一次，让我们计算该节点邻居的启发式值：



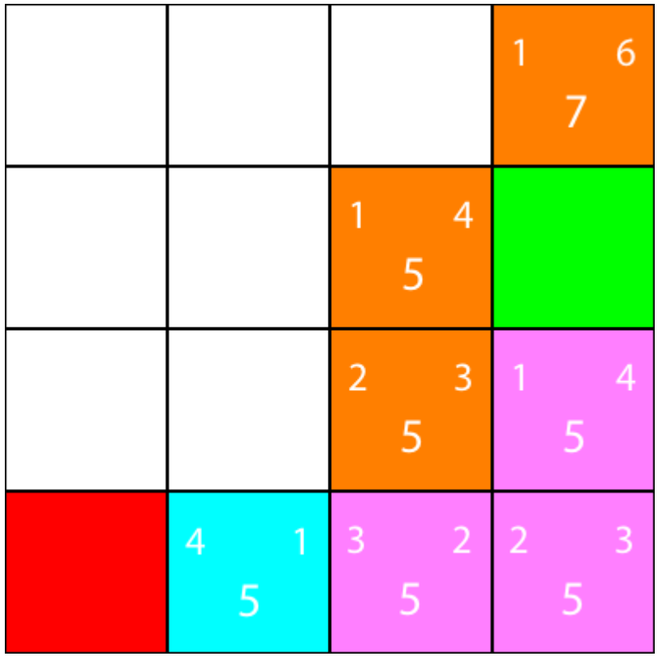
我们过去那里：



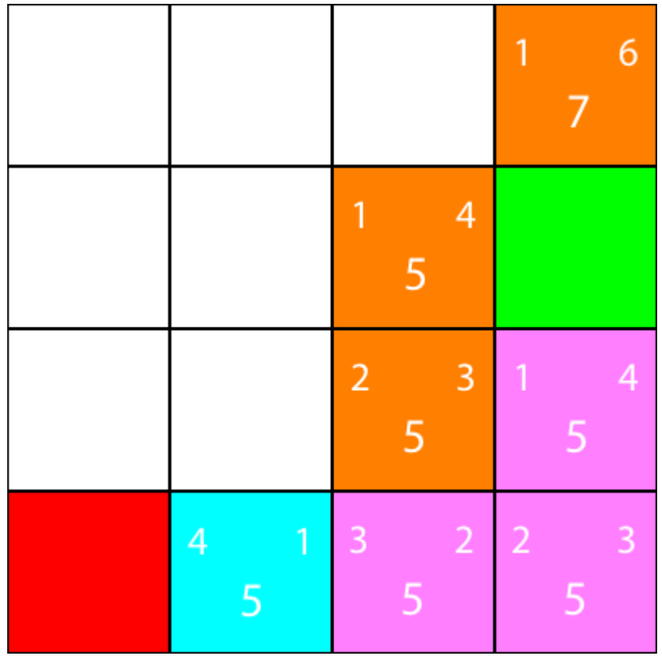
Once more, let's calculate the heuristics for the node's neighbour:



Let's move there:



最后，我们可以看到旁边有一个配对方块，所以我们移动过去，完成了。



Finally, we can see that we have a *winning square* beside us, so we move there, and we are done.

第14章：动态规划

动态规划是一个广泛使用的概念，通常用于优化。它指的是通过递归的方式将一个复杂的问题分解为更简单的子问题，通常采用自底向上的方法。有两个关键属性是问题必须具备的，动态规划才能适用："最优子结构"和"重叠子问题"。为了实现优化，动态规划使用了一个称为记忆化的概念。

第14.1节：编辑距离

问题描述是，如果给定两个字符串str1和str2，那么对str1执行多少最少的操作数才能将其转换为str2。

Java实现

```
public class EditDistance {

public static void main(String[] args) {
    // TODO 自动生成的方法存根
    String str1 = "march";
    String str2 = "cart";

EditDistance ed = new EditDistance();
    System.out.println(ed.getMinConversions(str1, str2));
}

public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
dp[i][j] = j;
            else if(j==0)
dp[i][j] = i;
            否则如果(str1.charAt(i-1) == str2.charAt(j-1))
              dp[i][j] = dp[i-1][j-1];
            否则{
dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    返回 dp[str1.length()][str2.length()];
}
```

}

输出

3

第14.2节：加权作业调度算法

加权作业调度算法也可以称为加权活动选择算法。

问题是，给定若干作业及其开始时间和结束时间，以及完成该作业时获得的利润，
在不允许两个作业同时执行的情况下，最大能获得多少利润？

Chapter 14: Dynamic Programming

Dynamic programming is a widely used concept and its often used for optimization. It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner usually a bottom-up approach. There are two key attributes that a problem must have in order for dynamic programming to be applicable "Optimal substructure" and "Overlapping sub-problems". To achieve its optimization, dynamic programming uses a concept called memoization

Section 14.1: Edit Distance

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2.

Implementation in Java

```
public class EditDistance {

public static void main(String[] args) {
    // TODO Auto-generated method stub
    String str1 = "march";
    String str2 = "cart";

    EditDistance ed = new EditDistance();
    System.out.println(ed.getMinConversions(str1, str2));
}

public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

}

Output

3

Section 14.2: Weighted Job Scheduling Algorithm

Weighted Job Scheduling Algorithm can also be denoted as Weighted Activity Selection Algorithm.

The problem is, given certain jobs with their start time and end time, and a profit you make when you finish the job, what is the maximum profit you can make given no two jobs can be executed in parallel?

这看起来像是使用贪心算法的活动选择问题，但有一个额外的变化。也就是说，我们不是最大化完成的作业数量，而是专注于获得最大利润。完成的作业数量在这里并不重要。

让我们来看一个例子：

名称	A	B	C	D	E	F
(开始时间, 结束时间)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
利润	6	4	2	5	11	5

这些任务用名称、开始和结束时间以及利润表示。经过几次迭代后，我们可以发现如果执行任务A和任务E，就能获得最大利润17。现在，如何用算法找出这个结果呢？

我们首先做的是按任务的结束时间进行非递减排序。为什么要这样做？这是因为如果我们选择一个完成时间较短的任务，就会为选择其他任务留下更多时间。我们有：

名称	D	A	F	B	E	C
(开始时间, 结束时间)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
利润	5	6	5	4	11	2

我们将有一个额外的临时数组Acc_Prof，大小为n（这里，n表示任务总数）。该数组将包含执行任务所获得的最大累计利润。不理解？请继续看。我们将用每个任务的利润初始化该数组的值。也就是说，Acc_Prof[i]最初将保存执行第i个任务的利润。

累计利润	5	6	5	4	11	2

现在我们用位置2表示i，用位置1表示j。我们的策略是让j从1迭代到i-1，每次迭代后，i加1，直到i变为n+1。

```
                              j       i
```

名称	D	A	F	B	E	C
(开始时间, 结束时间)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
利润	5	6	5	4	11	2
累计利润	5	6	5	4	11	2

This one looks like Activity Selection using Greedy Algorithm, but there's an added twist. That is, instead of maximizing the number of jobs finished, we focus on making the maximum profit. The number of jobs performed doesn't matter here.

Let's look at an example:

Name	A	B	C	D	E	F
(Start Time, Finish Time)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
Profit	6	4	2	5	11	5

The jobs are denoted with a name, their start and finishing time and profit. After a few iterations, we can find out if we perform **Job-A** and **Job-E**, we can get the maximum profit of 17. Now how to find this out using an algorithm?

The first thing we do is sort the jobs by their finishing time in non-decreasing order. Why do we do this? It's because if we select a job that takes less time to finish, then we leave the most amount of time for choosing other jobs. We have:

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2

We'll have an additional temporary array **Acc_Prof** of size **n** (Here, **n** denotes the total number of jobs). This will contain the maximum accumulated profit of performing the jobs. Don't get it? Wait and watch. We'll initialize the values of the array with the profit of each jobs. That means, **Acc_Prof[i]** will at first hold the profit of performing **i-th** job.

Acc_Prof	5	6	5	4	11	2

Now let's denote **position 2** with **i**, and **position 1** will be denoted with **j**. Our strategy will be to iterate **j** from **1** to **i-1** and after each iteration, we will increment **i** by 1, until **i** becomes **n+1**.

```
                              j       i
```

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

我们检查Job[i]和Job[j]是否重叠，也就是说，如果Job[j]的完成时间大于Job[i]的开始时间，那么这两个任务不能同时完成。然而，如果它们不重叠，我们将检查Acc_Prof[j] + Profit[i] > Acc_Prof[i]。如果成立，我们将更新Acc_Prof[i] = Acc_Prof[j] + Profit[i]。即：

```
if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
   endif
endif
```

这里Acc_Prof[j] + Profit[i]表示完成这两个任务的累计利润。让我们用我们的例子来验证：

这里Job[j]与Job[i]重叠。因此这两个任务不能同时完成。由于我们的j等于 i-1，我们将 i的值增加到 i+1，即3。并且将j = 1。

```
j                 i

+---------------------+--------+--------+--------+--------+--------+--------+
|         名称        |   D    |   A    |   F    |   B    |   E    |   C    |
+---------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)  | (1,3)  | (2,5)  | (4,6)  | (6,7)  | (5,8)  | (7,9)  |
+---------------------+--------+--------+--------+--------+--------+--------+
|         利润        |   5    |   6    |   5    |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
|       累计利润      |   5    |   6    |   5    |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
```

现在Job[j]和Job[i]不重叠。选择这两个任务可以获得的总利润是：Acc_Prof[j] + Profit[i] = 5 + 5 = 10，**这大于Acc_Prof[i]。因此我们更新Acc_Prof[i] = 10。我们也将j加1。**

我们得到，

```
j        i

+---------------------+--------+--------+--------+--------+--------+--------+
|         名称        |   D    |   A    |   F    |   B    |   E    |   C    |
+---------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)  | (1,3)  | (2,5)  | (4,6)  | (6,7)  | (5,8)  | (7,9)  |
+---------------------+--------+--------+--------+--------+--------+--------+
|         利润        |   5    |   6    |   5    |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
|       累计利润      |   5    |   6    |   10   |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
```

这里，Job[j] 与 Job[i] 重叠，并且 j 也等于 i-1。因此我们将 i 增加1，并令 j = 1。我们得到，

```
j                         i

+---------------------+--------+--------+--------+--------+--------+--------+
|         名称        |   D    |   A    |   F    |   B    |   E    |   C    |
+---------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)  | (1,3)  | (2,5)  | (4,6)  | (6,7)  | (5,8)  | (7,9)  |
+---------------------+--------+--------+--------+--------+--------+--------+
|         利润        |   5    |   6    |   5    |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
|       累计利润      |   5    |   6    |   10   |   4    |   11   |   2    |
+---------------------+--------+--------+--------+--------+--------+--------+
```


We check if **Job[i]** and **Job[j]** overlap, that is, if the **finish time** of **Job[j]** is greater than **Job[i]**'s start time, then these two jobs can't be done together. However, if they don't overlap, we'll check if **Acc_Prof[j] + Profit[i] > Acc_Prof[i]**. If this is the case, we will update **Acc_Prof[i] = Acc_Prof[j] + Profit[i]**. That is:

```
if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
    endif
endif
```

Here **Acc_Prof[j] + Profit[i]** represents the accumulated profit of doing these two jobs toegther. Let's check it for our example:

Here **Job[j]** overlaps with **Job[i]**. So these to can't be done together. Since our **j** is equal to **i-1**, we increment the value of **i** to **i+1** that is **3**. And we make **j = 1**.

```
                                j                 i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Now **Job[j]** and **Job[i]** don't overlap. The total amount of profit we can make by picking these two jobs is: **Acc_Prof[j] + Profit[i] = 5 + 5 = 10** which is greater than **Acc_Prof[i]**. So we update **Acc_Prof[i] = 10**. We also increment **j** by 1. We get,

```
                                      j        i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Here, **Job[j]** overlaps with **Job[i]** and **j** is also equal to **i-1**. So we increment **i** by 1, and make **j = 1**. We get,

```
                                j                         i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

```
+------------------------+---------+---------+---------+---------+---------+---------+
```

现在，**作业[j]**和**作业[i]**不重叠，我们得到累计利润**5 + 4 = 9**，大于**累计利润[i]**。我们更新**累计利润[i] = 9**，并将**j**加1。

```
j                  i

+----------------------+--------+--------+--------+--------+--------+--------+
|          名称          |   D    |   A    |   F    |   B    |   E    |   C    |
+----------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)      |  (1,3) |  (2,5) |  (4,6) |  (6,7) |  (5,8) |  (7,9) |
+----------------------+--------+--------+--------+--------+--------+--------+
|         利润           |   5    |   6    |   5    |   4    |   11   |   2    |
+----------------------+--------+--------+--------+--------+--------+--------+
|       累计利润          |   5    |   6    |   10   |   9    |   11   |   2    |
+----------------------+--------+--------+--------+--------+--------+--------+
```

再次**作业[j]**和**作业[i]**不重叠。累计利润为：**6 + 4 = 10**，大于**累计利润[i]**。我们再次更新**累计利润[i] = 10**。我们将**j**加1。得到：

```
j        i

+----------------------+--------+--------+--------+--------+--------+--------+
|          名称          |   D    |   A    |   F    |   B    |   E    |   C    |
+----------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)      |  (1,3) |  (2,5) |  (4,6) |  (6,7) |  (5,8) |  (7,9) |
+----------------------+--------+--------+--------+--------+--------+--------+
|         利润           |   5    |   6    |   5    |   4    |   11   |   2    |
+----------------------+--------+--------+--------+--------+--------+--------+
|       累计利润          |   5    |   6    |   10   |   10   |   11   |   2    |
+----------------------+--------+--------+--------+--------+--------+--------+
```

如果我们继续这个过程，使用i遍历整个表格后，我们的表格最终将如下所示：

```
+----------------------+--------+--------+--------+--------+--------+--------+
|          名称          |   D    |   A    |   F    |   B    |   E    |   C    |
+----------------------+--------+--------+--------+--------+--------+--------+
|(开始时间, 结束时间)      |  (1,3) |  (2,5) |  (4,6) |  (6,7) |  (5,8) |  (7,9) |
+----------------------+--------+--------+--------+--------+--------+--------+
|         利润           |   5    |   6    |   5    |   4    |   11   |   2    |
+----------------------+--------+--------+--------+--------+--------+--------+
|       累计利润          |   5    |   6    |   10   |   14   |   17   |   8    |
+----------------------+--------+--------+--------+--------+--------+--------+
```

* 为了使文档更简短，省略了一些步骤。

如果我们遍历数组Acc_Prof，就可以找到最大利润为17！伪代码如下：

```
过程 加权作业调度(作业)
根据完成时间对作业进行非递减排序
对于 i 从2到 n
    对于 j 从1到 i-1
        if Job[j].finish_time <= Job[i].start_time
            如果 Acc_Prof[j] + 利润[i] > Acc_Prof[i]
                Acc_Prof[i] = Acc_Prof[j] + 利润[i]
```



```
+------------------------+---------+---------+---------+---------+---------+---------+
```

Now, **Job[j]** and **Job[i]** don't overlap, we get the accumulated profit **5 + 4 = 9**, which is greater than **Acc_Prof[i]**. We update **Acc_Prof[i] = 9** and increment **j** by 1.

```
                                    j                  i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    9    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

Again **Job[j]** and **Job[i]** don't overlap. The accumulated profit is: **6 + 4 = 10**, which is greater than **Acc_Prof[i]**. We again update **Acc_Prof[i] = 10**. We increment **j** by 1. We get:

```
                                          j         i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    10   |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

If we continue this process, after iterating through the whole table using **i**, our table will finally look like:

```
+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    14   |    17   |    8    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

* A few steps have been skipped to make the document shorter.

If we iterate through the array **Acc_Prof**, we can find out the maximum profit to be **17**! The pseudo-code:

```
Procedure WeightedJobScheduling(Job)
sort Job according to finish time in non-decreasing order
for i -> 2 to n
    for j -> 1 to i-1
        if Job[j].finish_time <= Job[i].start_time
            if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
                Acc_Prof[i] = Acc_Prof[j] + Profit[i]
```

```
结束如果
        结束如果
    结束循环
结束循环

最大利润 = 0
对于 i 从1到 n
    如果 最大利润 < Acc_Prof[i]
        最大利润 = Acc_Prof[i]
返回 最大利润
```

填充Acc_Prof数组的复杂度是O(n2)。数组遍历的复杂度是O(n)。所以该算法的总复杂度是O(n2)。

现在，如果我们想找出为了获得最大利润而执行了哪些工作，我们需要反向遍历数组，如果Acc_Prof与maxProfit匹配，我们将把该工作的name压入stack，并从maxProfit中减去该工作的Profit。我们将一直这样做，直到maxProfit > 0或到达Acc_Prof数组的起点。伪代码如下：

```
过程 FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
对于 i 从 n 到 0 且 maxProfit > 0
    如果 maxProfit 等于 Acc_Prof[i]
S.push(Job[i].name
        maxProfit = maxProfit - Job[i].profit
    结束条件
结束循环
```

该过程的时间复杂度为：O(n)。

需要记住的一点是，如果存在多个作业调度方案能给出最大利润，该过程只能找到其中一个作业调度方案。

第14.3节：最长公共子序列

给定两个字符串，我们需要找到它们共同存在的最长公共子序列。

示例

输入序列“ABCDGH”和“AEDFHR”的最长公共子序列是“ADH”，长度为3。

输入序列“AGGTAB”和“GXTXAYB”的最长公共子序列是“GTAB”，长度为4。

Java实现

```
public class LCS {

    public static void main(String[] args) {
        // TODO 自动生成的方法存根
        String str1 = "AGGTAB";
        String str2 = "GXTXAYB";
LCS obj = new LCS();
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        System.out.println(obj.lcs2(str1, str2));
    }

    //递归函数
    public int lcs(String str1, String str2, int m, int n){
```



```
            endif
        endif
    endfor
endfor

maxProfit = 0
for i -> 1 to n
    if maxProfit < Acc_Prof[i]
        maxProfit = Acc_Prof[i]
return maxProfit
```

The complexity of populating the **Acc_Prof** array is **O(n2).** The array traversal takes **O(n)**. So the total complexity of this algorithm is **O(n2).**

Now, If we want to find out which jobs were performed to get the maximum profit, we need to traverse the array in reverse order and if the **Acc_Prof** matches the **maxProfit**, we will push the **name** of the job in a **stack** and subtract **Profit** of that job from **maxProfit**. We will do this until our **maxProfit > 0** or we reach the beginning point of the **Acc_Prof** array. The pseudo-code will look like:

```
Procedure FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
for i -> n down to 0 and maxProfit > 0
    if maxProfit is equal to Acc_Prof[i]
        S.push(Job[i].name
        maxProfit = maxProfit - Job[i].profit
    endif
endfor
```

The complexity of this procedure is: **O(n)**.

One thing to remember, if there are multiple job schedules that can give us maximum profit, we can only find one job schedule via this procedure.

Section 14.3: Longest Common Subsequence

If we are given with the two strings we have to find the longest common sub-sequence present in both of them.

Example

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

Implementation in Java

```
public class LCS {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str1 = "AGGTAB";
        String str2 = "GXTXAYB";
        LCS obj = new LCS();
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        System.out.println(obj.lcs2(str1, str2));
    }

    //Recursive function
    public int lcs(String str1, String str2, int m, int n){
```

```
        if(m==0 || n==0)
            return 0;
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }

    //迭代函数
    public int lcs2(String str1, String str2){
        int lcs[][] = new int[str1.length()+1][str2.length()+1];

        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                if(i==0 || j== 0){
lcs[i][j] = 0;
                }
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{
lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
                }
            }
        }

        return lcs[str1.length()][str2.length()];
    }

}
```

输出

```
4
```

第14.4节：斐波那契数

使用动态规划的自底向上方法打印第n个斐波那契数。

递归树

```
fib(5)
                        /           \
fib(4)                  fib(3)
                /       \                   /     \
          fib(3)        fib(2)          fib(2)    fib(1)
         /     \          /    \         /    \
fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

重叠子问题

这里 fib(0)、fib(1) 和 fib(3) 是重叠的子问题。fib(0) 被重复了 3 次，fib(1) 被重复了 5 次，fib(3) 被重复了 2 次。

实现

```
public int fib(int n){
        int f[] = new int[n+1];
```



```
        if(m==0 || n==0)
            return 0;
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }

    //Iterative function
    public int lcs2(String str1, String str2){
        int lcs[][] = new int[str1.length()+1][str2.length()+1];

        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                if(i==0 || j== 0){
                    lcs[i][j] = 0;
                }
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{
                    lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
                }
            }
        }

        return lcs[str1.length()][str2.length()];
    }

}
```

Output

```
4
```

Section 14.4: Fibonacci Number

Bottom up approach for printing the nth Fibonacci number using Dynamic Programming.

Recursive Tree

```
                        fib(5)
                      /         \
                fib(4)            fib(3)
              /      \            /     \
         fib(3)      fib(2)    fib(2)    fib(1)
        /     \       /   \     /   \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Overlapping Sub-problems

Here fib(0),fib(1) and fib(3) are the overlapping sub-problems.fib(0) is getting repeated 3 times, fib(1) is getting repeated 5 times and fib(3) is getting repeated 2 times.

Implementation

```
public int fib(int n){
        int f[] = new int[n+1];
```

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

时间复杂度

O(n)

第14.5节：最长公共子串

给定两个字符串str1和str2，我们需要找到它们之间最长公共子串的长度。

示例

输入：X = "abcdxyz"，y = "xyzabcd" 输出：4

最长公共子串是 "abcd"，长度为4。

输入：X = "zxabcdezy"，y = "yzabcdezx" 输出：6

最长公共子串是 "abcdez"，长度为6。

Java实现

```
public int getLongestCommonSubstring(String str1,String str2){
        int arr[][] = new int[str2.length()+1][str1.length()+1];
        int max = Integer.MIN_VALUE;
        for(int i=1;i<=str2.length();i++){
            for(int j=1;j<=str1.length();j++){
                if(str1.charAt(j-1) == str2.charAt(i-1)){
                    arr[i][j] = arr[i-1][j-1]+1;
                    if(arr[i][j]>max)
max = arr[i][j];
                }
                否则
arr[i][j] = 0;
            }
        }
        return max;
    }
```

时间复杂度

O(m*n)


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

Time Complexity

O(n)

Section 14.5: Longest Common Substring

Given 2 string str1 and str2 we have to find the length of the longest common substring between them.

Examples

Input : X = "abcdxyz", y = "xyzabcd" Output : 4

The longest common substring is "abcd" and is of length 4.

Input : X = "zxabcdezy", y = "yzabcdezx" Output : 6

The longest common substring is "abcdez" and is of length 6.

Implementation in Java

```
public int getLongestCommonSubstring(String str1,String str2){
        int arr[][] = new int[str2.length()+1][str1.length()+1];
        int max = Integer.MIN_VALUE;
        for(int i=1;i<=str2.length();i++){
            for(int j=1;j<=str1.length();j++){
                if(str1.charAt(j-1) == str2.charAt(i-1)){
                    arr[i][j] = arr[i-1][j-1]+1;
                    if(arr[i][j]>max)
                        max = arr[i][j];
                }
                else
                    arr[i][j] = 0;
            }
        }
        return max;
    }
```

Time Complexity

O(m*n)

第15章：动态规划的应用

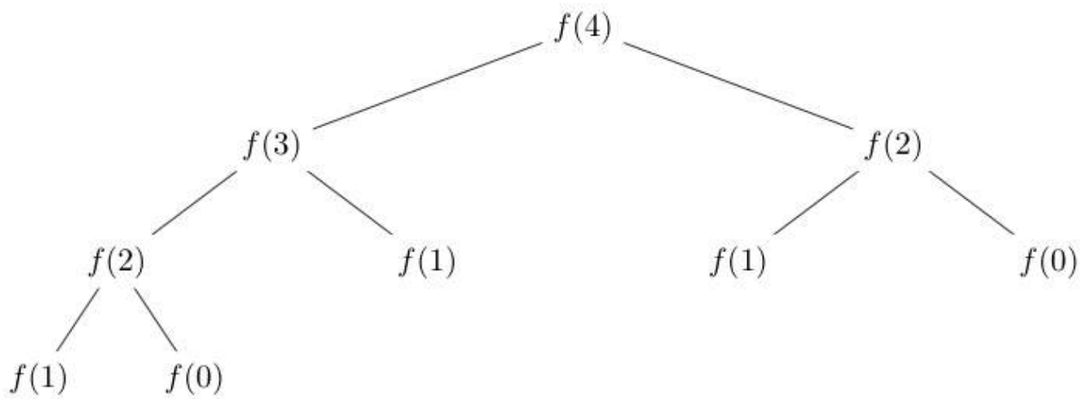
动态规划的基本思想是将一个复杂问题分解为几个重复出现的小而简单的问题。如果你能识别出一个反复计算的简单子问题，那么很可能存在该问题的动态规划方法。

由于本章标题为动态规划的应用，重点将更多放在应用上，而非动态规划算法的创建过程。

第15.1节：斐波那契数列

斐波那契数列是动态规划的典型例子，因为传统的递归方法会进行大量重复计算。在这些例子中，我将使用基准情况f(0) = f(1) = 1。

下面是计算fibonacci(4)的递归树示例，注意重复计算的部分：



非动态规划O(2^n)运行时间复杂度，O(n)栈空间复杂度

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

这是编写该问题最直观的方法。最多栈空间将是O(n)，因为你沿着第一个递归分支调用fibonacci(n-1)，直到达到基准情况 n < 2。

这里可以看到O(2^n)的运行时间复杂度证明：Fibonacci序列的计算复杂度。主要需要注意的是运行时间是指数级的，这意味着每增加一个项，运行时间都会翻倍，fibonacci(15)的运行时间将是fibonacci(14)的两倍。

带备忘录的O(n)运行时间复杂度，O(n)空间复杂度，O(n)栈复杂度

```
memo = []
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonacci(n):
    if len(memo) > n:
        return memo[n]
```


Chapter 15: Applications of Dynamic Programming

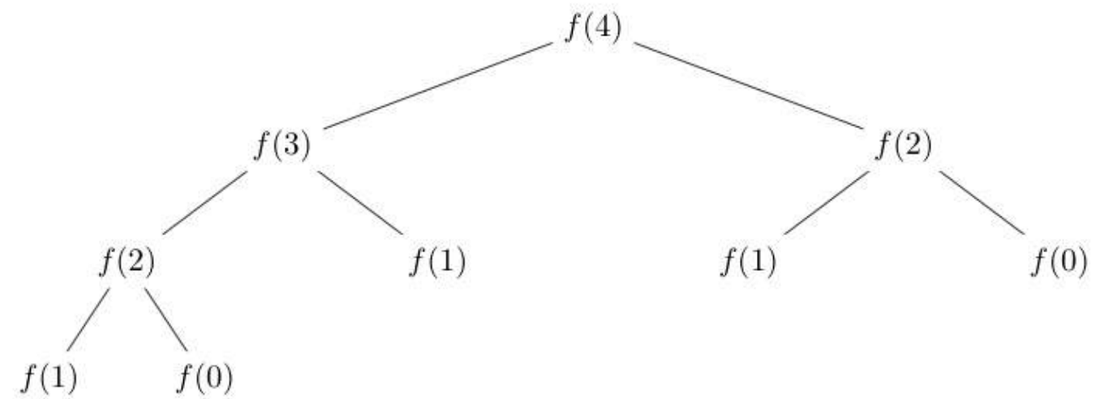
The basic idea behind dynamic programming is breaking a complex problem down to several small and simple problems that are repeated. If you can identify a simple subproblem that is repeatedly calculated, odds are there is a dynamic programming approach to the problem.

As this topic is titled *Applications of Dynamic Programming*, it will focus more on applications rather than the process of creating dynamic programming algorithms.

Section 15.1: Fibonacci Numbers

Fibonacci Numbers are a prime subject for dynamic programming as the traditional recursive approach makes a lot of repeated calculations. In these examples I will be using the base case of `f(0) = f(1) = 1`.

Here is an example recursive tree for `fibonacci(4)`, note the repeated computations:



Non-Dynamic Programming `O(2^n)` Runtime Complexity, `O(n)` Stack complexity

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

This is the most intuitive way to write the problem. At most the stack space will be `O(n)` as you descend the first recursive branch making calls to `fibonacci(n-1)` until you hit the base case `n < 2`.

The `O(2^n)` runtime complexity proof that can be seen here: Computational complexity of Fibonacci Sequence. The main point to note is that the runtime is exponential, which means the runtime for this will double for every subsequent term, `fibonacci(15)` will take twice as long as `fibonacci(14)`.

Memoized `O(n)` Runtime Complexity, `O(n)` Space complexity, `O(n)` Stack complexity

```
memo = []
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonacci(n):
    if len(memo) > n:
        return memo[n]
```

```
结果 = 斐波那契数列( n-1) + 斐波那契数列( n-2)
    备忘录.添加(结果) # f(n) = f(n-1) + f(n-2)
    返回 结果
```

通过备忘录方法，我们引入了一个数组，可以看作是所有之前的函数调用。
位置 `备忘录[n]` 是函数调用 `斐波那契数列(n)` 的结果。这使我们能够用 `O(n)` 的空间复杂度来换取一个 `O(n)` 的运行时间复杂度，因为我们不再需要计算重复的函数调用。

迭代动态规划 O(n) 运行时间复杂度，O(n) 空间复杂度，无递归栈

```
def fibonacci(n):
备忘录 = [1,1] # f(0) = 1, f(1) = 1

    对于 i 在范围(2, n+1) 内:
备忘录.添加(备忘录[i-1] + 备忘录[i-2])

    return memo[n]
```

如果我们将问题分解为核心元素，你会注意到为了计算 斐波那契数列(n)，我们需要 斐波那契数列(n-1) 和 斐波那契数列(n-2)。同时我们也可以注意到，基本情况会出现在上面递归树的末端。

有了这些信息，现在从基本情况开始向上计算解决方案是合理的。为了计算 斐波那契数列(n)，我们首先计算 所有直到 n 的斐波那契数。

这里的主要好处是我们现在消除了递归栈，同时保持了O(n)的运行时间。
不幸的是，我们仍然有一个O(n)的空间复杂度，但这也可以改变。

高级迭代动态规划O(n)运行时复杂度，O(1)空间复杂度，无递归栈

```
def fibonacci(n):
memo= [1,1] # f(1) = 1, f(2) = 1

    for i in range (2, n):
memo[i%2] = memo[0] + memo[1]

    return memo[n%2]
```

如上所述，迭代动态规划方法从基本情况开始，逐步计算到最终结果。为了将空间复杂度降到O(1)（常数级），关键的观察与我们对递归栈的观察相同——我们只需要fibonacci(n-1)和fibonacci(n-2)来构建fibonacci(n)。这意味着在迭代过程中，我们只需保存fibonacci(n-1)和fibonacci(n-2)的结果。

为了存储这两个最新的结果，我使用了一个大小为2的数组，并通过使用 `i % 2`来切换赋值的索引，这样索引会交替出现：0, 1, 0, 1, 0, 1, ...，即 `i % 2`。

我将数组的两个索引对应的值相加，因为我们知道加法是交换律的（`5 + 6 = 11` 且 `6 + 5 = 11`）。然后将结果赋值给较旧的那个位置（由 `i % 2`表示）。最终结果存储在位置 `n % 2`。

注释

- 需要注意的是，有时为执行大量重复计算的函数设计一个迭代的备忘录解决方案是最好的，因为你会建立一个函数调用结果的缓存，后续调用如果已经计算过，可能达到O(1)的时间复杂度。


```
    result = fibonacci(n-1) + fibonacci(n-2)
    memo.append(result) # f(n) = f(n-1) + f(n-2)
    return result
```

With the memoized approach we introduce an array that can be thought of as all the previous function calls. The location `memo[n]` is the result of the function call `fibonacci(n)`. This allows us to trade space complexity of `O(n)` for a `O(n)` runtime as we no longer need to compute duplicate function calls.

Iterative Dynamic Programming `O(n)` Runtime complexity, `O(n)` Space complexity, No recursive stack

```
def fibonacci(n):
    memo = [1,1] # f(0) = 1, f(1) = 1

    for i in range(2, n+1):
        memo.append(memo[i-1] + memo[i-2])

    return memo[n]
```

If we break the problem down into it's core elements you will notice that in order to compute `fibonacci(n)` we need `fibonacci(n-1)` and `fibonacci(n-2)`. Also we can notice that our base case will appear at the end of that recursive tree as seen above.

With this information, it now makes sense to compute the solution backwards, starting at the base cases and working upwards. Now in order to calculate `fibonacci(n)` we first calculate **all** the fibonacci numbers up to and through `n`.

This main benefit here is that we now have eliminated the recursive stack while keeping the `O(n)` runtime. Unfortunately, we still have an `O(n)` space complexity but that can be changed as well.

Advanced Iterative Dynamic Programming `O(n)` Runtime complexity, `O(1)` Space complexity, No recursive stack

```
def fibonacci(n):
    memo = [1,1] # f(1) = 1, f(2) = 1

    for i in range (2, n):
        memo[i%2] = memo[0] + memo[1]

    return memo[n%2]
```

As noted above, the iterative dynamic programming approach starts from the base cases and works to the end result. The key observation to make in order to get to the space complexity to `O(1)` (constant) is the same observation we made for the recursive stack - we only need `fibonacci(n-1)` and `fibonacci(n-2)` to build `fibonacci(n)`. This means that we only need to save the results for `fibonacci(n-1)` and `fibonacci(n-2)` at any point in our iteration.

To store these last 2 results I use an array of size 2 and simply flip which index I am assigning to by using `i % 2` which will alternate like so: `0, 1, 0, 1, 0, 1, ..., i % 2`.

I add both indexes of the array together because we know that addition is commutative (`5 + 6 = 11` and `6 + 5 == 11`). The result is then assigned to the older of the two spots (denoted by `i % 2`). The final result is then stored at the position `n%2`

Notes

- It is important to note that sometimes it may be best to come up with a iterative memoized solution for

对于执行大量重复计算的函数，设计一个迭代的备忘录解决方案非常重要，因为你会建立函数调用结果的缓存，后续调用如果已经计算过，可能达到O(1)的时间复杂度。

functions that perform large calculations repeatedly as you will build up a cache of the answer to the function calls and subsequent calls may be `O(1)` if it has already been computed.

第16章：克鲁斯卡尔算法

第16.1节：基于最优不相交集的实现

我们可以做两件事来改进简单且次优的不相交集子算法：

1. 路径压缩启发式：findSet不需要处理高度大于2的树。如果它
最终遍历了这样的树，它可以将较低的节点直接连接到根节点，从而优化未来的遍历；

```
子算法 findSet(v:一个节点)：
    如果 v.parent != v
v.parent = findSet(v.parent)
    返回 v.父节点
```

2. **基于高度的合并启发式：对于每个节点，存储其子树的高度。合并时，将较高的树作为较小树的父节点，从而不增加任何节点的高度。**
较高的树作为较小树的父节点，从而不增加任何节点的高度。

```
子算法 unionSet(u, v: 节点):
     vRoot = findSet(v)
    uRoot = findSet(u)

     如果 vRoot == uRoot:
        返回

    如果 vRoot.高度 < uRoot.高度:
vRoot.父节点 = uRoot
    否则如果 vRoot.高度 > uRoot.高度:
uRoot.父节点 = vRoot
    else:
uRoot.parent = vRoot
        uRoot.height =  uRoot.height + 1
```

这导致每次操作的时间复杂度为O(alpha(n))，其中alpha是快速增长的Ackermann函数的反函数，因此增长非常缓慢，实际上可以视为O(1)。

这使得整个Kruskal算法的时间复杂度为O(m log m + m) = O(m log m)，因为初始排序的缘故。

注意

路径压缩可能会降低树的高度，因此在合并操作中比较树的高度
可能不是一件简单的事情。为了避免存储和计算树高度的复杂性，
可以随机选择合并后的父节点：

```
子算法 unionSet(u, v: 节点):
         vRoot = findSet(v)
        uRoot = findSet(u)

         如果 vRoot == uRoot:
            返回
        如果 random() % 2 == 0:
vRoot.parent = uRoot
        else:
uRoot.parent = vRoot
```

在实际操作中，这种随机算法结合路径压缩的findSet操作将导致

Chapter 16: Kruskal's Algorithm

Section 16.1: Optimal, disjoint-set based implementation

We can do two things to improve the simple and sub-optimal disjoint-set subalgorithms:

1. **Path compression heuristic**: `findSet` does not need to ever handle a tree with height bigger than 2. If it ends up iterating such a tree, it can link the lower nodes directly to the root, optimizing future traversals;

```
subalgo findSet(v: a node):
    if v.parent != v
        v.parent = findSet(v.parent)
    return v.parent
```

2. **Height-based merging heuristic**: for each node, store the height of its subtree. When merging, make the taller tree the parent of the smaller one, thus not increasing anyone's height.

```
subalgo unionSet(u, v: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    if vRoot == uRoot:
        return

    if vRoot.height < uRoot.height:
        vRoot.parent = uRoot
    else if vRoot.height > uRoot.height:
        uRoot.parent = vRoot
    else:
        uRoot.parent = vRoot
        uRoot.height =  uRoot.height + 1
```

This leads to `O(alpha(n))` time for each operation, where `alpha` is the inverse of the fast-growing Ackermann function, thus it is very slow growing, and can be considered `O(1)` for practical purposes.

This makes the entire Kruskal's algorithm `O(m log m + m) = O(m log m)`, because of the initial sorting.

Note

Path compression may reduce the height of the tree, hence comparing heights of the trees during union operation might not be a trivial task. Hence to avoid the complexity of storing and calculating the height of the trees the resulting parent can be picked randomly:

```
    subalgo unionSet(u, v: nodes):
        vRoot = findSet(v)
        uRoot = findSet(u)

        if vRoot == uRoot:
            return
        if random() % 2 == 0:
            vRoot.parent = uRoot
        else:
            uRoot.parent = vRoot
```

In practice this randomised algorithm together with path compression for `findSet` operation will result in

性能相当，但实现起来更简单。

第16.2节：简单且更详细的实现

为了高效处理环检测，我们将每个节点视为树的一部分。添加边时，我们检查其两个组成节点是否属于不同的树。最初，每个节点构成一个单节点树。

```
algorithm kruskalMST(G: a graph)
按值对G的边进行排序
MST = 一片森林，最初每棵树是图中的一个节点
    对于 G中的每条边 e：
        如果 e.first 所属树的根节点与
       e.second 所属树的根节点不同：
将其中一个根连接到另一个根，从而合并两棵树

    返回 MST，现在是一片单一-树的森林
```

第16.3节：基于简单不相交集的实现

上述森林方法实际上是一种不相交集数据结构，涉及三个主要操作：

```
子算法 makeSet(v: 一个节点):
v.parent = v     <- 创建一个以 v 为根的新树


子算法 findSet(v:一个节点)：
    如果 v.parent == v:
        返回 v
    返回 findSet(v.parent)

子算法 unionSet(v, u: 节点):
vRoot = findSet(v)
    uRoot = findSet(u)

    uRoot.parent = vRoot

算法 kruskalMST(G: 一个图):
按值对G的边进行排序
     对于 G 中的每个节点 n:
makeSet(n)
     对于 G中的每条边 e：
         if findSet(e.first) != findSet(e.second):
unionSet(e.first, e.second)
```

这种简单实现导致管理不相交集合数据结构的时间复杂度为`O(n log n)`，整个克鲁斯卡尔算法的时间复杂度为`O(m*n log n)`。

第16.4节：简单的高级实现

按权值排序边，并按排序顺序将每条边加入最小生成树，前提是不形成环。

```
算法 kruskalMST(G: 一个图)
    按权值排序G的边
     MST = 一个空图
     对于 G中的每条边 e：
         如果 将e加入MST不形成环:
将e加入MST
```


comparable performance, yet much simpler to implement.

Section 16.2: Simple, more detailed implementation

In order to efficiently handle cycle detection, we consider each node as part of a tree. When adding an edge, we check if its two component nodes are part of distinct trees. Initially, each node makes up a one-node tree.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = a forest of trees, initially each tree is a node in the graph
    for each edge e in G:
        if the root of the tree that e.first belongs to is not the same
        as the root of the tree that e.second belongs to:
            connect one of the roots to the other, thus merging two trees

    return MST, which now a single-tree forest
```

Section 16.3: Simple, disjoint-set based implementation

The above forest methodology is actually a disjoint-set data structure, which involves three main operations:

```
subalgo makeSet(v: a node):
    v.parent = v    <- make a new tree rooted at v


subalgo findSet(v: a node):
    if v.parent == v:
        return v
    return findSet(v.parent)

subalgo unionSet(v, u: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    uRoot.parent = vRoot

algorithm kruskalMST(G: a graph):
    sort Gs edges by their value
    for each node n in G:
        makeSet(n)
    for each edge e in G:
        if findSet(e.first) != findSet(e.second):
            unionSet(e.first, e.second)
```

This naive implementation leads to `O(n log n)` time for managing the disjoint-set data structure, leading to `O(m*n log n)` time for the entire Kruskal's algorithm.

Section 16.4: Simple, high level implementation

Sort the edges by value and add each one to the MST in sorted order, if it doesn't create a cycle.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = an empty graph
    for each edge e in G:
        if adding e to MST does not create a cycle:
            add e to MST
```

```
返回 MST
```

```
return MST
```

第17章：贪心算法

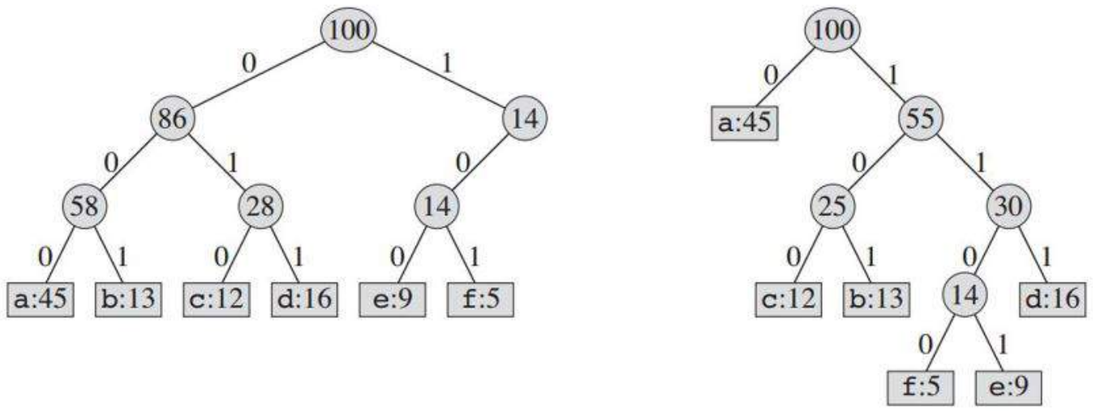
第17.1节：霍夫曼编码

霍夫曼编码是一种特定类型的最优前缀编码，常用于无损数据压缩。它能够非常有效地压缩数据，节省20%到90%的内存，具体取决于被压缩数据的特性。我们将数据视为字符序列。霍夫曼的贪心算法利用一个表格，给出每个字符出现的频率（即频率），以构建一种将每个字符表示为二进制字符串的最优方式。霍夫曼编码由大卫·A·霍夫曼于1951年提出。

假设我们有一个包含100,000个字符的数据文件，想要将其紧凑存储。我们假设该文件中只有6种不同的字符。字符的频率如下所示：

字符	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5

我们有多种方式来表示这样一个信息文件。在这里，我们考虑设计一种二进制字符编码，其中每个字符由唯一的二进制字符串表示，我们称之为“码字”。



Fixed-length Codeword

Variable-length Codeword

构建的树将为我们提供：

字符	a	b	c	d	e	f
固定长度码字	000	001	010	011	100	101
可变长度码字	0	101	100	111	1101	1100

如果我们使用固定长度编码，表示6个字符需要3位二进制数。该方法编码整个文件需要300,000位。现在的问题是，我们能做得更好吗？

Chapter 17: Greedy Algorithms

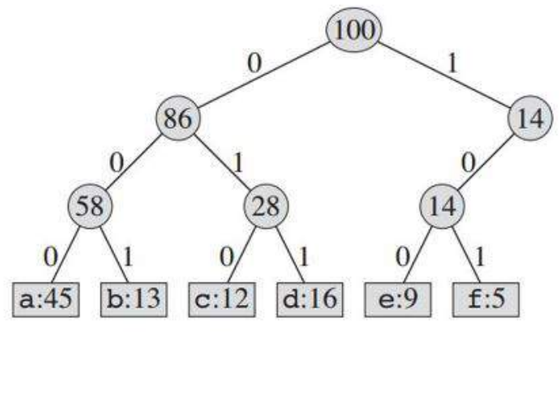
Section 17.1: Huffman Coding

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Huffman code was proposed by David A. Huffman in 1951.

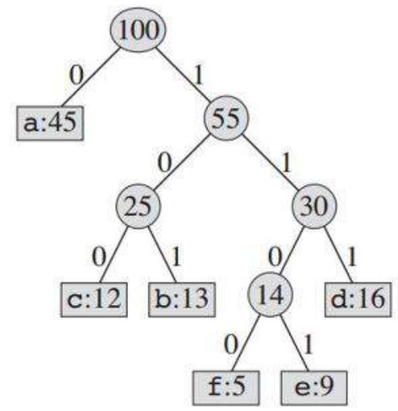
Suppose we have a 100,000-character data file that we wish to store compactly. We assume that there are only 6 different characters in that file. The frequency of the characters are given by:

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *Binary Character Code* in which each character is represented by a unique binary string, which we call a **codeword**.



Fixed-length Codeword



Variable-length Codeword

The constructed tree will provide us with:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

If we use a **fixed-length code**, we need three bits to represent 6 characters. This method requires 300,000 bits to code the entire file. Now the question is, can we do better?

可变长度编码比固定长度编码效果好得多，因为它为频繁出现的字符分配短码字，为不频繁出现的字符分配长码字。该编码需要：(45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 +5 X 4) X 1000 = 224000 位来表示该文件，节省了大约 25% 的内存。

有一点需要记住，我们这里仅考虑没有任何码字是其他码字前缀的编码。这些被称为前缀码。对于变长编码，我们将3字符文件abc编码为0.101.100 =0101100，其中“.”表示连接操作。

前缀码是理想的，因为它们简化了解码过程。由于没有码字是其他码字的前缀，编码文件开头的码字是明确无歧义的。我们可以简单地识别初始码字，将其译回原始字符，然后对剩余的编码文件重复解码过程。例如，001011101可以唯一解析为0.0.101.1101，解码为aabe。简而言之，所有二进制表示的组合都是唯一的。举例来说，如果一个字母用110表示，则不会有其他字母用1101或1100表示。这是因为否则你可能会在选择110还是继续连接下一个比特时产生混淆。

压缩技术：

该技术通过创建一个二叉树节点来实现。这些节点可以存储在一个常规数组中，数组大小取决于符号数量 n。一个节点可以是叶节点或内部节点。最初所有节点都是叶节点，包含符号本身、其频率以及可选的指向子节点的链接。按照约定，位'0'表示左子节点，位'1'表示右子节点。使用优先队列存储节点，弹出时提供频率最低的节点。过程如下：

1.为每个符号创建一个叶节点并将其加入优先队列。
2.当队列中节点数多于一个时：
    1.从队列中移除两个优先级最高的节点。
    2.创建一个新的内部节点，将这两个节点作为子节点，频率为两个节点频率之和。
    3.将新节点加入队列。
3.剩余的节点即为根节点，霍夫曼树构建完成。

对于我们的示例：

A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. This code requires: (45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 + 5 X 4) X 1000 = 224000 bits to represent the file, which saves approximately 25% of memory.

One thing to remember, we consider here only codes in which no codeword is also a prefix of some other codeword. These are called *prefix codes.* For variable-length coding, we code the 3-character file *abc* as 0.101.100 = 0101100, where "." denotes the concatenation.

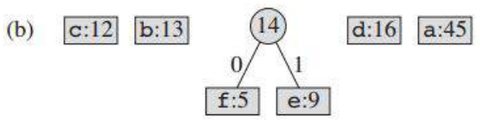
Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. For example, 001011101 parses uniquely as 0.0.101.1101, which decodes to *aabe*. In short, all the combinations of binary representations are unique. Say for example, if one letter is denoted by 110, no other letter will be denoted by 1101 or 1100. This is because you might face confusion on whether to select 110 or to continue on concatenating the next bit and select that one.

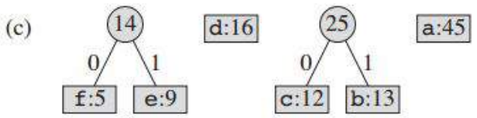
Compression Technique:

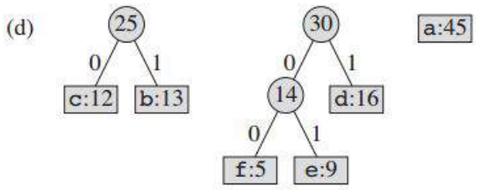
The technique works by creating a *binary tree* of nodes. These can stored in a regular array, the size of which depends on the number of symbols, **n**. A node can either be a *leaf node* or an *internal node*. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. *Priority queue* is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

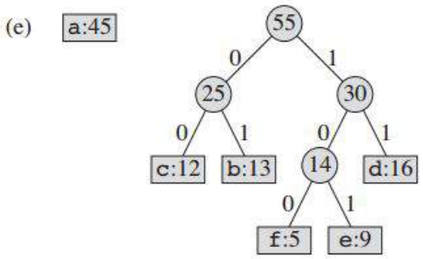
1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the two nodes of highest priority from the queue.
    2. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.
    3. Add the new node to the queue.
3. The remaining node is the root node and the Huffman tree is complete.

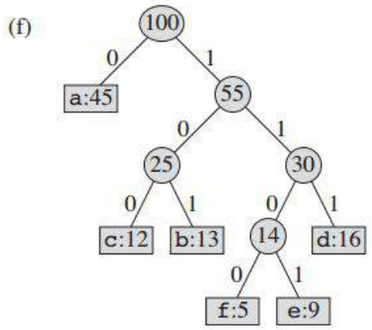
For our example:











伪代码如下：

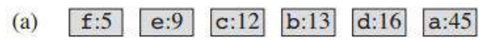
```
过程 Huffman(C):     // C 是包含 n 个字符及相关信息的集合
n = C.大小
Q = 优先队列()
对于 i = 1 到 n
n = 节点(C[i])
Q.入队(n)
结束 循环
当 Q.大小() 不等于 1
    Z = 新 节点()
Z.左子节点 = x = Q.出队
Z.右子节点 = y = Q.出队
Z.频率 = x.频率 + y.频率
Q.push(Z)
end while
Return Q
```

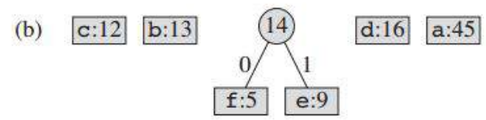
虽然在输入已排序的情况下是线性时间，但对于任意输入的通用情况，使用该算法需要预先排序。因此，由于排序在一般情况下需要O(nlogn)时间，两个方法的复杂度相同。

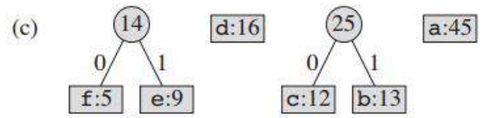
由于此处的 n 是字母表中的符号数量，通常是一个非常小的数字（相较于要编码的消息长度），因此时间复杂度在选择该算法时并不十分重要。

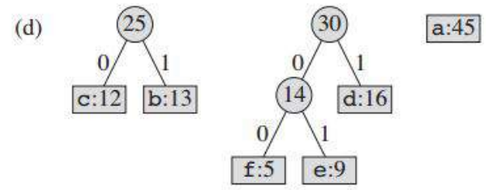
解压缩技术：

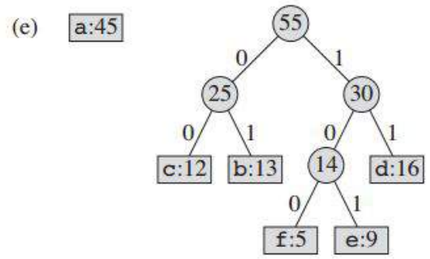
解压缩过程仅仅是将前缀码流转换为单个字节值，通常通过在读取输入流的每个位时逐节点遍历哈夫曼树来实现。到达叶子节点必然终止对该特定字节值的搜索。叶子节点的值表示所需的

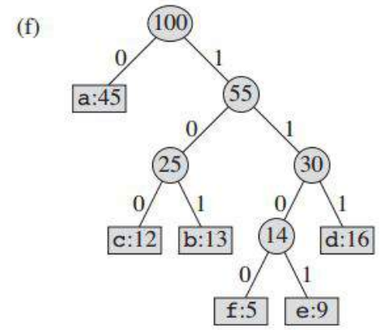












The pseudo-code looks like:

```
Procedure Huffman(C):     // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
end while
Return Q
```

Although linear-time given sorted input, in general cases of arbitrary input, using this algorithm requires pre-sorting. Thus, since sorting takes **O(nlogn)** time in general cases, both methods have same complexity.

Since **n** here is the number of symbols in the alphabet, which is typically very small number (compared to the length of the message to be encoded), time complexity is not very important in the choice of this algorithm.

Decompression Technique:

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired

字符。通常哈夫曼树是使用每次压缩周期中统计调整的数据构建的，因此重建相当简单。否则，重建树的信息必须单独发送。伪代码如下：

```
过程 HuffmanDecompression(root, S):   // root 表示哈夫曼树的根节点
n := S.length                         // S 指要解压的比特流
for i := 1 到 n
current = root
    while current.left != NULL 且 current.right != NULL
        if S[i] 等于 '0'
current := current.left
         否则
current := current.right
       endif
i := i+1
    endwhile
print current.symbol
endfor
```

贪心算法解释：
哈夫曼编码通过观察每个字符的出现情况，并以最优方式将其存储为二进制字符串。其思想是为输入字符分配可变长度的编码，编码长度基于相应字符的频率。我们创建一棵二叉树，并自底向上操作，使得频率最低的两个字符尽可能远离根节点。这样，出现频率最高的字符获得最短的编码，出现频率最低的字符获得最长的编码。

参考文献：

- 算法导论 - 查尔斯·E·莱瑟森、克利福德·斯坦、罗纳德·里维斯特、托马斯·H·科尔曼
- 哈夫曼编码 - 维基百科
- 离散数学及其应用 - 肯尼斯·H·罗森

第17.2节：活动选择问题

问题描述

你有一组要做的事情（活动）。每个活动都有一个开始时间和结束时间。你不允许同时进行多个活动。你的任务是找到一种方法来完成最多数量的活动。

例如，假设你有一组选课可供选择。

活动编号	开始时间	结束时间
1	上午10:20	上午11:00
2	上午10:30	上午11:30
3	上午11:00	下午12:00
4	上午10:00	上午11:30
5	上午9:00	上午11:00

请记住，你不能同时参加两门课程。这意味着你不能同时选第1和第2节课，因为它们在上午10:30到11:00有重叠时间。然而，你可以选第1和第3节课，因为它们没有重叠时间。所以你的任务是在不重叠的情况下尽可能多地选课。你该如何做到呢？

分析

character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately. The pseudo-code:

```
Procedure HuffmanDecompression(root, S):   // root represents the root of Huffman Tree
n := S.length                              // S refers to bit-stream to be decompressed
for i := 1 to n
    current = root
    while current.left != NULL and current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
    print current.symbol
endfor
```

Greedy Explanation:
Huffman coding looks at the occurrence of each character and stores it as a binary string in an optimal way. The idea is to assign variable-length codes to input input characters, length of the assigned codes are based on the frequencies of corresponding characters. We create a binary tree and operate on it in bottom-up manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

References:

- Introduction to Algorithms - Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen
- Huffman Coding - Wikipedia
- Discrete Mathematics and Its Applications - Kenneth H. Rosen

Section 17.2: Activity Selection Problem

The Problem

You have a set of things to do (activities). Each activity has a start time and a end time. You aren't allowed to perform more than one activity at a time. Your task is to find a way to perform the maximum number of activities.

For example, suppose you have a selection of classes to choose from.

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

Remember, you can't take two classes at the same time. That means you can't take class 1 and 2 because they share a common time 10.30 A.M to 11.00 A.M. However, you can take class 1 and 3 because they don't share a common time. So your task is to take maximum number of classes as possible without any overlap. How can you do that?

Analysis

让我们通过贪心算法来思考解决方案。首先我们随机选择一种方法，检查它是否可行。

- 按开始时间排序活动也就是说，哪个活动先开始，我们就先选哪个。然后从排序后的列表中从第一个到最后一个依次检查，判断当前活动是否与之前选择的活动有交集。如果当前活动与之前选择的活动没有交集，我们就执行该活动，否则不执行。该方法适用于某些情况，比如

活动编号	开始时间	结束时间
1	上午11:00	下午1:30
2	上午11:30	中午12:00
3	下午1:30	下午2:00
4	上午10:00	上午11:00

排序顺序将是4-->1-->2-->3。活动4-->1-->3将被执行，活动2将被跳过。
最多可以执行3个活动。该方法适用于这类情况，但在某些情况下会失败。让我们将此方法应用于以下情况

活动编号	开始时间	结束时间
1	上午11:00	下午1:30
2	上午11:30	中午12:00
3	下午1:30	下午2:00
4	上午10:00	下午3:00

排序顺序将是4-->1-->2-->3，只有活动4会被执行，但正确答案可能是活动1-->3或2-->3被执行。因此，我们的方法在上述情况下不起作用。让我们尝试另一种方法。

- **按时间持续时间对活动进行排序**也就是说先执行持续时间最短的活动。这样可以解决之前的问题。
  尽管问题没有完全解决，仍然存在一些情况可能导致解决方案失败。
  将此方法应用于以下案例。

活动编号	开始时间	结束时间
1	上午6:00	上午11:40
2	上午11:30	中午12:00
3	晚上11:40	下午2:00

如果我们按活动持续时间排序，排序顺序将是2 --> 3 --> 1。如果我们先执行活动2，那么就无法执行其他活动。但答案是先执行活动1，然后执行活动3。因此我们最多可以执行2个活动。所以这不能作为该问题的解决方案。我们应该尝试不同的方法。

解决方案

- **按结束时间排序活动**也就是说先结束的活动排在前面。算法如下
  所示

> 1. 按活动的结束时间排序。
> 2. 如果要执行的活动与之前执行的活动没有时间重叠，则执行该活动。

让我们分析第一个例子

Lets think for the solution by greedy approach.First of all we randomly chose some approach and check that will work or not.

- **sort the activity by start time** that means which activity start first we will take them first. then take first to last from sorted list and check it will intersect from previous taken activity or not. If the current activity is not intersect with the previously taken activity, we will perform the activity otherwise we will not perform. this approach will work for some cases like

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	11.00AM

the sorting order will be 4-->1-->2-->3 .The activity 4--> 1--> 3 will be performed and the activity 2 will be skipped. the maximum 3 activity will be performed. It works for this type of cases. but it will fail for some cases. Lets apply this approach for the case

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	3.00P.M

The sort order will be 4-->1-->2-->3 and only activity 4 will be performed but the answer can be activity 1-->3 or 2-->3 will be performed. So our approach will not work for the above case. Let's try another approach

- **Sort the activity by time duration** that means perform the shortest activity first. that can solve the previous problem . Although the problem is not completely solved. There still some cases that can fail the solution. apply this approach on the case bellow.

Activity No.	start time	end time
1	6.00 A.M	11.40A.M
2	11.30 A.M	12.00P.M
3	11.40 P.M	2.00P.M

if we sort the activity by time duration the sort order will be 2--> 3 --->1 . and if we perform activity No. 2 first then no other activity can be performed. But the answer will be perform activity 1 then perform 3 . So we can perform maximum 2 activity.So this can not be a solution of this problem. We should try a different approach.

The solution

- **Sort the Activity by ending time** that means the activity finishes first that come first. the algorithm is given below

> 1. Sort the activities by its ending times.
> 2. If the activity to be performed do not share a common time with the activities that previously performed, perform the activity.

Lets analyse the first example

活动编号	开始时间	结束时间
1	上午10:20	上午11:00
2	上午10:30	上午11:30
3	上午11:00	下午12:00
4	上午10:00	上午11:30
5	上午9:00	上午11:00

按结束时间排序活动，排序顺序为1 --> 5 --> 2 --> 4 --> 3。答案是1 --> 3，这两个活动将被执行。这就是答案。以下是伪代码。

> 1.排序：活动
> 2.执行排序后活动列表中的第一个活动。
> 3.设置：当前活动 := 第一个活动
> 4.设置：结束时间 := 当前活动的结束时间
> 5.如果存在下一个活动则转到下一个活动，否则终止。
> 6.如果当前活动的开始时间 <= 结束时间：执行该活动并转到步骤4
> 7.否则：转到步骤5。

查看此处获取编码帮助 http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

第17.3节：找零问题

给定一种货币系统，判断是否可以给出一定数量的硬币，以及如何找到对应该金额的最小硬币集合。

规范货币系统。对于某些货币系统，比如我们现实生活中使用的，"直观"的解决方案完全有效。例如，如果不同的欧元硬币和纸币（不包括分币）是1€、2€、5€、10€，每次给出最高面额的硬币或纸币直到达到金额，重复此过程将得到最小的硬币集合。

我们可以用OCaml递归实现：

```
(* 假设货币系统按降序排列 *)
let change_make money_system amount =
let rec loop given amount =
    if amount = 0 then given
    否则
      (* 我们找到第一个小于或等于剩余金额的值 *)
     let coin = List.find ((>=) amount) money_system in
     loop (coin::given) (amount - coin)
in loop [] amount
```

这些系统的设计使得找零变得简单。当涉及任意货币系统时，问题变得更加复杂。

一般情况。 如何用10欧元、7欧元和5欧元的硬币凑出99欧元？这里，如果一直用10欧元硬币，直到剩下9欧元，显然无法解决。更糟的是，可能根本不存在解决方案。这个问题实际上是NP难的，但存在结合贪心和记忆化的可接受解决方案。其思路是探索所有可能性，并选择硬币数量最少的方案。

要给出金额X > 0，我们选择货币系统中的一枚硬币P，然后解决对应于X-P的子问题。我们对系统中的所有硬币都尝试一遍。如果存在解决方案，则该方案是导致0的最短路径。

下面是对应该方法的OCaml递归函数。如果不存在解决方案，则返回None。

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

sort the activity by its ending times , So sort order will be 1-->5-->2-->4-->3.. the answer is 1-->3 these two activities will be performed. ans that's the answer. here is the sudo code.

> 1. sort: activities
> 2. perform first activity from the sorted list of activities.
> 3. Set : Current_activity := first activity
> 4. set: end_time := end_time of Current activity
> 5. go to next activity if exist, if not exist terminate .
> 6. if start_time of current activity <= end_time : perform the activity and go to 4
> 7. else: got to 5.

see here for coding help http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

Section 17.3: Change-making problem

Given a money system, is it possible to give an amount of coins and how to find a minimal set of coins corresponding to this amount.

Canonical money systems. For some money system, like the ones we use in the real life, the "intuitive" solution works perfectly. For example, if the different euro coins and bills (excluding cents) are 1€, 2€, 5€, 10€, giving the highest coin or bill until we reach the amount and repeating this procedure will lead to the minimal set of coins.

We can do that recursively with OCaml :

```
(* assuming the money system is sorted in decreasing order *)
let change_make money_system amount =
  let rec loop given amount =
    if amount = 0 then given
    else
      (* we find the first value smaller or equal to the remaining amount *)
      let coin = List.find ((>=) amount) money_system in
      loop (coin::given) (amount - coin)
  in loop [] amount
```

These systems are made so that change-making is easy. The problem gets harder when it comes to arbitrary money system.

General case. How to give 99€ with coins of 10€, 7€ and 5€? Here, giving coins of 10€ until we are left with 9€ leads obviously to no solution. Worse than that a solution may not exist. This problem is in fact np-hard, but acceptable solutions mixing **greediness** and **memoization** exist. The idea is to explore all the possibilies and pick the one with the minimal number of coins.

To give an amount X > 0, we choose a piece P in the money system, and then solve the sub-problem corresponding to X-P. We try this for all the pieces of the system. The solution, if it exists, is then the smallest path that led to 0.

Here an OCaml recursive function corresponding to this method. It returns None, if no solution exists.

```
(* option工具函数 *)
let optmin x y =
  match x,y with
  | 无,a | a,无 -> a
  | 有 x, 有 y-> 有 (min x y)

let optsucc = function
  | 有 x -> 有 (x+1)
  | 无 -> 无

(* 找零问题*)
let change_make money_system amount =
  let rec loop n =
let onepiece acc piece =
match n - piece with
      | 0 -> (*用一枚硬币解决问题*)
有 1
      | x -> if x < 0 then
                (*我们没有达到0，放弃这个方案*)
              无
             else
 (*我们搜索与剩余棋子不同且非None的最短路径*)
               optmin (optsucc (loop x)) acc
在
 (*我们称所有硬币为onepiece*)
    List.fold_left onepiece None money_system
 在循环金额中
```

注意：我们可以注意到该过程可能会多次计算相同数值的找零集合。在实际应用中，使用记忆化来避免这些重复计算会带来更快（快得多）的结果。


```
(* option utilities *)
let optmin x y =
  match x,y with
  | None,a | a,None -> a
  | Some x, Some y-> Some (min x y)

let optsucc = function
  | Some x -> Some (x+1)
  | None -> None

(* Change-making problem*)
let change_make money_system amount =
  let rec loop n =
    let onepiece acc piece =
      match n - piece with
      | 0 -> (*problem solved with one coin*)
             Some 1
      | x -> if x < 0 then
               (*we don't reach 0, we discard this solution*)
               None
             else
               (*we search the smallest path different to None with the remaining pieces*)
               optmin (optsucc (loop x)) acc
    in
    (*we call onepiece forall the pieces*)
    List.fold_left onepiece None money_system
  in loop amount
```

Note: We can remark that this procedure may compute several times the change set for the same value. In practice, using memoization to avoid these repetitions leads to faster (way faster) results.

第18章：贪心技术的应用

第18.1节：离线缓存

缓存问题源于有限空间的限制。假设我们的缓存C有k页。现在我们想处理一个长度为m的请求序列，这些请求必须在处理前被放入缓存中。当然，如果m<=k，那么我们只需将所有元素放入缓存即可，但通常m>>k。

当请求的项目已在缓存中时，我们称之为缓存命中（cache hit），否则称为缓存未命中（cache miss）。在这种情况下，假设缓存已满，我们必须将请求的项目加载到缓存中并驱逐另一个项目。目标是制定一个最小化驱逐次数的驱逐计划。

针对该问题有许多贪心策略，下面来看一些：

1. 先进先出（FIFO）：最旧的页面被驱逐
2. 后进先出（LIFO）：最新的页面被驱逐
3. 最近最少使用（LRU）：淘汰最近访问时间最早的页面
4. 最不经常请求（LFU）：淘汰请求次数最少的页面
5. 最长前向距离（LFD）：淘汰缓存中未来最长时间内不会被请求的页面。

注意： 对于以下示例，如果有多个页面可以被淘汰，我们将淘汰索引最小的页面。

示例（FIFO）

设缓存大小为 k=3，初始缓存为 a,b,c，请求序列为 a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c：

请求序列	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
缓存 1	a	a	d	d	d	d	a	a	a	d	d	d	f	f	f	c
缓存 2	b	b	b	e	e	e	e	c	c	c	e	e	e	b	b	b
缓存 3	c	c	c	c	b	b	b	b	f	f	f	a	a	a	e	e
缓存未命中	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

十六次请求中有十三次缓存未命中，听起来并不是很理想，我们尝试用另一种

策略来做同样的例子：

示例（LFD）

设缓存大小为 k=3，初始缓存为 a,b,c，请求序列为 a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c：

请求序列	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
缓存 1	a	a	d	e	e	e	e	e	e	e	e	e	e	e	e	c
缓存 2	b	b	b	b	b	b	a	a	a	a	a	a	f	f	f	f
缓存 3	c	c	c	c	c	c	c	c	f	d	d	d	d	b	b	b
缓存未命中	x	x		x		x	x		x	x		x	x		x	

八次缓存未命中要好得多。

自测：对LIFO、LFU、RFU进行示例测试，看看结果如何。

以下示例程序（用C++编写）由两部分组成：

Chapter 18: Applications of Greedy technique

Section 18.1: Offline Caching

The caching problem arises from the limitation of finite space. Lets assume our cache `C` has `k` pages. Now we want to process a sequence of `m` item requests which must have been placed in the cache before they are processed.Of course if `m<=k` then we just put all elements in the cache and it will work, but usually is `m>>k`.

We say a request is a **cache hit**, when the item is already in cache, otherwise its called a **cache miss**. In that case we must bring the requested item into cache and evict another, assuming the cache is full. The Goal is a eviction schedule that **minimizes the number of evictions**.

There are numerous greedy strategies for this problem, lets look at some:

1. **First in, first out (FIFO)**: The oldest page gets evicted
2. **Last in, first out (LIFO)**: The newest page gets evicted
3. **Last recent out (LRU)**: Evict page whose most recent access was earliest
4. **Least frequently requested(LFU)**: Evict page that was least frequently requested
5. **Longest forward distance (LFD)**: Evict page in the cache that is not requested until farthest in the future.

Attention: For the following examples we evict the page with the smallest index, if more than one page could be evicted.

Example (FIFO)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	d	d	d	a	a	a	d	d	d	f	f	f	c
cache 2	b	b	b	e	e	e	e	c	c	c	e	e	e	b	b	b
cache 3	c	c	c	c	b	b	b	b	f	f	f	a	a	a	e	e
cache miss	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Thirteen cache misses by sixteen requests does not sound very optimal, lets try the same example with another strategy:

Example (LFD)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	e	e	e	e	e	e	e	e	e	e	e	e	c
cache 2	b	b	b	b	b	b	a	a	a	a	a	a	f	f	f	f
cache 3	c	c	c	c	c	c	c	c	f	d	d	d	d	b	b	b
cache miss	x	x		x		x	x		x	x		x	x		x	

Eight cache misses is a lot better.

Selftest: Do the example for LIFO, LFU, RFU and look what happend.

The following example programm (written in C++) consists of two parts:

框架是一个应用程序，根据所选的贪心策略解决问题：

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize     = 3;
const int requestLength = 16;

const char request[]    = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c'};
char cache[]            = {'a','b','c'};

// 用于重置
char originalCache[]    = {'a','b','c'};


class Strategy {

public:
Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    // 计算应该使用哪个缓存位置
    virtual int apply(int requestIndex)                                    = 0;

    // 更新策略所需的信息
    virtual void update(int cachePlace, int requestIndex, bool cacheMiss)    = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    // 计算请求放置的位置
    int cachePlace = strategy->apply(requestIndex);

    // 判断是缓存命中还是缓存未命中
    bool isMiss = request[requestIndex] != cache[cachePlace];

    // 更新策略（例如重新计算距离）
strategy->update(cachePlace, requestIndex, isMiss);

    // 写入缓存
cache[cachePlace] = request[requestIndex];

    return isMiss;
}

int main()
{
Strategy* selectedStrategy[] = { new FIFO, new LIFO, new LRU, new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {
        // 重置缓存
        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];cout <<"策略: " <

        < selectedStrategy[strat]->strategyName << endl;cout << "缓存初始状态: (";


        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] <<
```


The skeleton is a application, which solves the problem dependent on the chosen greedy strategy:

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize     = 3;
const int requestLength = 16;

const char request[]    = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c'};
char cache[]            = {'a','b','c'};

// for reset
char originalCache[]    = {'a','b','c'};


class Strategy {

public:
    Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    // calculate which cache place should be used
    virtual int apply(int requestIndex)                                    = 0;

    // updates information the strategy needs
    virtual void update(int cachePlace, int requestIndex, bool cacheMiss)    = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    // calculate where to put request
    int cachePlace = strategy->apply(requestIndex);

    // proof whether its a cache hit or a cache miss
    bool isMiss = request[requestIndex] != cache[cachePlace];

    // update strategy (for example recount distances)
    strategy->update(cachePlace, requestIndex, isMiss);

    // write to cache
    cache[cachePlace] = request[requestIndex];

    return isMiss;
}

int main()
{
    Strategy* selectedStrategy[] = { new FIFO, new LIFO, new LRU, new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {
        // reset cache
        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];

        cout <<"\nStrategy: " << selectedStrategy[strat]->strategyName << endl;

        cout << "\nCache initial: (";
        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] << ",";
```

```
        cout << cache[cacheSize-1] << ")";cout << "Re

        quest";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {
            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;

            cout << "  " << request[i] << "";
            for (int l=0; l < cacheSize; ++l) cout << "  " << cache[l] << "";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "Total cache misses: " << cntMisses << endl;}


    for(int i=0; i<5; ++i) delete selectedStrategy[i];
}
```

基本思想很简单：对于每个请求，我有两个调用给我的策略：

1. apply: 该策略必须告诉调用者使用哪个页面
2. 更新：调用者使用该位置后，会告诉策略是否未命中。然后策略
   可能会更新其内部数据。例如，**LFU**策略必须更新缓存页面的命中频率，
   而**LFD**策略则必须重新计算缓存页面的距离。

现在让我们来看一下五种策略的示例实现：

先进先出（FIFO）

```
类 FIFO : 公有 策略 {
公有：
FIFO() : 策略("FIFO")
    {
        对于 (int i=0; i<缓存大小; ++i) 年龄[i] = 0;
    }

    int 应用(int 请求索引) 重写
    {
        int 最老 = 0;

        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
```



```
        cout << cache[cacheSize-1] << ")\n\n";

        cout << "Request\t";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "\t";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {
            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;

            cout << "  " << request[i] << "\t";
            for (int l=0; l < cacheSize; ++l) cout << "  " << cache[l] << "\t";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "\nTotal cache misses: " << cntMisses << endl;
    }

    for(int i=0; i<5; ++i) delete selectedStrategy[i];
}
```

The basic idea is simple: for every request I have two calls two my strategy:

1. **apply**: The strategy has to tell the caller which page to use
2. **update**: After the caller uses the place, it tells the strategy whether it was a miss or not. Then the strategy may update its internal data. The strategy **LFU** for example has to update the hit frequency for the cache pages, while the **LFD** strategy has to recalculate the distances for the cache pages.

Now lets look of example implementations for our five strategies:

FIFO

```
class FIFO : public Strategy {
public:
    FIFO() : Strategy("FIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
```

```
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            否则
age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO只需要知道页面在缓存中存在了多久（当然是相对于其他页面而言）。所以唯一要做的就是等待缺页，然后将未被淘汰的页面变得更旧。对于上面的示例，程序解决方案是：

```
策略：FIFO

缓存初始：(a,b,c)

请求    缓存 0    缓存 1    缓存 2    缓存未命中
  a         a         b         c
a         a         b         c
  d         d         b         c         x
  e         d         e         c         x
  b         d         e         b         x
  b         d         e         b
  a           a           e           b           x
  c           a           c           b           x
  f           a           c           f           x
  d           d           c           f           x
  e           d           e           f           x
  a           d           e           a           x
  f           f           e           a           x
  b           f           b           a           x
  e           f           b           e           x
  c           c           b           e           x

缓存未命中总数：13
```

这正是上面给出的解决方案。

LIFO

```
类 LIFO : 公有 策略 {
公有:
LIFO() : 策略("LIFO")
    {
        对于 (int i=0; i<缓存大小; ++i) 年龄[i] = 0;
    }

    int 应用(int 请求索引) 重写
    {
        int 最新 = 0;
```



```
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO just needs the information how long a page is in the cache (and of course only relative to the other pages). So the only thing to do is wait for a miss and then make the pages, which where not evicted older. For our example above the program solution is:

```
Strategy: FIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          d          e          c          x
  b          d          e          b          x
  b          d          e          b
  a          a          e          b          x
  c          a          c          b          x
  f          a          c          f          x
  d          d          c          f          x
  e          d          e          f          x
  a          d          e          a          x
  f          f          e          a          x
  b          f          b          a          x
  e          f          b          e          x
  c          c          b          e          x

Total cache misses: 13
```

Thats exact the solution from above.

LIFO

```
class LIFO : public Strategy {
public:
    LIFO() : Strategy("LIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int newest = 0;
```

```
        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            否则如果(年龄[i] < 年龄[最新])
              最新 = i;
        }

        返回 最新;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            否则
age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

LIFO 的实现或多或少与 **FIFO** 相同，但我们驱逐的是最新的页面，而不是最旧的页面。
程序结果如下：

```
策略: LIFO

缓存初始: (a,b,c)

请求    缓存 0    缓存 1    缓存 2    缓存未命中
  a         a         b         c
a         a         b         c
  d         d         b         c         x
  e         e         b         c         x
  b         e         b         c
b         e         b         c
  a         a         b         c         x
  c         a         b         c
f         f         b         c         x
  d         d         b         c         x
  e         e         b         c         x
  a         a         b         c         x
  f         f         b         c         x
  b         f         b         c
e         e         b         c         x
  c         e         b         c

总缓存未命中次数: 9
```

LRU


```
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] < age[newest])
                newest = i;
        }

        return newest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

The implementation of **LIFO** is more or less the same as by **FIFO** but we evict the youngest not the oldest page. The program results are:

```
Strategy: LIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          e          b          c          x
  b          e          b          c
  b          e          b          c
  a          a          b          c          x
  c          a          b          c
  f          f          b          c          x
  d          d          b          c          x
  e          e          b          c          x
  a          a          b          c          x
  f          f          b          c          x
  b          f          b          c
  e          e          b          c          x
  c          e          b          c

Total cache misses: 9
```

LRU

```
类 LRU : 公有 策略 {
公有:
LRU() : 策略("LRU")
    {
        对于 (int i=0; i<缓存大小; ++i) 年龄[i] = 0;
    }

    // 这里的 oldest 意味着最长时间未被使用
    int 应用(int 请求索引) 重写
    {
        int 最老 = 0;

        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // 所有旧页面变得更旧，被使用的页面变为0
        对于(int i=0; i<缓存大小; ++i)
        {
            if(i != cachePos)
                age[i]++;

            否则
age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

在LRU的情况下，策略与缓存页中的内容无关，其唯一关注点是最后一次使用时间。程序结果如下：

```
策略: LRU

缓存初始: (a,b,c)

请求    缓存 0    缓存 1    缓存 2    缓存未命中
  a         a         b         c
a         a         b         c
  d         a         d         c         x
  e         a         d         e         x
  b         b         d         e         x
  b         b         d         e
a         b         a         e         x
  c         b         a         c         x
  f         f         a         c         x
  d         f         d         c         x
  e         f         d         e         x
  a         a         d         e         x
```



```
class LRU : public Strategy {
public:
    LRU() : Strategy("LRU")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    // here oldest mean not used the longest
    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // all old pages get older, the used one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

In case of **LRU** the strategy is independent from what is at the cache page, its only interest is the last usage. The programm results are:

```
Strategy: LRU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          b          d          e          x
  b          b          d          e
  a          b          a          e          x
  c          b          a          c          x
  f          f          a          c          x
  d          f          d          c          x
  e          f          d          e          x
  a          a          d          e          x
```

```
  f          a          f          e          x
  b          a          f          b          x
  e          e          f          b          x
  c          e          c          b          x

总缓存未命中次数: 13
```

LFU

```
class LFU : public Strategy {
public:
LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int 应用(int 请求索引) 重写
    {
        int least = 0;

        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(cacheMiss)
requestFrequency[cachePos] = 1;

        否则
            ++requestFrequency[cachePos];
    }

private:

    // 页面被使用的频率
    int requestFrequency[cacheSize];
};
```

LFU 淘汰使用频率最低的页面。所以更新策略就是统计每次访问。当然，缺页后计数会重置。程序结果如下：

```
策略: LFU

缓存初始: (a,b,c)

请求    缓存 0    缓存 1    缓存 2    缓存未命中
  a         a         b         c
a         a         b         c
  d         a         d         c         x
  e         a         d         e         x
  b         a         b         e         x
  b         a         b         e
  a           a           b           e
```



```
  f          a          f          e          x
  b          a          f          b          x
  e          e          f          b          x
  c          e          c          b          x

Total cache misses: 13
```

LFU

```
class LFU : public Strategy {
public:
    LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int least = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(cacheMiss)
            requestFrequency[cachePos] = 1;

        else
            ++requestFrequency[cachePos];
    }

private:

    // how frequently was the page used
    int requestFrequency[cacheSize];
};
```

LFU evicts the page uses least often. So the update strategy is just to count every access. Of course after a miss the count resets. The program results are:

```
Strategy: LFU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          a          b          e          x
  b          a          b          e
  a          a          b          e
```

```
c         a         b         c         x
  f         a         b         f         x
  d         a         b         d         x
  e         a         b         e         x
  a         a         b         e
f         a         b         f         x
  b         a         b         f
   e          a          b          e          x
   c          a          b          c          x
```

总缓存未命中次数: 10

LFD

```
类 LFD : 公有 策略 {
公有:
LFD() : 策略("LFD")
    {
        // 在开始满足请求之前预先计算下一次使用
        for (int i=0; i<cacheSize; ++i) nextUse[i] = calcNextUse(-1, cache[i]);
    }

    int 应用(int 请求索引) 重写
    {
        int latest = 0;

        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(nextUse[i] > nextUse[latest])
                latest = i;
        }

        return latest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
    }

private:

    int calcNextUse(int requestPosition, char pageItem)
    {
        for(int i = requestPosition+1; i < requestLength; ++i)
        {
            if (request[i] == pageItem)
                return i;
        }

        return requestLength + 1;
    }

    // next usage of page
    int nextUse[cacheSize];
};
```

最近最远使用（LFD）策略不同于之前的所有策略。它是唯一一个使用未来请求来决定淘汰哪个页面的策略。该实现使用函数calcNextUse来获取下次使用时间最远的页面。


```
  c         a         b         c         x
  f         a         b         f         x
  d         a         b         d         x
  e         a         b         e         x
  a         a         b         e
  f         a         b         f         x
  b         a         b         f
  e         a         b         e         x
  c         a         b         c         x
```

Total cache misses: 10

LFD

```
class LFD : public Strategy {
public:
    LFD() : Strategy("LFD")
    {
        // precalc next usage before starting to fullfill requests
        for (int i=0; i<cacheSize; ++i) nextUse[i] = calcNextUse(-1, cache[i]);
    }

    int apply(int requestIndex) override
    {
        int latest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(nextUse[i] > nextUse[latest])
                latest = i;
        }

        return latest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
            nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
    }

private:

    int calcNextUse(int requestPosition, char pageItem)
    {
        for(int i = requestPosition+1; i < requestLength; ++i)
        {
            if (request[i] == pageItem)
                return i;
        }

        return requestLength + 1;
    }

    // next usage of page
    int nextUse[cacheSize];
};
```

The **LFD** strategy is different from everyone before. Its the only strategy that uses the future requests for its decission who to evict. The implementation uses the function `calcNextUse` to get the page which next use is

程序的解决方案与上面手工计算的结果相同：

```
策略: LFD

缓存初始: (a,b,c)

请求    缓存 0    缓存 1    缓存 2    缓存未命中
  a         a         b         c
a         a         b         c
  d         a         b         d         x
  e         a         b         e         x
  b         a         b         e
b         a         b         e
  a         a         b         e
  c         a         c         e         x
  f         a         f         e         x
  d         a         d         e         x
  e         a         d         e
a         a         d         e
  f         f         d         e         x
  b         b         d         e         x
  e         b         d         e
c         c         d         e         x

总缓存未命中次数: 8
```

贪心策略LFD确实是所展示的五种策略中唯一的最优策略。证明相当冗长，可以在此处找到，或者参见Jon Kleinberg和Eva Tardos的书（见下方备注中的参考资料）。

算法与现实

LFD策略是最优的，但存在一个大问题。它是一个最优的离线解决方案。在实际中，缓存通常是一个在线问题，这意味着该策略无用，因为我们无法知道下一次何时需要特定的项目。其他四种策略也是在线策略。对于在线问题，我们需要一种完全不同的通用方法。

第18.2节：售票机

第一个简单示例：

你有一台自动售票机，可以用面值为1、2、5、10和20的硬币找零。找零的过程可以看作是一系列硬币的投放，直到找出正确的金额。我们称当找零的硬币数量最少时，该找零是最优的。

令M在[1,50]内为票价T，P在[1,50]内为某人支付的T的金额，且P >= M。令D=P-M。我们将一步的benefit定义为D与D-c之间的差值，其中c是自动售货机在这一步中吐出的硬币。

兑换的Greedy Technique如下伪算法方法：

步骤1：当`D > 20`时，吐出一枚20硬币，并设置`D = D - 20`
步骤2：当`D > 10`时，吐出一枚10硬币，并设置`D = D - 10`
步骤3：当`D > 5`时，吐出一枚5硬币，并设置`D = D - 5`
步骤4：当`D > 2`时，吐出一枚2硬币，并设置`D = D - 2`
步骤5：当`D > 1`时，吐出一枚1硬币，并设置`D = D - 1`

farthest away in the future. The program solution is equal to the solution by hand from above:

```
Strategy: LFD

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          b          d          x
  e          a          b          e          x
  b          a          b          e
  b          a          b          e
  a          a          b          e
  c          a          c          e          x
  f          a          f          e          x
  d          a          d          e          x
  e          a          d          e
  a          a          d          e
  f          f          d          e          x
  b          b          d          e          x
  e          b          d          e
  c          c          d          e          x

Total cache misses: 8
```

The greedy strategy **LFD** is indeed the only optimal strategy of the five presented. The proof is rather long and can be found here or in the book by Jon Kleinberg and Eva Tardos (see sources in remarks down below).

Algorithm vs Reality

The **LFD** strategy is optimal, but there is a big problem. Its an optimal **offline** solution. In praxis caching is usually an **online** problem, that means the strategy is useless because we cannot now the next time we need a particular item. The other four strategies are also **online** strategies. For online problems we need a general different approach.

Section 18.2: Ticket automat

First simple Example:

You have a ticket automat which gives exchange in coins with values 1, 2, 5, 10 and 20. The dispension of the exchange can be seen as a series of coin drops until the right value is dispensed. We say a dispension is **optimal** when its **coin count is minimal** for its value.

Let `M` in `[1,50]` be the price for the ticket `T` and `P` in `[1,50]` the money somebody paid for `T`, with `P >= M`. Let `D=P-M`. We define the **benefit** of a step as the difference between `D` and `D-c` with `c` the coin the automat dispense in this step.

The **Greedy Technique** for the exchange is the following pseudo algorithmic approach:

Step 1: while `D > 20` dispense a 20 coin and set `D = D - 20`
Step 2: while `D > 10` dispense a 10 coin and set `D = D - 10`
Step 3: while `D > 5` dispense a 5 coin and set `D = D - 5`
Step 4: while `D > 2` dispense a 2 coin and set `D = D - 2`
Step 5: while `D > 1` dispense a 1 coin and set `D = D - 1`

之后所有硬币的总和显然等于D。这是一个greedy algorithm，因为在每一步及每次重复步骤后，收益都被最大化。我们无法吐出另一枚具有更高收益的硬币。

现在，售票机的程序（用C++编写）：

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// 读取一些硬币面值，按降序排序，
// 去除重复并保证包含1分硬币
std::vector<unsigned int> readInCoinValues();

int main()
{
std::vector<unsigned int> coinValues;    // 硬币面值数组，升序排列
    int ticketPrice;                     // 示例中的M
    int paidMoney;                       // 示例中的P

    // 生成硬币面值
coinValues = readInCoinValues();

    cout << "票价: ";
    cin >> ticketPrice;

    cout << "支付金额: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "无找零" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    // 贪心算法开始

    // 保存需要找出的硬币数量
   std::vector<unsigned int> coinCount;

    for(auto coinValue  = coinValues.begin();
             coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;

        while (diffValue >= *coinValue)
        {
diffValue -= *coinValue;
            countCoins++;
        }

coinCount.push_back(countCoins);
    }

    // 输出结果
    cout << "the difference " << paidMoney - ticketPrice
         << " is paid with: " << endl;
```


Afterwards the sum of all coins clearly equals D. Its a **greedy algorithm** because after each step and after each repitition of a step the benefit is maximized. We cannot dispense another coin with a higher benefit.

Now the ticket automat as program (in C++):

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// read some coin values, sort them descending,
// purge copies and guaratee the 1 coin is in it
std::vector<unsigned int> readInCoinValues();

int main()
{
    std::vector<unsigned int> coinValues;   // Array of coin values ascending
    int ticketPrice;                        // M in example
    int paidMoney;                          // P in example

    // generate coin values
    coinValues = readInCoinValues();

    cout << "ticket price: ";
    cin >> ticketPrice;

    cout << "money paid: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "No exchange money" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    // Here starts greedy

    // we save how many coins we have to give out
    std::vector<unsigned int> coinCount;

    for(auto coinValue  = coinValues.begin();
             coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;

        while (diffValue >= *coinValue)
        {
            diffValue -= *coinValue;
            countCoins++;
        }

        coinCount.push_back(countCoins);
    }

    // print out result
    cout << "the difference " << paidMoney - ticketPrice
         << " is paid with: " << endl;
```

```
    for(unsigned int i=0; i < coinValues.size(); ++i)
    {
        if(coinCount[i] > 0)
            cout << coinCount[i] << " coins with value "
                 << coinValues[i] << endl;
    }

    return 0;
}

std::vector<unsigned int> readInCoinValues()
{
    // 硬币面值
std::vector<unsigned int> coinValues;

    // 确保1在向量中
    coinValues.push_back(1);

    // 读取硬币面值（注意：省略错误处理）
    while(true)
    {
        int coinValue;

        cout << "硬币面值（输入小于1停止）：";
        cin >> coinValue;

        if(coinValue > 0)
coinValues.push_back(coinValue);

        否则
            break;
    }

    // 排序面值
sort(coinValues.begin(), coinValues.end(), std::greater<int>());

    // 删除相同值的副本
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    // 打印数组
    cout << "硬币面值: ";

    for(auto i : coinValues)
        cout << i << " ";

    cout << endl;

    return coinValues;
}
```

请注意，为了保持示例简单，目前没有输入检查。一个示例输出：

```
硬币面值 (<1 停止): 2
硬币面值 (<1 停止): 4
硬币面值 (<1 停止): 7
硬币面值 (<1 停止): 9
硬币面值 (<1 停止): 14
硬币面值 (<1 停止): 4
硬币面值 (<1 停止): 0
```



```
    for(unsigned int i=0; i < coinValues.size(); ++i)
    {
        if(coinCount[i] > 0)
            cout << coinCount[i] << " coins with value "
                 << coinValues[i] << endl;
    }

    return 0;
}

std::vector<unsigned int> readInCoinValues()
{
    // coin values
    std::vector<unsigned int> coinValues;

    // make sure 1 is in vectore
    coinValues.push_back(1);

    // read in coin values (attention: error handling is omitted)
    while(true)
    {
        int coinValue;

        cout << "Coin value (<1 to stop): ";
        cin >> coinValue;

        if(coinValue > 0)
            coinValues.push_back(coinValue);

        else
            break;
    }

    // sort values
    sort(coinValues.begin(), coinValues.end(), std::greater<int>());

    // erase copies of same value
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    // print array
    cout << "Coin values: ";

    for(auto i : coinValues)
        cout << i << " ";

    cout << endl;

    return coinValues;
}
```

Be aware there is now input checking to keep the example simple. One example output:

```
Coin value (<1 to stop): 2
Coin value (<1 to stop): 4
Coin value (<1 to stop): 7
Coin value (<1 to stop): 9
Coin value (<1 to stop): 14
Coin value (<1 to stop): 4
Coin value (<1 to stop): 0
```

```
硬币面值: 14 9 7 4 2 1
票价: 34
支付金额: 67
差额 33 由以下硬币支付:
2面值为 14 的硬币
1面值为 4 的硬币
1面值为1的硬币
```

只要1在我们已知的硬币面值中，算法就会终止，因为：

- D在每一步都会严格减少
- D永远不会>0且同时小于最小的面值为1的硬币

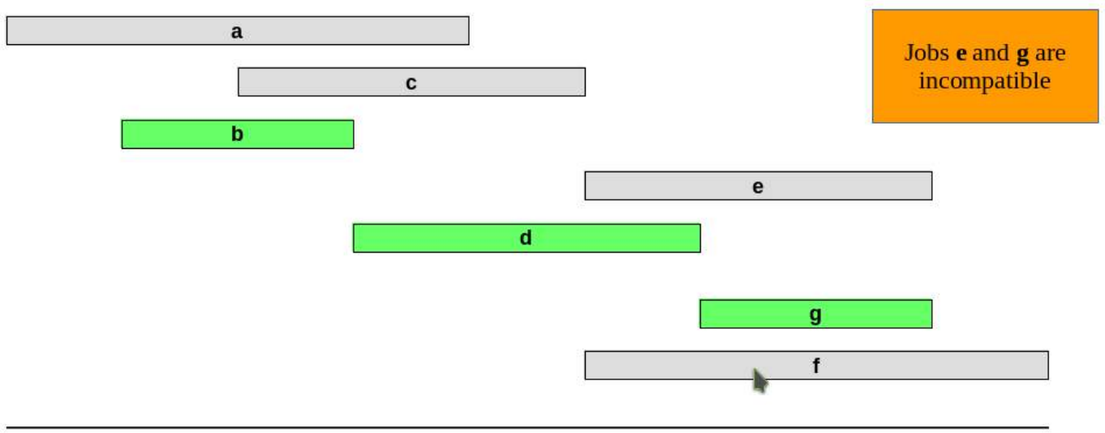
但该算法有两个陷阱：

1. 设C为最大硬币面值。只要D/C是多项式，运行时间才是多项式，因为D的表示只使用log D位，且运行时间至少与D/C成线性关系。
2. 在每一步中，我们的算法选择局部最优解。但这不足以保证算法找到全局最优解（更多信息见there或Korte和Vygen的书中）。

一个简单的反例：硬币面值为1、3、4，且D=6。最优解显然是两个面值为3的硬币，但贪心算法第一步选择了4，因此第二步和第三步必须选择1，结果不是最优解。该例的一个可能的最优算法基于动态规划。

第18.3节：区间调度

我们有一组作业J={a,b,c,d,e,f,g}。设j in J为一个作业，其开始时间为sj，结束时间为fj。两个作业如果不重叠，则称为兼容。下面是一个示例图：



目标是找到最大互相兼容的作业子集。针对这个问题，有几种贪心算法：

1. 最早开始时间：按sj的升序考虑作业
2. 最早完成时间：按fj的升序考虑作业
3. 最短区间：按fj-sj的升序考虑作业
4. 最少冲突：对于每个作业j，计算冲突作业的数量cj

现在的问题是，哪种方法真正有效。最早开始时间绝对不是，以下是一个反例


```
Coin values: 14 9 7 4 2 1
ticket price: 34
money paid: 67
the difference 33 is paid with:
2 coins with value 14
1 coins with value 4
1 coins with value 1
```

As long as 1 is in the coin values we now, that the algorithm will terminate, because:

- D strictly decreases with every step
- D is never >0 and smaller than than the smallest coin 1 at the same time

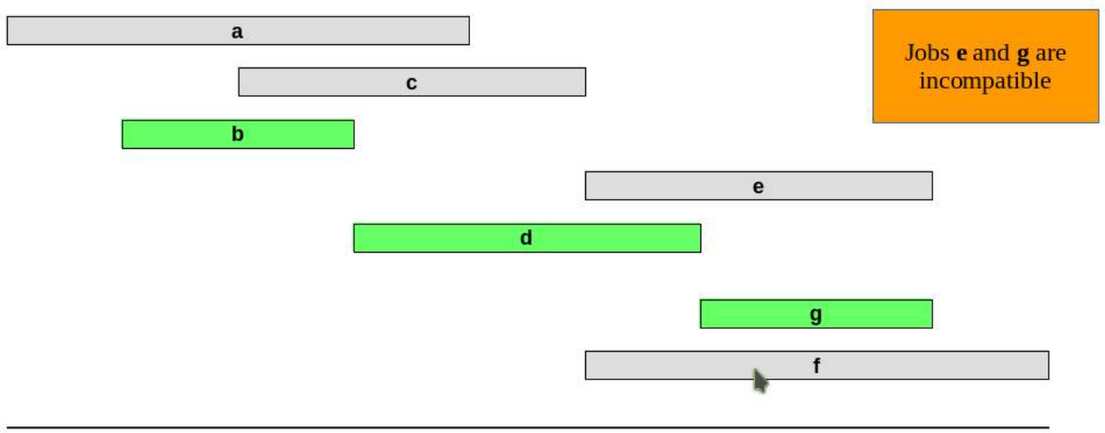
But the algorithm has two pitfalls:

1. Let C be the biggest coin value. The runtime is only polynomial as long as D/C is polynomial, because the representation of D uses only log D bits and the runtime is at least linear in D/C.
2. In every step our algorithm chooses the local optimum. But this is not sufficient to say that the algorithm finds the global optimal solution (see more information here or in the Book of Korte and Vygen).

A simple counter example: the coins are 1, 3, 4 and D=6. The optimal solution is clearly two coins of value 3 but greedy chooses 4 in the first step so it has to choose 1 in step two and three. So it gives no optimal soution. A possible optimal Algorithm for this example is based on **dynamic programming**.

Section 18.3: Interval Scheduling

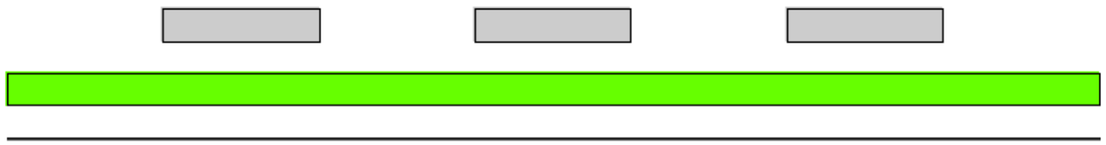
We have a set of jobs J={a,b,c,d,e,f,g}. Let j in J be a job than its start at sj and ends at fj. Two jobs are compatible if they don't overlap. A picture as example:



The goal is to find the **maximum subset of mutually compatible jobs**. There are several greedy approaches for this problem:

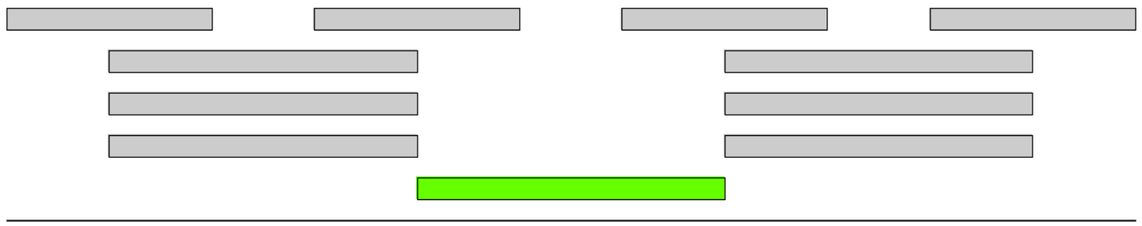
1. **Earliest start time**: Consider jobs in ascending order of sj
2. **Earliest finish time**: Consider jobs in ascending order of fj
3. **Shortest interval**: Consider jobs in ascending order of fj-sj
4. **Fewest conflicts**: For each job j, count the number of conflicting jobs cj

The question now is, which approach is really successfull. **Early start time** definetly not, here is a counter example



最短区间也不是最优的

而最少冲突听起来可能最优，但这里有一个针对该方法的问题案例：

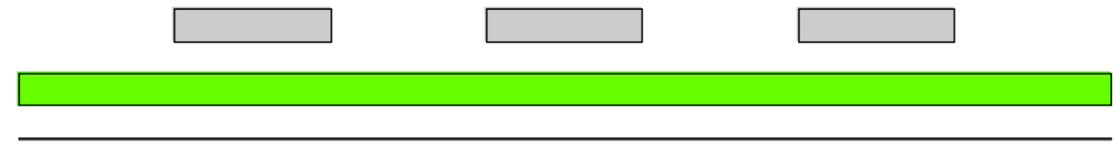


这就剩下了最早完成时间。伪代码非常简单：

1. 按完成时间排序作业，使得 f1<=f2<=...<=fn
2. 令 A 为一个空集
3. 对于 j=1 到 n，若 j 与 A 集合中的 所有 作业兼容，则 A=A+{j}
4. **A 是一个 最大互相兼容作业子集**

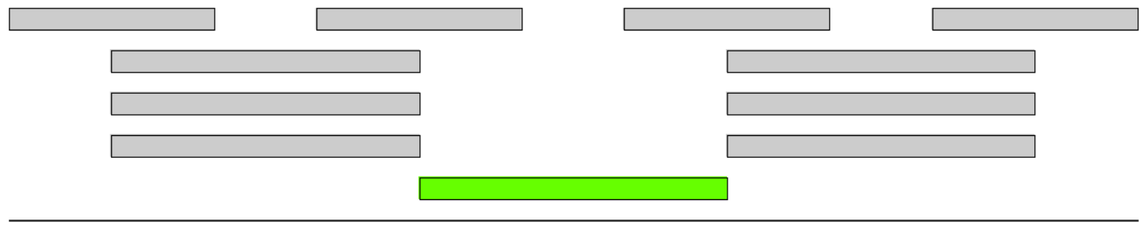
或者作为 C++ 程序：

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// 作业开始时间
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};

// 作业结束时间
const int endTimes[]   = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};

using namespace std;

int main()
{
vector<pair<int,int>> jobs;

     for(int i=0; i<jobCnt; ++i)
jobs.push_back(make_pair(startTimes[i], endTimes[i]));

    // 第一步：排序
sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                    { return p1.second < p2.second; });
```

Shortest interval is not optimal either

and **fewest conflicts** may indeed sound optimal, but here is a problem case for this approach:



Which leaves us with **earliest finish time**. The pseudo code is quiet simple:

1. Sort jobs by finish time so that `f1<=f2<=...<=fn`
2. Let `A` be an empty set
3. for `j=1` to `n` if `j` is compatible to **all** jobs in `A` set `A=A+{j}`
4. `A` is a **maximum subset of mutually compatible jobs**

Or as C++ program:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};

// Job end times
const int endTimes[]   = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(startTimes[i], endTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                        { return p1.second < p2.second; });
```

```
    // 第二步：空集合 A
    vector<int> A;

    // 第三步：
    for(int i=0; i<jobCnt; ++i)
    {
        auto job = jobs[i];
        bool isCompatible = true;

        for(auto jobIndex : A)
        {
            // 测试实际任务和来自 A 的任务是否不兼容
            if(job.second >= jobs[jobIndex].first &&
job.first  <= jobs[jobIndex].second)
            {
isCompatible = false;
                break;
            }
        }

        if(isCompatible)
A.push_back(i);
    }

    // 第4步：打印 A
    cout << "Compatible: ";

    for(auto i : A)
        cout << "(" << jobs[i].first << "," << jobs[i].second << ") ";
    cout << endl;

    return 0;
}
```

该示例的输出为：Compatible: (1,3) (4,5) (6,8) (9,10)该算法的实现显然是 Θ(n^2)

。也有一个 Θ(n log n) 的实现，有兴趣的读者可以继续阅读下面的（Java 示例）。

现在我们有了一个用于区间调度问题的贪心算法，但它是最优的吗？

命题： 贪心算法 最早完成时间 是最优的。

证明：（反证法）

假设贪心算法不是最优的，i1,i2,...,ik表示贪心算法选择的作业集合。令j1,j2,...,jm表示一个最优解中的作业集合，且对于最大的可能值r，有i1=j1,i2=j2,...,ir=jr。

作业i(r+1)存在且在作业j(r+1)之前完成（最早完成）。但那么，j1,j2,...,jr,i(r+1),j(r+2),...,jm也是一个最优解，并且对于所有k在[1,(r+1)]中，有jk=ik。这与r的最大性矛盾。由此完成证明。

第二个例子表明，通常存在许多可能的贪心策略，但只有部分甚至没有策略能在每个实例中找到最优解。

下面是一个运行时间为Θ(n log n)的Java程序

```
import java.util.Arrays;
```



```
// step 2: empty set A
vector<int> A;

// step 3:
for(int i=0; i<jobCnt; ++i)
{
    auto job = jobs[i];
    bool isCompatible = true;

    for(auto jobIndex : A)
    {
        // test whether the actual job and the job from A are incompatible
        if(job.second >= jobs[jobIndex].first &&
           job.first  <= jobs[jobIndex].second)
        {
            isCompatible = false;
            break;
        }
    }

    if(isCompatible)
        A.push_back(i);
}

//step 4: print A
cout << "Compatible: ";

for(auto i : A)
    cout << "(" << jobs[i].first << "," << jobs[i].second << ") ";
cout << endl;

return 0;
}
```

The output for this example is: `Compatible: (1,3) (4,5) (6,8) (9,10)`

The implementation of the algorithm is clearly in Θ(n^2). There is a Θ(n log n) implementation and the interested reader may continue reading below (Java Example).

Now we have a greedy algorithm for the interval scheduling problem, but is it optimal?

Proposition: The greedy algorithm **earliest finish time** is optimal.

Proof:*(by contradiction)*

Assume greedy is not optimal and `i1,i2,...,ik` denote the set of jobs selected by greedy. Let `j1,j2,...,jm` denote the set of jobs in an **optimal** solution with `i1=j1,i2=j2,...,ir=jr` for the **largest possible** value of `r`.

The job `i(r+1)` exists and finishes before `j(r+1)` (earliest finish). But than is `j1,j2,...,jr,i(r+1),j(r+2),...,jm` also a **optimal** solution and for all `k` in `[1,(r+1)]` is `jk=ik`. thats a **contradiction** to the maximality of `r`. This concludes the proof.

This second example demonstrates that there are usually many possible greedy strategies but only some or even none might find the optimal solution in every instance.

Below is a Java program that runs in Θ(n log n)

```
import java.util.Arrays;
```

```java
import java.util.Comparator;

class Job
{
    int start, finish, profit;

Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}


class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}

public class WeightedIntervalScheduling
{
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;

        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                否则
                    return mid;
            }
            否则
hi = mid - 1;
        }

        return -1;
    }

    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());

        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;

        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
inclProf += table[l];

            table[i] = Math.max(inclProf, table[i-1]);
```



```java
import java.util.Comparator;

class Job
{
    int start, finish, profit;

    Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}


class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}

public class WeightedIntervalScheduling
{
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;

        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                else
                    return mid;
            }
            else
                hi = mid - 1;
        }

        return -1;
    }

    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());

        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;

        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
                inclProf += table[l];

            table[i] = Math.max(inclProf, table[i-1]);
```

```
        }

        return table[n-1];
    }

    public static void main(String[] args)
    {
Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                    new Job(6, 19, 100), new Job(2, 100, 200)};

        System.out.println("Optimal profit is " + schedule(jobs));
    }
}
```

预期输出为：

```
Optimal profit is 250
```

第18.4节：最小化延迟

有许多最小化延迟的问题，这里我们有一个单一资源，该资源一次只能处理一个作业。作业j需要j个处理时间单位，截止时间为dj。如果j在时间sj开始，则完成时间为fj=sj+j。我们定义延迟L=max{0,fj-dj}对所有j。目标是最小化**最大延迟**L。

```
   1 2 3 4 5  6
tj 3 2 1 432
dj 6 8 9 9 10 11
 作业3 2 2 5 5 5 4 4 4411166
时间 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 Lj-8-5          -4        1         7      4
```

解L=7显然不是最优的。让我们看看一些贪心策略：

1. 最短处理时间优先：按处理时间j`的升序安排作业
2. 最早截止时间优先: 按截止时间 dj升序安排作业
3. 最小松弛时间: 按松弛时间 dj-tj升序安排作业

很容易看出最短处理时间优先并非最优，一个很好的反例是

```
   1  2
tj1  5
dj10 5
```

最小松弛时间方案也有类似的问题

```
   1 2
tj1 5
dj3 5
```

最后一种策略看起来有效，所以我们从一些伪代码开始：

1. 将 n个作业按截止时间排序，使得 d1<=d2<=...<=dn
2. 设置 t=0
3. 对于 j=1 到 n
    - 将作业 j 分配到区间 [t,t+tj]


```
        }

        return table[n-1];
    }

    public static void main(String[] args)
    {
        Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                    new Job(6, 19, 100), new Job(2, 100, 200)};

        System.out.println("Optimal profit is " + schedule(jobs));
    }
}
```

And the expected output is:

```
Optimal profit is 250
```

Section 18.4: Minimizing Lateness

There are numerous problems minimizing lateness, here we have a single resource which can only process one job at a time. Job `j` requires `tj` units of processing time and is due at time `dj`. if `j` starts at time `sj` it will finish at time `fj=sj+tj`. We define lateness `L=max{0,fj-dh}` for all `j`. The goal is to minimize the **maximum lateness** L.

```
   1 2 3 4 5  6
tj 3 2 1 4 3  2
dj 6 8 9 9 10 11
 Job  3 2 2 5 5 5 4 4 4 4  1  1  1  6  6
Time 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 Lj  -8  -5    -4        1         7      4
```

The solution `L=7` is obviously not optimal. Lets look at some greedy strategies:

1. **Shortest processing time first**: schedule jobs in ascending order og processing time j`
2. **Earliest deadline first**: Schedule jobs in ascending order of deadline `dj`
3. **Smallest slack**: schedule jobs in ascending order of slack `dj-tj`

Its easy to see that **shortest processing time first** is not optimal a good counter example is

```
   1  2
tj  1  5
dj 10 5
```

the **smallest stack** solution has simillar problems

```
   1 2
tj 1 5
dj 3 5
```

the last strategy looks valid so we start with some pseudo code:

1. Sort `n` jobs by due time so that `d1<=d2<=...<=dn`
2. Set `t=0`
3. for `j=1` to `n`
    - Assign job `j` to interval `[t,t+tj]`

- 设置 `sj=t` 和 `fj=t+tj`
- 设置 `t=t+tj`

4. 返回区间 [s1,f1],[s2,f2],...,[sn,fn]

以及 C++ 中的实现：

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// 作业开始时间
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};

// 作业结束时间
const int dueTimes[]     = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};

using namespace std;

int main()
{
vector<pair<int,int>> jobs;

     for(int i=0; i<jobCnt; ++i)
jobs.push_back(make_pair(processTimes[i], dueTimes[i]));

    // 第一步：排序
sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                  { return p1.second < p2.second; });

    // 第2步：设置 t=0
    int t = 0;

    // 第三步：
vector<pair<int,int>> jobIntervals;

    for(int i=0; i<jobCnt; ++i)
    {
jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }

    //步骤4：打印区间cout << "区
    间:" << endl;int 延迟 = 0;



    for(int i=0; i<jobCnt; ++i)
    {
        auto 对 = 任务区间[i];

延迟 = max(延迟, 对.second-任务[i].second);

        cout << "(" << 对.first << "," << 对.second << ") "
             << "延迟: " << 对.second-任务[i].second << std::endl;
    }

    cout << "最大延迟是 " << 延迟 << endl;
```


- set `sj=t` and `fj=t+tj`
- set `t=t+tj`

4. return intervals `[s1,f1],[s2,f2],...,[sn,fn]`

And as implementation in C++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};

// Job end times
const int dueTimes[]     = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(processTimes[i], dueTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                  { return p1.second < p2.second; });

    // step 2: set t=0
    int t = 0;

    // step 3:
    vector<pair<int,int>> jobIntervals;

    for(int i=0; i<jobCnt; ++i)
    {
        jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }

    //step 4: print intervals
    cout << "Intervals:\n" << endl;

    int lateness = 0;

    for(int i=0; i<jobCnt; ++i)
    {
        auto pair = jobIntervals[i];

        lateness = max(lateness, pair.second-jobs[i].second);

        cout << "(" << pair.first << "," << pair.second << ") "
             << "Lateness: " << pair.second-jobs[i].second << std::endl;
    }

    cout << "\nmaximal lateness is " << lateness << endl;
```

```
    return 0;
}
```

该程序的输出是：

```
区间:

(0,2)   延迟:-2
(2,5)   延迟:-2
(5,8)   延迟: 0
(8,9)   延迟: 0
(9,12)  延迟: 3
(12,17) 延迟: 6
(17,21) 延迟: 8
(21,23) 延迟: 6
(23,25) 延迟: 3
(25,26) 延迟: 1

最大延迟是 8
```

该算法的运行时间显然是 Θ(n log n)，因为排序是该算法的主导操作。
现在我们需要证明它是最优的。显然，一个最优的调度没有空闲时间。最早截止时间优先（earliest deadline first）调度也没有空闲时间。

假设作业编号为d1<=d2<=...<=dn。我们称调度中的逆序为一对作业i和j，满足i<j但j排在i之前。根据定义，最早截止时间优先调度没有逆序。当然，如果一个调度有逆序，那么必定存在一对相邻的逆序作业。

命题：交换两个相邻的逆序作业会使逆序数减少1，且不会增加最大延迟时间。

证明：设L为交换前的延迟时间，M为交换后的延迟时间。由于交换两个相邻作业不会改变其他作业的位置，因此对于所有k != i,j，有Lk=Mk。

显然Mi<=Li，因为作业i被提前调度。如果作业j迟到，则根据定义有：

```
Mj = fi-dj      (定义)
                <= fi-di      (因为交换了i和j)
                <= Li
```

这意味着交换后的延迟时间小于或等于交换前的延迟时间。证毕。

命题：最早截止时间优先调度S是最优的。

证明：（反证法）

假设S*是具有最少逆序数的最优调度。我们可以假设S*没有空闲时间。如果S*没有逆序，则S=S*，证明完成。如果S*有逆序，则必有相邻逆序。上一命题表明，我们可以交换相邻逆序作业，且不会增加延迟时间，但会减少逆序数。
这与S*的定义矛盾。

最小化延迟问题及其密切相关的最小完工时间问题，即寻找最短调度时间的问题，在现实世界中有广泛应用。但通常你不只有一台机器，而是多台机器，它们以不同速度处理相同任务。这些问题很快变成NP完全问题。


```
    return 0;
}
```

And the output for this program is:

```
Intervals:

(0,2)   Lateness:-2
(2,5)   Lateness:-2
(5,8)   Lateness: 0
(8,9)   Lateness: 0
(9,12)  Lateness: 3
(12,17) Lateness: 6
(17,21) Lateness: 8
(21,23) Lateness: 6
(23,25) Lateness: 3
(25,26) Lateness: 1

maximal lateness is 8
```

The runtime of the algorithm is obviously Θ(n log n) because sorting is the dominating operation of this algorithm. Now we need to show that it is optimal. Clearly an optimal schedule has no **idle time**. the **earliest deadline first** schedule has also no idle time.

Lets assume the jobs are numbered so that `d1<=d2<=...<=dn`. We say a **inversion** of a schedule is a pair of jobs `i` and `j` so that `i<j` but j is scheduled before `i`. Due to its definition the **earliest deadline first** schedule has no inversions. Of course if a schedule has an inversion it has one with a pair of inverted jobs scheduled consecutively.

Proposition: Swapping two adjacent, inverted jobs reduces the number of inversions by **one** and **does not increase** the maximal lateness.

Proof: Let `L` be the lateness before the swap and `M` the lateness afterwards. Because exchanging two adjacent jobs does not move the other jobs from their position it is `Lk=Mk` for all `k != i,j`.

Clearly it is `Mi<=Li` since job `i` got scheduled earlier. if job `j` is late, so follows from the definition:

```
                    Mj = fi-dj    (definition)
                      <= fi-di    (since i and j are exchanged)
                      <= Li
```

That means the lateness after swap is less or equal than before. This concludes the proof.

Proposition: The **earliest deadline first** schedule `S` is optimal.

Proof:*(by contradiction)*

Lets assume `S*` is optimal schedule with the **fewest possible** number of inversions. we can assume that `S*` has no idle time. If `S*` has no inversions, then `S=S*` and we are done. If `S*` has an inversion, than it has an adjacent inversion. The last Proposition states that we can swap the adjacent inversion without increasing lateness but with decreasing the number of inversions. This contradicts the definition of `S*`.

The minimizing lateness problem and its near related **minimum makespan** problem, where the question for a minimal schedule is asked have lots of applications in the real world. But usually you don't have only one machine but many and they handle the same task at different rates. These problems get NP-complete really fast.

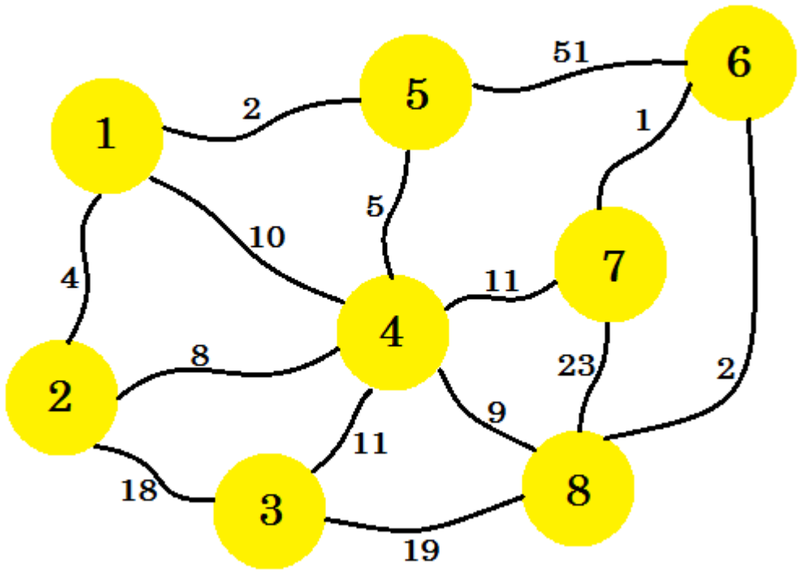
如果我们不考虑离线问题，即我们手头有所有任务和数据，而是考虑在线变体，即任务在执行过程中出现，那么另一个有趣的问题就出现了。

Another interesting question arises if we don't look at the **offline** problem, where we have all tasks and data at hand but at the **online** variant, where tasks appear during execution.

第19章：普里姆算法

第19.1节：普里姆算法简介

假设我们有8栋房子。我们想在这些房子之间架设电话线。房子之间的边表示在两栋房子之间架设线路的成本。



我们的任务是以一种方式架设线路，使所有房子都连接起来，并且整个连接的成本最小。那我们如何找到这个方案呢？我们可以使用普里姆算法。

普里姆算法是一种贪心算法，用于找到加权无向图的最小生成树。这意味着它找到一组边的子集，形成一个包含每个节点的树，且树中所有边的总权重最小。该算法由捷克数学家沃伊切赫·雅尔尼克（Vojtěch Jarník）于1930年开发，后来由计算机科学家罗伯特·克莱·普里姆（Robert Clay Prim）于1957年和埃兹格·韦布·戴克斯特拉（Edsger Wybe Dijkstra）于1959年重新发现并发表。它也被称为DJP算法、雅尔尼克算法、普里姆-雅尔尼克算法或普里姆-戴克斯特拉算法。

现在让我们先来看一些技术术语。如果我们用无向图G的一些节点和边创建一个图S，那么S称为图G的子图。只有当满足以下条件时，S才称为生成树：

- 它包含G的所有节点。
- 它是一棵树，也就是说没有环且所有节点都相连。
- 树中有(n-1)条边，其中n是G中的节点数。

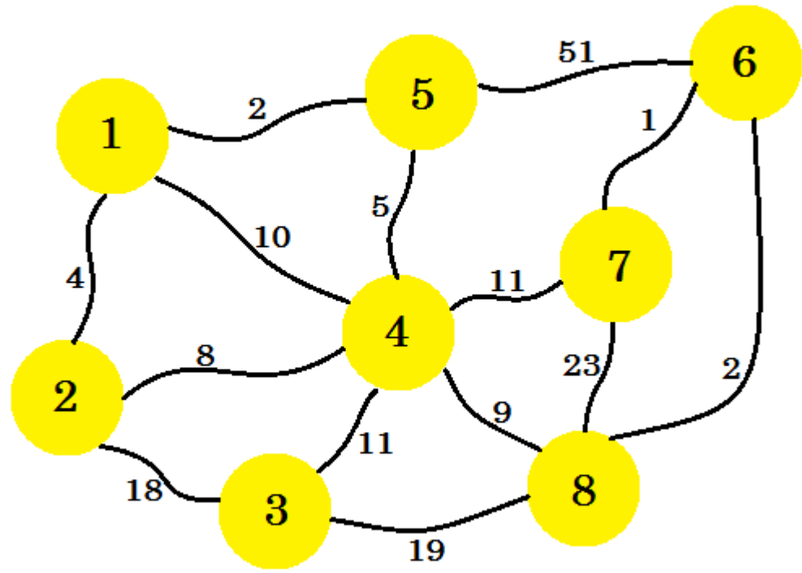
一个图可以有许多生成树。加权无向图的最小生成树是一棵树，使得边的权重之和最小。现在我们将使用普里姆算法来找出
最小生成树，也就是如何在我们的示例图中设置电话线路，使得安装成本最低。

首先我们将选择一个**源**节点。假设**节点-1**是我们的**源**。现在我们将从**节点-1**添加具有
最小代价的边到我们的子图中。在这里，我们用颜色**蓝色**标记子图中的边。这里**1-5**是

Chapter 19: Prim's Algorithm

Section 19.1: Introduction To Prim's Algorithm

Let's say we have **8** houses. We want to setup telephone lines between these houses. The edge between the houses represent the cost of setting line between two houses.



Our task is to set up lines in such a way that all the houses are connected and the cost of setting up the whole connection is minimum. Now how do we find that out? We can use **Prim's Algorithm**.

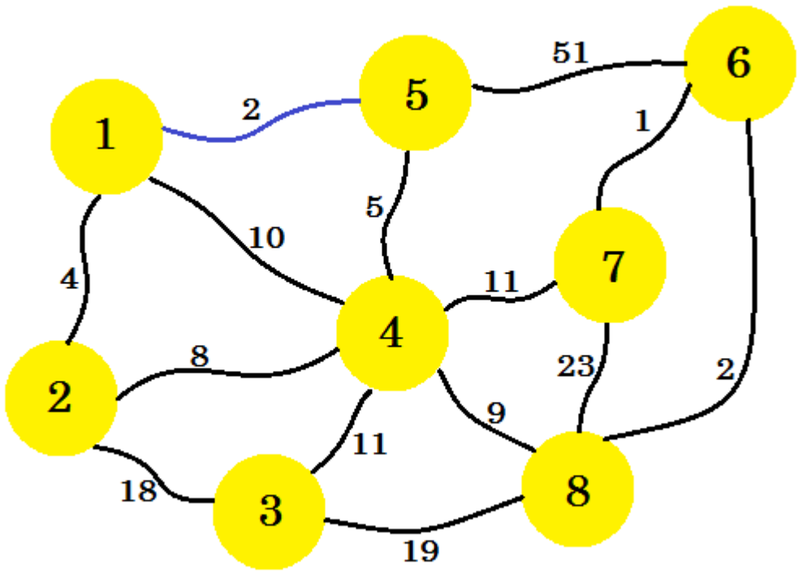
Prim's Algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every node, where the total weight of all the edges in the tree are minimized. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientist Robert Clay Prim in 1957 and Edsger Wybe Dijkstra in 1959. It is also known as **DJP algorithm**, **Jarnik's algorithm**, **Prim-Jarnik algorithm** or **Prim-Dijsktra algorithm**.

Now let's look at the technical terms first. If we create a graph, **S** using some nodes and edges of an undirected graph **G**, then **S** is called a **subgraph** of the graph **G**. Now **S** will be called a **Spanning Tree** if and only if:

- It contains all the nodes of **G**.
- It is a tree, that means there is no cycle and all the nodes are connected.
- There are **(n-1)** edges in the tree, where **n** is the number of nodes in **G**.

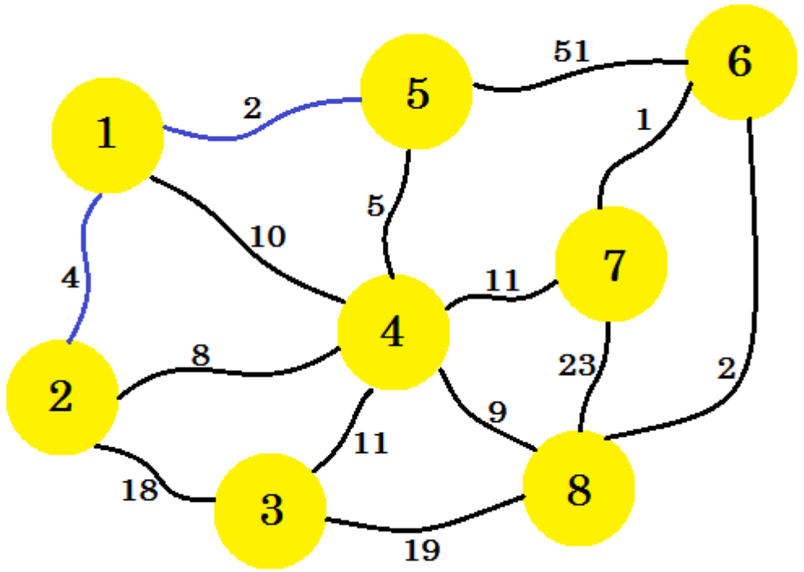
There can be many **Spanning Tree**'s of a graph. The **Minimum Spanning Tree** of a weighted undirected graph is a tree, such that sum of the weight of the edges is minimum. Now we'll use **Prim's algorithm** to find out the minimum spanning tree, that is how to set up the telephone lines in our example graph in such way that the cost of set up is minimum.

At first we'll select a **source** node. Let's say, **node-1** is our **source**. Now we'll add the edge from **node-1** that has the minimum cost to our subgraph. Here we mark the edges that are in the subgraph using the color **blue**. Here **1-5** is



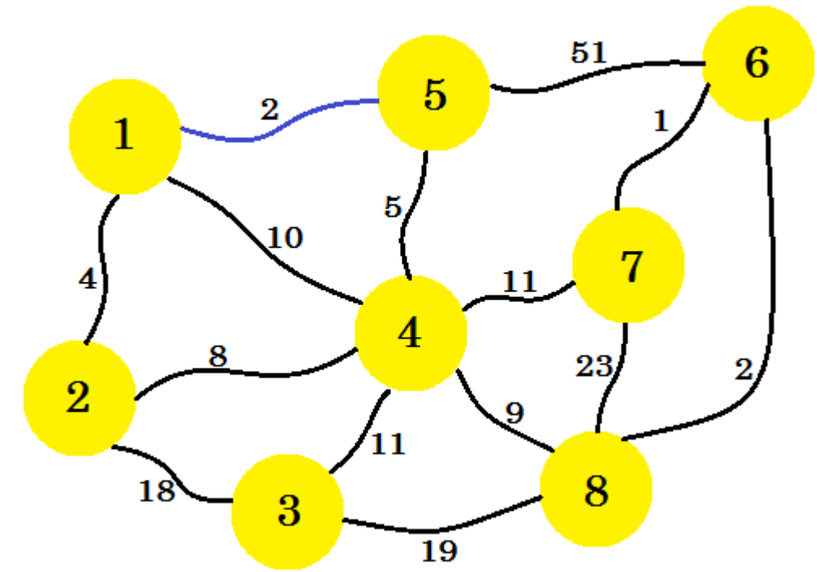
我们期望的优势。

现在我们考虑来自节点-1和节点-5的所有边，并取最小值。由于1-5已经被标记，我们



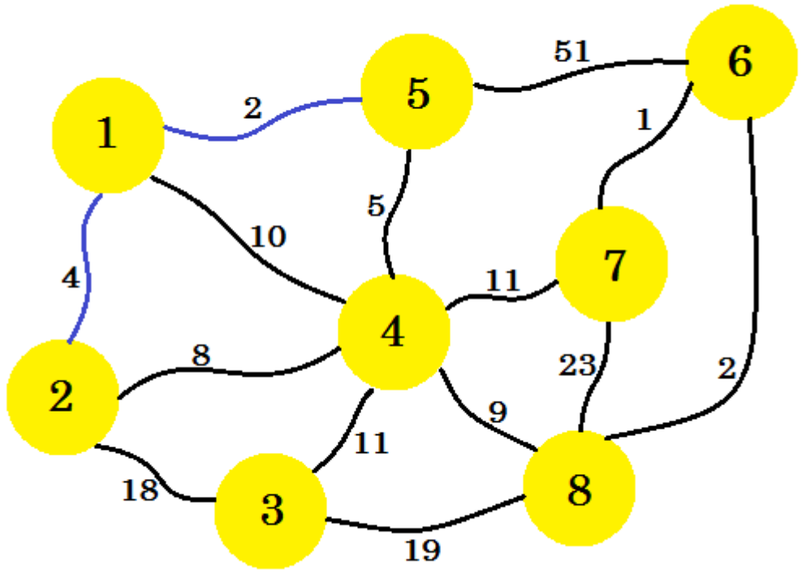
采取 1-2。

这次，我们考虑节点-1、节点-2和节点-5，并取最小边5-4。



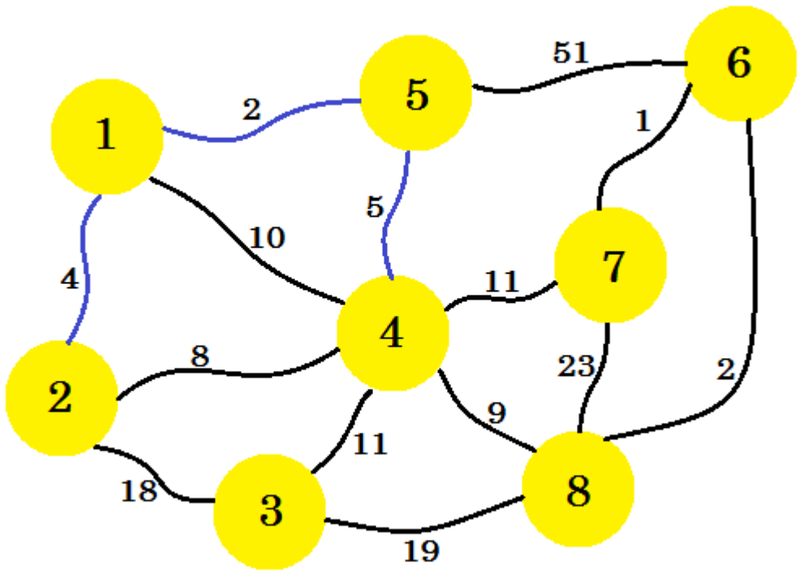
our desired edge.

Now we consider all the edges from **node-1** and **node-5** and take the minimum. Since **1-5** is already marked, we

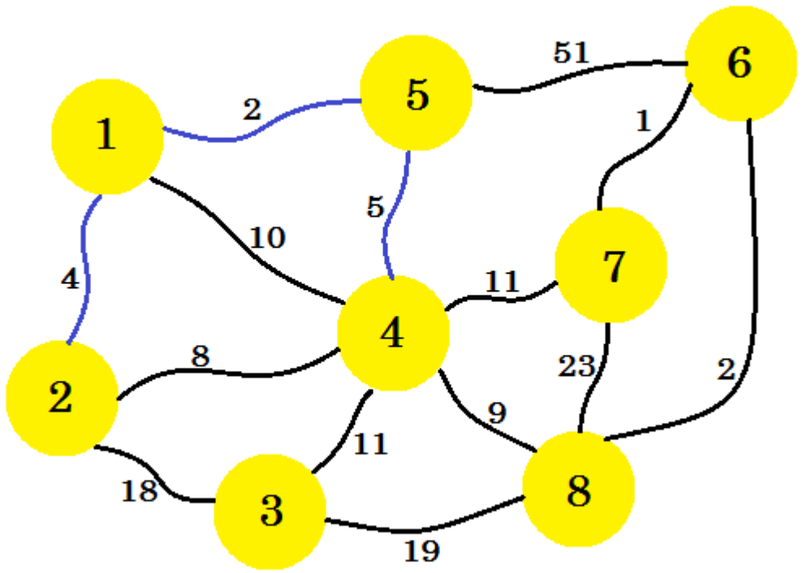


take **1-2**.

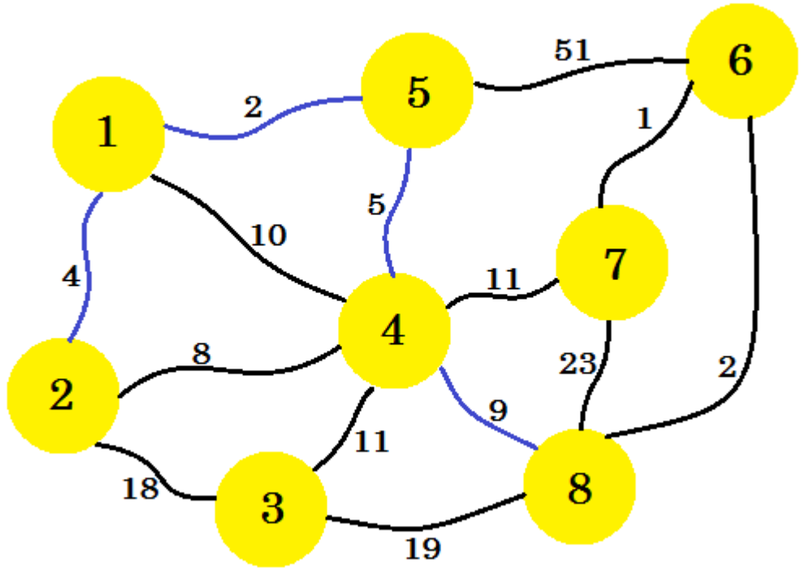
This time, we consider **node-1**, **node-2** and **node-5** and take the minimum edge which is **5-4**.



下一步很重要。从**节点-1**、**节点-2**、**节点-5**和**节点-4**中，最小的边是**2-4**。但如果我们选择那条边，它会在我们的子图中形成一个环。这是因为**节点-2**和**节点-4**已经在我们的子图中。所以选择边**2-4**对我们没有好处。*我们将以添加新节点到子图中的方式选择边*。所以我们

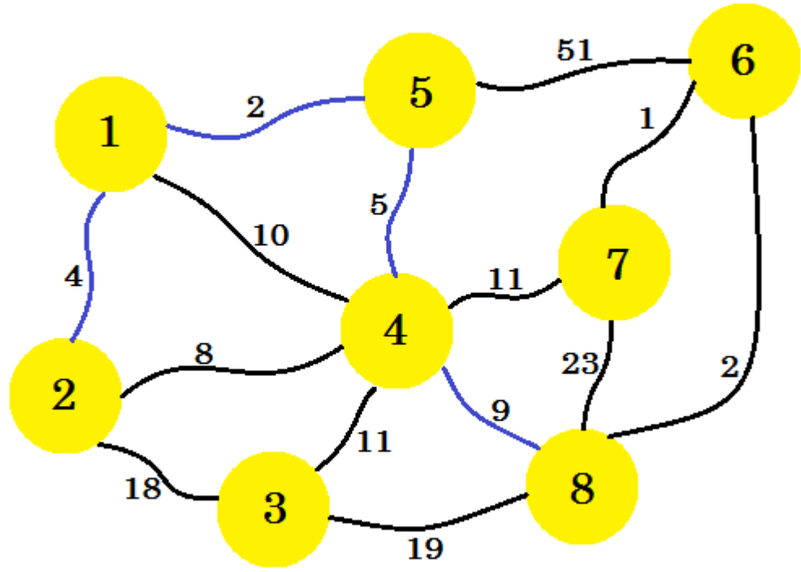


The next step is important. From **node-1**, **node-2**, **node-5** and **node-4**, the minimum edge is **2-4**. But if we select that one, it'll create a cycle in our subgraph. This is because **node-2** and **node-4** are already in our subgraph. So taking edge **2-4** doesn't benefit us. *We'll select the edges in such way that it adds a new node in our subgraph*. So we



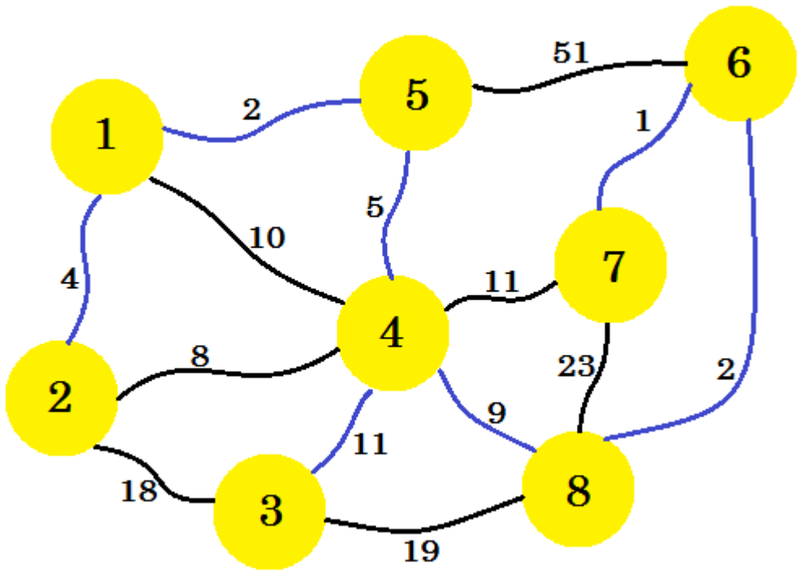
选择边缘4-8.

如果我们继续这样做，我们将选择边8-6、6-7和4-3。我们的子图将如下所示：

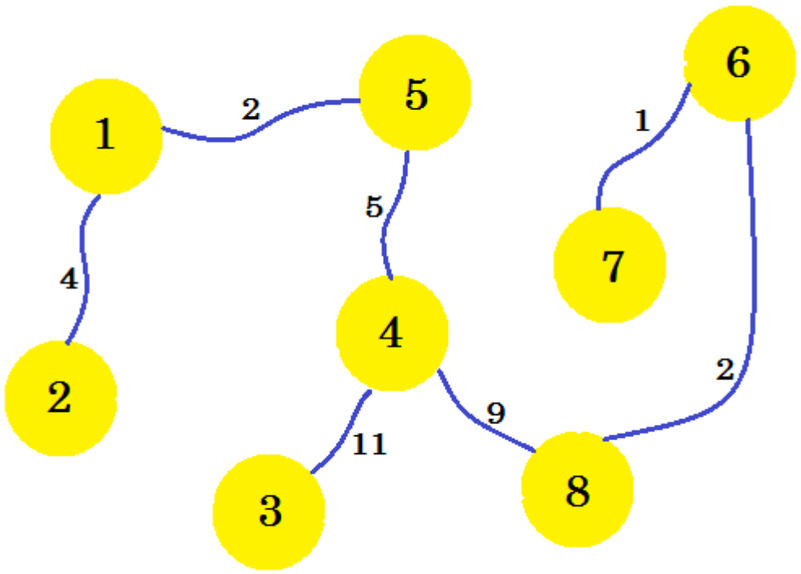


select edge **4-8**.

If we continue this way, we'll select edge **8-6**, **6-7** and **4-3**. Our subgraph will look like:



这是我们期望的子图，它将给我们最小生成树。如果我们移除未选择的边

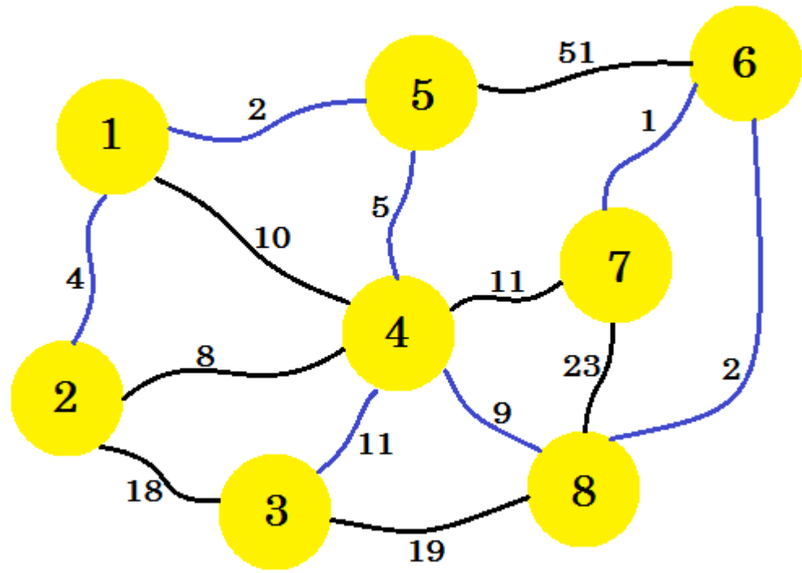


，我们将得到：

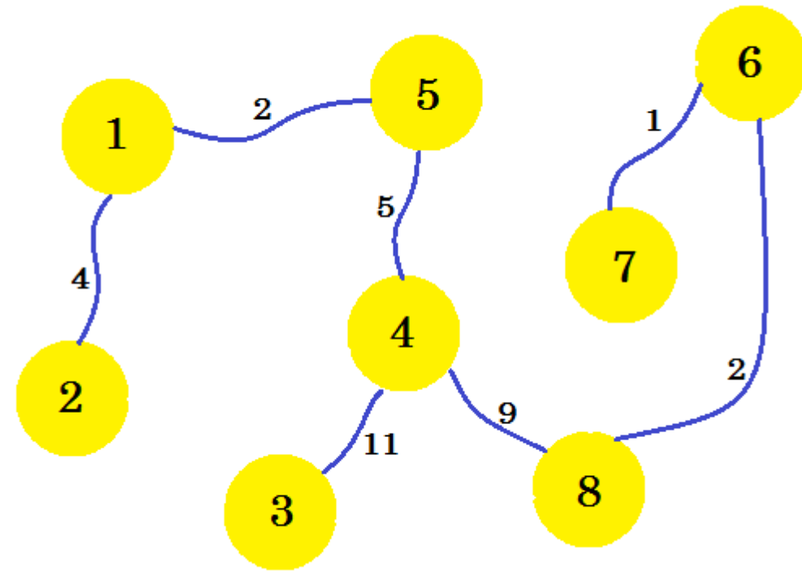
这是我们的最小生成树（MST）。因此建立电话连接的成本是：4 + 2 + 5 + 11 + 9 + 2 + 1 = 34。图中显示了房屋及其连接。一个图可能有多个MST，取决于我们选择的源节点。

算法的伪代码如下：

```
过程 PrimsMST(Graph):      // 这里Graph是一个非空连通加权图
Vnew[] = {x}                    // 新子图Vnew，包含源节点x
```

This is our desired subgraph, that'll give us the minimum spanning tree. If we remove the edges that we didn't



select, we'll get:

This is our **minimum spanning tree** (MST). So the cost of setting up the telephone connections is: **4** + **2** + **5** + **11** + **9** + **2** + **1** = **34**. And the set of houses and their connections are shown in the graph. There can be multiple **MST** of a graph. It depends on the **source** node we choose.

The pseudo-code of the algorithm is given below:

```
Procedure PrimsMST(Graph):     // here Graph is a non-empty connected weighted graph
Vnew[] = {x}                   // New subgraph Vnew with source node x
```

```
Enew[] = {}
while Vnew 不等于 V
u -> Vnew 中的一个节点
v -> 一个不在 Vnew 中且边 u-v 具有最小代价的节点
                                    // 如果两个节点权重相同，任选其一
将 v 添加到 Vnew
    将边 (u, v) 添加到 Enew
end while
返回 Vnew 和 Enew
```

复杂度：

上述朴素方法的时间复杂度为 O(V²)。它使用邻接矩阵。我们可以使用优先队列来降低复杂度。当我们将新节点添加到 Vnew 时，可以将其相邻的边加入优先队列。然后从中弹出权重最小的边。这样复杂度将变为：O(ElogE)，其中 E 是边的数量。
同样，可以构建二叉堆将复杂度降低到 O(ElogV)。

使用优先队列的伪代码如下：

```
过程 MSTPrim(图, 源点):
对于 V 中的每个 u
    key[u] := 无穷大
    parent[u] := 空
结束 对于
key[source] := 0
Q = Priority_Queue()
Q = V
while Q 不为空
   u -> Q.pop
    for 每个与 i 相邻的 v
        if v 属于 Q 且 Edge(u,v) < key[v]     // 这里 Edge(u, v) 表示
                                              // 边(u, v)的代价
parent[v] := u
            key[v] := Edge(u, v)
        end if
end for
end while
```

这里 **key[]** 存储遍历 **node-v** 的最小代价。 **parent[]** 用于存储父节点。它对于遍历和打印树结构很有用。

下面是一个简单的 Java 程序：

```
import java.util.*;

public class Graph
{
   private static int infinite = 9999999;
   int[][]  LinkCost;
   int      NNodes;
   Graph(int[][] mat)
   {
      int i, j;
NNodes = mat.length;
      LinkCost = new int[NNodes][NNodes];
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
         {
```



```
Enew[] = {}
while Vnew is not equal to V
    u -> a node from Vnew
    v -> a node that is not in Vnew such that edge u-v has the minimum cost
                                    // if two nodes have same weight, pick any of them
    add v to Vnew
    add edge (u, v) to Enew
end while
Return Vnew and Enew
```

Complexity:

Time complexity of the above naive approach is **O(V²)**. It uses adjacency matrix. We can reduce the complexity using priority queue. When we add a new node to **Vnew**, we can add its adjacent edges in the priority queue. Then pop the minimum weighted edge from it. Then the complexity will be: **O(ElogE)**, where **E** is the number of edges. Again a Binary Heap can be constructed to reduce the complexity to **O(ElogV)**.

The pseudo-code using Priority Queue is given below:

```
Procedure MSTPrim(Graph, source):
for each u in V
    key[u] := inf
    parent[u] := NULL
end for
key[source] := 0
Q = Priority_Queue()
Q = V
while Q is not empty
    u -> Q.pop
    for each v adjacent to i
        if v belongs to Q and Edge(u,v) < key[v]     // here Edge(u, v) represents
                                                     // cost of edge(u, v)
            parent[v] := u
            key[v] := Edge(u, v)
        end if
    end for
end while
```

Here **key[]** stores the minimum cost of traversing **node-v**. **parent[]** is used to store the parent node. It is useful for traversing and printing the tree.

Below is a simple program in Java:

```
import java.util.*;

public class Graph
{
   private static int infinite = 9999999;
   int[][]  LinkCost;
   int      NNodes;
   Graph(int[][] mat)
   {
      int i, j;
      NNodes = mat.length;
      LinkCost = new int[NNodes][NNodes];
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
         {
```

```
LinkCost[i][j] = mat[i][j];
            if ( LinkCost[i][j] == 0 )
               LinkCost[i][j] = infinite;
         }
      }
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
            if ( LinkCost[i][j] < infinite )
               System.out.print( " " + LinkCost[i][j] + " " );
            否则
               System.out.print(" * " );
         System.out.println();
      }
   }
   public int unReached(boolean[] r)
   {
      boolean done = true;
      for ( int i = 0; i < r.length; i++ )
         if ( r[i] == false )
            return i;
      return -1;
   }
   public void Prim( )
   {
      int i, j, k, x, y;
      boolean[] Reached = new boolean[NNodes];
      int[] predNode = new int[NNodes];
Reached[0] = true;
      for ( k = 1; k < NNodes; k++ )
      {
Reached[k] = false;
      }
predNode[0] = 0;
      printReachSet( Reached );
      for (k = 1; k < NNodes; k++)
      {
x = y = 0;
         for ( i = 0; i < NNodes; i++ )
            for ( j = 0; j < NNodes; j++ )
            {
                if ( Reached[i] && !Reached[j] &&
                     LinkCost[i][j] < LinkCost[x][y] )
                {
x = i;
                   y = j;
                }
            }
         System.out.println("最小代价边: (" +
                                 +x + "," +
                                 +y + ")" +
                                 "代价 = " + LinkCost[x][y]);
        predNode[y] = x;
Reached[y] = true;
printReachSet( Reached );
         System.out.println();
      }
      int[] a= predNode;
   for ( i = 0; i < NNodes; i++ )
         System.out.println( a[i] + " --> " + i );
   }
   void printReachSet(boolean[] Reached )
```



```
            LinkCost[i][j] = mat[i][j];
            if ( LinkCost[i][j] == 0 )
               LinkCost[i][j] = infinite;
         }
      }
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
            if ( LinkCost[i][j] < infinite )
               System.out.print( " " + LinkCost[i][j] + " " );
            else
               System.out.print(" * " );
         System.out.println();
      }
   }
   public int unReached(boolean[] r)
   {
      boolean done = true;
      for ( int i = 0; i < r.length; i++ )
         if ( r[i] == false )
            return i;
      return -1;
   }
   public void Prim( )
   {
      int i, j, k, x, y;
      boolean[] Reached = new boolean[NNodes];
      int[] predNode = new int[NNodes];
      Reached[0] = true;
      for ( k = 1; k < NNodes; k++ )
      {
         Reached[k] = false;
      }
      predNode[0] = 0;
      printReachSet( Reached );
      for (k = 1; k < NNodes; k++)
      {
         x = y = 0;
         for ( i = 0; i < NNodes; i++ )
            for ( j = 0; j < NNodes; j++ )
            {
                if ( Reached[i] && !Reached[j] &&
                     LinkCost[i][j] < LinkCost[x][y] )
                {
                   x = i;
                   y = j;
                }
            }
         System.out.println("Min cost edge: (" +
                                 + x + "," +
                                 + y + ")" +
                                 "cost = " + LinkCost[x][y]);
         predNode[y] = x;
         Reached[y] = true;
         printReachSet( Reached );
         System.out.println();
      }
      int[] a= predNode;
   for ( i = 0; i < NNodes; i++ )
         System.out.println( a[i] + " --> " + i );
   }
   void printReachSet(boolean[] Reached )
```

```
   {
      System.out.print("ReachSet = ");
      for (int i = 0; i < Reached.length; i++ )
         if ( Reached[i] )
           System.out.print( i + " ");
      //System.out.println();
   }
 public static void main(String[] args)
   {
      int[][] conn = {{0,3,0,2,0,0,0,0,4},  // 0
                      {3,0,0,0,0,0,0,4,0},  // 1
                      {0,0,0,6,0,1,0,2,0},  // 2
                      {2,0,6,0,1,0,0,0,0},  // 3
                      {0,0,0,1,0,0,0,0,8},  // 4
                      {0,0,1,0,0,0,8,0,0},  // 5
                      {0,0,0,0,0,8,0,0,0},  // 6
                      {0,4,2,0,0,0,0,0,0},  // 7
                      {4,0,0,0,8,0,0,0,0}   // 8
                     };
Graph G = new Graph(conn);
G.Prim();
   }
}
```

使用 `javac Graph.java` 编译上述代码

输出：

```
$ java Graph
 *  3  *  2  *  *  *  *  4
 3  *  *  *  *  *  *  4  *
 *  *  *  6  *  1  *  2  *
 2  *  6  *  1  *  *  *  *
 *  *  *  1  *  *  *  *  8
 *  *  1  *  *  *  8  *  *
 *  *  *  *  *  8  *  *  *
 *  4  2  *  *  *  *  *  *
 4  *  *  *  8  *  *  *  *
ReachSet = 0 最小代价边: (0,3)代价 = 2
ReachSet = 0 3
最小代价边: (3,4)代价 = 1
ReachSet = 0 3 4
最小代价边: (0,1)代价 = 3
ReachSet = 0 1 3 4
最小代价边: (0,8)代价 = 4
ReachSet = 0 1 3 4 8
最小代价边: (1,7)代价 = 4
ReachSet = 0 1 3 4 7 8
最小代价边: (7,2)代价= 2
可达集合= 0 1 2 3 4 7 8
最小代价边: (2,5)代价= 1
可达集合= 0 1 2 3 4 5 7 8
最小代价边: (5,6)代价= 8
可达集合= 0 1 2 3 4 5 6 7 8
0 --> 0
0 --> 1
7 --> 2
0 --> 3
3 --> 4
2 --> 5
5 --> 6
```



```
   {
      System.out.print("ReachSet = ");
      for (int i = 0; i < Reached.length; i++ )
         if ( Reached[i] )
           System.out.print( i + " ");
      //System.out.println();
   }
 public static void main(String[] args)
   {
      int[][] conn = {{0,3,0,2,0,0,0,0,4},  // 0
                      {3,0,0,0,0,0,0,4,0},  // 1
                      {0,0,0,6,0,1,0,2,0},  // 2
                      {2,0,6,0,1,0,0,0,0},  // 3
                      {0,0,0,1,0,0,0,0,8},  // 4
                      {0,0,1,0,0,0,8,0,0},  // 5
                      {0,0,0,0,0,8,0,0,0},  // 6
                      {0,4,2,0,0,0,0,0,0},  // 7
                      {4,0,0,0,8,0,0,0,0}   // 8
                     };
      Graph G = new Graph(conn);
      G.Prim();
   }
}
```

Compile the above code using `javac Graph.java`

Output:

```
$ java Graph
 *  3  *  2  *  *  *  *  4
 3  *  *  *  *  *  *  4  *
 *  *  *  6  *  1  *  2  *
 2  *  6  *  1  *  *  *  *
 *  *  *  1  *  *  *  *  8
 *  *  1  *  *  *  8  *  *
 *  *  *  *  *  8  *  *  *
 *  4  2  *  *  *  *  *  *
 4  *  *  *  8  *  *  *  *
ReachSet = 0 Min cost edge: (0,3)cost = 2
ReachSet = 0 3
Min cost edge: (3,4)cost = 1
ReachSet = 0 3 4
Min cost edge: (0,1)cost = 3
ReachSet = 0 1 3 4
Min cost edge: (0,8)cost = 4
ReachSet = 0 1 3 4 8
Min cost edge: (1,7)cost = 4
ReachSet = 0 1 3 4 7 8
Min cost edge: (7,2)cost = 2
ReachSet = 0 1 2 3 4 7 8
Min cost edge: (2,5)cost = 1
ReachSet = 0 1 2 3 4 5 7 8
Min cost edge: (5,6)cost = 8
ReachSet = 0 1 2 3 4 5 6 7 8
0 --> 0
0 --> 1
7 --> 2
0 --> 3
3 --> 4
2 --> 5
5 --> 6
```

```
1 --> 7
0 --> 8
```


1 --> 7
0 --> 8

第20章：贝尔曼-福特算法

第20.1节：单源最短路径算法（假设图中存在负权环）

在阅读本例之前，需要对边松弛有一个简要的了解。你可以从这里学习。

贝尔曼-福特算法用于计算加权有向图中从单一源点到所有其他顶点的最短路径。尽管它比迪杰斯特拉算法慢，但它适用于边权为负的情况，并且还能检测图中的负权环。迪杰斯特拉算法的问题在于，如果存在负权环，你会不断地绕着环走，距离会不断减小。

该算法的思路是以某种随机顺序逐一遍历图中的所有边。顺序可以是任意的随机顺序。但必须保证，如果 u-v（其中 u 和 v是图中的两个顶点）是遍历顺序中的一条边，那么图中必须存在从 u到 v的边。通常这个顺序直接取自输入的顺序。再次强调，任何随机顺序都可以。

选定顺序后，我们将根据松弛公式对边进行松弛。对于给定的边 u-v，从 u到 v的松弛公式为：

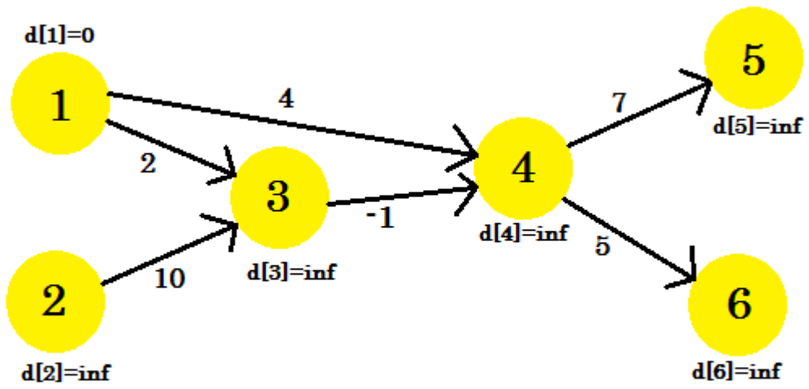
```
如果 distance[u] + cost[u][v] < d[v]
    d[v] = d[u] + cost[u][v]
```

也就是说，如果从源点到任意顶点 u的距离加上边 u-v的权重小于从源点到另一顶点 v的距离，则更新从源点到 v的距离。我们最多需要对边进行(V-1)次松弛，其中V是图中顶点的数量。为什么是(V-1)次？我们将在另一个例子中解释。此外，我们还会记录每个顶点的父节点，也就是当我们松弛一条边时，会设置：

```
parent[v] = u
```

这意味着我们找到了另一条通过 u到达 v的更短路径。稍后我们将用它来打印从源点到目标顶点的最短路径。

让我们来看一个例子。我们有一个图：



Chapter 20: Bellman–Ford Algorithm

Section 20.1: Single Source Shortest Path Algorithm (Given there is a negative cycle in a graph)

Before reading this example, it is required to have a brief idea on edge-relaxation. You can learn it from here

Bellman-Ford Algorithm is computes the shortest paths from a single source vertex to all of the other vertices in a weighted digraph. Even though it is slower than Dijkstra's Algorithm, it works in the cases when the weight of the edge is negative and it also finds negative weight cycle in the graph. The problem with Dijkstra's Algorithm is, if there's a negative cycle, you keep going through the cycle again and again and keep reducing the distance between two vertices.

The idea of this algorithm is to go through all the edges of this graph one-by-one in some random order. It can be any random order. But you must ensure, if **u-v** (where **u** and **v** are two vertices in a graph) is one of your orders, then there must be an edge from **u** to **v**. Usually it is taken directly from the order of the input given. Again, any random order will work.

After selecting the order, we will *relax* the edges according to the relaxation formula. For a given edge **u-v** going from **u** to **v** the relaxation formula is:

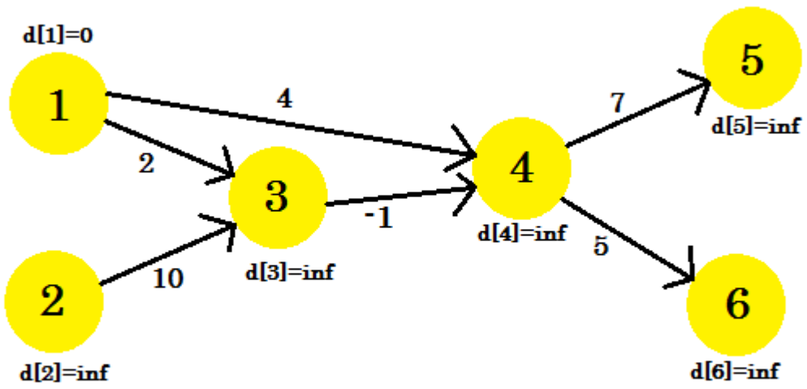
```
if distance[u] + cost[u][v] < d[v]
    d[v] = d[u] + cost[u][v]
```

That is, if the distance from **source** to any vertex **u** + the weight of the **edge u-v** is less than the distance from **source** to another vertex **v**, we update the distance from **source** to **v**. We need to *relax* the edges at most **(V-1)** times where **V** is the number of edges in the graph. Why **(V-1)** you ask? We'll explain it in another example. Also we are going to keep track of the parent vertex of any vertex, that is when we relax an edge, we will set:

```
parent[v] = u
```

It means we've found another shorter path to reach **v** via **u**. We will need this later to print the shortest path from **source** to the destined vertex.

Let's look at an example. We have a graph:



我们选择了**1**作为**源点**顶点。我们想要找出从**源点**到所有其他

顶点的最短路径。

首先，d[1] = 0因为它是源点。其余的都是无穷大，因为我们还不知道它们的距离。

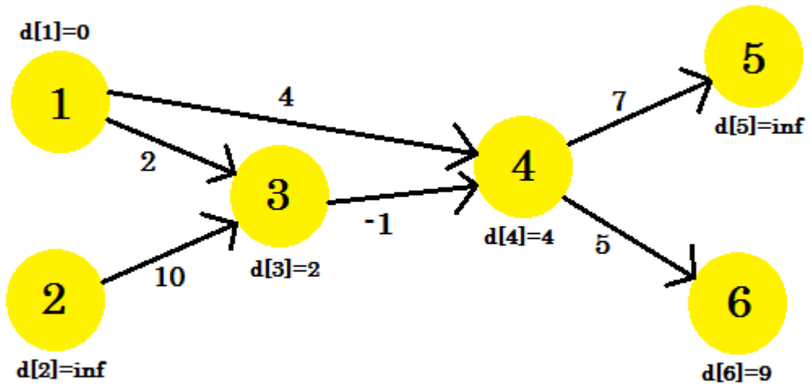
我们将按照以下顺序松弛边：

序号	1	2	3	4	5	6
边	4->5	3->4	1->3	1->4	4->6	2->3

你可以选择任何顺序。如果我们只松弛一次边，会得到什么？我们得到的是从源点到所有其他顶点的路径距离，这条路径最多使用1条边。现在让我们松弛边并更新d[]的值。我们得到：

1. d[4] + cost[4][5] = 无穷大 + 7 = 无穷大。我们无法更新这个值。
2. d[2] + cost[3][4] = 无穷大。我们无法更新这个值。
3. **d[1] + cost[1][3] = 0 + 2 = 2 < d[2]。所以d[3] = 2。同时parent[1] = 1。**
4. **d[1] + cost[1][4] = 4。所以d[4] = 4 < d[4]。parent[4] = 1。**
5. **d[4] + cost[4][6] = 9. d[6] = 9 < d[6]. parent[6] = 4.**
6. d[2] + cost[2][3] = infinity。我们无法更新这个值。

我们无法更新某些顶点，因为条件 d[u] + cost[u][v] < d[v] 不满足。正如之前所说，我们找到了从 source 到其他节点使用最多1条边的路径。



我们的第二次迭代将为我们提供使用2个节点的路径。结果是：

1. **d[4] + cost[4][5] = 12 < d[5]。 d[5] = 12。 parent[5] = 4。**
2. **d[3] + cost[3][4] = 1 < d[4]。 d[4] = 1。 parent[4] = 3。**
3. d[3] 保持不变。
4. d[4] 保持不变。
5. **d[4] + cost[4][6] = 6 < d[6]。 d[6] = 6。 parent[6] = 4。**
6. d[3] 保持不变。

We have selected **1** as the **source** vertex. We want to find out the shortest path from the **source** to all other vertices.

At first, **d[1] = 0** because it is the source. And rest are *infinity*, because we don't know their distance yet.

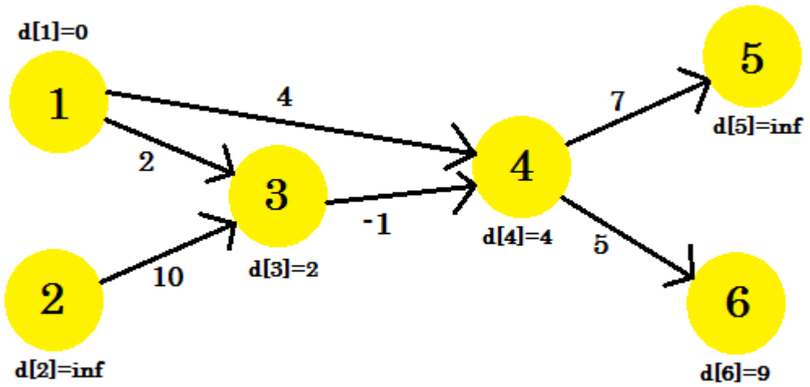
We will relax the edges in this sequence:

Serial	1	2	3	4	5	6
Edge	4->5	3->4	1->3	1->4	4->6	2->3

You can take any sequence you want. If we *relax* the edges once, what do we get? We get the distance from **source** to all other vertices of the path that uses at most 1 edge. Now let's relax the edges and update the values of **d[]**. We get:

1. **d[4]** + **cost[4][5]** = **infinity** + **7** = **infinity**. We can't update this one.
2. **d[2]** + **cost[3][4]** = **infinity**. We can't update this one.
3. **d[1]** + **cost[1][3]** = **0** + **2** = **2** < **d[2]**. So **d[3]** = **2**. Also **parent[1] = 1**.
4. **d[1]** + **cost[1][4]** = **4**. So **d[4] = 4** < **d[4]**. **parent[4]** = **1**.
5. **d[4]** + **cost[4][6]** = **9**. **d[6]** = **9** < **d[6]**. **parent[6]** = **4**.
6. **d[2]** + **cost[2][3]** = **infinity**. We can't update this one.

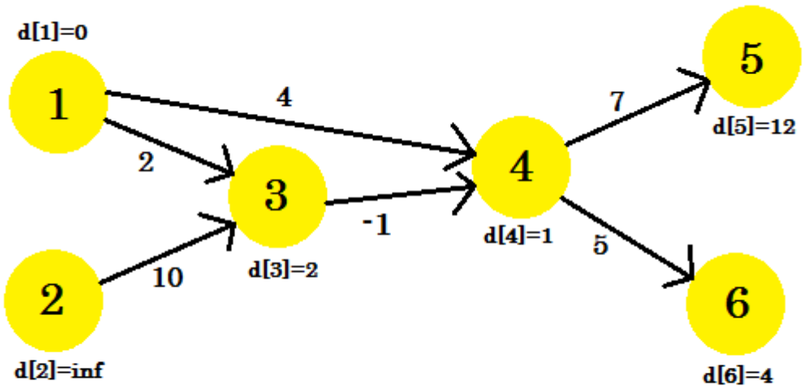
We couldn't update some vertices, because the `d[u] + cost[u][v] < d[v]` condition didn't match. As we have said before, we found the paths from **source** to other nodes using maximum 1 edge.



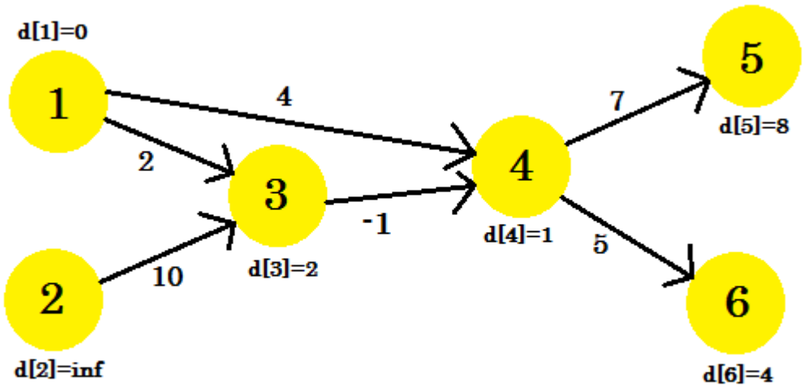
Our second iteration will provide us with the path using 2 nodes. We get:

1. **d[4]** + **cost[4][5]** = **12** < **d[5]**. **d[5]** = **12**. **parent[5]** = **4**.
2. **d[3]** + **cost[3][4]** = **1** < **d[4]**. **d[4]** = **1**. **parent[4]** = **3**.
3. **d[3]** remains unchanged.
4. **d[4]** remains unchanged.
5. **d[4]** + **cost[4][6]** = **6** < **d[6]**. **d[6]** = **6**. **parent[6]** = **4**.
6. **d[3]** remains unchanged.

我们的图将如下所示：



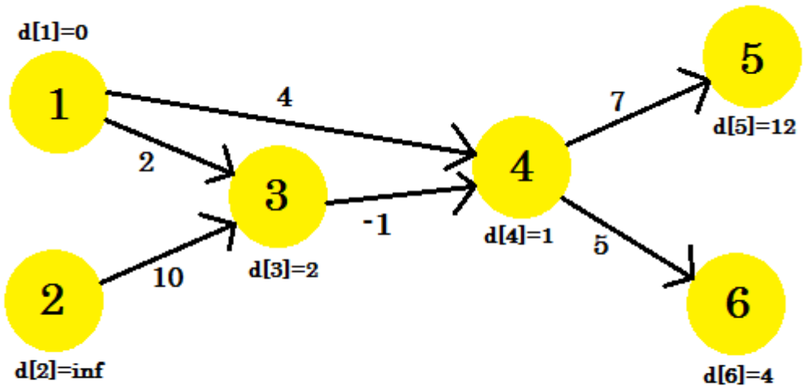
我们的第3次迭代将只更新顶点5，其中d[5]将是8。我们的图将如下所示：



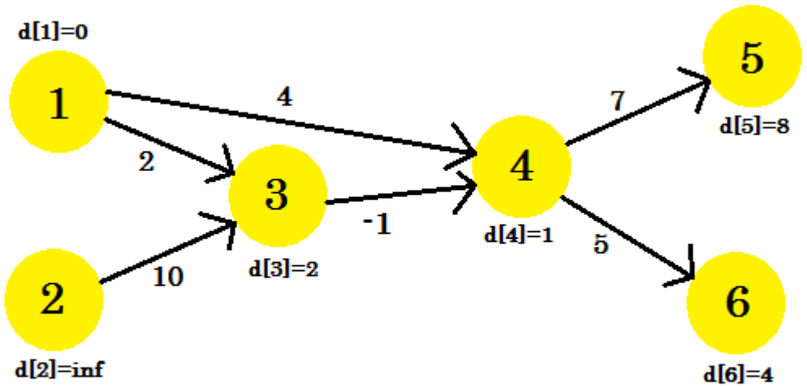
之后无论我们进行多少次迭代，距离都将保持不变。因此我们将保持一个标志来检查是否发生了任何更新。如果没有，我们将直接跳出循环。我们的伪代码将是：

```
过程 Bellman-Ford(图, 源点):
n := 图中顶点的数量
对于 i 从 1 到 n
    d[i] := 无穷大
    parent[i] := 空
结束 对于
d[源点] := 0
对于 i 从 1 到 n-1
    标志 := 假
    对于 图中所有边 (u,v)
        如果 d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
标志 := 真
```


Our graph will look like:



Our 3rd iteration will only update **vertex 5**, where **d[5]** will be **8**. Our graph will look like:



After this no matter how many iterations we do, we'll have the same distances. So we will keep a flag that checks if any update takes place or not. If it doesn't, we'll simply break the loop. Our pseudo-code will be:

```
Procedure Bellman-Ford(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
```

```
end if
    end for
    if flag == false
        break
end for
Return d
```

为了跟踪负环，我们可以使用此处描述的过程修改代码。我们完整的伪代码将是：

```
Bellman-Ford-带-负环-检测过程(图, 源点):
n := 图中顶点的数量
对于 i 从 1 到 n
    d[i] := 无穷大
    parent[i] := 空
结束 对于
d[源点] := 0
对于 i 从 1 到 n-1
    标志 := 假
    对于 图中所有边 (u,v)
        如果 d[u] + cost[u][v] < d[v]
          d[v] := d[u] + cost[u][v]
          parent[v] := u
flag := true
        end if
end for
    if flag == false
        break
end for
对于图中所有边 (u,v)
    如果 d[u] + cost[u][v] < d[v]
        返回 "检测到负环"
    end if
end for
返回 d
```

打印路径：

要打印到某个顶点的最短路径，我们将迭代回溯到它的父节点，直到找到NULL，然后打印这些顶点。伪代码如下：

```
过程 PathPrinting(u)
v := parent[u]
如果 v == NULL
    返回
PathPrinting(v)
打印 -> u
```

复杂度：

由于我们需要对边进行最多 (V-1) 次松弛，该算法的时间复杂度将等于 O(V * **E)** 其中 E 表示边的数量，如果我们使用 邻接表 来表示图。然而，如果使用 邻接矩阵 来表示图，时间复杂度将是 O(V^3)。原因是当使用 邻接表 时，我们可以在 **O(E) 时间内遍历所有边，但使用 邻接矩阵 时需要 O(V^2) 时间。**

第20.2节：图中负环的检测

要理解此示例，建议先对 Bellman-Ford 算法有一个简要了解，相关内容可在


```
        end if
    end for
    if flag == false
        break
end for
Return d
```

To keep track of negative cycle, we can modify our code using the procedure described here. Our completed pseudo-code will be:

```
Procedure Bellman-Ford-With-Negative-Cycle-Detection(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return d
```

Printing Path:

To print the shortest path to a vertex, we'll iterate back to its parent until we find **NULL** and then print the vertices. The pseudo-code will be:

```
Procedure PathPrinting(u)
v := parent[u]
if v == NULL
    return
PathPrinting(v)
print -> u
```

Complexity:

Since we need to relax the edges maximum **(V-1)** times, the time complexity of this algorithm will be equal to **O(V * E)** where **E** denotes the number of edges, if we use `adjacency list` to represent the graph. However, if `adjacency matrix` is used to represent the graph, time complexity will be **O(V^3)**. Reason is we can iterate through all edges in **O(E)** time when `adjacency list` is used, but it takes **O(V^2)** time when `adjacency matrix` is used.

Section 20.2: Detecting Negative Cycle in a Graph

To understand this example, it is recommended to have a brief idea about Bellman-Ford algorithm which can be found

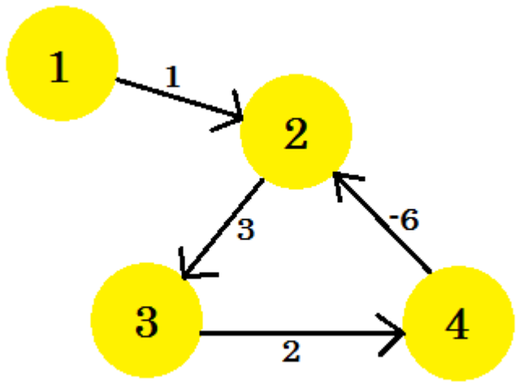
这里

使用贝尔曼-福特算法，我们可以检测图中是否存在负权环。我们知道，为了找出最短路径，我们需要对图中的所有边进行*松弛*操作**(V-1)**次，其中**V**是图中顶点的数量。我们已经看到，在这个例子中，经过(V-1)次迭代后，无论我们进行多少

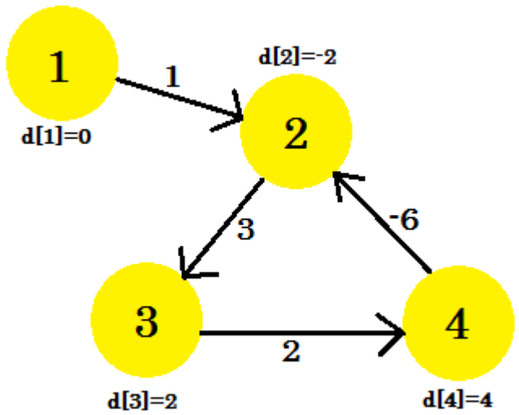
次迭代，都无法更新d[]。或者说，我们能更新吗？

如果图中存在负权环，即使经过**(V-1)**次迭代，我们仍然可以更新**d[]**。这是因为每次迭代中，经过负权环都会不断降低最短路径的代价。这也是贝尔曼-福特算法限制迭代次数为**(V-1)**的原因。如果这里使用迪杰斯特拉算法，我们将陷入无限循环。不过，现在我们先专注于寻找负权环。

假设我们有一个图：



我们选择**顶点1**作为**源点**。应用贝尔曼-福特单源最短路径算法后，我们将得到从**源点**到所有其他顶点的距离。



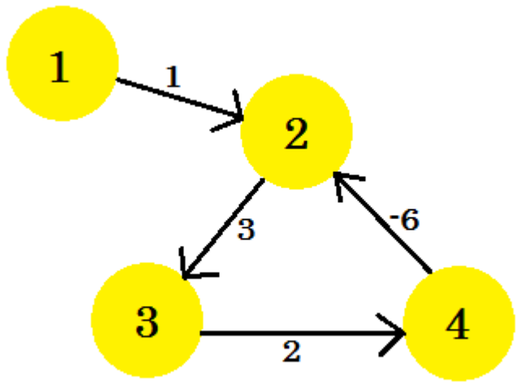
经过(V-1) = 3次迭代后，图的状态如图所示。由于有4条边，最多需要3次迭代才能找到最短路径。因此，这要么是最终结果，要么图中存在负权环。为了确认，经过(V-1)次迭代后，我们再进行一次最终迭代，如果距离继续减小，则说明图中肯定存在负权环。

here

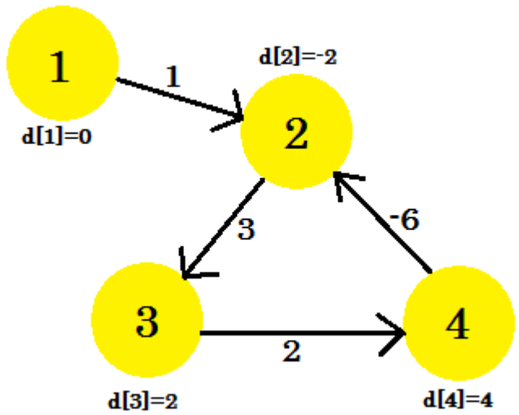
Using Bellman-Ford algorithm, we can detect if there is a negative cycle in our graph. We know that, to find out the shortest path, we need to *relax* all the edges of the graph **(V-1)** times, where **V** is the number of vertices in a graph. We have already seen that in this example, after **(V-1)** iterations, we can't update **d[]**, no matter how many iterations we do. Or can we?

If there is a negative cycle in a graph, even after **(V-1)** iterations, we can update **d[]**. This happens because for every iteration, traversing through the negative cycle always decreases the cost of the shortest path. This is why Bellman-Ford algorithm limits the number of iterations to **(V-1)**. If we used Dijkstra's Algorithm here, we'd be stuck in an endless loop. However, let's concentrate on finding negative cycle.

Let's assume, we have a graph:



Let's pick **vertex 1** as the **source**. After applying Bellman-Ford's single source shortest path algorithm to the graph, we'll find out the distances from the **source** to all the other vertices.



This is how the graph looks like after **(V-1)** = **3** iterations. It should be the result since there are **4** edges, we need at most **3** iterations to find out the shortest path. So either this is the answer, or there is a negative weight cycle in the graph. To find that, after **(V-1)** iterations, we do one more final iteration and if the distance continues to decrease, it means that there is definitely a negative weight cycle in the graph.

在这个例子中：如果我们检查2-3，d[2] + cost[2][3]将得到1，
这小于d[3]。因此我们可以断定图中存在负权环。

那么我们如何找出负权环呢？我们对贝尔曼-福特过程做一点修改：

```
过程 NegativeCycleDetector(图, 源点):
n := 图中顶点的数量
对于 i 从 1 到 n
    d[i] := 无穷大
结束 对于
d[源点] := 0
对于 i 从 1 到 n-1
    标志 := 假
    对于图中所有边(u,v)
        如果 d[u] + cost[u][v] < d[v]
          d[v] := d[u] + cost[u][v]
          flag := true
end if
    end for
    if flag == false
        break
end for
对于图中所有边 (u,v)
    如果 d[u] + cost[u][v] < d[v]
        返回 "检测到负环"
    end if
end for
返回"无负环"
```

这就是我们如何判断图中是否存在负环的方法。我们也可以修改贝尔曼-福特算法来跟踪负环。

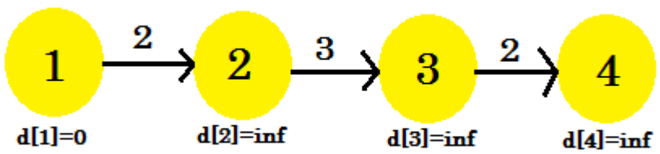
第20.3节：为什么我们需要对所有边最多松弛(V-1)次

要理解这个例子，建议先对贝尔曼-福特单源最短路径算法有一个简要了解，相关内容可以在这里找到

在贝尔曼-福特算法中，为了找到最短路径，我们需要松弛图中的所有边。这个过程最多重复(V-1)次，其中V是图中顶点的数量。

从源点到所有其他顶点找到最短路径所需的迭代次数，取决于我们选择松弛边的顺序。

让我们看一个例子：



这里，源点是1。我们将找出源点与所有其他顶点之间的最短距离。
我们可以清楚地看到，要到达顶点4，在最坏情况下，需要经过(V-1)条边。现在根据发现边的顺序，可能需要(V-1)次才能发现顶点4。没明白？让我们用贝尔曼-福特算法来说明

For this example: if we check **2-3**, **d[2]** + **cost[2][3]** will give us **1** which is less than **d[3]**. So we can conclude that there is a negative cycle in our graph.

So how do we find out the negative cycle? We do a bit modification to Bellman-Ford procedure:

```
Procedure NegativeCycleDetector(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return "No Negative Cycle"
```

This is how we find out if there is a negative cycle in a graph. We can also modify Bellman-Ford Algorithm to keep track of negative cycles.

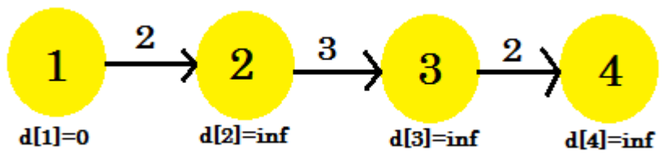
Section 20.3: Why do we need to relax all the edges at most (V-1) times

To understand this example, it is recommended to have a brief idea on Bellman-Ford single source shortest path algorithm which can be found here

In Bellman-Ford algorithm, to find out the shortest path, we need to *relax* all the edges of the graph. This process is repeated at most **(V-1)** times, where **V** is the number of vertices in the graph.

The number of iterations needed to find out the shortest path from **source** to all other vertices depends on the order that we select to *relax* the edges.

Let's take a look at an example:



Here, the **source** vertex is 1. We will find out the shortest distance between the **source** and all the other vertices. We can clearly see that, to reach **vertex 4**, in the worst case, it'll take **(V-1)** edges. Now depending on the order in which the edges are discovered, it might take **(V-1)** times to discover **vertex 4**. Didn't get it? Let's use Bellman-Ford

寻找最短路径的算法如下：

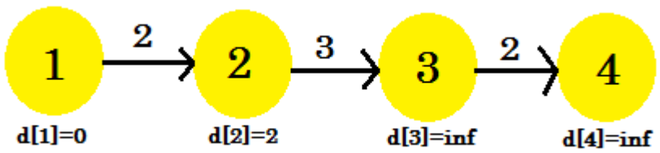
我们将使用以下序列：

```
+--------+--------+--------+--------+
| 序号   |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  边    |  3->4  |  2->3  |  1->2  |
+--------+--------+--------+--------+
```

第一次迭代：

1. d[3] + cost[3][4] = 无穷大。不会有任何变化。
2. d[2] + cost[2][3] = 无穷大。不会有任何变化。
3. **d[1] + cost[1][2] = 2 < d[2]。 d[2] = 2。 parent[2] = 1。**

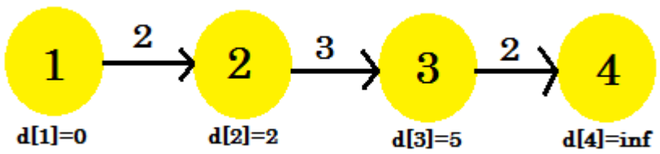
我们可以看到，松弛过程只改变了d[2]。我们的图将变成：



第二次迭代：

1. d[3] + cost[3][4] = 无穷大。不会有任何变化。
2. **d[2] + cost[2][3] = 5 < d[3]. d[3] = 5. parent[3] = 2.**
3. 它不会被改变。

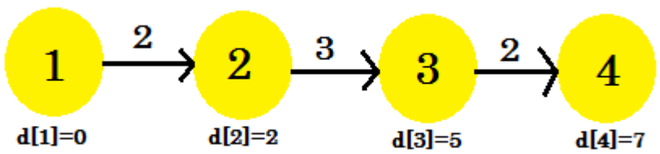
这次松弛过程改变了 d[3]。我们的图将如下所示：



第三次迭代：

1. **d[3] + cost[3][4] = 7 < d[4]。 d[4] = 7。 parent[4] = 3。**
2. 它不会被改变。
3. 它不会被改变。

我们的第三次迭代终于找到了从1到4的最短路径。我们的图将如下所示：



所以，找出最短路径共用了3次迭代。之后，无论我们松弛多少次边，

algorithm to find out the shortest path here:

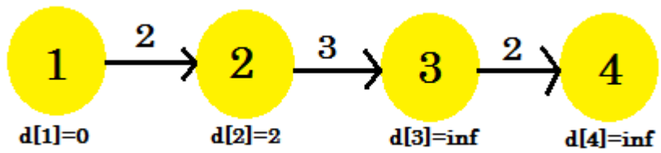
We're going to use this sequence:

```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  3->4  |  2->3  |  1->2  |
+--------+--------+--------+--------+
```

For our first iteration:

1. **d[3]** + **cost[3][4]** = **infinity**. It won't change anything.
2. **d[2]** + **cost[2][3]** = **infinity**. It won't change anything.
3. **d[1]** + **cost[1][2]** = **2** < **d[2]**. **d[2]** = **2**. **parent[2]** = **1**.

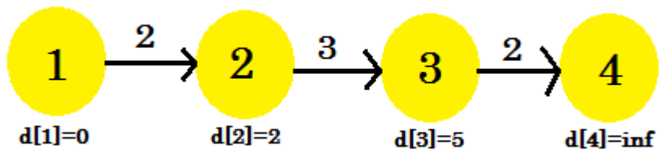
We can see that our *relaxation* process only changed **d[2]**. Our graph will look like:



Second iteration:

1. **d[3]** + **cost[3][4]** = **infinity**. It won't change anything.
2. **d[2]** + **cost[2][3]** = **5** < **d[3]**. **d[3]** = **5**. **parent[3]** = **2.**
3. It won't be changed.

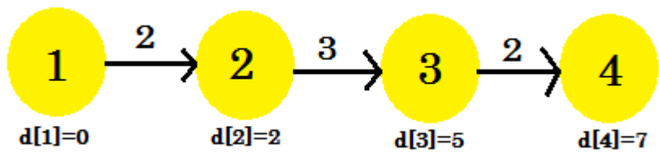
This time the *relaxation* process changed **d[3]**. Our graph will look like:



Third iteration:

1. **d[3]** + **cost[3][4]** = **7** < **d[4]**. **d[4]** = **7**. **parent[4]** = **3**.
2. It won't be changed.
3. It won't be changed.

Our third iteration finally found out the shortest path to **4** from **1**. Our graph will look like:



So, it took **3** iterations to find out the shortest path. After this one, no matter how many times we *relax* the edges,

d[]中的值都将保持不变。现在，如果我们考虑另一种序列：

```
+--------+--------+--------+--------+
| 序号   |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  边    |  1->2  |  2->3  |  3->4  |
+--------+--------+--------+--------+
```

我们得到：

1. **d[1] + cost[1][2] = 2 < d[2]. d[2] = 2.**
2. **d[2] + cost[2][3] = 5 < d[3]. d[3] = 5.**
3. **d[3] + cost[3][4] = 7 < d[4]. d[4] = 5.**

我们的第一次迭代已经找到了从source到所有其他节点的最短路径。另一种序列1->2，
3->4，2->3是可能的，这将在2次迭代后给出最短路径。我们可以得出结论，无论我们如何安排序列，在这个例子中，从source找到最短路径不会超过3次迭代。

我们可以总结出，最佳情况下，找到从source出发的最短路径只需1次迭代。最坏情况下，需要(V-1)次迭代，这就是为什么我们重复进行relaxation(V-1)次的原因。

the values in **d[]** will remain the same. Now, if we considered another sequence:

```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  1->2  |  2->3  |  3->4  |
+--------+--------+--------+--------+
```

We'd get:

1. **d[1]** + **cost[1][2]** = **2** < **d[2]**. **d[2]** = **2**.
2. **d[2]** + **cost[2][3]** = **5** < **d[3]**. **d[3]** = **5**.
3. **d[3]** + **cost[3][4]** = **7** < **d[4]**. **d[4]** = **5**.

Our very first iteration has found the shortest path from **source** to all the other nodes. Another sequence **1->2**, **3->4**, **2->3** is possible, which will give us shortest path after **2** iterations. We can come to the decision that, no matter how we arrange the sequence, it won't take more than **3** iterations to find out shortest path from the **source** in this example.

We can conclude that, for the best case, it'll take **1** iteration to find out the shortest path from **source**. For the worst case, it'll take **(V-1)** iterations, which is why we repeat the process of *relaxation* **(V-1)** times.

第21章：直线算法

直线绘制是通过计算两个指定端点位置之间直线路径上的中间位置来完成的。然后，输出设备被指示填充这些端点之间的位置。

第21.1节：Bresenham直线绘制算法

背景理论：布雷森汉姆线段绘制算法是一种高效且精确的光栅线生成算法，由布雷森汉姆开发。它仅涉及整数计算，因此既准确又快速。该算法还可以扩展用于显示圆和其他曲线。

在Bresenham直线绘制算法中：

对于斜率 |m|<1：
x的值要么增加
要么使用判定参数同时增加x和y的值。

对于斜率 |m|>1：
y的值要么增加
要么使用判定参数同时增加x和y的值。

斜率 |m|<1 的算法：

1. 输入直线的两个端点 (x1,y1) 和 (x2,y2)。
2. 绘制第一个点 (x1,y1)。
3. 计算
   Delx =| x2 – x1 |
   Dely = | y2 – y1 |
4. 获取初始决策参数为
   P = 2 * dely – delx
5. 对 I 从 0 到 delx，步长为 1

   如果 p < 0，则
   X1 = x1 + 1
   绘制点(x1,y1)
   P = p + 2*dely*

   否则
   X1 = x1 + 1
   Y1 = y1 + 1
   绘制点(x1,y1)
   P = p + 2dely – 2 * delx

   结束条件判断

   结束
6. 结束

源代码：

Chapter 21: Line Algorithm

Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints.

Section 21.1: Bresenham Line Drawing Algorithm

Background Theory: Bresenham's Line Drawing Algorithm is an efficient and accurate raster line generating algorithm developed by Bresenham. It involves only integer calculation so it is accurate and fast. It can also be extended to display circles another curves.

In Bresenham line drawing algorithm:

For Slope |m|<1:
Either value of x is increased
OR both x and y is increased using decision parameter.

For Slope |m|>1:
Either value of y is increased
OR both x and y is increased using decision parameter.

Algorithm for slope |m|<1:

1. Input two end points (x1,y1) and (x2,y2) of the line.
2. Plot the first point (x1,y1).
3. Calculate
   Delx =| x2 – x1 |
   Dely = | y2 – y1 |
4. Obtain the initial decision parameter as
   P = 2 * dely – delx
5. For I = 0 to delx in step of 1

   If p < 0 then
   X1 = x1 + 1
   Pot(x1,y1)
   P = p+ 2*dely*

   Else
   X1 = x1 + 1
   Y1 = y1 + 1
   Plot(x1,y1)
   P = p + 2dely – 2 * delx

   End if

   End for
6. END

Source Code:

```c
/* 一个用于实现|m|<1的Bresenham直线绘制算法的C程序 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>

int main()
{
 int gdriver=DETECT,gmode;
 int x1,y1,x2,y2,delx,dely,p,i;
 initgraph(&gdriver,&gmode,"c:\\TC\\BGI");

printf("请输入起始点: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("请输入终点: ");
scanf("%d",&x2);
scanf("%d",&y2);

putpixel(x1,y1,RED);

delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*dely)-delx;
for(i=0;i<delx;i++){
if(p<0)
{
x1=x1+1;
putpixel(x1,y1,RED);
 p=p+(2*dely);
}
否则
{
x1=x1+1;
y1=y1+1;
putpixel(x1,y1,RED);
 p=p+(2*dely)-(2*delx);
}
}
getch();
 closegraph();
 return 0;
}
```

斜率 |m| > 1 的算法：

1. 输入直线的两个端点 (x1,y1) 和 (x2,y2)。
2. 绘制第一个点 (x1,y1)。
3. 计算

   Delx =| x2 – x1 |

   Dely = | y2 – y1 |
4. 获取初始决策参数为

   P = 2 * delx – dely
5. 对 I 从 0 到 dely，步长为 1

   如果 p < 0 则

   y1 = y1 + 1

   Pot(x1,y1)


```c
/* A C program to implement Bresenham line drawing algorithm for |m|<1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>

int main()
{
 int gdriver=DETECT,gmode;
 int x1,y1,x2,y2,delx,dely,p,i;
 initgraph(&gdriver,&gmode,"c:\\TC\\BGI");

printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);

putpixel(x1,y1,RED);

delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*dely)-delx;
for(i=0;i<delx;i++){
if(p<0)
{
 x1=x1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely);
}
else
{
 x1=x1+1;
 y1=y1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely)-(2*delx);
}
}
 getch();
 closegraph();
 return 0;
}
```

Algorithm for slope |m|>1:

1. Input two end points (x1,y1) and (x2,y2) of the line.
2. Plot the first point (x1,y1).
3. Calculate

   Delx =| x2 – x1 |

   Dely = | y2 – y1 |
4. Obtain the initial decision parameter as

   P = 2 * delx – dely
5. For I = 0 to dely in step of 1

   If p < 0 then

   y1 = y1 + 1

   Pot(x1,y1)

P = p + 2delx

否则

X1 = x1 + 1

Y1 = y1 + 1

绘制点(x1,y1)

P = p + 2delx – 2 * dely

结束条件判断

结束循环

6.结束

源代码：

```
/* 一个用于实现|m|>1的Bresenham直线绘制算法的C程序 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
int main()
{
int gdriver=DETECT,gmode;
int x1,y1,x2,y2,delx,dely,p,i;
initgraph(&gdriver,&gmode,"c:\\TC\\BGI");
printf("请输入起始点: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("请输入终点: ");
scanf("%d",&x2);
scanf("%d",&y2);
putpixel(x1,y1,RED);
delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*delx)-dely;
for(i=0;i<delx;i++){
if(p<0)
{
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx);
}
否则
{
x1=x1+1;
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx)-(2*dely);
}
}
getch();
closegraph();
 return 0;
}
```


P = p+ 2*delx*

Else

X1 = x1 + 1

Y1 = y1 + 1

Plot(x1,y1)

*P = p + 2*delx – 2 * dely

End if

End for

6. END

Source Code:

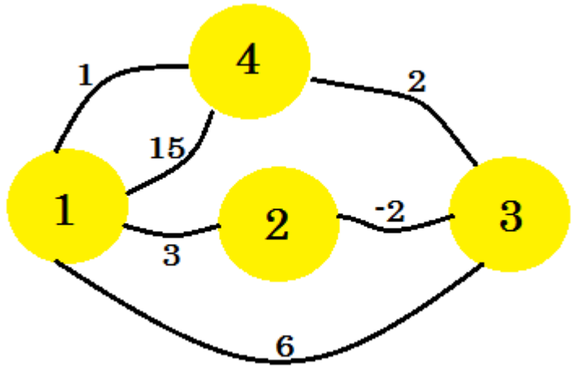
```
/* A C program to implement Bresenham line drawing algorithm for |m|>1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
int main()
{
int gdriver=DETECT,gmode;
int x1,y1,x2,y2,delx,dely,p,i;
initgraph(&gdriver,&gmode,"c:\\TC\\BGI");
printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);
putpixel(x1,y1,RED);
delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*delx)-dely;
for(i=0;i<delx;i++){
if(p<0)
{
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx);
}
else
{
x1=x1+1;
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx)-(2*dely);
}
}
getch();
closegraph();
 return 0;
}
```

第22章：弗洛伊德-沃尔沙尔算法

第22.1节：所有点对最短路径算法

弗洛伊德-沃尔沙尔算法用于在带权图中寻找最短路径，边权可以为正也可以为负。
算法执行一次即可找到所有顶点对之间最短路径的长度（权重和）。稍作变动，它还能打印最短路径并检测图中的负权环。弗洛伊德-
沃尔沙尔算法是一种动态规划算法。

让我们看一个例子。我们将对这个图应用弗洛伊德-沃尔沙尔算法：



首先，我们取两个二维矩阵。这些是邻接矩阵。矩阵的大小等于顶点总数。对于我们的图，我们将取4 * 4矩阵。距离矩阵将存储目前为止找到的两个顶点之间的最短距离。起初，对于边，如果存在一条边u-v且距离/权重为w，我们将存储：distance[u][v] = w。对于不存在的边，我们将设为无穷大。

路径矩阵用于重新生成两个顶点之间的最短路径。因此，最初如果存在从u到v的路径，我们将设置path[u][v] = u。这意味着从顶点-v到顶点-u的最佳方式是使用连接v和u的边。如果两个顶点之间没有路径，我们将在该处放置N，表示当前没有可用路径。我们图的两个表格将如下所示：

```
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
|  1  |  0  |  3  |  6  |  15 |          |  1  |  N  |  1  |  1  |  1  |
|  2  | inf |  0  | -2  | inf |          |  2  |  N  |  N  |  2  |  N  |
|  3  | inf | inf |  0  |  2  |          |  3  |  N  |  N  |  N  |  3  |
|  4  |  1  | inf | inf |  0  |          |  4  |  4  |  N  |  N  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
距离                                     路径
```

由于没有环，主对角线设置为N。顶点自身到自身的距离为0。

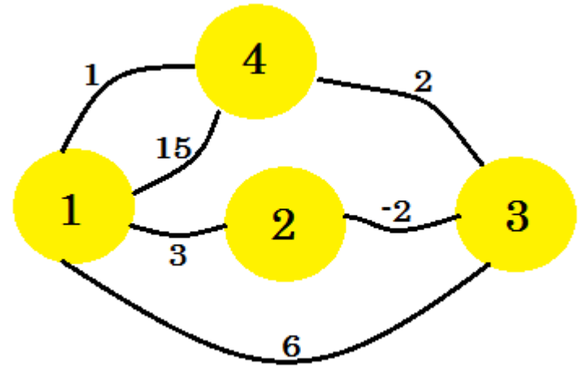
为了应用弗洛伊德-沃尔沙尔算法，我们将选择一个中间顶点k。然后对于每个顶点 i，我们将检查是否可以先从 i 到 k，再从 k 到另一个顶点 j，从而使从 i 到 j 的路径代价最小化。如果当前的 distance[i][j] 大于 distance[i][k] + distance[k][j]，我们将把 distance[i][j] 设为这两个距离的和。同时 path[i][j] 将被设置为 path[k][j]，因为先从 i 到 k 更优，

Chapter 22: Floyd-Warshall Algorithm

Section 22.1: All Pair Shortest Path Algorithm

Floyd-Warshall's algorithm is for finding shortest paths in a weighted graph with positive or negative edge weights. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pair of vertices. With a little variation, it can print the shortest path and can detect negative cycles in a graph. Floyd-Warshall is a Dynamic-Programming algorithm.

Let's look at an example. We're going to apply Floyd-Warshall's algorithm on this graph:



First thing we do is, we take two 2D matrices. These are adjacency matrices. The size of the matrices is going to be the total number of vertices. For our graph, we will take **4 * 4** matrices. The **Distance Matrix** is going to store the minimum distance found so far between two vertices. At first, for the edges, if there is an edge between **u-v** and the distance/weight is **w**, we'll store: `distance[u][v] = w`. For all the edges that doesn't exist, we're gonna put *infinity*. The **Path Matrix** is for regenerating minimum distance path between two vertices. So initially, if there is a path between **u** and **v**, we're going to put `path[u][v] = u`. This means the best way to come to **vertex-v** from **vertex-u** is to use the edge that connects **v** with **u**. If there is no path between two vertices, we're going to put **N** there indicating there is no path available now. The two tables for our graph will look like:

```
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  6  |  15 |          |  1  |  N  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  2  | inf |  0  | -2  | inf |          |  2  |  N  |  N  |  2  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  3  | inf | inf |  0  |  2  |          |  3  |  N  |  N  |  N  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  4  |  1  | inf | inf |  0  |          |  4  |  4  |  N  |  N  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
distance                                   path
```

Since there is no loop, the diagonals are set **N**. And the distance from the vertex itself is **0**.

To apply Floyd-Warshall algorithm, we're going to select a middle vertex **k**. Then for each vertex **i**, we're going to check if we can go from **i** to **k** and then **k** to **j**, where **j** is another vertex and minimize the cost of going from **i** to **j**. If the current **distance[i][j]** is greater than **distance[i][k]** + **distance[k][j]**, we're going to put **distance[i][j]** equals to the summation of those two distances. And the **path[i][j]** will be set to **path[k][j]**, as it is better to go from **i** to **k**,

然后从 k 到 j。所有顶点都会被选为 k。我们将有三个嵌套循环： k 从1到4， i 从1到4， j 从1到4。我们将检查：

```
如果 distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if
```

所以我们基本上是在检查，对于每一对顶点，通过另一个顶点是否能得到更短的距离？我们图的总操作次数是4 * 4 * 4 = 64。也就是说我们将进行64次这样的检查。
64让我们看其中几个例子：

当 k = 1， i = 2， j = 3时，distance[i][j]为-2，不大于 distance[i][k] + distance[k][j] = -2 + 0 = -2。所以它将保持不变。再次说明，当k=1，i=4且j=2时，distance[i][j]=无穷大，大于distance[i][k]+distance[k][j]=1+3=4。因此我们将distance[i][j]设为4，并将path[i][j]设为path[k][j]=1。这意味着，从顶点-4到顶点-2的路径4->1->2比现有路径更短。这就是我们如何填充这两个矩阵。每一步的计算过程显示在这里。经过必要的更改后，我们的矩阵将如下所示：

```
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  1  |  3  |          |  1  |  N  |  1  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  2  |  1  |  0  | -2  |  0  |          |  2  |  4  |  N  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  3  |  3  |  6  |  0  |  2  |          |  3  |  4  |  1  |  N  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  4  |  1  |  4  |  2  |  0  |          |  4  |  4  |  1  |  2  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
distance                                 path
```

这是我们的最短距离矩阵。例如，从1到4的最短距离是3，从4到3的最短距离是2。我们的伪代码如下：

```
过程 Floyd-Warshall(图):
对于k 从 1 到 V     // V表示顶点数量
    对于i 从 1 到 V
        对于j 从 1 到 V
            如果 distance[i][j] > distance[i][k] + distance[k][j]
              distance[i][j] := distance[i][k] + distance[k][j]
              path[i][j] := path[k][j]
结束 如果
       结束 循环
    结束 循环
结束 循环
```

打印路径：

要打印路径，我们将检查Path矩阵。要打印从u到v的路径，我们将从path[u][v]开始。我们会不断将v设置为path[u][v]，直到找到path[u][v]=u，并将path[u][v]的每个值压入栈中。找到u后，我们打印u，然后开始从栈中弹出元素并打印它们。这之所以可行，是因为path矩阵存储了从任意节点到v的最短路径所经过的顶点值。伪代码如下：

```
过程 PrintPath(源，目标)：
```


and then **k** to **j**. All the vertices will be selected as **k**. We'll have 3 nested loops: for **k** going from 1 to 4, **i** going from 1 to 4 and **j** going from 1 to 4. We're going check:

```
if distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if
```

So what we're basically checking is, *for every pair of vertices, do we get a shorter distance by going through another vertex?* The total number of operations for our graph will be **4 * 4 * 4** = **64**. That means we're going to do this check **64** times. Let's look at a few of them:

When **k** = **1**, **i** = **2** and **j** = **3**, **distance[i][j]** is **-2**, which is not greater than **distance[i][k]** + **distance[k][j]** = **-2** + **0** = **-2**. So it will remain unchanged. Again, when **k** = **1**, **i** = **4** and **j** = **2**, **distance[i][j]** = **infinity**, which is greater than **distance[i][k]** + **distance[k][j]** = **1** + **3** = **4**. So we put **distance[i][j]** = **4**, and we put **path[i][j]** = **path[k][j]** = **1**. What this means is, to go from **vertex-4** to **vertex-2**, the path **4->1->2** is shorter than the existing path. This is how we populate both matrices. The calculation for each step is shown here. After making necessary changes, our matrices will look like:

```
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |          |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  1  |  3  |          |  1  |  N  |  1  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  2  |  1  |  0  | -2  |  0  |          |  2  |  4  |  N  |  2  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  3  |  3  |  6  |  0  |  2  |          |  3  |  4  |  1  |  N  |  3  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
|  4  |  1  |  4  |  2  |  0  |          |  4  |  4  |  1  |  2  |  N  |
+-----+-----+-----+-----+-----+          +-----+-----+-----+-----+-----+
distance                                 path
```

This is our shortest distance matrix. For example, the shortest distance from **1** to **4** is **3** and the shortest distance between **4** to **3** is **2**. Our pseudo-code will be:

```
Procedure Floyd-Warshall(Graph):
for k from 1 to V     // V denotes the number of vertex
    for i from 1 to V
       for j from 1 to V
           if distance[i][j] > distance[i][k] + distance[k][j]
               distance[i][j] := distance[i][k] + distance[k][j]
               path[i][j] := path[k][j]
           end if
       end for
    end for
end for
```

Printing the path:

To print the path, we'll check the **Path** matrix. To print the path from **u** to **v**, we'll start from **path[u][v]**. We'll set keep changing **v** = **path[u][v]** until we find **path[u][v]** = **u** and push every values of **path[u][v]** in a stack. After finding **u**, we'll print **u** and start popping items from the stack and print them. This works because the **path** matrix stores the value of the vertex which shares the shortest path to **v** from any other node. The pseudo-code will be:

```
Procedure PrintPath(source, destination):
```

```
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
S.push(Path[source][destination])
    destination := Path[source][destination]
end while
print -> source
while S is not empty
    print -> S.pop
end while
```

寻找负边环：

要判断是否存在负边环，我们需要检查distance矩阵的主对角线。如果对角线上有任何值为负，则表示图中存在负环。

复杂度：

弗洛伊德-沃尔沙尔算法的时间复杂度是O(V^3)，空间复杂度是：O(V^2)。


```
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
    S.push(Path[source][destination])
    destination := Path[source][destination]
end while
print -> source
while S is not empty
    print -> S.pop
end while
```

Finding Negative Edge Cycle:

To find out if there is a negative edge cycle, we'll need to check the main diagonal of **distance** matrix. If any value on the diagonal is negative, that means there is a negative cycle in the graph.

Complexity:

The complexity of Floyd-Warshall algorithm is **O(V^3)** and the space complexity is: **O(V^2)**.

第23章：卡特兰数算法

第23.1节：卡特兰数算法基础信息

卡特兰数算法是一种动态规划算法。

在组合数学中，卡塔兰数 形成了一列自然数，这些数出现在各种计数问题中，通常涉及递归定义的对象。非负整数 n 上的卡塔兰数是一组数字，出现在树的枚举问题中，例如，“一个正 n 边形有多少种不同的方式被划分成 n-2 个三角形，如果不同的方向被分别计数？”

卡特兰数算法的应用：

1. 在一个平面上，将硬币堆叠在由 n 个连续硬币组成的底行上，要求不允许在底行硬币的两侧放置硬币，并且每个额外的硬币必须放在两个其他硬币的上方，这样的堆叠方式数量是第 n 个卡特兰数。

2. 将由 n 对括号组成的字符串分组的方法数，使得每个左括号都有对应的右括号，是第 n 个卡特兰数。

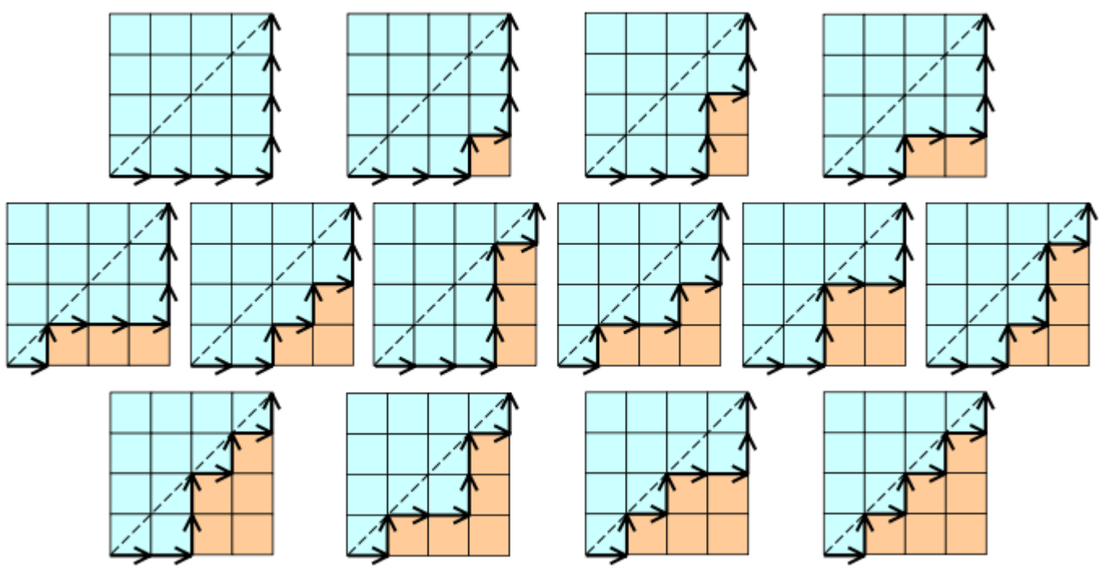
3. 将一个平面上有 n+2 个边的凸多边形通过连接顶点的直线（且这些直线不相交）切割成三角形的方式数是第 n 个卡特兰数。这是欧拉的一个应用。
感兴趣。

使用零基编号，第n个卡特兰数直接通过二项式系数由以下公式给出。

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

卡塔兰数示例：

这里 n 的值 = 4。（最佳示例 - 来自维基百科）



辅助空间：O(n)

Chapter 23: Catalan Number Algorithm

Section 23.1: Catalan Number Algorithm Basic Information

Catalan numbers algorithm is Dynamic Programming algorithm.

In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. The Catalan numbers on nonnegative integers n are a set of numbers that arise in tree enumeration problems of the type, 'In how many ways can a regular n-gon be divided into n-2 triangles if different orientations are counted separately?'

Application of Catalan Number Algorithm:

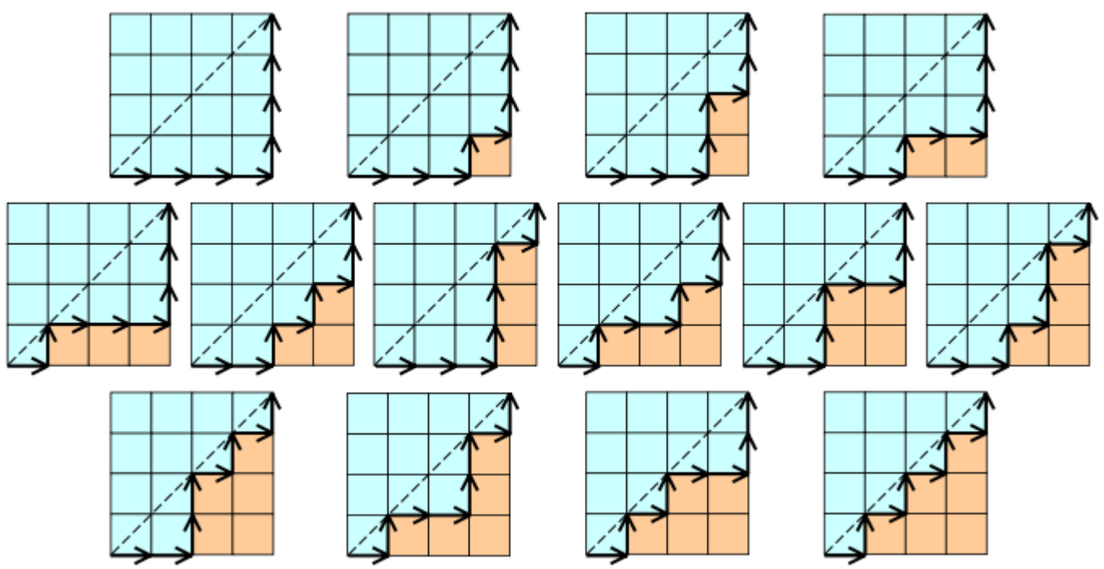
1. The number of ways to stack coins on a bottom row that consists of n consecutive coins in a plane, such that no coins are allowed to be put on the two sides of the bottom coins and every additional coin must be above two other coins, is the nth Catalan number.
2. The number of ways to group a string of n pairs of parentheses, such that each open parenthesis has a matching closed parenthesis, is the nth Catalan number.
3. The number of ways to cut an n+2-sided convex polygon in a plane into triangles by connecting vertices with straight, non-intersecting lines is the nth Catalan number. This is the application in which Euler was interested.

Using zero-based numbering, the *n*th Catalan number is given directly in terms of binomial coefficients by the following equation.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

Example of Catalan Number:

Here value of n = 4.(Best Example - From Wikipedia)



Auxiliary Space: O(n)

时间复杂度：O(n^2)

Time Complexity: `O(n^2)`

第24章：多线程算法

一些多线程算法的示例。

第24.1节：方阵乘法多线程

```
multiply-方阵-矩阵-并行(A, B)
    n = A.行数
C = 矩阵(n,n) //创建一个新的 n*n 矩阵
   并行 for i = 1 到 n
并行 for j = 1 到 n
          C[i][j] = 0
对于 k = 1 到 n
              C[i][j] = C[i][j] + A[i][k]*B[k][j]
    返回 C
```

第24.2节：矩阵向量乘法多线程

```
矩阵-向量(A,x)
n = A.行数
y = 向量(n) //创建一个长度为 n 的新向量
   并行 循环 i = 1 到 n
y[i] = 0
   并行 循环 i = 1 到 n
        循环 j = 1 到 n
y[i] = y[i] + A[i][j]*x[j]
    返回 y
```

第24.3节：归并排序多线程

A 是一个数组，p 和 q 是数组的索引，表示你将要排序的子数组 A[p..r]。B 是一个子数组，将由排序结果填充。

调用 p-merge-sort(A,p,r,B,s) 会对 A[p..r] 中的元素进行排序，并将结果放入 B[s..s+r-p]。

```
p-merge-sort(A,p,r,B,s)
    n = r - p + 1
    如果 n == 1
B[s] = A[p]
    否则
T = new Array(n) // 创建一个大小为 n 的新数组 T
        q = floor((p + r) / 2)
q_prime = q - p + 1
        spawn p-merge-sort(A,p,q,T,1)
        p-merge-sort(A,q + 1,r,T,q_prime + 1)
          sync
p-merge(T,1,q_prime,q_prime + 1,n,B,s)
```

这是执行并行合并的辅助函数。
p-merge 假设要合并的两个子数组在同一个数组中，但不假设它们在数组中是相邻的。这就是为什么我们需要 p1,r1,p2,r2。

```
p-merge(T,p1,r1,p2,r2,A,p3)
    n1 = r1-p1+1
n2 = r2-p2+1
    if n1<n2    //检查 n1 是否大于等于 n2
```


Chapter 24: Multithreaded Algorithms

Examples for some multithreaded algorithms.

Section 24.1: Square matrix multiplication multithread

```
multiply-square-matrix-parallel(A, B)
    n = A.lines
    C = Matrix(n,n) //create a new matrix n*n
    parallel for i = 1 to n
        parallel for j = 1 to n
            C[i][j] = 0
            pour k = 1 to n
                C[i][j] = C[i][j] + A[i][k]*B[k][j]
    return C
```

Section 24.2: Multiplication matrix vector multithread

```
matrix-vector(A,x)
    n = A.lines
    y = Vector(n) //create a new vector of length n
    parallel for i = 1 to n
        y[i] = 0
    parallel for i = 1 to n
        for j = 1 to n
            y[i] = y[i] + A[i][j]*x[j]
    return y
```

Section 24.3: merge-sort multithread

A is an array and *p* and *q* indexes of the array such as you gonna sort the sub-array *A[p..r]*. *B* is a sub-array which will be populated by the sort.

A call to *p-merge-sort(A,p,r,B,s)* sorts elements from *A[p..r]* and put them in *B[s..s+r-p]*.

```
p-merge-sort(A,p,r,B,s)
    n = r-p+1
    if n==1
        B[s] = A[p]
    else
        T = new Array(n) //create a new array T of size n
        q = floor((p+r)/2))
        q_prime = q-p+1
        spawn p-merge-sort(A,p,q,T,1)
        p-merge-sort(A,q+1,r,T,q_prime+1)
        sync
        p-merge(T,1,q_prime,q_prime+1,n,B,s)
```

Here is the auxiliary function that performs the merge in parallel.
p-merge assumes that the two sub-arrays to merge are in the same array but doesn't assume they are adjacent in the array. That's why we need *p1,r1,p2,r2*.

```
p-merge(T,p1,r1,p2,r2,A,p3)
    n1 = r1-p1+1
    n2 = r2-p2+1
    if n1<n2    //check if n1>=n2
```

```
交换 p1 和 p2
        交换 r1 和 r2
        交换 n1 和 n2
    if n1==0     //两者都为空？
        return
    else
q1 = floor((p1+r1)/2)
        q2 = dichotomic-search(T[q1],T,p2,r2)
        q3 = p3 + (q1-p1) + (q2-p2)
A[q3] = T[q1]
spawn p-merge(T,p1,q1-1,p2,q2-1,A,p3)
        p-merge(T,q1+1,r1,q2,r2,A,q3+1)
同步
```

下面是辅助函数二分查找。

x 是在子数组 T[p..r] 中要查找的关键字。

```
dichotomic-search(x,T,p,r)
    inf = p
sup = max(p,r+1)
    while inf<sup
half = floor((inf+sup)/2)
        if x<=T[half]
sup = half
        否则
inf = half+1
    return sup
```

```
        permute p1 and p2
        permute r1 and r2
        permute n1 and n2
    if n1==0     //both empty?
        return
    else
        q1 = floor((p1+r1)/2)
        q2 = dichotomic-search(T[q1],T,p2,r2)
        q3 = p3 + (q1-p1) + (q2-p2)
        A[q3] = T[q1]
        spawn p-merge(T,p1,q1-1,p2,q2-1,A,p3)
        p-merge(T,q1+1,r1,q2,r2,A,q3+1)
        sync
```

And here is the auxiliary function dichotomic-search.

x is the key to look for in the sub-array T[p..r].

```
dichotomic-search(x,T,p,r)
    inf = p
    sup = max(p,r+1)
    while inf<sup
        half = floor((inf+sup)/2)
        if x<=T[half]
            sup = half
        else
            inf = half+1
    return sup
```

第25章：克努斯-莫里斯-普拉特（KMP）算法

KMP是一种模式匹配算法，用于在主“文本字符串”S中搜索“单词”W的出现位置，通过观察当发生不匹配时，我们拥有足够的信息来确定下一个匹配可能开始的位置。我们利用这些信息来避免匹配那些我们已知肯定会匹配的字符。搜索模式的最坏情况复杂度降低到O(n)。

第25.1节：KMP示例

算法

该算法是一个两步过程。首先我们创建一个辅助数组lps[]，然后使用该数组来搜索模式。

预处理：

1. 我们预处理模式并创建一个辅助数组lps[]，用于在匹配时跳过字符。

2. 这里lps[]表示最长的既是前缀又是后缀的真前缀。真前缀是指不包含整个字符串的前缀。例如，字符串ABC的前缀有“ ”、“A”、“AB”和“ABC”。真前缀是“ ”、“A”和“AB”。该字符串的后缀有“ ”、“C”、“BC”和“ABC”。

搜索

1. 我们持续匹配字符 txt[i]和 pat[j]，并在 pat[j]和 txt[i]持续匹配时不断递增i和j。匹配。

2. 当我们看到不匹配时，我们知道字符pat[0..j-1]与 txt[i-j+1...i-1]匹配。我们也知道 lps[j-1] 是 **pat[0…j-1]** 中既是真前缀又是后缀的字符数。由此我们可以得出结论不需要将这 **lps[j-1]** 个字符与 **txt[i-j…i-1]** 进行匹配，因为我们知道这些字符无论如何都会匹配。

Java 实现

```
public class KMP {

    public static void main(String[] args) {
        // TODO 自动生成的方法存根
        String str = "abcabdabc";
        String pattern = "abc";
KMP obj = new KMP();
        System.out.println(obj.patternExistKMP(str.toCharArray(), pattern.toCharArray()));
    }

    public int[] computeLPS(char[] str){
        int lps[] = new int[str.length];

lps[0] = 0;
        int j = 0;
        for(int i =1;i<str.length;i++){
            if(str[j] == str[i]){
lps[i] = j+1;
                j++;
```


Chapter 25: Knuth Morris Pratt (KMP) Algorithm

The KMP is a pattern matching algorithm which searches for occurrences of a "word" **W** within a main "text string" **S** by employing the observation that when a mismatch occurs, we have the sufficient information to determine where the next match could begin.We take advantage of this information to avoid matching the characters that we know will anyway match.The worst case complexity for searching a pattern reduces to **O(n)**.

Section 25.1: KMP-Example

Algorithm

This algorithm is a two step process.First we create a auxiliary array lps[] and then use this array for searching the pattern.

Preprocessing :

1. We pre-process the pattern and create an auxiliary array lps[] which is used to skip characters while matching.
2. Here lps[] indicates longest proper prefix which is also suffix.A proper prefix is prefix in which whole string is not included.For example, prefixes of string **ABC** are “ ”, “**A**”, “**AB**” and “**ABC**”. Proper prefixes are “ ”, “**A**” and “**AB**”. Suffixes of the string are “ ”, “**C**”, “**BC**” and “**ABC**”.

Searching

1. We keep matching characters **txt[i]** and **pat[j]** and keep incrementing i and j while **pat[j]** and **txt[i]** keep matching.

2. When we see a mismatch,we know that characters **pat[0..j-1]** match with **txt[i-j+1…i-1]**.We also know that lps[j-1] is count of characters of **pat[0…j-1]** that are both proper prefix and suffix.From this we can conclude that we do not need to match these **lps[j-1]** characters with **txt[i-j…i-1]** because we know that these characters will match anyway.

Implementaion in Java

```
public class KMP {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str = "abcabdabc";
        String pattern = "abc";
        KMP obj = new KMP();
        System.out.println(obj.patternExistKMP(str.toCharArray(), pattern.toCharArray()));
    }

    public int[] computeLPS(char[] str){
        int lps[] = new int[str.length];

        lps[0] = 0;
        int j = 0;
        for(int i =1;i<str.length;i++){
            if(str[j] == str[i]){
                lps[i] = j+1;
                j++;
```

```
i++;
            }else{
                if(j!=0){
j = lps[j-1];
                }else{
lps[i] = j+1;
                    i++;
                }
            }

        }

        return lps;
    }

    public boolean patternExistKMP(char[] text,char[] pat){
        int[] lps = computeLPS(pat);
        int i=0,j=0;
        while(i<text.length && j<pat.length){
            if(text[i] == pat[j]){
i++;
                j++;
            }else{
                if(j!=0){
j = lps[j-1];
                }else{
i++;
                }
            }
        }

        if(j==pat.length)
            return true;
        return false;
    }

}
```



```
                i++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    lps[i] = j+1;
                    i++;
                }
            }

        }

        return lps;
    }

    public boolean patternExistKMP(char[] text,char[] pat){
        int[] lps = computeLPS(pat);
        int i=0,j=0;
        while(i<text.length && j<pat.length){
            if(text[i] == pat[j]){
                i++;
                j++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    i++;
                }
            }
        }

        if(j==pat.length)
            return true;
        return false;
    }

}
```

第26章：编辑距离动态算法

第26.1节：将字符串1转换为

字符串2所需的最小编辑次数

问题描述是，如果给定两个字符串str1和str2，那么对str1执行多少最少的操作次数，才能将其转换为str2。操作可以是：

1.**插入**
2.**删除**
3.**替换**

例如

```
输入: str1 = "geek", str2 = "gesek"
输出: 1
我们只需要在第一个字符串中插入 s

输入: str1 = "march", str2 = "cart"
输出: 3
我们需要将 m 替换为 c，然后删除字符 c，最后将 h 替换为 t
```

为了解决这个问题，我们将使用一个二维数组 dp[n+1][m+1]，其中 n 是第一个字符串的长度，m 是第二个字符串的长度。以我们的例子为例，如果 str1 是 azcef，str2 是 abcdef，那么我们的数组将是 dp[6][7]，最终答案将存储在 dp[5][6]。

```
(a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

对于 dp[1][1] 我们需要检查如何将 a 转换为 a。结果是 0。对于 dp[1][2] 我们需要检查如何将 a 转换为 ab。结果是 1，因为我们需要 插入 b。所以第一轮迭代后我们的数组将变成这样

```
          (a) (b) (c) (d) (e) (f)

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

(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |

(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```


Chapter 26: Edit Distance Dynamic Algorithm

Section 26.1: Minimum Edits required to convert string 1 to string 2

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2.The Operations can be:

1. **Insert**
2. **Remove**
3. **Replace**

For Example

```
Input: str1 = "geek", str2 = "gesek"
Output: 1
We only need to insert s in first string

Input: str1 = "march", str2 = "cart"
Output: 3
We need to replace m with c and remove character c and then replace h with t
```

To solve this problem we will use a 2D array dp[n+1][m+1] where n is the length of the first string and m is the length of the second string. For our example, if str1 is **azcef** and str2 is **abcdef** then our array will be dp[6][7]and our final answer will be stored at dp[5][6].

```
          (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

For **dp[1][1]** we have to check what can we do to convert **a** into **a**.It will be **0**.For **dp[1][2]** we have to check what can we do to convert **a** into **ab**.It will be **1** because we have to **insert b**.So after 1st iteration our array will look like

```
          (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
```

```
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

第二轮迭代

对于 dp[2][1] 我们需要检查将 az 转换为 a 需要删除 z，因此 dp[2][1] 为 1。同样地，对于 dp[2][2] 我们需要将 z 替换为 b，因此 dp[2][2] 为 1。所以第二轮迭代后我们的 dp[] 数组将变成这样。

```
(a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

所以我们的公式将会是

```
如果字符相同
    dp[i][j] = dp[i-1][j-1];
否则
dp[i][j] = 1 + Min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

最后一次迭代后我们的 dp[] 数组将会是

```
(a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(c)| 3 | 2 | 2 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+---+---+
(e)| 4 | 3 | 3 | 2 | 2 | 2 | 3 |
+---+---+---+---+---+---+---+
(f)| 5 | 4 | 4 | 2 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+
```

Java实现

```java
public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
```



```
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

For iteration 2

For **dp[2][1]** we have to check that to convert **az** to **a** we need to remove **z**, hence **dp[2][1]** will be **1**.Similary for **dp[2][2]** we need to replace **z** with **b**, hence **dp[2][2]** will be **1**.So after 2nd iteration our **dp[]** array will look like.

```
        (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

So our **formula** will look like

```
if characters are same
    dp[i][j] = dp[i-1][j-1];
else
    dp[i][j] = 1 + Min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

After last iteration our dp[] array will look like

```
         (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
(c)| 3 | 2 | 2 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+---+---+
(e)| 4 | 3 | 3 | 2 | 2 | 2 | 3 |
+---+---+---+---+---+---+---+
(f)| 5 | 4 | 4 | 2 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+
```

Implementation in Java

```java
public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
```

```
dp[i][j] = j;
                else if(j==0)
dp[i][j] = i;
                否则如果(str1.charAt(i-1) == str2.charAt(j-1))
                  dp[i][j] = dp[i-1][j-1];
                否则{
dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
                }
            }
        }
        返回 dp[str1.length()][str2.length()];
}
```

时间复杂度

```
O(n^2)
```



```
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

Time Complexity

```
O(n^2)
```

第27章：在线算法

理论

定义1：一个**优化问题**Π由一组**实例**ΣΠ组成。对于每个实例σ∈ΣΠ，存在一组Zσ的**解**和一个**目标函数** fσ : Zσ → ℜ≥0，它为每个解分配一个非负实数值。
我们称 OPT(σ) 为最优解的值，A(σ) 为问题 Π 的算法 A 的解，
wA(σ)=fσ(A(σ)) 为其值。

定义 2： 对于一个最小化问题 Π，若存在常数 τ∈ℜ，使得在线算法 A 的 竞争比 r ≥ 1 满足

> wA(σ) = fσ(A(σ)) ≤ r · OPT(&sigma) + τ

对所有实例 σ∈ΣΠ 都成立，则称 A 为 r-竞争性 在线算法。如果甚至

> wA(σ) ≤ r · OPT(&sigma)

对所有实例 σ∈ΣΠ 都成立，则称 A 为 严格 r-竞争性 在线算法。

命题 1.3：LRU 和 FWF 是标记算法。

证明： 在每个阶段开始时（除第一个阶段外），FWF 会发生缓存未命中并清空缓存。这意味着我们有 k 个空页。在每个阶段最多请求 k 个不同的页面，因此在该阶段不会发生驱逐。因此 FWF 是标记算法。

假设 LRU 不是标记算法。则存在一个实例 σ，使得 LRU 在第 i 阶段驱逐了一个已标记的页面 x。设 σt 为第 i 阶段驱逐 x 的请求。由于 x 已被标记，说明在同一阶段存在一个更早的请求 σt*，t* < t。t* 之后 x 是缓存中最新的页面，因此要在 t 时被驱逐，序列 σt*+1,...,σt 必须请求至少 k 个不同于 x 的页面。这意味着第 i 阶段请求了至少 k+1 个不同的页面，与阶段定义矛盾。因此 LRU 必须是标记算法。

命题 1.4： 每个标记算法 都是严格 k-竞争性 的。

证明： 设 σ 是分页问题的一个实例，l 是 σ 的阶段数。如果 l = 1，则每个标记算法都是最优的，且最优的离线算法不会更好。
我们假设 l ≥ 2。对于每个标记算法实例 σ，其成本被上界限制为 l · k，因为在每个阶段中，标记算法不能在不驱逐一个标记页的情况下驱逐超过 k 页。
现在我们尝试证明最优离线算法对于 σ 至少驱逐 k+l-2 页，第一阶段驱逐 k 页，之后每个阶段至少驱逐一页，除了最后一个阶段。为证明此点，我们定义 σ 的 l-2 个不相交子序列。
子序列 i ∈ {1,...,l-2} 从第 i+1 阶段的第二个位置开始，到第 i+2 阶段的第一个位置结束。
设 x 为第 i+1 阶段的第一页。在子序列 i 开始时，最优离线算法的缓存中有页面 x 和最多 k-1 个不同页面。子序列 i 中有 k 个不同于 x 的页面请求，因此最优离线算法必须为每个子序列驱逐至少一页。由于第一阶段开始时缓存仍为空，最优离线算法在第一阶段导致 k 次驱逐。这表明

> wA(σ) ≤ l·k ≤ (k+l-2)k ≤ OPT(σ) · k

推论 1.5：LRU 和 FWF 是 严格 k-竞争性 的。

Chapter 27: Online algorithms

Theory

Definition 1: An **optimization problem** Π consists of a set of **instances** ΣΠ. For every instance σ∈ΣΠ there is a set Zσ of **solutions** and a **objective function** fσ : Zσ → ℜ≥0 which assigns apositive real value to every solution.
We say OPT(σ) is the value of an optimal solution, A(σ) is the solution of an Algorithm A for the problem Π and wA(σ)=fσ(A(σ)) its value.

Definition 2: An online algorithm A for a minimization problem Π has a **competetive ratio** of r ≥ 1 if there is a constant τ∈ℜ with

> wA(σ) = fσ(A(σ)) ≤ r · OPT(&sigma) + τ

for all instances σ∈ΣΠ. A is called a **r-competitive** online algorithm. Is even

> wA(σ) ≤ r · OPT(&sigma)

for all instances σ∈ΣΠ then A is called a **strictly r-competitive** online algorithm.

Proposition 1.3: LRU and **FWF** are marking algorithm.

Proof: At the beginning of each phase (except for the first one) **FWF** has a cache miss and cleared the cache. that means we have k empty pages. In every phase are maximal k different pages requested, so there will be now eviction during the phase. So **FWF** is a marking algorithm.
Lets assume **LRU** is not a marking algorithm. Then there is an instance σ where **LRU** a marked page x in phase i evicted. Let σt the request in phase i where x is evicted. Since x is marked there has to be a earlier request σt* for x in the same phase, so t* < t. After t* x is the caches newest page, so to got evicted at t the sequence σt*+1,...,σt has to request at least k from x different pages. That implies the phase i has requested at least k+1 different pages which is a contradictory to the phase definition. So **LRU** has to be a marking algorithm.

Proposition 1.4: Every marking algorithm **is strictly k-competitive**.

Proof: Let σ be an instance for the paging problem and l the number of phases for σ. Is l = 1 then is every marking algorithm optimal and the optimal offline algorithm cannot be better.
We assume l ≥ 2. the cost of every marking algorithm for instance σ is bounded from above with l · k because in every phase a marking algorithm cannot evict more than k pages without evicting one marked page.
Now we try to show that the optimal offline algorithm evicts at least k+l-2 pages for σ, k in the first phase and at least one for every following phase except for the last one. For proof lets define l-2 disjunct subsequences of σ.
Subsequence i ∈ {1,...,l-2} starts at the second position of phase i+1 and end with the first position of phase i+2.
Let x be the first page of phase i+1. At the beginning of subsequence i there is page x and at most k-1 different pages in the optimal offline algorithms cache. In subsequence i are k page request different from x, so the optimal offline algorithm has to evict at least one page for every subsequence. Since at phase 1 beginning the cache is still empty, the optimal offline algorithm causes k evictions during the first phase. That shows that

> wA(σ) ≤ l·k ≤ (k+l-2)k ≤ OPT(σ) · k

Corollary 1.5: LRU and **FWF** are **strictly k-competitive**.

如果不存在常数 r 使得某在线算法 A 是 r-竞争性的，则称 A 为 不竞争性 的。

命题 1.6：LFU 和 LIFO 是 不竞争性 的。

证明：设 l ≥ 2 为常数，k ≥ 2 为缓存大小。不同的缓存页面编号为 1,...,k+1。我们观察以下序列：

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

首先请求页面 1 共 l 次，然后是页面 2，依此类推。最后有 (l-1) 次页面 k 和 k+1 的交替请求。

LFU和LIFO用页面1-k填充它们的缓存。当请求页面k+1时，页面k被驱逐，反之亦然。这意味着子序列(k,k+1)^{l-1}的每次请求都会驱逐一个页面。此外，首次使用页面1-(k-1)时会有k-1次缓存未命中。因此，LFU和LIFO总共驱逐了恰好k-1+2(l-1)个页面。
现在我们必须证明，对于每个常数τ∈ℜ和每个常数r ≤ 1，存在一个l使得

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

这等同于

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

要满足这个不等式，你只需选择足够大的l。因此，LFU和LIFO不是竞争性的。

命题1.7：不存在r-竞争性的确定性在线分页算法，其中 r < k。

来源
基础材料

1. 在线算法脚本（德文），Heiko Roeglin，波恩大学
2. 页面替换算法

进一步阅读

1. 《在线计算与竞争分析》 作者 Allan Borodin 和 Ran El-Yaniv

源代码

1. 用于离线缓存的源代码
2. 对抗游戏的源代码

第27.1节：分页（在线缓存）

前言

本章不从正式定义开始，而是通过一系列示例来介绍这些主题，沿途引入定义。备注部分理论将包含所有定义、定理和命题，方便你快速查找具体内容。

Is there no constant r for which an online algorithm A is r-competitive, we call A **not competitive**.

Proposition 1.6: LFU and **LIFO** are **not competitive**.

Proof: Let l ≥ 2 a constant, k ≥ 2 the cache size. The different cache pages are nubered 1,...,k+1. We look at the following sequence:

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

First page 1 is requested l times than page 2 and so one. At the end there are (l-1) alternating requests for page k and k+1.
LFU and **LIFO** fill their cache with pages 1-k. When page k+1 is requested page k is evicted and vice versa. That means every request of subsequence (k,k+1)l-1 evicts one page. In addition their are k-1 cache misses for the first time use of pages 1-(k-1). So **LFU** and **LIFO** evict exact k-1+2(l-1) pages.
Now we must show that for every constant τ∈ℜ and every constan r ≤ 1 there exists an l so that

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

which is equal to

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

To satisfy this inequality you just have to choose l sufficient big. So **LFU** and **LIFO** are not competetive.

Proposition 1.7: There is **no r-competetive** deterministic online algorithm for paging with **r < k**.

Sources
Basic Material

1. Script Online Algorithms (german), Heiko Roeglin, University Bonn
2. Page replacement algorithm

Further Reading

1. Online Computation and Competetive Analysis by Allan Borodin and Ran El-Yaniv

Source Code

1. Source code for offline caching
2. Source code for adversary game

Section 27.1: Paging (Online Caching)

Preface

Instead of starting with a formal definition, the goal is to approach these topic via a row of examples, introducing definitions along the way. The remark section **Theory** will consist of all definitions, theorems and propositions to give you all information to faster look up specific aspects.

备注部分的资料包括本主题的基础材料和进一步阅读的补充信息。此外，你还可以在那里找到示例的完整源代码。请注意，为了使示例的源代码更易读且更简洁，省略了错误处理等内容。同时也避免使用一些会影响示例清晰度的特定语言特性，如大量使用高级库等。

分页

分页问题源于有限空间的限制。假设我们的缓存C有k页。现在我们要处理一个长度为m的页面请求序列，这些页面必须在处理前被放入缓存。当然，如果m<=k，那么我们只需将所有元素放入缓存即可，但通常m>>k。

当请求的页面已经在缓存中时，我们称之为缓存命中，否则称为缓存未命中。在后一种情况下，必须将请求的页面调入缓存并驱逐另一个页面，前提是缓存已满。目标是制定一个最小化驱逐次数的驱逐计划。

针对这个问题有许多策略，下面让我们来看一些：

1. 先进先出（FIFO）：最旧的页面被驱逐
2. 后进先出（LIFO）：最新的页面被驱逐
3. 最近最少使用（LRU）：驱逐最近访问时间最早的页面
4. 最不经常使用（LFU）：驱逐请求次数最少的页面
5. 最长前向距离（LFD）：淘汰缓存中未来最长时间内不会被请求的页面。
6. 满时清空（FWF）：一旦发生缓存未命中，立即清空整个缓存

解决该问题有两种方法：

1. 离线：页面请求序列事先已知
2. 在线：页面请求序列事先未知

离线方法

对于第一种方法，请参考贪心技术的应用主题。其第三个示例**离线缓存**考虑了上述前五种策略，并为后续内容提供了良好的入门点。

示例程序扩展了FWF策略：

```
class FWF : public Strategy {
public:
FWF() : 策略("FWF")
    {
    }

    int 应用(int 请求索引) 重写
    {
        对于(int i=0; i<缓存大小; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            // 第一个空白页之后，所有其他页必须为空
            else if(cache[i] == emptyPage)
                return i;
        }

        // 没有空闲页
```


The remark section sources consists of the basis material used for this topic and additional information for further reading. In addition you will find the full source codes for the examples there. Please pay attention that to make the source code for the examples more readable and shorter it refrains from things like error handling etc. It also passes on some specific language features which would obscure the clarity of the example like extensive use of advanced libraries etc.

Paging

The paging problem arises from the limitation of finite space. Let's assume our cache `C` has `k` pages. Now we want to process a sequence of `m` page requests which must have been placed in the cache before they are processed. Of course if `m<=k` then we just put all elements in the cache and it will work, but usually is `m>>k`.

We say a request is a **cache hit**, when the page is already in cache, otherwise, its called a **cache miss**. In that case, we must bring the requested page into the cache and evict another, assuming the cache is full. The Goal is an eviction schedule that **minimizes the number of evictions**.

There are numerous strategies for this problem, let's look at some:

1. **First in, first out (FIFO)**: The oldest page gets evicted
2. **Last in, first out (LIFO)**: The newest page gets evicted
3. **Least recently used (LRU)**: Evict page whose most recent access was earliest
4. **Least frequently used (LFU)**: Evict page that was least frequently requested
5. **Longest forward distance (LFD)**: Evict page in the cache that is not requested until farthest in the future.
6. **Flush when full (FWF)**: clear the cache complete as soon as a cache miss happened

There are two ways to approach this problem:

1. **offline**: the sequence of page requests is known ahead of time
2. **online**: the sequence of page requests is not known ahead of time

Offline Approach

For the first approach look at the topic Applications of Greedy technique. It's third Example **Offline Caching** considers the first five strategies from above and gives you a good entry point for the following.

The example program was extended with the **FWF** strategy:

```
class FWF : public Strategy {
public:
    FWF() : Strategy("FWF")
    {
    }

    int apply(int requestIndex) override
    {
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            // after first empty page all others have to be empty
            else if(cache[i] == emptyPage)
                return i;
        }

        // no free pages
```

```
        return 0;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {

        // 没有空闲页 -> 未命中 -> 清空缓存
        if(cacheMiss && cachePos == 0)
        {
            for(int i = 1; i < cacheSize; ++i)
                cache[i] = emptyPage;
        }
    }
};
```

完整源码可在 here 获取。如果我们重用该主题中的示例，得到以下输出：

```
策略: FWF

缓存初始: (a,b,c)

请求 缓存 0 缓存 1 缓存 2 缓存未命中
  a       a       b       c
a     a     b     c
  d     d     X     X     x
  e     d     e     X
b     d     e     b
  b     d     e     b
  a     a     X     X     x
  c     a     c     X
f     a     c     f
  d     d     X     X     x
  e     d     e     X
a     d     e     a
  f     f     X     X     x
  b     f     b     X
  e       f       b       e
  c       c       X       X       x

缓存未命中总数: 5
```

尽管LFD是最优的，FWF的缓存未命中更少。但主要目标是最小化驱逐次数，对于FWF来说，五次未命中意味着15次驱逐，这使得它在此示例中是最差的选择。

在线方法

现在我们想要解决分页的在线问题。但首先我们需要了解如何去做。
显然，在线算法不可能比最优的离线算法更好。但它到底差多少？我们需要正式定义来回答这个问题：

定义 1.1： 一个 优化问题 Π 由一组 实例 ΣΠ 组成。对于每个实例 σ∈ΣΠ，存在一个 解集 Zσ 和一个 目标函数 fσ ： Zσ → ℜ≥0，该函数为每个解赋予一个正实数值。
我们称 OPT(σ) 为最优解的值，A(σ) 为问题 Π 的算法 A 的解，
wA(σ)=fσ(A(σ)) 为其值。

定义 1.2： 对于一个最小化问题 Π 的在线算法 A，如果存在一个常数 τ∈ℜ，使得其 竞争比 为 r ≥ 1，则满足


```
        return 0;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {

        // no pages free -> miss -> clear cache
        if(cacheMiss && cachePos == 0)
        {
            for(int i = 1; i < cacheSize; ++i)
                cache[i] = emptyPage;
        }
    }
};
```

The full sourcecode is available here. If we reuse the example from the topic, we get the following output:

```
Strategy: FWF

Cache initial: (a,b,c)

Request cache 0 cache 1 cache 2 cache miss
  a       a       b       c
  a       a       b       c
  d       d       X       X       x
  e       d       e       X
  b       d       e       b
  b       d       e       b
  a       a       X       X       x
  c       a       c       X
  f       a       c       f
  d       d       X       X       x
  e       d       e       X
  a       d       e       a
  f       f       X       X       x
  b       f       b       X
  e       f       b       e
  c       c       X       X       x

Total cache misses: 5
```

Even though **LFD** is optimal, **FWF** has fewer cache misses. But the main goal was to minimize the number of evictions and for **FWF** five misses mean 15 evictions, which makes it the poorest choice for this example.

Online Approach

Now we want to approach the online problem of paging. But first we need an understanding how to do it.
Obviously an online algorithm cannot be better than the optimal offline algorithm. But how much worse it is? We need formal definitions to answer that question:

Definition 1.1: An **optimization problem** Π consists of a set of **instances** ΣΠ. For every instance σ∈ΣΠ there is a set Zσ of **solutions** and a **objective function** fσ : Zσ → ℜ≥0 which assigns apositive real value to every solution.
We say OPT(σ) is the value of an optimal solution, A(σ) is the solution of an Algorithm A for the problem Π and wA(σ)=fσ(A(σ)) its value.

Definition 1.2: An online algorithm A for a minimization problem Π has a **competetive ratio** of r ≥ 1 if there is a constant τ∈ℜ with

> $wA(\sigma) = f\sigma(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$

对所有实例 σ∈ΣΠ 都成立，则称 A 为 r-竞争性 在线算法。如果甚至

> $wA(\sigma) \leq r \cdot OPT(\sigma)$

对所有实例 σ∈ΣΠ 都成立，则称 A 为 严格 r-竞争性 在线算法。

那么问题是，我们的在线算法与最优的离线算法相比，竞争力如何。在他们著名的书籍中，艾伦·博罗丁（Allan Borodin）和兰·埃尔-亚尼夫（Ran El-Yaniv）用另一种情景来描述在线分页问题：

有一个恶意对手，他知道你的算法和最优的离线算法。在每一步，他都会尝试请求一个对你最不利且对离线算法最有利的页面。你的算法的竞争因子就是你的算法相对于对手的最优离线算法表现有多差的因子。如果你想尝试成为对手，可以尝试对手游戏（尝试击败分页策略）。

标记算法

与其单独分析每个算法，不如看看一种针对分页问题的特殊在线算法家族，称为标记算法。

设 σ=(σ1,...,σp) 是我们问题的一个实例，k 是我们的缓存大小，则 σ 可以被划分为若干阶段：

- 阶段 1 是从开始到请求了最多 k 个不同页面的 σ 的最大子序列阶段 i ≥ 2 是从阶段 i-1 结束到请求
- 了最多 k 个不同页面的 σ 的最大子序列

例如，当 k = 3 时：

$$\sigma = \left(\overbrace{a, b, d, a}^{phase\ 1}, \overbrace{e, a, f, a, f}^{phase\ 2}, \overbrace{b, d, a}^{phase\ 3}, \overbrace{c, c, d}^{phase\ 4}\right)$$

一个标记算法（隐式或显式）维护页面是否被标记。在每个阶段开始时，所有页面都是未标记的。如果在某个阶段请求了某个页面，该页面就会被标记。一个算法是标记算法iff它从不从缓存中驱逐被标记的页面。这意味着在一个阶段内被使用的页面不会被驱逐。

命题 1.3：LRU 和 FWF 是标记算法。

证明： 在每个阶段开始时（除第一个阶段外），FWF 会发生缓存未命中并清空缓存。这意味着我们有 k 个空页。在每个阶段最多请求 k 个不同的页面，因此在该阶段不会发生驱逐。因此 FWF 是标记算法。

假设LRU不是标记算法。那么存在一个实例σ，在阶段 i 中LRU驱逐了一个被标记的页面 x。设σt为阶段 i 中驱逐 x 的请求。由于 x 被标记，说明在同一阶段中存在一个更早的请求σt*，即 t* < t。在 t* 之后，x 是缓存中最新的页面，因此为了在 t 时驱逐它，序列σt*+1,...,σt必须请求至少 k 个不同于 x 的页面。这意味着阶段 i 请求了至少 k+1 个不同的页面，这与阶段的定义矛盾。所以LRU必须是一个标记算法。

> $wA(\sigma) = f\sigma(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$

for all instances σ∈ΣΠ. A is called a **r-competitive** online algorithm. Is even

> $wA(\sigma) \leq r \cdot OPT(\sigma)$

for all instances σ∈ΣΠ then A is called a **strictly r-competitive** online algorithm.

So the question is how **competitive** is our online algorithm compared to an optimal offline algorithm. In their famous book Allan Borodin and Ran El-Yaniv used another scenario to describe the online paging situation:

There is an **evil adversary** who knows your algorithm and the optimal offline algorithm. In every step, he tries to request a page which is worst for you and simultaneously best for the offline algorithm. the **competitive factor** of your algorithm is the factor on how badly your algorithm did against the adversary's optimal offline algorithm. If you want to try to be the adversary, you can try the Adversary Game (try to beat the paging strategies).

Marking Algorithms

Instead of analysing every algorithm separately, let's look at a special online algorithm family for the paging problem called **marking algorithms**.

Let σ=(σ1,...,σp) an instance for our problem and k our cache size, than σ can be divided into phases:

- Phase 1 is the maximal subsequence of σ from the start till maximal k different pages are requested
- Phase i ≥ 2 is the maximal subsequence of σ from the end of pase i-1 till maximal k different pages are requested

For example with k = 3:

$$\sigma = \left(\overbrace{a, b, d, a}^{phase\ 1}, \overbrace{e, a, f, a, f}^{phase\ 2}, \overbrace{b, d, a}^{phase\ 3}, \overbrace{c, c, d}^{phase\ 4}\right)$$

A marking algorithm (implicitly or explicitly) maintains whether a page is marked or not. At the beginning of each phase are all pages unmarked. Is a page requested during a phase it gets marked. An algorithm is a marking algorithm **iff** it never evicts a marked page from cache. That means pages which are used during a phase will not be evicted.

Proposition 1.3: LRU and **FWF** are marking algorithm.

Proof: At the beginning of each phase (except for the first one) **FWF** has a cache miss and cleared the cache. that means we have k empty pages. In every phase are maximal k different pages requested, so there will be now eviction during the phase. So **FWF** is a marking algorithm.
Let's assume **LRU** is not a marking algorithm. Then there is an instance σ where **LRU** a marked page x in phase i evicted. Let σt the request in phase i where x is evicted. Since x is marked there has to be a earlier request σt* for x in the same phase, so t* < t. After t* x is the caches newest page, so to got evicted at t the sequence σt*+1,...,σt has to request at least k from x different pages. That implies the phase i has requested at least k+1 different pages which is a contradictory to the phase definition. So **LRU** has to be a marking algorithm.

命题 1.4： 每个标记算法 都是严格 k-竞争性 的。

证明： 设 σ 是分页问题的一个实例，l 是 σ 的阶段数。如果 l = 1，则每个标记算法都是最优的，且最优的离线算法不会更好。
我们假设 l ≥ 2。每个标记算法的代价，例如σ，上界为 l · k，因为在每个阶段中，标记算法不能在不驱逐任何被标记页面的情况下驱逐超过 k 个页面。
现在我们尝试证明最优离线算法对于 σ 至少驱逐 k+l-2 页，第一阶段驱逐 k 页，之后每个阶段至少驱逐一页，除了最后一个阶段。为证明此点，我们定义 σ 的 l-2 个不相交子序列。
子序列 i ∈ {1,...,l-2} 从第 i+1 阶段的第二个位置开始，到第 i+2 阶段的第一个位置结束。
设 x 为第 i+1 阶段的第一页。在子序列 i 开始时，最优离线算法的缓存中有页面 x 和最多 k-1 个不同页面。子序列 i 中有 k 个不同于 x 的页面请求，因此最优离线算法必须为每个子序列驱逐至少一页。由于第一阶段开始时缓存仍为空，最优离线算法在第一阶段导致 k 次驱逐。这表明

$$wA(\sigma) \leq l \cdot k \leq (k+l-2)k \leq OPT(\sigma) \cdot k$$

推论 1.5：LRU 和 FWF 是 严格 k-竞争性 的。

练习：证明FIFO不是标记算法，但它是严格 k-竞争的。

如果不存在常数 r 使得在线算法 A 是 r-竞争的，我们称 A 为**非竞争的**

命题 1.6：LFU和LIFO是非竞争的。

证明：设 l ≥ 2 为常数，k ≥ 2 为缓存大小。不同的缓存页面编号为 1,...,k+1。我们观察以下序列：

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

第一个页面 1 被请求 l 次，接着是页面 2，依此类推。最后，有 (l-1) 次交替请求页面 k 和 k+1。

LFU和LIFO用页面 1 到 k 填满它们的缓存。当请求页面 k+1 时，页面 k 被驱逐，反之亦然。这意味着子序列 (k, k+1)l-1 的每次请求都会驱逐一个页面。此外，首次使用页面 1 到 (k-1) 时会有 k-1 次缓存未命中。因此，LFU和LIFO总共驱逐了恰好 k-1+2(l-1) 个页面。
现在我们必须证明，对于每个常数τ∈ℜ和每个常数 r ≤ 1，存在一个 l 使得

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

这等同于

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

要满足这个不等式，你只需选择足够大的 l。因此，LFU 和 LIFO 并不具备竞争力。

命题1.7：不存在r-竞争性的确定性在线分页算法，其中 r < k。

Proposition 1.4: Every marking algorithm **is strictly k-competitive**.

Proof: Let σ be an instance for the paging problem and l the number of phases for σ. Is l = 1 then is every marking algorithm optimal and the optimal offline algorithm cannot be better.
We assume l ≥ 2. the cost of every marking algorithm, for instance, σ is bounded from above with l · k because in every phase a marking algorithm cannot evict more than k pages without evicting one marked page.
Now we try to show that the optimal offline algorithm evicts at least k+l-2 pages for σ, k in the first phase and at least one for every following phase except for the last one. For proof lets define l-2 disjunct subsequences of σ.
Subsequence i ∈ {1,...,l-2} starts at the second position of phase i+1 and end with the first position of phase i+2.
Let x be the first page of phase i+1. At the beginning of subsequence i there is page x and at most k-1 different pages in the optimal offline algorithms cache. In subsequence i are k page request different from x, so the optimal offline algorithm has to evict at least one page for every subsequence. Since at phase 1 beginning the cache is still empty, the optimal offline algorithm causes k evictions during the first phase. That shows that

$$wA(\sigma) \leq l \cdot k \leq (k+l-2)k \leq OPT(\sigma) \cdot k$$

Corollary 1.5: LRU and **FWF** are **strictly k-competitive**.

Excercise: Show that **FIFO** is no marking algorithm, but **strictly k-competitive**.

Is there no constant r for which an online algorithm A is r-competitive, we call A **not competitive**

Proposition 1.6: LFU and **LIFO** are **not competitive**.

Proof: Let l ≥ 2 a constant, k ≥ 2 the cache size. The different cache pages are nubered 1,...,k+1. We look at the following sequence:

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

The first page 1 is requested l times than page 2 and so one. At the end, there are (l-1) alternating requests for page k and k+1.
LFU and **LIFO** fill their cache with pages 1-k. When page k+1 is requested page k is evicted and vice versa. That means every request of subsequence (k,k+1)l-1 evicts one page. In addition, their are k-1 cache misses for the first time use of pages 1-(k-1). So **LFU** and **LIFO** evict exact k-1+2(l-1) pages.
Now we must show that for every constant τ∈ℜ and every constant r ≤ 1 there exists an l so that

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

which is equal to

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

To satisfy this inequality you just have to choose l sufficient big. So **LFU** and **LIFO** are not competitive.

Proposition 1.7: There is **no r-competetive** deterministic online algorithm for paging with **r < k**.

最后一个命题的证明相当冗长，基于LFD是一个最优的离线算法这一陈述。有兴趣的读者可以查阅Borodin和El-Yaniv的书（见下方参考资料）。

问题是我们是否能做得更好。为此，我们必须抛弃确定性方法，开始对算法进行随机化。显然，如果算法是随机化的，敌手要惩罚你的算法就困难得多。

随机分页将在后续的某个例子中讨论……

The proof for this last proposition is rather long and based of the statement that **LFD** is an optimal offline algorithm. The interested reader can look it up in the book of Borodin and El-Yaniv (see sources below).

The Question is whether we could do better. For that, we have to leave the deterministic approach behind us and start to randomize our algorithm. Clearly, its much harder for the adversary to punish your algorithm if it's randomized.

Randomized paging will be discussed in one of next examples...

第28章：排序

参数	描述
稳定性	如果排序算法在排序后保持相等元素的相对顺序，则称该算法为稳定的。
原地	如果一个排序算法只使用O(1)的辅助内存（不包括需要排序的数组）进行排序，则称该算法是就地排序算法。
最佳情况复杂度	如果一个排序算法的运行时间对于所有可能的输入至少是T(n)，则该算法的最佳情况时间复杂度为O(T(n))。
平均情况复杂度	如果一个排序算法的运行时间在所有可能输入上的平均值为T(n)，则该算法的平均情况时间复杂度为O(T(n))。
最坏情况复杂度	如果一个排序算法的运行时间最多为T(n)，则该算法的最坏情况时间复杂度为O(T(n))。

第28.1节：排序的稳定性

排序的稳定性是指排序算法是否保持原始输入中相等键的相对顺序在结果输出中不变。

如果两个具有相等键的对象在排序后的输出中保持与它们在未排序输入数组中相同的顺序，则该排序算法被称为稳定的。

考虑一组对：

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

现在我们将使用每对的第一个元素对列表进行排序。

对该列表进行稳定排序将输出以下列表：

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

因为(9, 3)在原列表中也出现在(9, 7)之后。

一个不稳定排序将输出以下列表：

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

不稳定排序可能生成与稳定排序相同的输出，但并非总是如此。

著名的稳定排序算法：

- 归并排序
- 插入排序
- 基数排序
- Tim排序
- 冒泡排序

著名的不稳定排序：

- 堆排序
- 快速排序

Chapter 28: Sorting

Parameter	Description
Stability	A sorting algorithm is **stable** if it preserves the relative order of equal elements after sorting.
In place	A sorting algorithm is **in-place** if it sorts using only `O(1)` auxiliary memory (not counting the array that needs to be sorted).
Best case complexity	A sorting algorithm has a best case time complexity of `O(T(n))` if its running time is **at least** `T(n)` for all possible inputs.
Average case complexity	A sorting algorithm has an average case time complexity of `O(T(n))` if its running time, **averaged over all possible inputs**, is `T(n)`.
Worst case complexity	A sorting algorithm has a worst case time complexity of `O(T(n))` if its running time is **at most** `T(n)`.

Section 28.1: Stability in Sorting

Stability in sorting means whether a sort algorithm maintains the relative order of the equals keys of the original input in the result output.

So a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Consider a list of pairs:

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

Now we will sort the list using the first element of each pair.

A **stable sorting** of this list will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

Because `(9, 3)` appears after `(9, 7)` in the original list as well.

An **unstable sorting** will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

Unstable sort may generate the same output as the stable sort but not always.

Well-known stable sorts:

- Merge sort
- Insertion sort
- Radix sort
- Tim sort
- Bubble Sort

Well-known unstable sorts:

- Heap sort
- Quick sort

第29章：冒泡排序

参数	描述
稳定	是
原地	是
最佳情况复杂度	O(n)
平均情况复杂度	O(n^2)
最坏情况复杂度	O(n^2)
空间复杂度	O(1)

第29.1节：冒泡排序

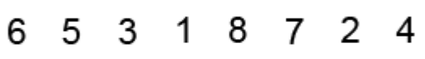
冒泡排序BubbleSort比较无序列表中每对相邻元素，如果它们的顺序不正确，则交换这两个元素。

下面的例子展示了列表{6,5,3,1,8,7,2,4}上的冒泡排序（每一步比较的元素对用"**"括起）：

```
{6,5,3,1,8,7,2,4}
{**5,6**,3,1,8,7,2,4} -- 5 < 6 -> 交换
{5,**3,6**,1,8,7,2,4} -- 3 < 6 -> 交换
{5,3,**1,6**,8,7,2,4} -- 1 < 6 -> 交换
{5,3,1,**6,8**,7,2,4} -- 8 > 6 -> 不交换
{5,3,1,6,**7,8**,2,4} -- 7 < 8 -> 交换
{5,3,1,6,7,**2,8**,4} -- 2 < 8 -> 交换
{5,3,1,6,7,2,**4,8**} -- 4 < 8 -> 交换
```

经过一次遍历列表后，我们得到{5,3,1,6,7,2,4,8}。注意，数组中最大的未排序值（本例中为8）总是会到达其最终位置。因此，为确保列表已排序，对于长度为n的列表，我们必须迭代n-1次。

图示：



第29.2节：C和C++中的实现

在C++中BubbleSort的示例实现：

```
void bubbleSort(vector<int>numbers)
{
    for(int i = numbers.size() - 1; i >= 0; i--) {
        for(int j = 1; j <= i; j++) {
            if(numbers[j-1] > numbers[j]) {
                swap(numbers[j-1],numbers(j));
```


Chapter 29: Bubble Sort

Parameter	Description
Stable	Yes
In place	Yes
Best case complexity	O(n)
Average case complexity	O(n^2)
Worst case complexity	O(n^2)
Space complexity	O(1)

Section 29.1: Bubble Sort

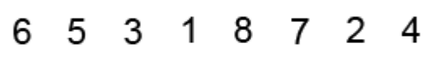
The `BubbleSort` compares each successive pair of elements in an unordered list and inverts the elements if they are not in order.

The following example illustrates the bubble sort on the list `{6,5,3,1,8,7,2,4}` (pairs that were compared in each step are encapsulated in '**'):

```
{6,5,3,1,8,7,2,4}
{**5,6**,3,1,8,7,2,4} -- 5 < 6 -> swap
{5,**3,6**,1,8,7,2,4} -- 3 < 6 -> swap
{5,3,**1,6**,8,7,2,4} -- 1 < 6 -> swap
{5,3,1,**6,8**,7,2,4} -- 8 > 6 -> no swap
{5,3,1,6,**7,8**,2,4} -- 7 < 8 -> swap
{5,3,1,6,7,**2,8**,4} -- 2 < 8 -> swap
{5,3,1,6,7,2,**4,8**} -- 4 < 8 -> swap
```

After one iteration through the list, we have `{5,3,1,6,7,2,4,8}`. Note that the greatest unsorted value in the array (8 in this case) will always reach its final position. Thus, to be sure the list is sorted we must iterate n-1 times for lists of length n.

Graphic:



Section 29.2: Implementation in C & C++

An example implementation of `BubbleSort` in `C++`:

```
void bubbleSort(vector<int>numbers)
{
    for(int i = numbers.size() - 1; i >= 0; i--) {
        for(int j = 1; j <= i; j++) {
            if(numbers[j-1] > numbers[j]) {
                swap(numbers[j-1],numbers(j));
```

```
            }
        }
    }
}
```

C语言实现

```
void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* 交换 */

t       = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

带指针的冒泡排序

```
void pointer_bubble_sort(long * list, long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if ( * (list + d ) > *(list+d+1))
      {
        /* 交换 */

t       = * (list + d );
        * (list + d )   = * (list + d + 1 );
        * (list + d + 1) = t;
      }
    }
  }
}
```

第29.3节：C#中的实现

冒泡排序也称为下沉排序。它是一种简单的排序算法，通过重复遍历待排序列表，比较每对相邻元素，如果顺序错误则交换它们。

冒泡排序示例


```
            }
        }
    }
}
```

C Implementation

```
void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */

        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

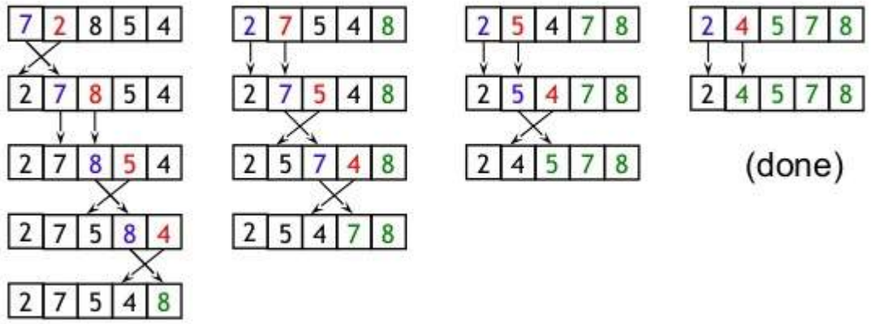
Bubble Sort with pointer

```
void pointer_bubble_sort(long * list, long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if ( * (list + d ) > *(list+d+1))
      {
        /* Swapping */

        t         = * (list + d );
        * (list + d )   = * (list + d + 1 );
        * (list + d + 1) = t;
      }
    }
  }
}
```

Section 29.3: Implementation in C#

Bubble sort is also known as **Sinking Sort**. It is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Bubble sort example



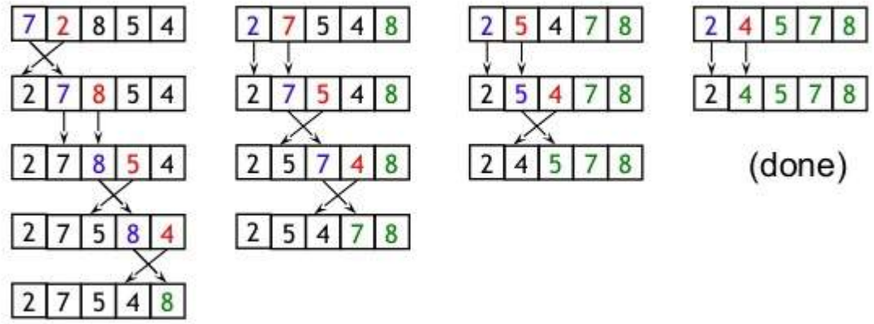
冒泡排序的实现

我使用C#语言实现了冒泡排序算法

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--)
            {
                if (input[j] <= input[j + 1]) continue;
                var temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
SortBubble(input);
        return input;
    }
}
```

第29.4节：Python实现

```
#!/usr/bin/python

input_list = [10,1,2,11]

for i in range(len(input_list)):
   for j in range(i):
      if int(input_list[j]) > int(input_list[j+1]):
input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list
```

Implementation of Bubble Sort

I used C# language to implement bubble sort algorithm

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--)
            {
                if (input[j] <= input[j + 1]) continue;
                var temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBubble(input);
        return input;
    }
}
```

Section 29.4: Python Implementation

```
#!/usr/bin/python

input_list = [10,1,2,11]

for i in range(len(input_list)):
  for j in range(i):
    if int(input_list[j]) > int(input_list[j+1]):
      input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list
```

第29.5节：Java中的实现

```
public class 我的冒泡排序 {

    public static void 冒泡排序(int 数组[]) {//主要逻辑
        int n = 数组.长度;
        int k;
        for (int m = n; m >= 0; m--) {
             for (int i = 0; i < n - 1; i++) {
               k = i + 1;
                  if (数组[i] > 数组[k]) {
                   交换数字(i, k, 数组);
                  }
             }
打印数字(数组);
        }
    }

    private static void 交换数字(int i, int j, int[] 数组) {

        int 临时变量;
临时变量 = 数组[i];
       数组[i] = 数组[j];
       数组[j] = 临时变量;
    }

    private static void 打印数字(int[] 输入) {

        for (int i = 0; i < 输入.长度; i++) {
            System.out.print(输入[i] + ", ");
        }
        System.out.println("");
    }

    public static void main(String[] args) {
        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
        bubble_srt(input);

    }
}
```

第29.6节：Javascript中的实现

```
function bubbleSort(a)
          {
var swapped;
                do {
swapped = false;
                    for (var i=0; i < a.length-1; i++) {
                        if (a[i] > a[i+1]) {
var temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                            swapped = true;
                        }
                    }
                } while (swapped);
            }

var a = [3, 203, 34, 746, 200, 984, 198, 764, 9];
```


Section 29.5: Implementation in Java

```
public class MyBubbleSort {

    public static void bubble_srt(int array[]) {//main logic
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
                if (array[i] > array[k]) {
                    swapNumbers(i, k, array);
                }
            }
            printNumbers(array);
        }
    }

    private static void swapNumbers(int i, int j, int[] array) {

        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void printNumbers(int[] input) {

        for (int i = 0; i < input.length; i++) {
            System.out.print(input[i] + ", ");
        }
        System.out.println("\n");
    }

    public static void main(String[] args) {
        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
        bubble_srt(input);

    }
}
```

Section 29.6: Implementation in Javascript

```
        function bubbleSort(a)
          {
                var swapped;
                do {
                    swapped = false;
                    for (var i=0; i < a.length-1; i++) {
                        if (a[i] > a[i+1]) {
                            var temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                            swapped = true;
                        }
                    }
                } while (swapped);
            }

   var a = [3, 203, 34, 746, 200, 984, 198, 764, 9];
```

```
bubbleSort(a);
console.log(a); //输出 [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]
```



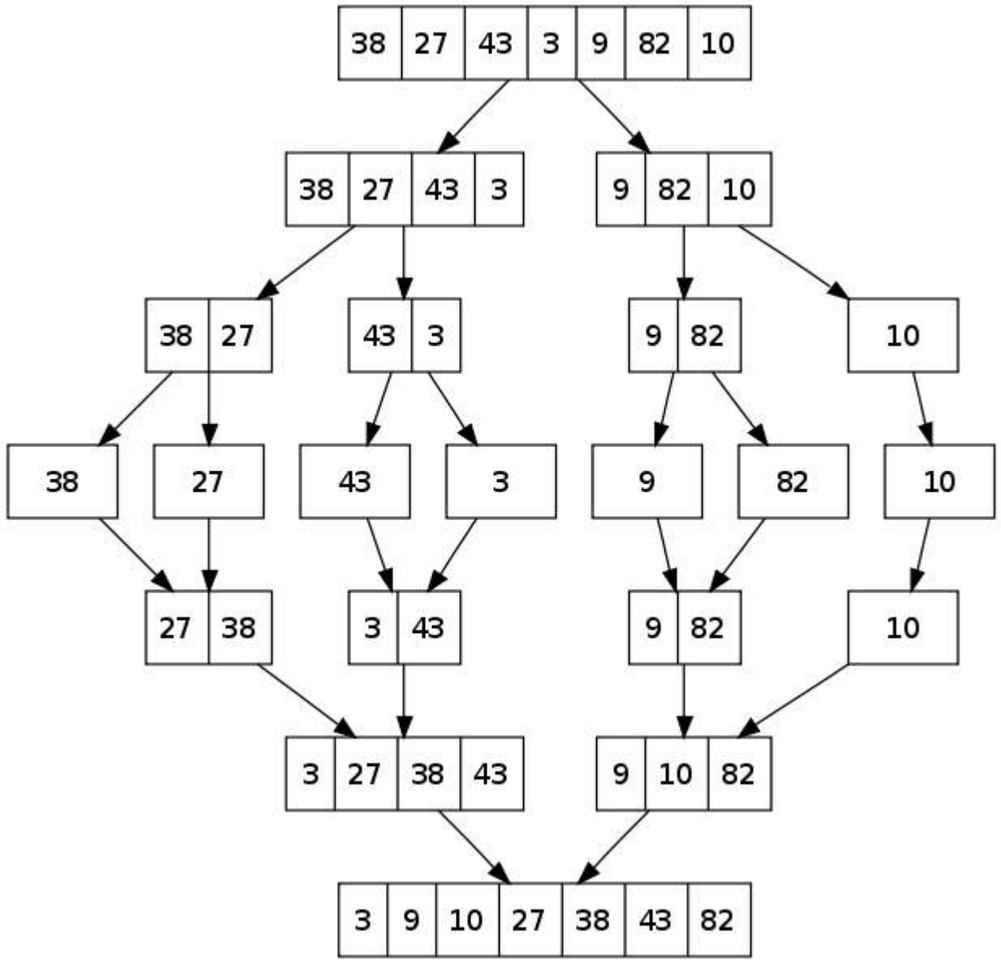
```
bubbleSort(a);
console.log(a); //logs [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]
```

第30章：归并排序

第30.1节：归并排序基础

归并排序是一种分治算法。它将长度为n的输入列表不断对半分割，直到
n个大小为1的列表。然后，将成对的列表合并，每一步都将两个列表中较小的第一个元素
加入。通过连续的合并和对第一个元素的比较，构建出排序后的列表。

一个例子：



时间复杂度: T(n) = 2T(n/2) + Θ(n)

上述递推式可以通过递归树方法或主方法求解。它属于主方法的第二种情况，递推式的解为 Θ(nLogn)。归并排序的时间复杂度在三种情况下（最坏、平均和最好）均为 Θ(nLogn)，因为归并排序总是将数组分成两半，并且合并两半所需时间为线性时间。

辅助空间: O(n)

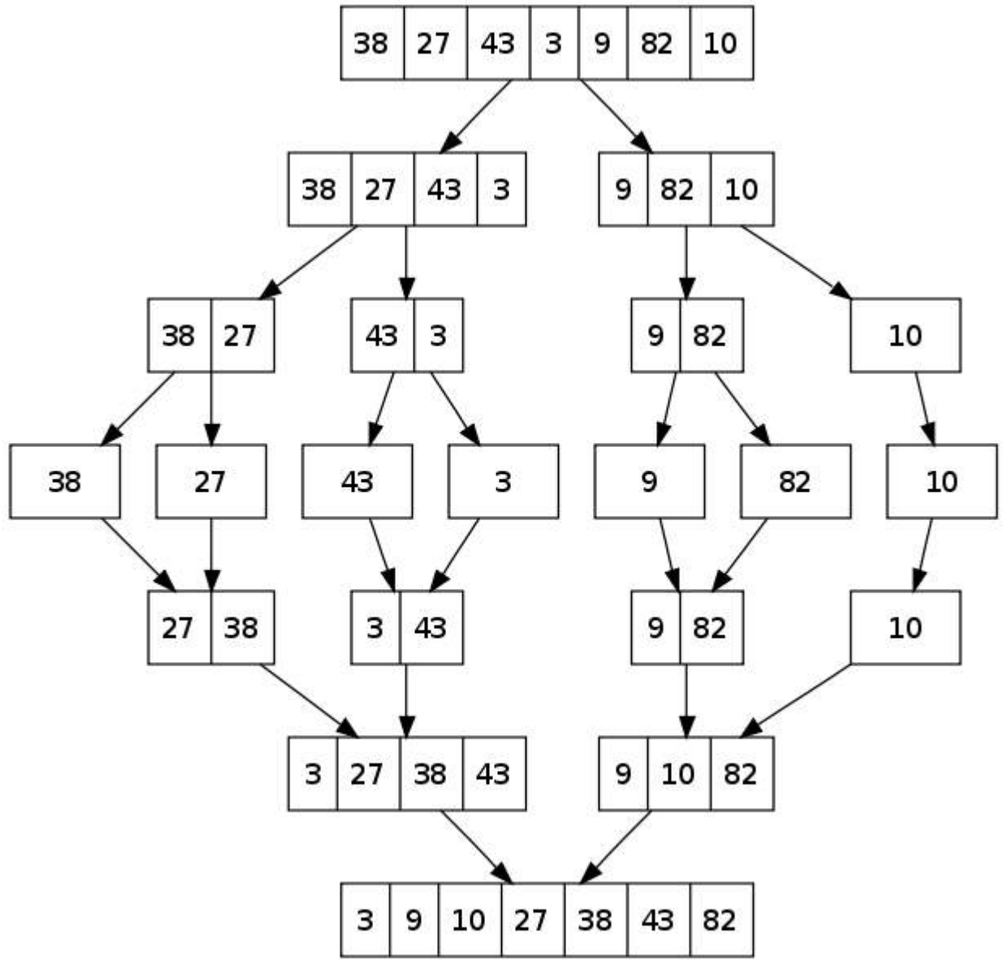
算法范式: 分治法

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

Merge Sort is a divide-and-conquer algorithm. It divides the input list of length n in half successively until there are n lists of size 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built.

An example:



Time Complexity: T(n) = 2T(n/2) + Θ(n)

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn). Time complexity of *Merge Sort* is Θ(nLogn) in all 3 cases (*worst, average and best*) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

原地排序：不是典型实现

稳定：是

第30.2节：Go语言中的归并排序实现

```
包 main

导入 "fmt"

func mergeSort(a []int) []int {
    如果 len(a) < 2 {
        返回 a
    }
m := (len(a)) / 2

    f := mergeSort(a[:m])
    s := mergeSort(a[m:])

    返回 merge(f, s)
}

func merge(f []int, s []int) []int {
   定义变量 i, j int
size := len(f) + len(s)

    a := make([]int, size, size)

    for z := 0; z < size; z++ {
        lenF := len(f)
        lenS := len(s)

        if i > lenF-1 && j <= lenS-1 {
            a[z] = s[j]
            j++
        } else if j > lenS-1 && i <= lenF-1 {
            a[z] = f[i]
            i++
        } else if f[i] < s[j] {
            a[z] = f[i]
            i++
        } else {
a[z] = s[j]
            j++
        }
    }

    返回 a
}

func main() {
a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
    fmt.Println(a)
fmt.Println(mergeSort(a))
}
```

第30.3节：C和C#中的归并排序实现

C语言归并排序

Sorting In Place: Not in a typical implementation

Stable: Yes

Section 30.2: Merge Sort Implementation in Go

```
package main

import "fmt"

func mergeSort(a []int) []int {
    if len(a) < 2 {
        return a
    }
    m := (len(a)) / 2

    f := mergeSort(a[:m])
    s := mergeSort(a[m:])

    return merge(f, s)
}

func merge(f []int, s []int) []int {
    var i, j int
    size := len(f) + len(s)

    a := make([]int, size, size)

    for z := 0; z < size; z++ {
        lenF := len(f)
        lenS := len(s)

        if i > lenF-1 && j <= lenS-1 {
            a[z] = s[j]
            j++
        } else if j > lenS-1 && i <= lenF-1 {
            a[z] = f[i]
            i++
        } else if f[i] < s[j] {
            a[z] = f[i]
            i++
        } else {
            a[z] = s[j]
            j++
        }
    }

    return a
}

func main() {
    a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
    fmt.Println(a)
    fmt.Println(mergeSort(a))
}
```

Section 30.3: Merge Sort Implementation in C & C#

C Merge Sort

```c
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // 两个临时数组，用于存放待合并的两个
数组
  int n1,n2,i,j,k;
  n1=m-l+1;
n2=h-m;

  for(i=0; i<n1; i++)
    arr1[i]=arr[l+i];
       for(j=0; j<n2; j++)
arr2[j]=arr[m+j+1];

arr1[i]=9999;  // 标记每个临时数组的结束
  arr2[j]=9999;

i=0;
  j=0;
  for(k=l; k<=h; k++) { //合并两个已排序数组的过程
    if(arr1[i]<=arr2[j])
arr[k]=arr1[i++];
    否则
arr[k]=arr2[j++];
  }

  return 0;
}

int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high) {
mid=(low+high)/2;
    // 分治法
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
    // 合并
merge(arr,low,mid,high);
  }

  return 0;
}
```

C# 归并排序

```c
public class MergeSort
    {
        static void Merge(int[] input, int l, int m, int r)
        {
            int i, j;
var n1 = m - l + 1;
            var n2 = r - m;

            var left = new int[n1];
            var right = new int[n2];

            for (i = 0; i < n1; i++)
            {
left[i] = input[l + i];
            }
```



```c
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // Two temporary arrays to
  hold the two arrays to be merged
  int n1,n2,i,j,k;
  n1=m-l+1;
  n2=h-m;

  for(i=0; i<n1; i++)
    arr1[i]=arr[l+i];
  for(j=0; j<n2; j++)
    arr2[j]=arr[m+j+1];

  arr1[i]=9999;  // To mark the end of each temporary array
  arr2[j]=9999;

  i=0;
  j=0;
  for(k=l; k<=h; k++) { //process of combining two sorted arrays
    if(arr1[i]<=arr2[j])
      arr[k]=arr1[i++];
    else
      arr[k]=arr2[j++];
  }

  return 0;
}

int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high) {
    mid=(low+high)/2;
    // Divide and Conquer
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
    // Combine
    merge(arr,low,mid,high);
  }

  return 0;
}
```

C# Merge Sort

```c
public class MergeSort
    {
        static void Merge(int[] input, int l, int m, int r)
        {
            int i, j;
            var n1 = m - l + 1;
            var n2 = r - m;

            var left = new int[n1];
            var right = new int[n2];

            for (i = 0; i < n1; i++)
            {
                left[i] = input[l + i];
            }
```

```
            for (j = 0; j < n2; j++)
            {
right[j] = input[m + j + 1];
            }

i = 0;
            j = 0;
            var k = l;

            while (i < n1 && j < n2)
            {
                if (left[i] <= right[j])
                {
input[k] = left[i];
                    i++;
                }
                否则
                {
input[k] = right[j];
                    j++;
                }
k++;
            }

            while (i < n1)
            {
input[k] = left[i];
                i++;
k++;
            }

            while (j < n2)
            {
input[k] = right[j];
                j++;
k++;
            }
        }

        static void SortMerge(int[] input, int l, int r)
        {
            if (l < r)
            {
                int m = l + (r - l) / 2;
                SortMerge(input, l, m);
                SortMerge(input, m + 1, r);
                Merge(input, l, m, r);
            }
        }

        public static int[] Main(int[] input)
        {
SortMerge(input, 0, input.Length - 1);
            return input;
        }
    }
```

第30.4节：Java中的归并排序实现

下面是使用泛型方法的Java实现。这是与上面介绍的相同的算法。


```
            for (j = 0; j < n2; j++)
            {
                right[j] = input[m + j + 1];
            }

            i = 0;
            j = 0;
            var k = l;

            while (i < n1 && j < n2)
            {
                if (left[i] <= right[j])
                {
                    input[k] = left[i];
                    i++;
                }
                else
                {
                    input[k] = right[j];
                    j++;
                }
                k++;
            }

            while (i < n1)
            {
                input[k] = left[i];
                i++;
                k++;
            }

            while (j < n2)
            {
                input[k] = right[j];
                j++;
                k++;
            }
        }

        static void SortMerge(int[] input, int l, int r)
        {
            if (l < r)
            {
                int m = l + (r - l) / 2;
                SortMerge(input, l, m);
                SortMerge(input, m + 1, r);
                Merge(input, l, m, r);
            }
        }

        public static int[] Main(int[] input)
        {
            SortMerge(input, 0, input.Length - 1);
            return input;
        }
    }
```

Section 30.4: Merge Sort Implementation in Java

Below there is the implementation in Java using a generics approach. It is the same algorithm, which is presented above.

```
public interface 原地排序<T extends Comparable<T>> {
void 排序(final T[] 元素); }


public class 归并排序 < T extends Comparable < T >> 实现 原地排序 < T > {


@Override
public void 排序(T[] 元素) {
    T[] arr = (T[]) new Comparable[元素.length];
    排序(元素, arr, 0, 元素.length - 1);
}

// 我们检查两边然后合并它们
private void 排序(T[] 元素, T[] arr, int 低, int 高) {
    if (低 >= 高) return;
    int 中间 = 低 + (高 - 低) / 2;
    排序(元素, arr, 低, 中间);
    排序(元素, arr, 中间 + 1, 高);
    合并(元素, arr, 低, 高, 中间);
}


private void 合并(T[] a, T[] b, int 低, int 高, int 中间) {
    int i = 低;
    int j = 中间 + 1;

    // 我们选择两个中较小的元素。然后将其放入 b
    for (int k = 低; k <= 高; k++) {

        if (i <= mid && j <= high) {
            if (a[i].compareTo(a[j]) >= 0) {
                b[k] = a[j++];
            } else {
b[k] = a[i++];
            }
        } else if (j > high && i <= mid) {
            b[k] = a[i++];
        } else if (i > mid && j <= high) {
            b[k] = a[j++];
        }
    }

    for (int n = low; n <= high; n++) {
        a[n] = b[n];
    }}}
```

第30.5节：Python中的归并排序实现

```
def merge(X, Y):
    " 合并两个已排序的列表 "
    p1 = p2 = 0
out = []
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
out.append(X[p1])
            p1 += 1
        else:
out.append(Y[p2])
            p2 += 1
out += X[p1:] + Y[p2:]
```



```
public interface InPlaceSort<T extends Comparable<T>> {
void sort(final T[] elements); }


public class MergeSort < T extends Comparable < T >> implements InPlaceSort < T > {


@Override
public void sort(T[] elements) {
    T[] arr = (T[]) new Comparable[elements.length];
    sort(elements, arr, 0, elements.length - 1);
}

// We check both our sides and then merge them
private void sort(T[] elements, T[] arr, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    sort(elements, arr, low, mid);
    sort(elements, arr, mid + 1, high);
    merge(elements, arr, low, high, mid);
}


private void merge(T[] a, T[] b, int low, int high, int mid) {
    int i = low;
    int j = mid + 1;

    // We select the smallest element of the two. And then we put it into b
    for (int k = low; k <= high; k++) {

        if (i <= mid && j <= high) {
            if (a[i].compareTo(a[j]) >= 0) {
                b[k] = a[j++];
            } else {
                b[k] = a[i++];
            }
        } else if (j > high && i <= mid) {
            b[k] = a[i++];
        } else if (i > mid && j <= high) {
            b[k] = a[j++];
        }
    }

    for (int n = low; n <= high; n++) {
        a[n] = b[n];
    }}}
```

Section 30.5: Merge Sort Implementation in Python

```
def merge(X, Y):
    " merge two sorted lists "
    p1 = p2 = 0
    out = []
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
            out.append(X[p1])
            p1 += 1
        else:
            out.append(Y[p2])
            p2 += 1
    out += X[p1:] + Y[p2:]
```

```
    return out

def mergeSort(A):
        if len(A) <= 1:
            return A
    if len(A) == 2:
        return sorted(A)

mid = len(A) / 2
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))

if __name__ == "__main__":
# 生成20个随机数并排序
    A = [randint(1, 100) for i in xrange(20)]
    print mergeSort(A)
```

第30.6节：自底向上的Java实现

```
public class MergeSortBU {
    private static Integer[] array = { 4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70,
60,80,0,9,67,54,51,52,24,54,7 };

    public MergeSortBU() {
    }

    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid, int hi) {

        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }

        int i = lo;
        int j = 中间 + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
arrayToSort[k] = aux[j++];
            else if (j > hi)
arrayToSort[k] = aux[i++];
            else if (isLess(aux[i], aux[j])) {
                arrayToSort[k] = aux[i++];
            } else {
arrayToSort[k] = aux[j++];
            }

        }
    }

    public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
        int N = arrayToSort.length;
        for (int sz = 1; sz < N; sz = sz + sz) {
            for (int low = 0; low < N; low = low + sz + sz) {
                System.out.println("Size:"+ sz);
merge(arrayToSort, aux, low, low + sz -1 ,Math.min(low + sz + sz - 1, N - 1));
                print(arrayToSort);
            }
        }

    }

    public static boolean isLess(Comparable a, Comparable b) {
        return a.compareTo(b) <= 0;
```



```
    return out

def mergeSort(A):
    if len(A) <= 1:
        return A
    if len(A) == 2:
        return sorted(A)

    mid = len(A) / 2
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))

if __name__ == "__main__":
    # Generate 20 random numbers and sort them
    A = [randint(1, 100) for i in xrange(20)]
    print mergeSort(A)
```

Section 30.6: Bottoms-up Java Implementation

```
public class MergeSortBU {
    private static Integer[] array = { 4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70,
60,80,0,9,67,54,51,52,24,54,7 };

    public MergeSortBU() {
    }

    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid, int hi) {

        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }

        int i = lo;
        int j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
                arrayToSort[k] = aux[j++];
            else if (j > hi)
                arrayToSort[k] = aux[i++];
            else if (isLess(aux[i], aux[j])) {
                arrayToSort[k] = aux[i++];
            } else {
                arrayToSort[k] = aux[j++];
            }

        }
    }

    public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
        int N = arrayToSort.length;
        for (int sz = 1; sz < N; sz = sz + sz) {
            for (int low = 0; low < N; low = low + sz + sz) {
                System.out.println("Size:"+ sz);
                merge(arrayToSort, aux, low, low + sz -1 ,Math.min(low + sz + sz - 1, N - 1));
                print(arrayToSort);
            }
        }

    }

    public static boolean isLess(Comparable a, Comparable b) {
        return a.compareTo(b) <= 0;
```

```
    }

    private static void print(Comparable[] array)
{http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        StringBuffer buffer = new
StringBuffer();http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        for (Comparable value : array) {
buffer.append(value);
            buffer.append(' ');
        }
        System.out.println(buffer);
    }

    public static void main(String[] args) {
        Comparable[] aux = new Comparable[array.length];
        print(array);
MergeSortBU.sort(array, aux, 0, array.length - 1);
    }
}
```



```
    }

    private static void print(Comparable[] array)
{http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        StringBuffer buffer = new
StringBuffer();http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        for (Comparable value : array) {
            buffer.append(value);
            buffer.append(' ');
        }
        System.out.println(buffer);
    }

    public static void main(String[] args) {
        Comparable[] aux = new Comparable[array.length];
        print(array);
        MergeSortBU.sort(array, aux, 0, array.length - 1);
    }
}
```

第31章：插入排序

第31.1节：Haskell实现

```
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: Ord a => a-> [a] -> [a]
insert n [] = [n]
insert n (x:xs) | n <= x    = (n:x:xs)
                | otherwise = x:insert n xs
```


Chapter 31: Insertion Sort

Section 31.1: Haskell Implementation

```
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: Ord a => a-> [a] -> [a]
insert n [] = [n]
insert n (x:xs) | n <= x    = (n:x:xs)
                | otherwise = x:insert n xs
```

第32章：桶排序

第32.1节：C#实现

```
public class BucketSort
{
    public static void SortBucket(ref int[] input)
    {
        int minValue = input[0];
        int maxValue = input[0];
        int k = 0;

        for (int i = input.Length - 1; i >= 1; i--)
        {
            if (input[i] > maxValue) maxValue = input[i];
            if (input[i] < minValue) minValue = input[i];
        }

List<int>[] bucket = new List<int>[maxValue - minValue + 1];

        for (int i = bucket.Length - 1; i >= 0; i--)
        {
bucket[i] = new List<int>();
        }

        foreach (int i in input)
        {
bucket[i - minValue].Add(i);
        }

        foreach (List<int> b in bucket)
        {
            if (b.Count > 0)
            {
                foreach (int t in b)
                {
input[k] = t;
                    k++;
                }
            }
        }
    }

    public static int[] Main(int[] input)
    {
SortBucket(ref input);
        return input;
    }
}
```


Chapter 32: Bucket Sort

Section 32.1: C# Implementation

```
public class BucketSort
{
    public static void SortBucket(ref int[] input)
    {
        int minValue = input[0];
        int maxValue = input[0];
        int k = 0;

        for (int i = input.Length - 1; i >= 1; i--)
        {
            if (input[i] > maxValue) maxValue = input[i];
            if (input[i] < minValue) minValue = input[i];
        }

        List<int>[] bucket = new List<int>[maxValue - minValue + 1];

        for (int i = bucket.Length - 1; i >= 0; i--)
        {
            bucket[i] = new List<int>();
        }

        foreach (int i in input)
        {
            bucket[i - minValue].Add(i);
        }

        foreach (List<int> b in bucket)
        {
            if (b.Count > 0)
            {
                foreach (int t in b)
                {
                    input[k] = t;
                    k++;
                }
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBucket(ref input);
        return input;
    }
}
```

第33章：快速排序

第33.1节：快速排序基础

快速排序是一种排序算法，它选择一个元素（“枢轴”），并重新排列数组形成两个分区，使得所有小于枢轴的元素位于其前面，所有大于枢轴的元素位于其后。然后递归地对这些分区应用该算法，直到列表排序完成。

1. Lomuto划分方案机制：

该方案选择一个枢轴，通常是数组中的最后一个元素。算法维护一个变量i作为放置枢轴的索引，每当找到一个小于或等于枢轴的元素时，索引i就会递增，并将该元素放置在枢轴之前。

```
partition(A, low, high) 是
pivot := A[high]
i := low
对于 j 从 low 到 high – 1 执行
    如果 A[j] ≤ pivot 则
        交换 A[i] 和 A[j]
        i := i + 1
交换 A[i] 和 A[high]
返回 i
```

快速排序机制：

```
quicksort(A, low, high) 是
如果 low < high 则
p := partition(A, low, high)
    quicksort(A, low, p – 1)
    quicksort(A, p + 1, high)
```

快速排序示例：

Chapter 33: Quicksort

Section 33.1: Quicksort Basics

Quicksort is a sorting algorithm that picks an element ("the pivot") and reorders the array forming two partitions such that all elements less than the pivot come before it and all elements greater come after. The algorithm is then applied recursively to the partitions until the list is sorted.

1. Lomuto partition scheme mechanism :

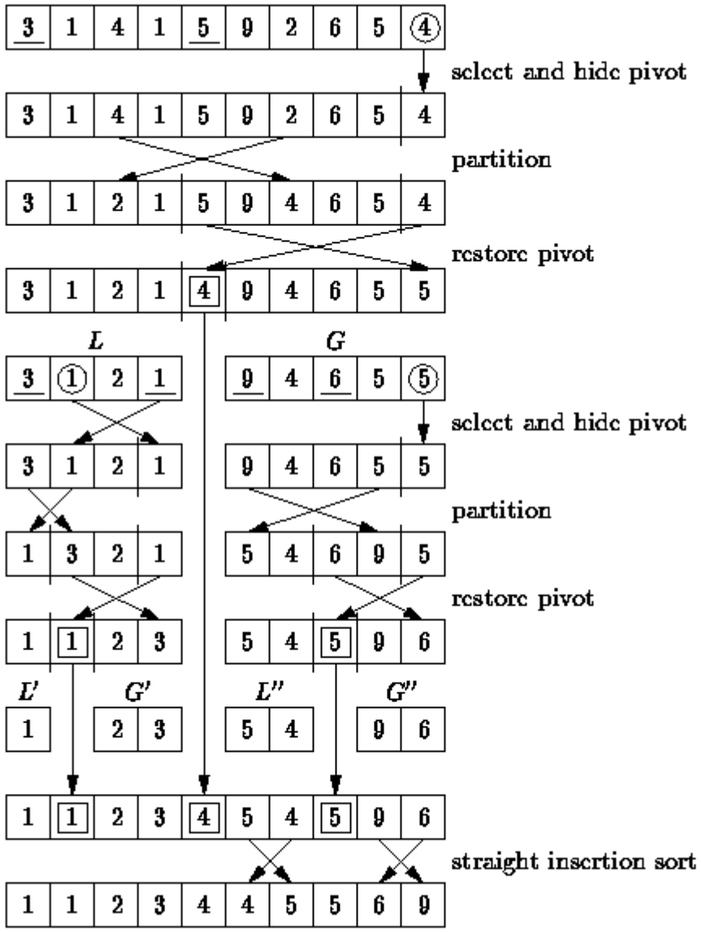
This scheme chooses a pivot which is typically the last element in the array. The algorithm maintains the index to put the pivot in variable i and each time it finds an element less than or equal to pivot, this index is incremented and that element would be placed before the pivot.

```
partition(A, low, high) is
pivot := A[high]
i := low
for j := low to high – 1 do
    if A[j] ≤ pivot then
        swap A[i] with A[j]
        i := i + 1
swap A[i] with A[high]
return i
```

Quick Sort mechanism :

```
quicksort(A, low, high) is
if low < high then
    p := partition(A, low, high)
    quicksort(A, low, p – 1)
    quicksort(A, p + 1, high)
```

Example of quick sort:



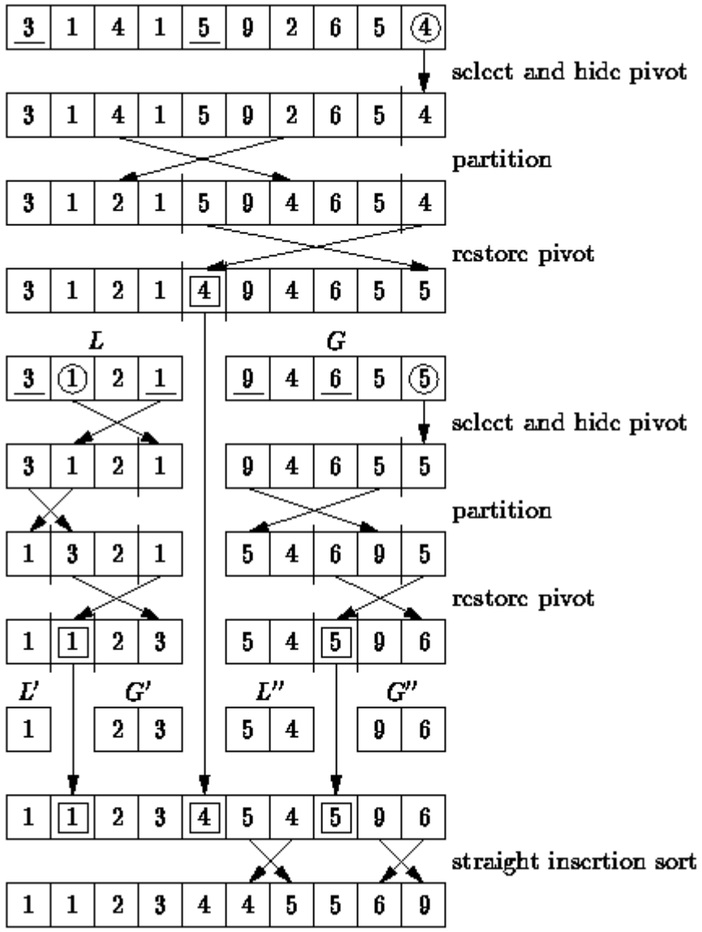
2. 霍尔分区方案：

它使用两个索引，分别从被分区数组的两端开始，然后向中间移动，直到它们检测到一个逆序：一对元素，其中一个大于或等于枢轴，另一个小于或等于枢轴，但它们的顺序相互颠倒。然后交换这对逆序元素。当索引相遇时，算法停止并返回最终索引。霍尔方案比洛穆托分区方案更高效，因为它平均减少了三倍的交换次数，并且即使所有值都相等，也能创建高效的分区。

```
快速排序(A, lo, hi) 是
如果 lo < hi 则
    p := 分区(A, lo, hi)
    快速排序(A, lo, p)
快速排序(A, p + 1, hi)
```

分区：

```
分区(A, lo, hi) 是
枢轴 := A[lo]
i := lo - 1
j := hi + 1
无限循环
    执行:
i := i + 1
```

2. Hoare partition scheme:

It uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater or equal than the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index. Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.

```
quicksort(A, lo, hi) is
if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Partition :

```
partition(A, lo, hi) is
pivot := A[lo]
i := lo - 1
j := hi + 1
loop forever
    do:
        i := i + 1
```

```
    while A[i] < pivot do

    do:
j := j - 1
    while A[j] > pivot do

    if i >= j then
        return j

swap A[i] with A[j]
```

第33.2节：Python中的快速排序

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
pivot = arr[len(arr) / 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])
```

打印 "[1, 1, 2, 3, 6, 8, 10]"

第33.3节：Lomuto划分法Java实现

```
public class Solution {

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] ar = new int[n];
    for(int i=0; i<n; i++)
      ar[i] = sc.nextInt();
     quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high)
 {
    if(low<high)
    {
        int p = partition(ar, low, high);
        quickSort(ar, 0 , p-1);
        quickSort(ar, p+1, high);
    }
 }
public static int partition(int[] ar, int l, int r)
 {
    int pivot = ar[r];
    int i =l;
    for(int j=l; j<r; j++)
     {
        if(ar[j] <= pivot)
         {
            int t = ar[j];
            ar[j] = ar[i];
            ar[i] = t;
            i++;
        }
```



```
    while A[i] < pivot do

    do:
        j := j - 1
    while A[j] > pivot do

    if i >= j then
        return j

    swap A[i] with A[j]
```

Section 33.2: Quicksort in Python

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) / 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])
```

Prints "[1, 1, 2, 3, 6, 8, 10]"

Section 33.3: Lomuto partition java implementation

```
public class Solution {

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] ar = new int[n];
    for(int i=0; i<n; i++)
      ar[i] = sc.nextInt();
     quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high)
 {
    if(low<high)
    {
        int p = partition(ar, low, high);
        quickSort(ar, 0 , p-1);
        quickSort(ar, p+1, high);
    }
 }
public static int partition(int[] ar, int l, int r)
 {
    int pivot = ar[r];
    int i =l;
    for(int j=l; j<r; j++)
     {
        if(ar[j] <= pivot)
         {
            int t = ar[j];
            ar[j] = ar[i];
            ar[i] = t;
            i++;
        }
```

```
    }
    int t = ar[i];
    ar[i] = ar[r];
    ar[r] = t;

    return i;
}
```



```
    }
    int t = ar[i];
    ar[i] = ar[r];
    ar[r] = t;

    return i;
}
```

第34章：计数排序

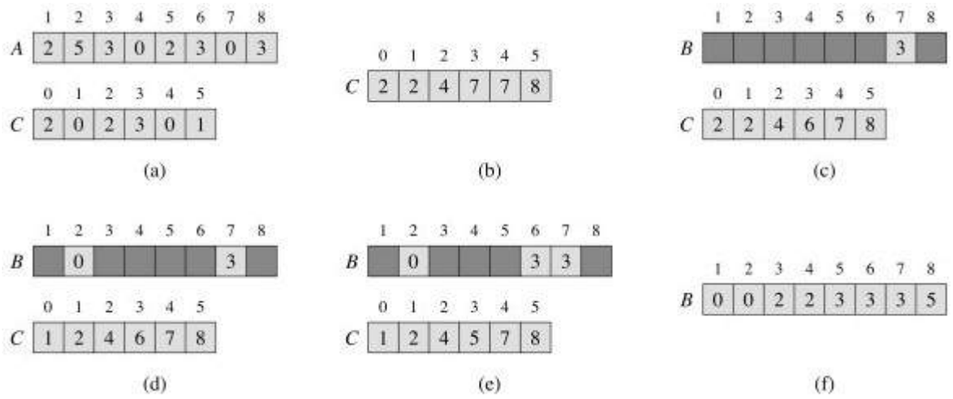
第34.1节：计数排序基本信息

计数排序是一种整数排序算法，用于对一组对象根据对象的键进行排序。

步骤

1. 构造一个工作数组C，其大小等于输入数组A的取值范围。
2. 遍历A，根据x在A中出现的次数为C[x]赋值。
3. 将C转换为一个数组，其中C[x]表示数组中小于等于x的值的数量，通过遍历数组实现，将每个C[x]赋值为其之前的值与C中所有先前值的总和。
4. 从后向前遍历A，将每个值放入新排序数组B中，位置由C中记录的索引决定。对于给定的A[x]，通过将B[C[A[x]]]赋值为A[x]来完成，并在原始未排序数组中存在重复值时，递减C[A[x]]。

计数排序示例



辅助空间：O(n+k)
时间复杂度：最坏情况：O(n+k)，最好情况：O(n)，平均情况：O(n+k)

第34.2节：伪代码实现

约束条件：

1. 输入（待排序数组）
2. 输入元素数量（n）
3. 键值范围为*0..k-1*（k）
4.计数（数字数组）

伪代码：

```
对于 x 在输入中：
count[key(x)] += 1
total = 0
对于 i 在范围内(k)：
oldCount = count[i]
    count[i] = total
    total += oldCount
```


Chapter 34: Counting Sort

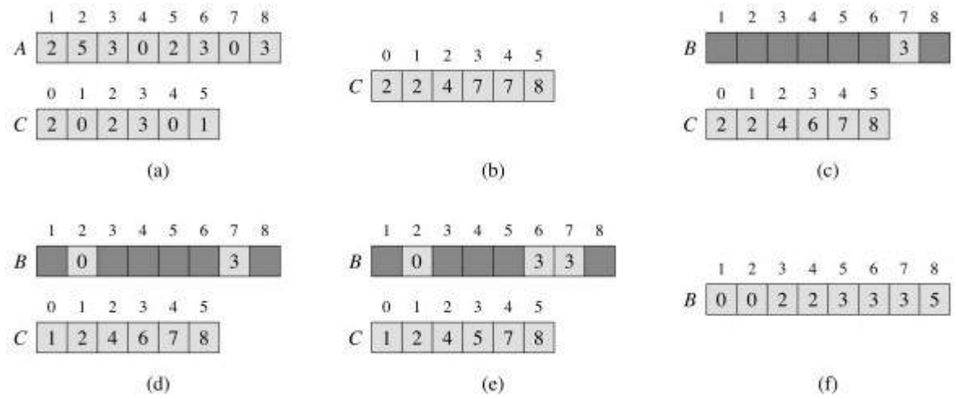
Section 34.1: Counting Sort Basic Information

Counting sort is an integer sorting algorithm for a collection of objects that sorts according to the keys of the objects.

Steps

1. Construct a working array *C* that has size equal to the range of the input array *A*.
2. Iterate through *A*, assigning *C*[x] based on the number of times x appeared in *A*.
3. Transform *C* into an array where *C*[x] refers to the number of values ≤ x by iterating through the array, assigning to each *C*[x] the sum of its prior value and all values in *C* that come before it.
4. Iterate backwards through *A*, placing each value in to a new sorted array *B* at the index recorded in *C*. This is done for a given *A*[x] by assigning *B*[*C*[*A*[x]]] to *A*[x], and decrementing *C*[*A*[x]] in case there were duplicate values in the original unsorted array.

Example of Counting Sort



Auxiliary Space: O(n+k)
Time Complexity: Worst-case: O(n+k), Best-case: O(n), Average-case O(n+k)

Section 34.2: Psuedocode Implementation

Constraints:

1. Input (an array to be sorted)
2. Number of element in input (n)
3. Keys in the range of *0..k-1* (k)
4. Count (an array of number)

Pseudocode:

```
for x in input:
    count[key(x)] += 1
total = 0
for i in range(k):
    oldCount = count[i]
    count[i] = total
    total += oldCount
```

```
对于 x 在输入中：
output[count[key(x)]] = x
    count[key(x)] += 1
返回 output
```

```
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1
return output
```

第35章：堆排序

第35.1节：C#实现

```
public class 堆排序
{
    public static void 堆化(int[] 输入, int n, int i)
    {
        int 最大值索引 = i;
        int 左子节点 = i + 1;
        int 右子节点 = i + 2;

        if (左子节点 < n && 输入[左子节点] > 输入[最大值索引])
        最大值索引 = 左子节点;

        if (右子节点 < n && 输入[右子节点] > 输入[最大值索引])
        最大值索引 = 右子节点;

        if (最大值索引 != i)
        {
var 临时变量 = 输入[i];
            输入[i] = 输入[最大值索引];
            输入[最大值索引] = 临时变量;
            堆化(输入, n, 最大值索引);
        }
    }

    public static void 堆排序(int[] 输入, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
SortHeap(input, input.Length);
        return input;
    }
}
```

第35.2节：堆排序基本信息

堆排序是一种基于二叉堆数据结构的比较排序技术。它类似于选择排序，首先找到最大元素并将其放在数据结构的末尾。然后对剩余的元素重复相同的过程。

堆排序的伪代码：

```
function heapsort(input, count)
```


Chapter 35: Heap Sort

Section 35.1: C# Implementation

```
public class HeapSort
{
    public static void Heapify(int[] input, int n, int i)
    {
        int largest = i;
        int l = i + 1;
        int r = i + 2;

        if (l < n && input[l] > input[largest])
            largest = l;

        if (r < n && input[r] > input[largest])
            largest = r;

        if (largest != i)
        {
            var temp = input[i];
            input[i] = input[largest];
            input[largest] = temp;
            Heapify(input, n, largest);
        }
    }

    public static void SortHeap(int[] input, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
            Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
            var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
        SortHeap(input, input.Length);
        return input;
    }
}
```

Section 35.2: Heap Sort Basic Information

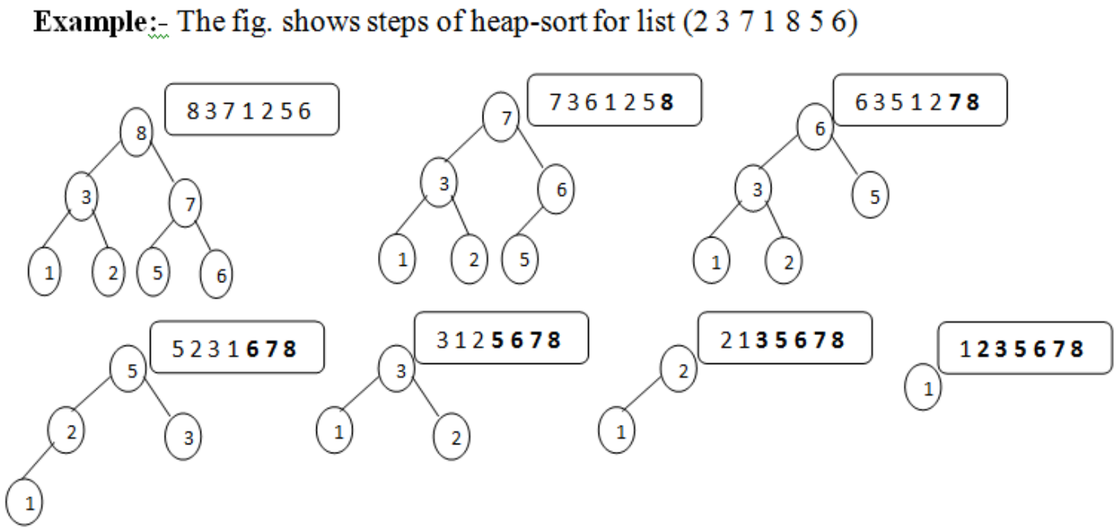
Heap sort is a comparison based sorting technique on binary heap data structure. It is similar to selection sort in which we first find the maximum element and put it at the end of the data structure. Then repeat the same process for the remaining items.

Pseudo code for Heap Sort:

```
function heapsort(input, count)
```

```
heapify(a,count)
    end <- count - 1
    while end -> 0 do
    swap(a[end],a[0])
    end<-end-1
restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

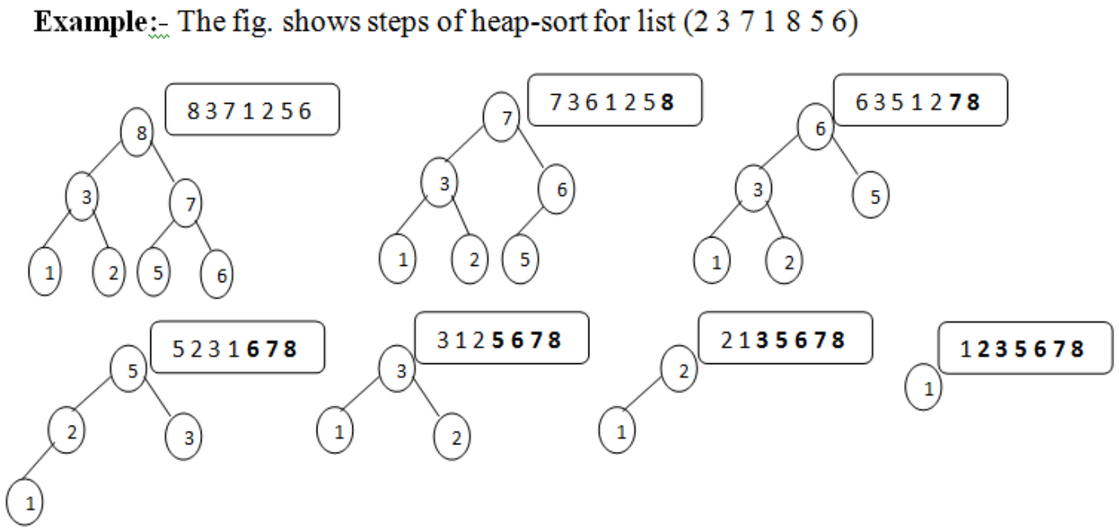
堆排序示例：



辅助空间：O(1)
时间复杂度：O(nlogn)


```
    heapify(a,count)
    end <- count - 1
    while end -> 0 do
    swap(a[end],a[0])
    end<-end-1
    restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

Example of Heap Sort:



Auxiliary Space: O(1)
Time Complexity: O(nlogn)

第36章：循环排序

第36.1节：伪代码实现

```
(input)
输出 = 0
for cycleStart 从 0 到 length(array) - 2
    item = array[cycleStart]
pos = cycleStart
    for i 从 cycleStart + 1 到 length(array) - 1
        if array[i] < item:
pos += 1
    if pos == cycleStart:
        continue
    while item == array[pos]:
pos += 1
array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
pos = cycleStart
        for i 从 cycleStart + 1 到 length(array) - 1
            if array[i] < item:
pos += 1
        while item == array[pos]:
pos += 1
array[pos], item = item, array[pos]
        writes += 1
return outout
```


Chapter 36: Cycle Sort

Section 36.1: Pseudocode Implementation

```
(input)
output = 0
for cycleStart from 0 to length(array) - 2
    item = array[cycleStart]
    pos = cycleStart
    for i from cycleStart + 1 to length(array) - 1
        if array[i] < item:
            pos += 1
    if pos == cycleStart:
        continue
    while item == array[pos]:
        pos += 1
    array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
        pos = cycleStart
        for i from cycleStart + 1 to length(array) - 1
            if array[i] < item:
                pos += 1
        while item == array[pos]:
            pos += 1
        array[pos], item = item, array[pos]
        writes += 1
return outout
```

第37章：奇偶排序

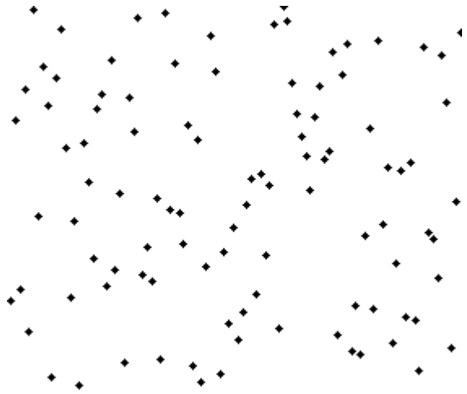
第37.1节：奇偶排序基本信息

奇偶排序（Odd-Even Sort）或称砖排序（brick sort）是一种简单的排序算法，专为具有局部互连的并行处理器设计。它通过比较列表中所有奇数/偶数索引的相邻元素对，如果一对元素顺序错误，则交换它们。下一步对偶数/奇数索引的元素对重复此操作。然后在奇数/偶数和偶数/奇数步骤之间交替进行，直到列表排序完成。

奇偶排序的伪代码：

```
if n>2 then
    1. 递归地对偶数子序列 a0, a2, ..., an-2 和奇数子序列 a1, a3, ..., an-1 应用奇偶归并（odd-even merge）（n/2）

    2. 对所有 i 元素 {1, 3, 5, 7, ..., n-3} 进行比较 [i : i+1]
否则
比较 [0 : 1]
```

维基百科有关于奇偶排序的最佳示例：



奇偶排序示例：

Chapter 37: Odd-Even Sort

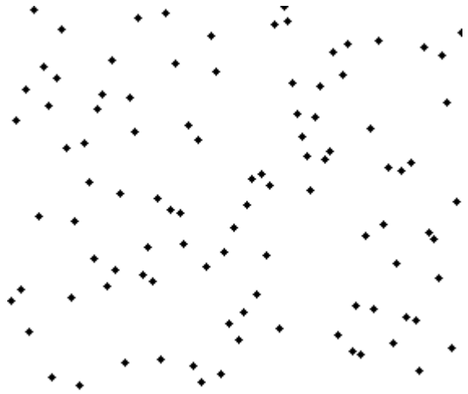
Section 37.1: Odd-Even Sort Basic Information

An Odd-Even Sort or brick sort is a simple sorting algorithm, which is developed for use on parallel processors with local interconnection. It works by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order the elements are switched. The next step repeats this for even/odd indexed pairs. Then it alternates between odd/even and even/odd steps until the list is sorted.

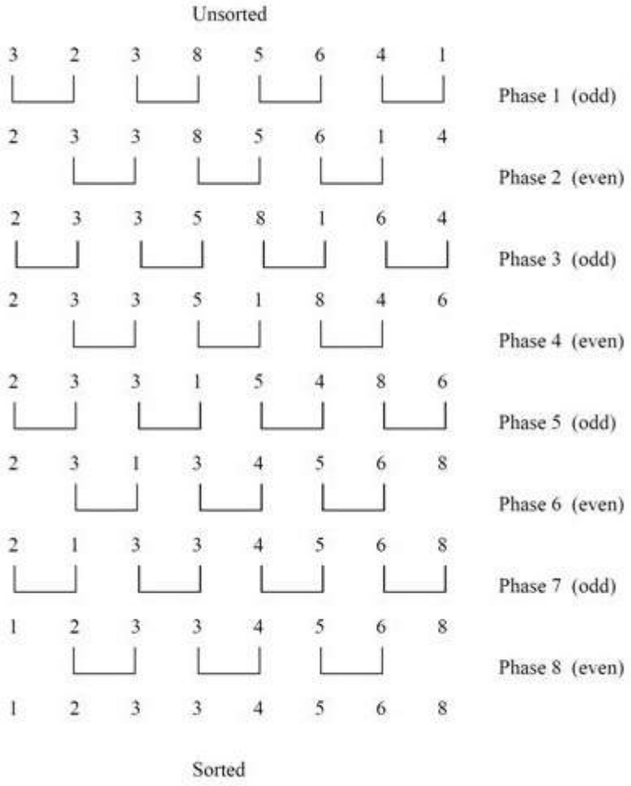
Pseudo code for Odd-Even Sort:

```
if n>2 then
    1. apply odd-even merge(n/2) recursively to the even subsequence a0, a2, ..., an-2 and to the
odd subsequence a1, a3, , ..., an-1
    2. comparison [i : i+1] for all i element {1, 3, 5, 7, ..., n-3}
else
    comparison [0 : 1]
```

Wikipedia has best illustration of Odd-Even sort:



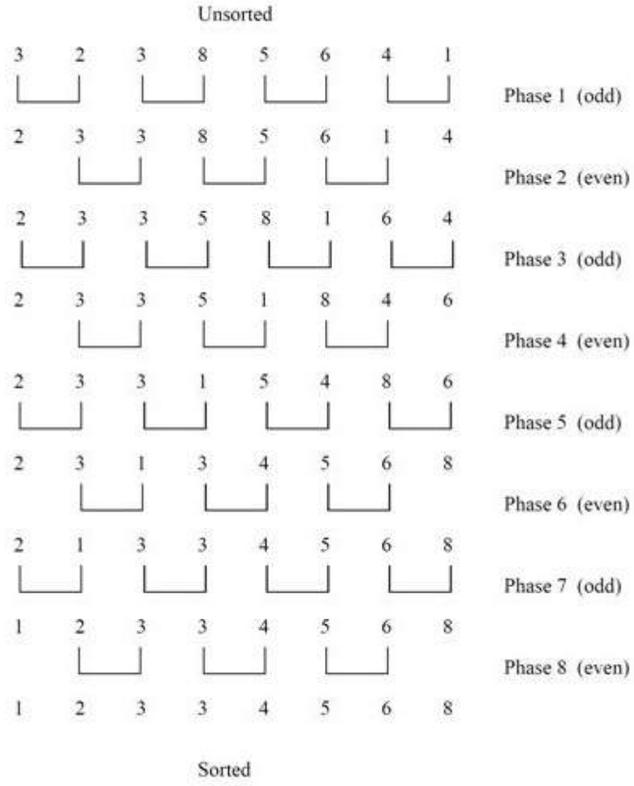
Example of Odd-Even Sort:



实现：

我使用C#语言实现了奇偶排序算法。

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
var sort = false;

        while (!sort)
        {
sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
            for (var i = 0; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
```

Implementation:

I used C# language to implement Odd-Even Sort Algorithm.

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
        var sort = false;

        while (!sort)
        {
            sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
            for (var i = 0; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
```

```
        }
    }

    public static int[] Main(int[] input)
    {
SortOddEven(input, input.Length);
        return input;
    }
}
```

辅助空间：O(n)
时间复杂度：O(n)


```
        }
    }

    public static int[] Main(int[] input)
    {
        SortOddEven(input, input.Length);
        return input;
    }
}
```

Auxiliary Space: O(n)
Time Complexity: O(n)

第38章：选择排序

第38.1节：Elixir实现

```
defmodule Selection do

  def sort(list) 当 is_list(list) do
    do_selection(list, [])
  end

def do_selection([head|[]], acc) do
    acc ++ [head]
  end

def do_selection(list, acc) do
    min = min(list)
do_selection(:lists.delete(min, list), acc ++ [min])
  end

defp min([first|[second|[]]]) do
    smaller(first, second)
  end

defp min([first|[second|tail]]) do
    min([smaller(first, second)|tail])
  end

defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    否则
      e2
    end
  end
end

Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

第38.2节：选择排序基础信息

选择排序是一种排序算法，具体来说是一种原地比较排序。它的时间复杂度为O(n2)，因此在大规模列表上效率较低，通常表现比类似的插入排序差。选择排序以其简单性著称，并且在某些情况下，尤其是辅助内存有限时，相较于更复杂的算法具有性能优势。

该算法将输入列表分为两部分：已排序的子列表，该子列表从左到右在列表的前端（左侧）逐步构建；以及剩余未排序的子列表，占据列表的其余部分。
最初，已排序的子列表为空，未排序的子列表是整个输入列表。算法通过在未排序子列表中找到最小（或最大，取决于排序顺序）元素，将其与最左侧的未排序元素交换（置于已排序位置），然后将子列表边界向右移动一个元素来进行。

选择排序的伪代码：

```
function select(list[1..n], k)
 for i 从 1 到 k
```


Chapter 38: Selection Sort

Section 38.1: Elixir Implementation

```
defmodule Selection do

  def sort(list) when is_list(list) do
    do_selection(list, [])
  end

  def do_selection([head|[]], acc) do
    acc ++ [head]
  end

  def do_selection(list, acc) do
    min = min(list)
    do_selection(:lists.delete(min, list), acc ++ [min])
  end

  defp min([first|[second|[]]]) do
    smaller(first, second)
  end

  defp min([first|[second|tail]]) do
    min([smaller(first, second)|tail])
  end

  defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    else
      e2
    end
  end
end

Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

Section 38.2: Selection Sort Basic Information

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

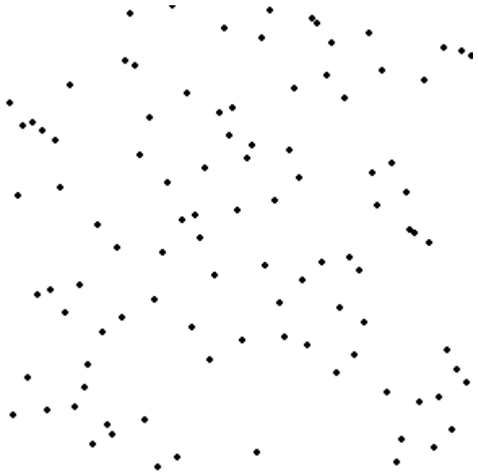
The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Pseudo code for Selection sort:

```
function select(list[1..n], k)
 for i from 1 to k
```

```
minIndex = i
      minValue = list[i]
      for j 从 i+1 到 n
           if list[j] < minValue
              minIndex = j
              minValue = list[j]
     交换 list[i] 和 list[minIndex]
 return list[k]
```

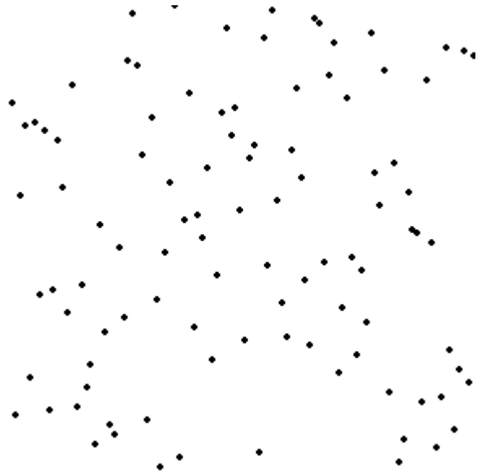
选择排序的可视化：



选择排序示例：


```
    minIndex = i
    minValue = list[i]
    for j from i+1 to n
        if list[j] < minValue
            minIndex = j
            minValue = list[j]
    swap list[i] and list[minIndex]
return list[k]
```

Visualization of selection sort:



Example of Selection sort:

辅助空间：O(n)
时间复杂度：O(n^2)

第38.3节：C#中选择排序的实现

我使用C#语言实现了选择排序算法。

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
var temp = input[minId];
```


Auxiliary Space: O(n)
Time Complexity: O(n^2)

Section 38.3: Implementation of Selection sort in C#

I used C# language to implement Selection sort algorithm.

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
            var temp = input[minId];
```

```
input[minId] = input[i];
            input[i] = temp;
        }
    }

    public static int[] Main(int[] input)
    {
SortSelection(input, input.Length);
        return input;
    }
}
```

```
            input[minId] = input[i];
            input[i] = temp;
        }
    }

    public static int[] Main(int[] input)
    {
        SortSelection(input, input.Length);
        return input;
    }
}
```

第39章：搜索

第39.1节：二分查找

介绍

二分查找是一种分治搜索算法。它使用O(log n)时间来查找元素在搜索空间中的位置，其中 n是搜索空间的大小。

二分查找通过在每次迭代中将搜索空间减半来工作，方法是将目标值与搜索空间的中间值进行比较。

要使用二分查找，搜索空间必须以某种方式有序（排序）。重复条目（根据比较函数比较为相等的条目）无法区分，尽管它们不违反二分查找的性质。

传统上，我们使用小于号（<）作为比较函数。如果 a < b，则返回真。如果 a 不小于 b 且 b 不小于 a，则 a 和 b 相等。

示例问题

你是一名经济学家，虽然水平相当差。你被分配到一个任务，寻找大米的均衡价格（即供给等于需求的价格）。

记住，价格越高，供给越大，需求越少。

由于你们公司在计算市场力量方面非常高效，当大米价格设定为某个价格 p 时，你可以立即获得大米的供给和需求单位数。

你的老板希望尽快得到均衡价格，但告诉你均衡价格是一个正整数，最大不超过 10^17，并且保证在该范围内有且仅有一个正整数解。所以赶快开始工作，别丢了饭碗！

你可以调用函数 getSupply(k) 和 getDemand(k)，这两个函数的功能正如题目所述。

示例说明

这里我们的搜索空间是从 1 到 10^17。因此线性搜索不可行。

但是，注意到随着 k 的增加，getSupply(k) 增加，而 getDemand(k) 减少。因此，对于任意 x > y，有 getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)。由此，该搜索空间是单调的，我们可以使用二分查找。

以下伪代码演示了二分查找的用法：

```
high = 100000000000000000      <- 搜索空间的上界
low = 1                        <- 搜索空间的下界
当高-低> 1
    mid = (high + low) / 2     <- 取中间值
    supply = getSupply(mid)
    demand = getDemand(mid)
     如果 供应 > 需求
        高=中<-解在搜索空间的下半部分
```


Chapter 39: Searching

Section 39.1: Binary Search

Introduction

Binary Search is a Divide and Conquer search algorithm. It uses `O(log n)` time to find the location of an element in a search space where n is the size of the search space.

Binary Search works by halving the search space at each iteration after comparing the target value to the middle value of the search space.

To use Binary Search, the search space must be ordered (sorted) in some way. Duplicate entries (ones that compare as equal according to the comparison function) cannot be distinguished, though they don't violate the Binary Search property.

Conventionally, we use less than (<) as the comparison function. If a < b, it will return true. if a is not less than b and b is not less than a, a and b are equal.

Example Question

You are an economist, a pretty bad one though. You are given the task of finding the equilibrium price (that is, the price where supply = demand) for rice.

Remember the higher a price is set, the larger the supply and the lesser the demand

As your company is very efficient at calculating market forces, you can instantly get the supply and demand in units of rice when the price of rice is set at a certain price p.

Your boss wants the equilibrium price ASAP, but tells you that the equilibrium price can be a positive integer that is at most `10^17` and there is guaranteed to be exactly 1 positive integer solution in the range. So get going with your job before you lose it!

You are allowed to call functions `getSupply(k)` and `getDemand(k)`, which will do exactly what is stated in the problem.

Example Explanation

Here our search space is from 1 to `10^17`. Thus a linear search is infeasible.

However, notice that as the k goes up, `getSupply(k)` increases and `getDemand(k)` decreases. Thus, for any `x > y`, `getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)`. Therefore, this search space is monotonic and we can use Binary Search.

The following psuedocode demonstrates the usage of Binary Search:

```
high = 100000000000000000      <- Upper bound of search space
low = 1                        <- Lower bound of search space
while high - low > 1
    mid = (high + low) / 2     <- Take the middle value
    supply = getSupply(mid)
    demand = getDemand(mid)
    if supply > demand
        high = mid              <- Solution is in lower half of search space
```

```
    否则如果 需求 > 供应
       低 = 中              <- 解决方案位于搜索空间的上半部分
    else                      <- 供给==需求条件
        return mid             <- 找到的解
```

该算法运行时间约为~O(log 10^17)。该算法可以推广到~O(log S)时间复杂度，其中S是搜索空间的大小，因为在每次while循环迭代中，我们将搜索空间减半（从[low:high]变为[low:mid]或[mid:high]）。

C语言递归实现二分查找

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
binsearch(a, x, low, mid - 1);
    } else {
binsearch(a, x, mid + 1, high);
    }
}
```

第39.2节：拉宾-卡普算法

拉宾-卡普算法或卡普-拉宾算法是一种字符串搜索算法，利用哈希在文本中查找一组模式字符串中的任意一个。其平均和最好情况下的运行时间为O(n+m)，空间复杂度为O(p)，但最坏情况下的时间复杂度为O(nm)，其中n是文本长度，m是模式长度。

字符串匹配的Java算法实现

```
void RabinfindPattern(String text,String pattern){
    /*
q 一个质数
p 模式的哈希值
t 文本的哈希值
d 是输入字母表中唯一字符的数量
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash 值计算函数
    for (i=0;i<m-1;i++)
h = (h*d)%q;
        for (i=0;i<m;i++){
p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//搜索模式
```



```
    else if demand > supply
        low = mid              <- Solution is in upper half of search space
    else                       <- supply==demand condition
        return mid             <- Found solution
```

This algorithm runs in ~O(log 10^17) time. This can be generalized to ~O(log S) time where S is the size of the search space since at every iteration of the **while** loop, we halved the search space (*from [low:high] to either [low:mid] or [mid:high]*).

C Implementation of Binary Search with Recursion

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Section 39.2: Rabin Karp

The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text.Its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm) where n is the length of the text and m is the length of the pattern.

Algorithm implementation in java for string matching

```
void RabinfindPattern(String text,String pattern){
    /*
    q a prime number
    p hash value for pattern
    t hash value for text
    d is the number of unique characters in input alphabet
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash value calculating function
    for (i=0;i<m-1;i++)
            h = (h*d)%q;
        for (i=0;i<m;i++){
        p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//search for the pattern
```

```
    for(i=0;i<end-m;i++){
        if(p==t){
//如果哈希值匹配，则逐字符匹配
                for(j=0;j<m;j++)
                    if(text.charAt(j+i)!=pattern.charAt(j))
                    break;
                if(j==m && i>=start)
                                    System.out.println("在索引处找到模式匹配 "+i);
        }
        if(i<end-m){
t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
                if(t<0)
t=t+q;
        }
    }
}
```

在计算哈希值时，我们通过一个质数进行除法以避免冲突。除以质数后，冲突的可能性会降低，但仍然存在两个字符串哈希值相同的情况，因此当匹配成功时，我们必须逐字符检查以确保确实匹配。

t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;

这是重新计算模式的哈希值，首先移除最左边的字符，然后添加文本中的新字符。

第39.3节：线性搜索的分析（最坏、平均和最佳情况）

我们可以有三种情况来分析一个算法：

1. 最坏情况
2. 平均情况
3. 最佳情况

```
#include <stdio.h>

// 在线性数组arr[]中搜索x。如果x存在则返回索引，

// 否则返回-1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/* 测试上述函数的驱动程序 */

```
int main()
```



```
    for(i=0;i<end-m;i++){
        if(p==t){
//if the hash value matches match them character by character
            for(j=0;j<m;j++)
                if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
        }
        if(i<end-m){
            t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
            if(t<0)
                t=t+q;
        }
    }
}
```

While calculating hash value we are dividing it by a prime number in order to avoid collision.After dividing by prime number the chances of collision will be less, but still ther is a chance that the hash value can be same for two strings,so when we get a match we have to check it character by character to make sure that we got a proper match.

t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;

This is to recalculate the hash value for pattern,first by removing the left most character and then adding the new character from the text.

Section 39.3: Analysis of Linear search (Worst, Average and Best Cases)

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/* Driver program to test above functions*/

```
int main()
```

```
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

最坏情况分析（通常进行）

在最坏情况分析中，我们计算算法运行时间的上界。我们必须知道导致执行最多操作的情况。对于线性搜索，最坏情况发生在要搜索的元素（上面代码中的 x）不在数组中时。当 x 不存在时，search() 函数会将其与 arr[] 中的所有元素逐一比较。因此，线性搜索的最坏情况时间复杂度为 Θ(n)

平均情况分析（有时进行）

在平均情况分析中，我们考虑所有可能的输入并计算所有输入的计算时间。将所有计算值相加后除以输入总数。我们必须知道（或预测）情况的分布。对于线性搜索问题，假设所有情况均匀分布（包括 x 不存在于数组中的情况）。因此，我们将所有情况相加后除以 (n+1)。以下是平均情况时间复杂度的值。

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)} = \frac{\theta((n+1)*(n+2)/2)}{(n+1)} = \Theta(n)$$

最好情况分析（无效）

在最佳情况分析中，我们计算算法运行时间的下界。我们必须知道导致执行操作次数最少的情况。在线性搜索问题中，最佳情况发生在x出现在第一个位置时。最佳情况下的操作次数是常数（不依赖于n）。因此，最佳情况的时间复杂度为 Θ(1)。大多数情况下，我们进行最坏情况分析来分析算法。在最坏情况分析中，我们保证算法运行时间的上界，这是一条有用的信息。平均情况分析在大多数实际情况下不容易进行，且很少进行。在平均情况分析中，我们必须知道（或预测）所有可能输入的数学分布。最佳情况分析是无效的。保证算法的下界并不提供任何信息，因为在最坏情况下，算法可能需要数年才能运行完毕。

对于某些算法，所有情况渐近相同，即不存在最坏和最佳情况。例如，归并排序。归并排序在所有情况下都执行Θ(nLogn)次操作。大多数其他排序算法都有最坏和最佳情况。例如，在快速排序的典型实现中（枢轴选为角元素），最坏情况发生在输入数组已经排序时，最佳情况发生在枢轴元素总是将数组分成两半时。对于插入排序，最坏情况发生在数组逆序排序时，最佳情况


```
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n)

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)} = \frac{\theta((n+1)*(n+2)/2)}{(n+1)} = \Theta(n)$$

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1) Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort. Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case

发生在数组与输出顺序相同的排序时。

第39.4节：二分查找：针对已排序数字

用伪代码展示数字的二分查找是最简单的

```
int array[1000] = { 已排序的数字列表 };
int N = 100;  // 搜索空间中的条目数；
int high, low, mid; // 临时变量
int x; // 要搜索的值

low = 0;
high = N -1;
while(low < high)
{
mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    否则
high = mid;
}
if(array[low] == x)
    // 找到，索引为 low
else
    // 未找到
```

不要尝试通过比较 array[mid] 是否等于 x 来提前返回。额外的比较只会使代码变慢。注意需要对 low 加一，以避免因整数除法总是向下取整而陷入死循环。

有趣的是，上述版本的二分查找允许你找到数组中 x 的最小出现位置。如果数组中包含 x 的重复项，可以稍作修改算法，使其返回 x 的最大出现位置，只需在 if 条件中添加：

```
while(low < high)
    {
mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        否则
high = mid;
    }
```

注意，与其使用 mid = (low + high) / 2，不如尝试使用 mid = low + ((high - low)/ 2)，这对于 Java 等实现来说，可以降低在处理非常大输入时发生溢出的风险。

第39.5节：线性搜索

线性搜索是一种简单的算法。它遍历所有项目直到找到查询项，这使得它成为一种线性算法——其复杂度为O(n)，其中n是需要遍历的项目数量。

为什么是O(n)？在最坏的情况下，你必须遍历所有n个项目。

它可以比作在一堆书中寻找一本书——你需要逐本翻阅，直到找到你想要的那本书。

下面是一个Python实现：

occurs when the array is sorted in the same order as output.

Section 39.4: Binary Search: On Sorted Numbers

It's easiest to show a binary search on numbers using pseudo-code

```
int array[1000] = { sorted list of numbers };
int N = 100;  // number of entries in search space;
int high, low, mid; // our temporaries
int x; // value to search for

low = 0;
high = N -1;
while(low < high)
{
    mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    else
        high = mid;
}
if(array[low] == x)
    // found, index is low
else
    // not found
```

Do not attempt to return early by comparing array[mid] to x for equality. The extra comparison can only slow the code down. Note you need to add one to low to avoid becoming trapped by integer division always rounding down.

Interestingly, the above version of binary search allows you to find the smallest occurrence of x in the array. If the array contains duplicates of x, the algorithm can be modified slightly in order for it to return the largest occurrence of x by simply adding to the if conditional:

```
while(low < high)
    {
        mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        else
            high = mid;
    }
```

Note that instead of doing `mid = (low + high) / 2`, it may also be a good idea to try `mid = low + ((high - low) / 2)` for implementations such as Java implementations to lower the risk of getting an overflow for really large inputs.

Section 39.5: Linear search

Linear search is a simple algorithm. It loops through items until the query has been found, which makes it a linear algorithm - the complexity is O(n), where n is the number of items to go through.

Why O(n)? In worst-case scenario, you have to go through all of the n items.

It can be compared to looking for a book in a stack of books - you go through them all until you find the one that you want.

Below is a Python implementation:

```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #返回 True
```



```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #returns True
```

第40章：子串搜索

第40.1节：Knuth-Morris-Pratt（KMP）算法简介

假设我们有一个文本和一个模式。我们需要确定模式是否存在于文本中。例如：

```
+-------+---+---+---+---+---+---+---+---+
| 索引  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+
|  文本  | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| 索引    | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| 模式    | b | c | g | l |
+---------+---+---+---+---+
```

这个模式确实存在于文本中。因此我们的子串搜索应该返回3，即该模式开始的位置索引。那么我们的暴力子串搜索过程是如何工作的呢？

我们通常的做法是：从文本的第0个索引和模式的第0个索引开始比较Text[0]与Pattern[0]。由于不匹配，我们移动到文本的下一个索引，比较Text[1]与Pattern[0]。由于匹配，我们同时增加模式和文本的索引。接着比较Text[2]与Pattern[1]。它们也匹配。按照之前的步骤，我们现在比较Text[3]与Pattern[2]。由于不匹配，我们从开始匹配的下一个位置重新开始，即文本的索引2。比较Text[2]与Pattern[0]。它们不匹配。然后增加文本的索引，比较Text[3]与Pattern[0]。它们匹配。接着Text[4]与Pattern[1]匹配，Text[5]与Pattern[2]匹配，Text[6]与Pattern[3]匹配。由于已经到达模式的末尾，我们返回匹配开始的索引，即3。如果我们的模式是：bcgll，意味着如果模式不存在于文本中，我们的搜索应该返回异常或-1或其他预定义值。我们可以清楚地看到，在最坏情况下，该算法的时间复杂度为O(mn)，其中m是文本的长度，n是模式的长度。我们如何降低这个时间复杂度呢？这就是KMP子串搜索算法的用武之地。

Knuth-Morris-Pratt字符串搜索算法（简称KMP算法）通过观察当发生不匹配时，字符串本身包含足够的信息来确定下一个匹配可能开始的位置，从而避免重新检查之前匹配的字符，来搜索主字符串中的“模式”出现位置。该算法由唐纳德·克努斯（Donald Knuth）、沃恩·普拉特（Vaughan Pratt）和詹姆斯·H·莫里斯（James H. Morris）于1970年独立提出，并于1977年共同发表。

让我们扩展一下示例文本和模式以便更好地理解：

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 索引  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  文本  |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
```


Chapter 40: Substring Search

Section 40.1: Introduction To Knuth-Morris-Pratt (KMP) Algorithm

Suppose that we have a *text* and a *pattern*. We need to determine if the pattern exists in the text or not. For example:

```
+-------+---+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+---+
|  Text | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| Index   | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| Pattern | b | c | g | l |
+---------+---+---+---+---+
```

This *pattern* does exist in the *text*. So our substring search should return **3**, the index of the position from which this *pattern* starts. So how does our brute force substring search procedure work?

What we usually do is: we start from the **0th** index of the *text* and the **0th** index of our *pattern and we compare **Text[0]** with **Pattern[0]**. Since they are not a match, we go to the next index of our *text* and we compare **Text[1]** with **Pattern[0]**. Since this is a match, we increment the index of our *pattern* and the index of the *Text* also. We compare **Text[2]** with **Pattern[1]**. They are also a match. Following the same procedure stated before, we now compare **Text[3]** with **Pattern[2]**. As they do not match, we start from the next position where we started finding the match. That is index **2** of the *Text*. We compare **Text[2]** with **Pattern[0]**. They don't match. Then incrementing index of the *Text*, we compare **Text[3]** with **Pattern[0]**. They match. Again **Text[4]** and **Pattern[1]** match, **Text[5]** and **Pattern[2]** match and **Text[6]** and **Pattern[3]** match. Since we've reached the end of our *Pattern*, we now return the index from which our match started, that is **3**. If our *pattern* was: `bcgll`, that means if the *pattern* didn't exist in our *text*, our search should return exception or **-1** or any other predefined value. We can clearly see that, in the worst case, this algorithm would take `O(mn)` time where **m** is the length of the *Text* and **n** is the length of the *Pattern*. How do we reduce this time complexity? This is where KMP Substring Search Algorithm comes into the picture.

The Knuth-Morris-Pratt String Searching Algorithm or KMP Algorithm searches for occurrences of a "Pattern" within a main "Text" by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The algorithm was conceived in 1970 by Donuld Knuth and Vaughan Pratt and independently by James H. Morris. The trio published it jointly in 1977.

Let's extend our example *Text* and *Pattern* for better understanding:

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Index |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Text |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
```

```
+---------+---+---+---+---+---+---+---+---+
| 模式    | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

起初，我们的文本和模式匹配到索引2。文本[3]和模式[3]不匹配。因此，我们的目标是不在该文本中向后退，也就是说，在不匹配的情况下，我们不希望匹配从最初开始匹配的位置重新开始。为实现这一点，我们会在不匹配发生之前，在我们的模式中寻找一个后缀（子串abc），该后缀也是模式子串的前缀。以我们的例子来说，由于所有字符都是唯一的，没有后缀，也就是匹配子串的前缀。这意味着我们的下一次比较将从索引0开始。稍等一下，你会明白我们为什么这样做。接下来，我们进行比较

文本[3] 与 模式[0] 不匹配。之后，对于 文本 从索引 4 到索引 9 以及对于 模式 从索引 **0** 到索引5，我们找到匹配。在Text[10]和Pattern[6]处发现不匹配。因此我们取Pattern就在发生不匹配的点之前（子串abcdabc），我们检查该子串的一个后缀，该后缀同时也是该子串的前缀。我们可以看到这里ab既是该子串的后缀也是前缀。这意味着，因为我们已经匹配到Text[10]，错配前的字符是ab。我们可以推断，由于ab也是我们取的子串的前缀，我们不必再次检查ab，下一次检查可以从Text[10]和Pattern[2]开始。我们不需要回头查看整个Text，可以直接从发生错配的位置开始。现在我们检查Text[10]和Pattern[2]，由于不匹配，且错配前的子串（abc）不包含既是后缀又是前缀的部分，我们检查Text[10]和Pattern[0]，它们不匹配。之后，对于Text从索引11到索引17，和Pattern从索引0到索引6，我们在Text[18]和Pattern[7]发现了错配。于是我们再次检查错配前的子串（子串abcdabc），发现abc既是后缀又是前缀。由于我们匹配到了Pattern[7]，abc必定出现在Text[18]之前。这意味着，我们不需要比较到Text[17]，比较将从Text[18]和Pattern[3]开始。这样我们会找到匹配，并返回15作为匹配的起始索引。这就是我们利用后缀和前缀信息进行KMP子串搜索的工作原理。

现在，我们如何高效地计算后缀是否与前缀相同，以及在Text和Pattern之间出现字符不匹配时从何处开始检查。让我们看一个例子：

```
+---------+---+---+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| 模式    | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

我们将生成一个包含所需信息的数组。我们称该数组为S。数组的大小与模式的长度相同。由于Pattern的第一个字母不可能是任何前缀的后缀，我们将S[0]设为0。我们首先取 i = 1和 j = 0。每一步我们比较Pattern[i]和Pattern[j]并递增 i。如果匹配，我们将S[i]设为j + 1并递增 j；如果不匹配，我们检查 j的前一个值位置（如果有），并将 j设为S[j-1]（如果 j不等于0），我们持续这样做直到S[j]与S[i]不匹配或 j变为0。对于后一种情况，我们将S[i]设为0。以我们的例子为例：

```
           j   i
+---------+---+---+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| 模式    | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Pattern[j]和Pattern[i]不匹配，所以我们递增 i，由于 j为0，我们不检查前一个值，直接将Pattern[i]设为0。如果我们继续递增 i，当 i = 4时，会匹配，所以我们将S[i]设为S[4] = j + 1 = 0 + 1 = 1并


```
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

At first, our *Text* and *Pattern* matches till index **2**. **Text[3]** and **Pattern[3]** doesn't match. So our aim is to not go backwards in this *Text*, that is, in case of a mismatch, we don't want our matching to begin again from the position that we started matching with. To achieve that, we'll look for a **suffix** in our *Pattern* right before our mismatch occurred (substring **abc**), which is also a **prefix** of the substring of our *Pattern*. For our example, since all the characters are unique, there is no suffix, that is the prefix of our matched substring. So what that means is, our next comparison will start from index **0**. Hold on for a bit, you'll understand why we did this. Next, we compare **Text[3]** with **Pattern[0]** and it doesn't match. After that, for *Text* from index **4** to index **9** and for *Pattern* from index **0** to index **5**, we find a match. We find a mismatch in **Text[10]** and **Pattern[6]**. So we take the substring from *Pattern* right before the point where mismatch occurs (substring **abcdabc**), we check for a suffix, that is also a prefix of this substring. We can see here **ab** is both the suffix and prefix of this substring. What that means is, since we've matched until **Text[10]**, the characters right before the mismatch is **ab**. What we can infer from it is that since **ab** is also a prefix of the substring we took, we don't have to check **ab** again and the next check can start from **Text[10]** and **Pattern[2]**. We didn't have to look back to the whole *Text*, we can start directly from where our mismatch occurred. Now we check **Text[10]** and **Pattern[2]**, since it's a mismatch, and the substring before mismatch (**abc**) doesn't contain a suffix which is also a prefix, we check **Text[10]** and **Pattern[0]**, they don't match. After that for *Text* from index **11** to index **17** and for *Pattern* from index **0** to index **6**. We find a mismatch in **Text[18]** and **Pattern[7]**. So again we check the substring before mismatch (substring **abcdabc**) and find **abc** is both the suffix and the prefix. So since we matched till **Pattern[7]**, **abc** must be before **Text[18]**. That means, we don't need to compare until **Text[17]** and our comparison will start from **Text[18]** and **Pattern[3]**. Thus we will find a match and we'll return **15** which is our starting index of the match. This is how our KMP Substring Search works using suffix and prefix information.

Now, how do we efficiently compute if suffix is same as prefix and at what point to start the check if there is a mismatch of character between *Text* and *Pattern*. Let's take a look at an example:

```
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

We'll generate an array containing the required information. Let's call the array **S**. The size of the array will be same as the length of the pattern. Since the first letter of the *Pattern* can't be the suffix of any prefix, we'll put **S[0]** = **0**. We take **i** = **1** and **j** = **0** at first. At each step we compare **Pattern[i]** and **Pattern[j]** and increment **i**. If there is a match we put **S[i]** = **j** + **1** and increment **j**, if there is a mismatch, we check the previous value position of **j** (if available) and set **j** = **S[j-1]** (if **j** is not equal to **0**), we keep doing this until **S[j]** doesn't match with **S[i]** or **j** doesn't become **0**. For the later one, we put **S[i]** = **0**. For our example:

```
           j   i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Pattern[j] and **Pattern[i]** don't match, so we increment **i** and since **j** is **0**, we don't check the previous value and put **Pattern[i]** = **0**. If we keep incrementing **i**, for **i** = **4**, we'll get a match, so we put **S[i]** = **S[4]** = **j** + **1** = **0** + **1** = **1** and

递增 j和 i。我们的数组将如下所示：

```
j             i
+---------+---+---+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| 模式    | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

由于Pattern[1]和Pattern[5]匹配，我们将S[i]设为S[5] = j + 1 = 1 + 1 = 2。如果继续，我们会发现当 j = 3且 i = 7时不匹配。由于 j不等于0，我们将 j设为S[j-1]。然后比较 i和 j处的字符是否相同，由于相同，我们将S[i]设为 j + 1。最终完成的数组如下：

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

这是我们需要的数组。这里 S[i] 的非零值表示在该子串（从 0 到 i）中存在一个长度为 S[i] 的后缀与前缀相同，下一次比较将从模式串（Pattern）的 S[i] + 1 位置开始。我们的生成该数组的算法如下：

```
过程 生成后缀数组(Pattern)：
i := 1
j := 0
n := Pattern.length
当 i 小于 n
    如果 Pattern[i] 等于 Pattern[j]
       S[i] := j + 1
       j := j + 1
i := i + 1
    否则
         如果 j 不等于 0
           j := S[j-1]
         否则
S[i] := 0
            i := i + 1
       结束 如果
结束 如果
结束 当
```

构建该数组的时间复杂度为 O(n)，空间复杂度也为 O(n)。为了确保你完全理解该算法，尝试为模式串 aabaabaa 生成一个数组，并检查结果是否与此数组匹配。

现在让我们使用以下示例进行子字符串搜索：

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   文本  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  索引   | 0 | 1 | 2 | 3 | 4 | 5 |
```


increment **j** and **i**. Our array will look like:

```
              j               i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

Since **Pattern[1]** and **Pattern[5]** is a match, we put **S[i]** = **S[5]** = **j** + **1** = **1** + **1** = **2**. If we continue, we'll find a mismatch for **j** = **3** and **i** = **7**. Since **j** is not equal to **0**, we put **j** = **S[j-1]**. And we'll compare the characters at **i** and **j** are same or not, since they are same, we'll put **S[i]** = **j** + 1. Our completed array will look like:

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

This is our required array. Here a nonzero-value of **S[i]** means there is a **S[i]** length suffix same as the prefix in that substring (substring from **0** to **i**) and the next comparison will start from **S[i]** + **1** position of the *Pattern*. Our algorithm to generate the array would look like:

```
Procedure GenerateSuffixArray(Pattern):
i := 1
j := 0
n := Pattern.length
while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i] := j + 1
        j := j + 1
        i := i + 1
    else
        if j is not equal to 0
            j := S[j-1]
        else
            S[i] := 0
            i := i + 1
        end if
    end if
end while
```

The time complexity to build this array is `O(n)` and the space complexity is also `O(n)`. To make sure if you have completely understood the algorithm, try to generate an array for pattern `aabaabaa` and check if the result matches with this one.

Now let's do a substring search using the following example:

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   Text  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 |
```

```
+---------+---+---+---+---+---+---+
| 模式 | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

我们有一个文本，一个模式和一个预先计算好的数组S，使用之前定义的逻辑。我们比较Text[0]和Pattern[0]，它们相同。 Text[1]和Pattern[1]也相同。 Text[2]和Pattern[2]不相同。我们检查不匹配位置之前的位置的值。由于S[1]是0，说明我们的子串中没有与前缀相同的后缀，比较从位置S[1]开始，即0。 所以Pattern[0]不等于Text[2]，我们继续。 Text[3]等于Pattern[0]，且匹配一直持续到Text[8]和Pattern[5]。我们在S数组中回溯一步，找到2。这意味着长度为2的前缀也是该子串（abcab）的后缀，即ab。 这也意味着在Text[8]之前有一个ab。所以我们可以安全地忽略Pattern[0]和Pattern[1]， 从Pattern[2]和Text[8]开始下一次比较。如果继续，我们会在Text中找到Pattern。我们的过程将如下：

```
过程 KMP(文本, 模式)
生成后缀数组(模式)
m := 文本.长度
n := 模式.长度
i := 0
j := 0
当 i 小于 m
    如果 模式[j] 等于 文本[i]
       j := j + 1
i := i + 1
    如果 j 等于 n
        返回 (j-i)
    否则如果 i < m 且 模式[j] 不等于 文本[i]
        如果 j 不等于 0
j = S[j-1]
        否则
i := i + 1
        end if
end if
end while
Return -1
```

除了后缀数组计算外，该算法的时间复杂度为 `O(m)`。由于 *GenerateSuffixArray* 需要 `O(n)`，KMP算法的总时间复杂度为：`O(m+n)`。

附注：如果你想在 Text 中找到 Pattern 的多个出现位置，不要返回值，而是打印或存储它，并将 j := S[j-1]。还要保持一个 flag 来跟踪是否找到任何出现，并相应地处理。

第40.2节：Rabin-Karp算法简介

Rabin-Karp算法 是由 理查德·M·卡普（Richard M. Karp） 和 迈克尔·O·拉宾（Michael O. Rabin） 创建的字符串搜索算法，利用哈希来查找文本中一组模式字符串中的任意一个。

字符串的子串是另一个出现在该字符串中的字符串。例如，ver 是 stackoverflow 的子串。不要与子序列混淆，因为 cover 是同一字符串的子序列。换句话说，字符串中任意连续字母的子集都是该字符串的子串。

在Rabin-Karp算法中，我们将生成要查找的 pattern 的哈希值，并检查 text 的滚动哈希是否与 pattern 匹配。如果不匹配，我们可以确定 pattern 不存在于 text 中。


```
+---------+---+---+---+---+---+---+
| Pattern | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

We have a *Text*, a *Pattern* and a pre-calculated array *S* using our logic defined before. We compare **Text[0]** and **Pattern[0]** and they are same. **Text[1]** and **Pattern[1]** are same. **Text[2]** and **Pattern[2]** are not same. We check the value at the position right before the mismatch. Since **S[1]** is **0**, there is no suffix that is same as the prefix in our substring and our comparison starts at position **S[1]**, which is **0**. So **Pattern[0]** is not same as **Text[2]**, so we move on. **Text[3]** is same as **Pattern[0]** and there is a match till **Text[8]** and **Pattern[5]**. We go one step back in the **S** array and find **2**. So this means there is a prefix of length **2** which is also the suffix of this substring (**abcab)** which is **ab**. That also means that there is an **ab** before **Text[8]**. So we can safely ignore **Pattern[0]** and **Pattern[1]** and start our next comparison from **Pattern[2]** and **Text[8]**. If we continue, we'll find the *Pattern* in the *Text*. Our procedure will look like:

```
Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)
m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
    if j is equal to n
        Return (j-i)
    else if i < m and Pattern[j] is not equal t Text[i]
        if j is not equal to 0
            j = S[j-1]
        else
            i := i + 1
        end if
    end if
end while
Return -1
```

The time complexity of this algorithm apart from the Suffix Array Calculation is `O(m)`. Since *GenerateSuffixArray* takes `O(n)`, the total time complexity of KMP Algorithm is: `O(m+n)`.

PS: If you want to find multiple occurrences of *Pattern* in the *Text*, instead of returning the value, print it/store it and set `j := S[j-1]`. Also keep a `flag` to track whether you have found any occurrence or not and handle it accordingly.

Section 40.2: Introduction to Rabin-Karp Algorithm

Rabin-Karp Algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any one of a set of pattern strings in a text.

A substring of a string is another string that occurs in. For example, *ver* is a substring of *stackoverflow*. Not to be confused with subsequence because *cover* is a subsequence of the same string. In other words, any subset of consecutive letters in a string is a substring of the given string.

In Rabin-Karp algorithm, we'll generate a hash of our *pattern* that we are looking for & check if the rolling hash of our *text* matches the *pattern* or not. If it doesn't match, we can guarantee that the *pattern* **doesn't exist** in the *text*.

但是，如果匹配成功，模式可以出现在文本中。让我们来看一个例子：

假设我们有一段文本：yeminsajid，我们想要判断模式 nsa是否存在于该文本中。为了计算哈希值和滚动哈希，我们需要使用一个质数。这个质数可以是任意一个质数。这里我们取 prime = 11作为示例。我们将使用以下公式来确定哈希值：

```
(第1个字母) X (质数) + (第2个字母) X (质数)¹ + (第3个字母) X (质数)² X + .....
```

我们定义：

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

nsa的哈希值将是：

```
14X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

现在我们计算文本的滚动哈希。如果滚动哈希与模式的哈希值匹配，我们将检查字符串是否相同。由于我们的模式有3个字母，我们取文本的前3个字母yem并计算哈希值。结果是：

```
25X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

该值与模式的哈希值不匹配，因此字符串不存在于此。接下来我们需要考虑下一步。为了计算下一个字符串emi的哈希值，我们可以使用公式计算，但这比较繁琐且耗费资源。相反，我们采用另一种技术。

- 我们从当前的哈希值中减去前一个字符串的首字母的值。在本例中是y。我们得到，1653 - 25 = 1628。
- 我们用我们的素数（本例中为11）除以差值。我们得到，1628 / 11 = 148。
- 我们将新字母 X (prime)$^{-1}$，其中m是模式的长度，加到商i = 9上。我们得到，148 + 9 X 11² = 1237。

新的哈希值不等于我们模式的哈希值。继续，对于 n我们得到：

```
前一个字符串： emi
前一个字符串的首字母： e(5)
新字母： n(14)
新字符串："min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

不匹配。之后，对于 s，我们得到：

```
前一个字符串： 最小值
前一个字符串的首字母： m(13)
新 字母： s(19)
新 字符串："ins"
1806 - 13 = 1793
1793 / 11 = 163
```


However, if it does match, the *pattern* **can** be present in the *text*. Let's look at an example:

Let's say we have a text: **yeminsajid** and we want to find out if the pattern **nsa** exists in the text. To calculate the hash and rolling hash, we'll need to use a prime number. This can be any prime number. Let's take **prime** = **11** for this example. We'll determine hash value using this formula:

```
(1st letter) X (prime) + (2nd letter) X (prime)¹ + (3rd letter) X (prime)² X + ......
```

We'll denote:

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

The hash value of **nsa** will be:

```
14 X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

Now we find the rolling-hash of our text. If the rolling hash matches with the hash value of our pattern, we'll check if the strings match or not. Since our pattern has **3** letters, we'll take 1st **3** letters **yem** from our text and calculate hash value. We get:

```
25 X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

This value doesn't match with our pattern's hash value. So the string doesn't exists here. Now we need to consider the next step. To calculate the hash value of our next string **emi**. We can calculate this using our formula. But that would be rather trivial and cost us more. Instead, we use another technique.

- We subtract the value of the **First Letter of Previous String** from our current hash value. In this case, **y**. We get, `1653 - 25 = 1628`.
- We divide the difference with our **prime**, which is **11** for this example. We get, `1628 / 11 = 148`.
- We add **new letter X (prime)□$^{-1}$**, where **m** is the length of the pattern, with the quotient, which is **i** = **9**. We get, `148 + 9 X 11² = 1237`.

The new hash value is not equal to our patterns hash value. Moving on, for **n** we get:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
New String: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

It doesn't match. After that, for **s**, we get:

```
Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
New String: "ins"
1806 - 13 = 1793
1793 / 11 = 163
```

```
163 + 19 X 11² = 2462
```

不匹配。接下来，对于 a，我们得到：

```
之前的 字符串： ins
之前字符串的第一个字母： i(9)
新 字母： a(1)
新 字符串："nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

匹配成功！现在我们将模式与当前字符串进行比较。由于两个字符串匹配，子串存在于该字符串中。
我们返回子串的起始位置。

伪代码如下：

哈希计算：

```
过程 计算-哈希(字符串, 素数, x)：
哈希 := 0                                    // 这里 x 表示要考虑的长度
对于 m 从 1 到 x                          // 用于计算哈希值
哈希 := 哈希 + (字符串[m] 的值)^-1
结束 循环
返回 哈希
```

哈希重新计算：

```
过程 重新计算-哈希(字符串, 当前, 素数, 哈希)：
哈希 := 哈希 - 字符串[当前] 的值   // 这里 当前 表示前一个字符串的首字母
哈希 := 哈希 / 素数
m := String.length
New := Curr + m - 1
Hash := Hash + (String[New] 的值)^-1
返回 Hash
```

字符串匹配：

```
过程 字符串-匹配(文本, 模式, m)：
对于 i 从 m 到 模式-长度 + m - 1
    如果 文本[i] 不等于 模式[i]
        返回 假
结束 如果
结束 循环
返回 真
```

拉宾-卡普算法：

```
过程 拉宾-卡普(文本, 模式, 素数)：
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i 从 1 到 Text.length - m
    if HashValue == CurrValue 且 String-Match(Text, Pattern, i) 为 true
        返回 i
结束 if
CurrValue := 重新计算-Hash(String, i+1, Prime, CurrValue)
结束 for
```



```
163 + 19 X 11² = 2462
```

It doesn't match. Next, for **a**, we get:

```
Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
New String: "nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

It's a match! Now we compare our pattern with the current string. Since both the strings match, the substring exists in this string. And we return the starting position of our substring.

The pseudo-code will be:

Hash Calculation:

```
Procedure Calculate-Hash(String, Prime, x):
hash := 0                                    // Here x denotes the length to be considered
for m from 1 to x                            // to find the hash value
    hash := hash + (Value of String[m])□^-1
end for
Return hash
```

Hash Recalculation:

```
Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]  //here Curr denotes First Letter of Previous String
Hash := Hash / Prime
m := String.length
New := Curr + m - 1
Hash := Hash + (Value of String[New])□^-1
Return Hash
```

String Match:

```
Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m - 1
    if Text[i] is not equal to Pattern[i]
        Return false
    end if
end for
Return true
```

Rabin-Karp:

```
Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length - m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for
```

```
返回 -1
```

如果算法未找到任何匹配项，则简单返回 -1。

该算法用于检测抄袭。给定源材料，算法可以快速搜索论文中来自源材料的句子实例，忽略大小写和标点等细节。由于所查找字符串数量众多，单字符串搜索算法在此不切实际。同样，Knuth-

Morris-Pratt算法或Boyer-Moore字符串搜索算法比Rabin-Karp更快的单模式字符串搜索算法。然而，它是多模式搜索的首选算法。如果我们想在文本中查找大量固定长度的模式，比如k个，我们可以创建Rabin-Karp算法的一个简单变体。

对于长度为 n 的文本和组合长度为 m 的 p 个模式，其平均和最佳运行时间为 O(n+m)，空间复杂度为 O(p)，但最坏情况下时间复杂度为 O(nm)。

第40.3节：KMP算法的Python实现

Haystack：需要在其中搜索给定模式的字符串。
Needle：需要搜索的模式。

时间复杂度：搜索部分（strstr方法）的复杂度为O(n)，其中 n 是haystack的长度，但由于needle也需要预处理以构建前缀表，构建前缀表需要O(m)，其中 m 是needle的长度。

因此，KMP的总体时间复杂度为 **O(n+m)**
空间复杂度： **O(m)**，因needle的前缀表所致。

注意：以下实现返回匹配在haystack中的起始位置（如果存在匹配），否则返回-1，针对边界情况如needle/haystack为空字符串或needle未在haystack中找到。

```
def get_prefix_table(needle):
prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
prefix_table[delimeter] = delimeter - j + 1
                break
j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
# m: 表示在 S 中潜在匹配 W 的起始位置
    # i: 表示当前考虑的 W 中字符的索引。
    haystack_len = len(haystack)
    needle_len = len(needle)
    如果(needle_len > haystack_len) 或者 (没有 haystack_len) 或者 (没有 needle_len):
        return -1
prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
i += 1
```



```
Return -1
```

If the algorithm doesn't find any match, it simply returns **-1**.

This algorithm is used in detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical here. Again, **Knuth-Morris-Pratt algorithm** or **Boyer-Moore String Search algorithm** is faster single pattern string searching algorithm, than **Rabin-Karp**. However, it is an algorithm of choice for multiple pattern search. If we want to find any of the large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin-Karp algorithm.

For text of length **n** and **p** patterns of combined length **m**, its average and best case running time is **O(n+m)** in space **O(p)**, but its worst-case time is **O(nm)**.

Section 40.3: Python Implementation of KMP algorithm

Haystack: The string in which given pattern needs to be searched.
Needle: The pattern to be searched.

Time complexity: Search portion (strstr method) has the complexity O(n) where `n` is the length of haystack but as needle is also pre parsed for building prefix table O(m) is required for building prefix table where `m` is the length of the needle.
Therefore, overall time complexity for KMP is **O(n+m)**
Space complexity: **O(m)** because of prefix table on needle.

Note: Following implementation returns the start position of match in haystack (if there is a match) else returns -1, for edge cases like if needle/haystack is an empty string or needle is not found in haystack.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
        prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
                prefix_table[delimeter] = delimeter - j + 1
                break
            j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
    # m: denoting the position within S where the prospective match for W begins
    # i: denoting the index of the currently considered character in W.
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
```

```
m += 1
            else:
                if i != 0:
i = prefix_table[i-1]
                else:
m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

第40.4节：C语言中的KMP算法

给定一个文本 txt和一个模式 pat，本程序的目标是打印出 pat在 txt中所有出现的位置。

示例：

输入：

```
txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"
```

输出：

```
模式在索引10处被找到
```

输入：

```
txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"
```

输出：

```
在索引0处找到模式
   在索引9处找到模式
   在索引13处找到模式
```

C语言实现：

```
// KMP模式搜索算法的C程序实现

#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // 创建lps[]，用于存储最长前缀后缀
```



```
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

Section 40.4: KMP Algorithm in C

Given a text *txt* and a pattern *pat*, the objective of this program will be to print all the occurance of *pat* in *txt*.

Examples:

Input:

```
  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"
```

output:

```
Pattern found at index 10
```

Input:

```
  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"
```

output:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

C Language Implementation:

```
// C program for implementation of KMP pattern searching
// algorithm
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
```

```
    // 模式的值
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // pat[] 的索引

    // 预处理模式（计算 lps[] 数组）
computeLPSArray(pat, M, lps);

    int i = 0;  // txt[] 的索引
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
    j++;
        i++;
      }

      if (j == M)
      {
printf("Found pattern at index %d ", i-j);          j = lps[j-1];

      }

      // j 匹配后不匹配
      else if (i < N && pat[j] != txt[i])
      {
        // 不要匹配 lps[0..lps[j-1]] 的字符，
        // 它们无论如何都会匹配上
        if (j != 0)
j = lps[j-1];
        否则
i = i+1;
      }
    }
free(lps); // 避免内存泄漏
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // 前一个最长前缀后缀的长度
    int i;

lps[0] = 0; // lps[0] 始终为 0
   i = 1;

    // 循环计算 i = 1 到 M-1 的 lps[i]
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // 这很棘手。考虑示例
           // AAACAAAA 和 i = 7。
len = lps[len-1];

           // 另外，注意这里我们不增加 i
```



```
    // values for pattern
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // length of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // This is tricky. Consider the example
           // AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
```

```
        }
        else // if (len == 0)
        {
lps[i] = 0;
          i++;
        }
      }
    }
```



```
        }
        else // if (len == 0)
        {
          lps[i] = 0;
          i++;
        }
      }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Output:

```
Found pattern at index 10
```

Reference:

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

第41章：广度优先搜索

第41.1节：从源点到其他节点寻找最短路径

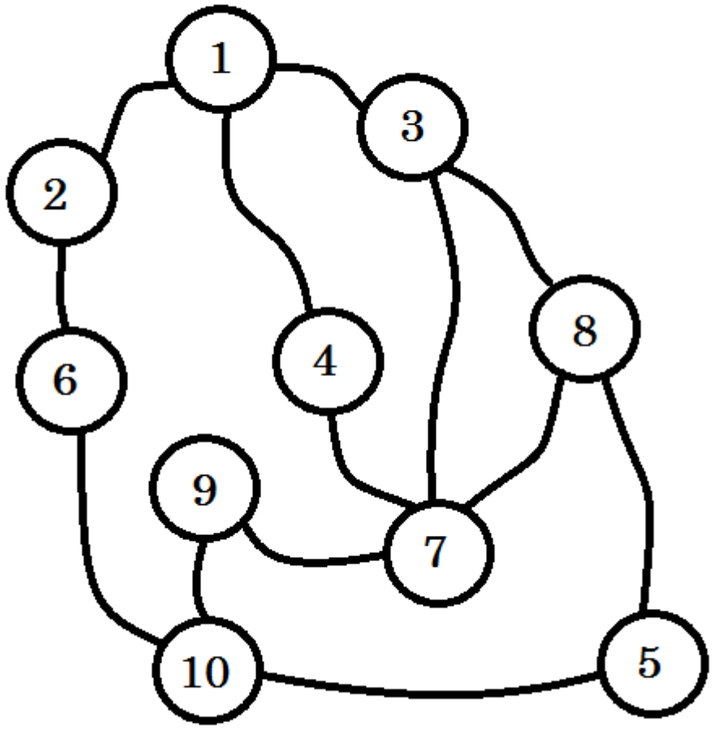
广度优先搜索（BFS）是一种遍历或搜索树或图数据结构的算法。它从树的根节点（或图中的某个任意节点，有时称为“搜索键”）开始，先探索邻近节点，然后再移动到下一层的邻居节点。BFS由爱德华·福雷斯特·摩尔于20世纪50年代末发明，他用它来寻找迷宫的最短路径；1961年，C. Y. 李独立发现了该算法，并将其用作布线算法。

BFS算法的过程基于以下假设：

1. 我们不会遍历任何节点超过一次。
2. 源节点或我们开始的节点位于第0层。
3. 我们可以直接从源节点到达的节点是第1层节点，可以直接从第1层节点到达的节点是第2层节点，依此类推。

4. 层数表示从源节点出发的最短路径距离。

让我们看一个例子：



假设这个图表示多个城市之间的连接，每个节点代表一个城市，两个节点之间的边表示它们之间有一条道路相连。我们想从节点1到节点10。因此，节点1是我们的源点，位于第0层。我们将节点1标记为已访问。从这里我们可以到达节点2、节点3和节点4，所以它们是第(0+1)层=第1层节点。现在我们将它们标记为已访问，并开始处理它们。

Chapter 41: Breadth-First Search

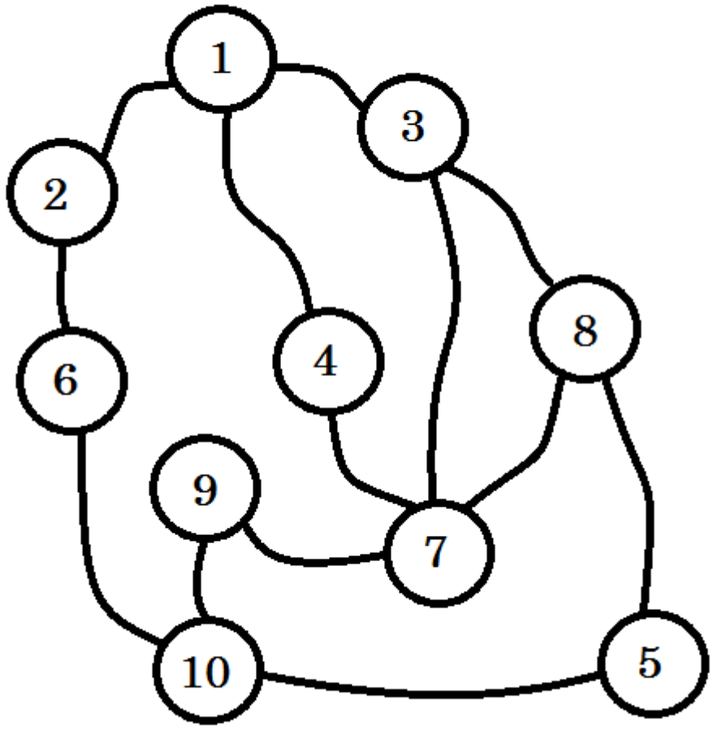
Section 41.1: Finding the Shortest Path from Source to other Nodes

Breadth-first-search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. BFS was invented in the late 1950s by Edward Forrest Moore, who used it to find the shortest path out of a maze and discovered independently by C. Y. Lee as a wire routing algorithm in 1961.

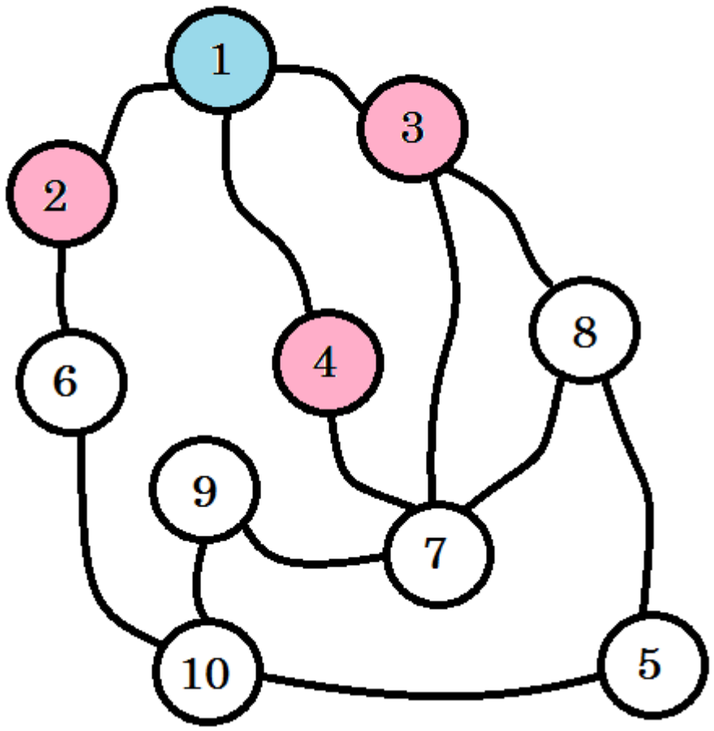
The processes of BFS algorithm works under these assumptions:

1. We won't traverse any node more than once.
2. Source node or the node that we're starting from is situated in level 0.
3. The nodes we can directly reach from source node are level 1 nodes, the nodes we can directly reach from level 1 nodes are level 2 nodes and so on.
4. The level denotes the distance of the shortest path from the source.

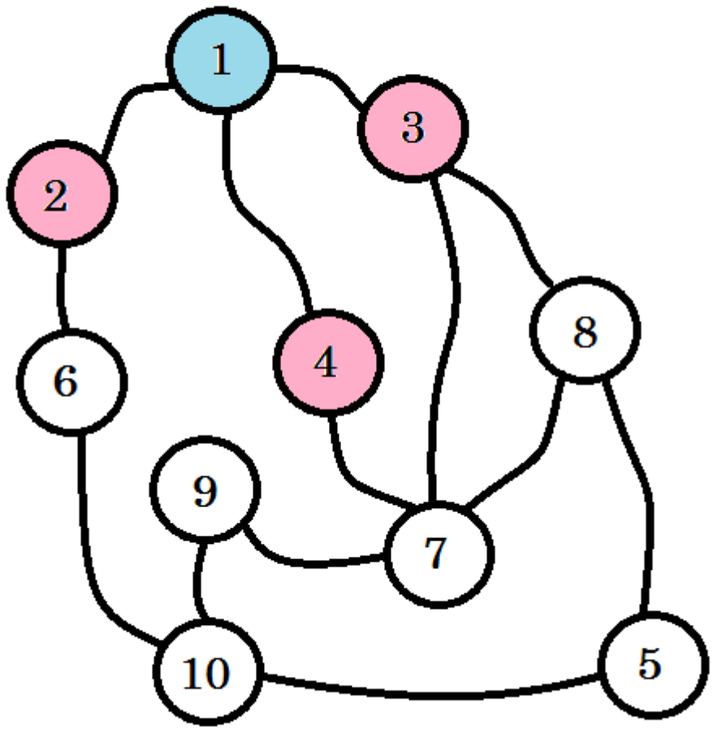
Let's see an example:



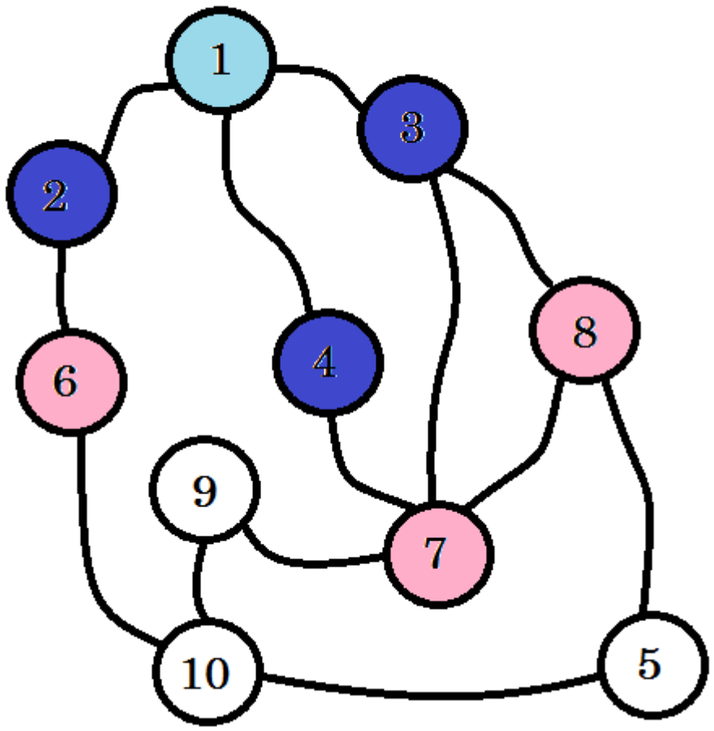
Let's assume this graph represents connection between multiple cities, where each node denotes a city and an edge between two nodes denote there is a road linking them. We want to go from **node 1** to **node 10**. So **node 1** is our **source**, which is **level 0**. We mark **node 1** as visited. We can go to **node 2**, **node 3** and **node 4** from here. So they'll be **level (0+1)** = **level 1** nodes. Now we'll mark them as visited and work with them.



彩色节点表示已访问的节点。我们当前正在处理的节点将标记为粉色。我们不会访问同一个节点两次。从节点2、节点3和节点4，我们可以到达节点6、节点7和节点8。让我们将它们标记为已访问。这些节点的层级将是层级(1+1)=层级2。

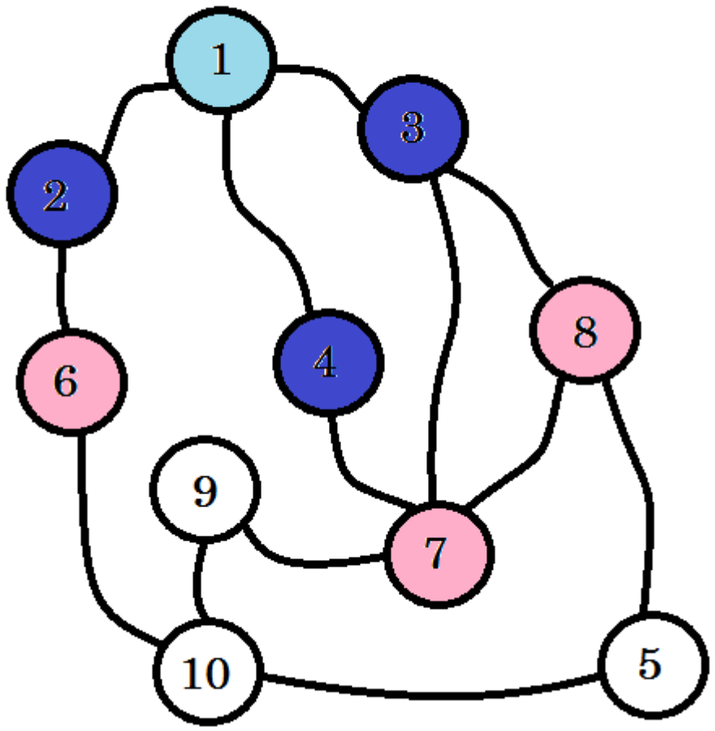


The colored nodes are visited. The nodes that we're currently working with will be marked with pink. We won't visit the same node twice. From **node 2**, **node 3** and **node 4**, we can go to **node 6, node 7** and **node 8**. Let's mark them as visited. The level of these nodes will be **level (1+1)** = **level 2**.



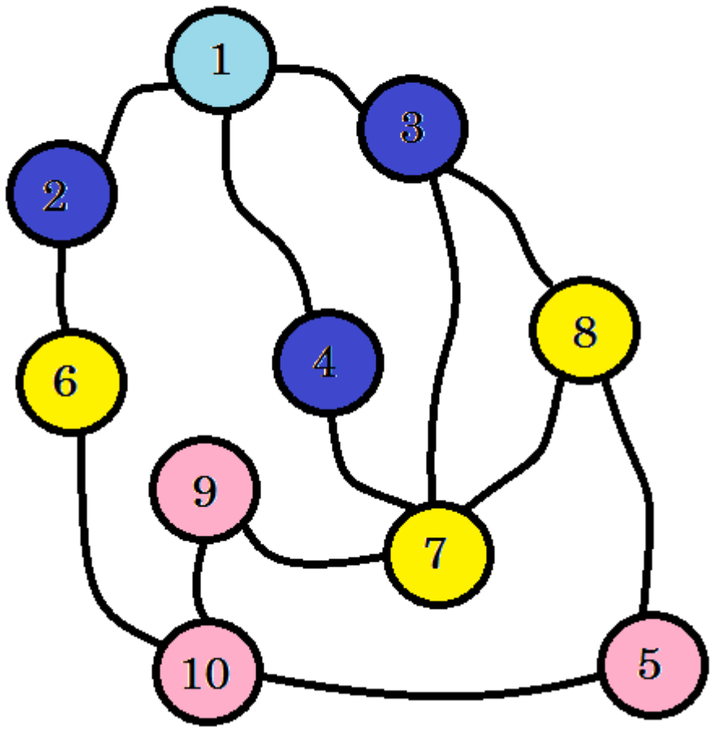
如果你还没有注意到，节点的层级仅表示从源点到该节点的最短路径距离。例如：
我们在层级2找到了节点8。所以从源点到节点8的距离是2。

我们还没有到达目标节点，即**节点10**。所以让我们访问下一个节点。我们可以直接从**节点6**、**节点7**和**节点8**前往。

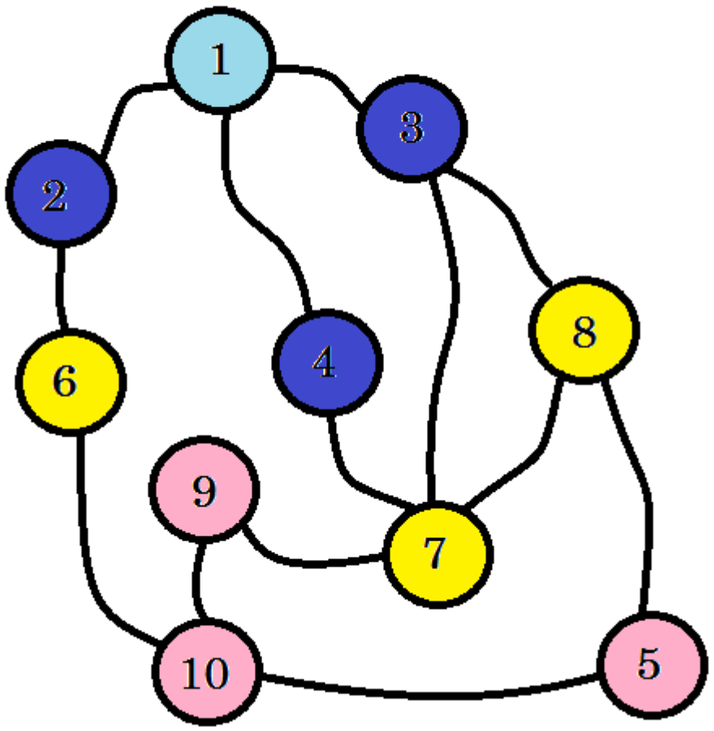


If you haven't noticed, the level of nodes simply denote the shortest path distance from the **source**. For example: we've found **node 8** on **level 2**. So the distance from **source** to **node 8** is **2**.

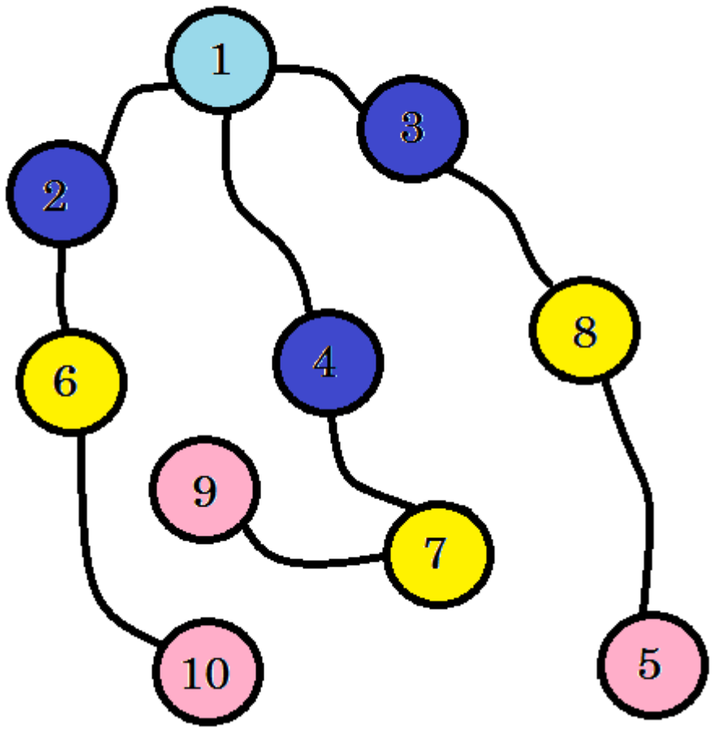
We didn't yet reach our target node, that is **node 10**. So let's visit the next nodes. we can directly go to from **node 6**, **node 7** and **node 8**.



我们可以看到，在层级3找到了节点10。所以从源点到节点10的最短路径是3。我们按层级搜索图，找到了最短路径。现在让我们删除未使用的边：



We can see that, we found **node 10** at **level 3**. So the shortest path from **source** to **node 10** is **3.** We searched the graph level by level and found the shortest path. Now let's erase the edges that we didn't use:



删除未使用的边后，我们得到一棵称为BFS树的树。这棵树显示了从源点到所有其他节点的最短路径。

所以我们的任务是，从源点到层级1的节点。然后从层级1到层级2的节点，依此类推，直到到达目的地。我们可以使用队列来存储将要处理的节点。也就是说，对于每个我们要处理的节点，我们会将所有可以直接遍历且尚未遍历的其他节点推入队列。

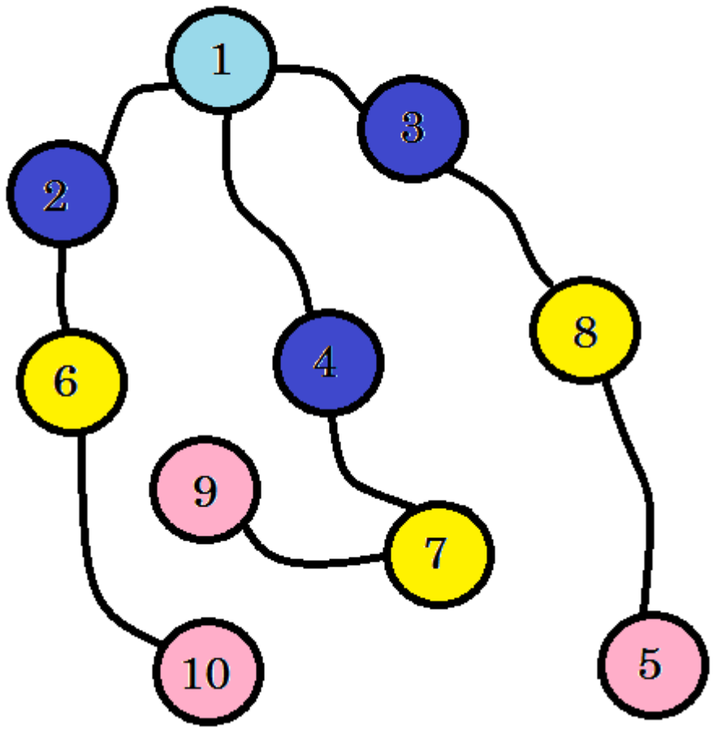
我们的示例模拟：

首先我们将源点加入队列。我们的队列将如下所示：

```
队首

|  1  |
+-----+
```

节点 1 的层级将是 0。level[1] = 0。现在我们开始广度优先搜索（BFS）。首先，我们从队列中弹出一个节点。得到节点 1。我们可以从该节点访问节点 4、节点 3 和节点 2。我们是从节点 1 到达这些节点的。因此，level[4] = level[3] = level[2] = level[1] + 1 = 1。现在我们将它们标记为已访问并加入队列。

```
队首

|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

After removing the edges that we didn't use, we get a tree called BFS tree. This tree shows the shortest path from **source** to all other nodes.

So our task will be, to go from **source** to **level 1** nodes. Then from **level 1** to **level 2** nodes and so on until we reach our destination. We can use *queue* to store the nodes that we are going to process. That is, for each node we're going to work with, we'll push all other nodes that can be directly traversed and not yet traversed in the queue.

The simulation of our example:

First we push the source in the queue. Our queue will look like:

```
 front
+-----+
|  1  |
+-----+
```

The level of **node 1** will be 0. **level[1]** = **0**. Now we start our BFS. At first, we pop a node from our queue. We get **node 1**. We can go to **node 4**, **node 3** and **node 2** from this one. We've reached these nodes from **node 1**. So **level[4]** = **level[3]** = **level[2]** = **level[1]** + **1** = **1**. Now we mark them as visited and push them in the queue.

```
                  front
+-----+  +-----+  +-----+
|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

现在我们弹出节点 4 并处理它。我们可以从节点 4 访问节点 7。level[7] = level[4] + 1 = 2。我们将节点 7 标记为已访问并加入队列。

```
队首

|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

从节点 3，我们可以访问节点 7 和节点 8。由于节点 7 已经被标记为已访问，我们将节点 8 标记为已访问，更新 level[8] = level[3] + 1 = 2。我们将节点 8 加入队列。

```
队首

|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

该过程将持续，直到我们到达目标节点或队列为空。level 数组将为我们提供从源点到各节点的最短路径距离。我们可以用无穷大（infinity）值初始化 level 数组，以标记节点尚未被访问。我们的伪代码如下：

```
BFS过程(图，源点):
Q = 队列();
level[] = 无穷大
level[源点] := 0
Q.入队(源点)
当 Q 不为空时
u -> Q.出队()
    对于 从u到v的所有边，在邻接表中
        如果 level[v] == 无穷大
level[v] := level[u] + 1
Q.入队(v)
        结束 如果
结束 循环
结束 循环
返回 level
```

通过遍历level数组，我们可以得知每个节点到源点的距离。例如：节点10到源点的距离将存储在level[10]中。

有时我们不仅需要打印最短距离，还需要打印从源点到目标节点的路径。为此，我们需要维护一个parent数组。对于parent[源点]，其值为NULL。每次更新level数组时，我们只需在伪代码的for循环中添加parent[v] := u。完成BFS后，若要找到路径，我们将沿着parent数组回溯，直到达到值为NULL的源点。

伪代码如下：

```
过程 PrintPath(u):  //递归    |  过程 PrintPath(u):  //迭代
如果 parent[u] 不等于 null      |    S = Stack()
PrintPath(parent[u])           |  while parent[u] 不等于 null
end if                         |      S.push(u)
print -> u                     |          u := parent[u]
                               |   end while
                               |  while S 不为空
                               |      print -> S.pop
                               |  end while
```


Now we pop **node 4** and work with it. We can go to **node 7** from **node 4**. **level[7]** = **level[4]** + **1** = **2**. We mark **node 7** as visited and push it in the queue.

```
                  front
+-----+  +-----+  +-----+
|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

From **node 3**, we can go to **node 7** and **node 8**. Since we've already marked **node 7** as visited, we mark **node 8** as visited, we change **level[8]** = **level[3]** + **1** = **2**. We push **node 8** in the queue.

```
                  front
+-----+  +-----+  +-----+
|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

This process will continue till we reach our destination or the queue becomes empty. The **level** array will provide us with the distance of the shortest path from **source**. We can initialize **level** array with *infinity* value, which will mark that the nodes are not yet visited. Our pseudo-code will be:

```
Procedure BFS(Graph, source):
Q = queue();
level[] = infinity
level[source] := 0
Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
        end if
    end for
end while
Return level
```

By iterating through the **level** array, we can find out the distance of each node from source. For example: the distance of **node 10** from **source** will be stored in **level[10]**.

Sometimes we might need to print not only the shortest distance, but also the path via which we can go to our destined node from the **source**. For this we need to keep a **parent** array. **parent[source]** will be NULL. For each update in **level** array, we'll simply add `parent[v] := u` in our pseudo code inside the for loop. After finishing BFS, to find the path, we'll traverse back the **parent** array until we reach **source** which will be denoted by NULL value. The pseudo-code will be:

```
Procedure PrintPath(u):  //recursive    |  Procedure PrintPath(u):  //iterative
if parent[u] is not equal to null       |  S = Stack()
    PrintPath(parent[u])                |  while parent[u] is not equal to null
end if                                  |      S.push(u)
print -> u                              |      u := parent[u]
                                        |  end while
                                        |  while S is not empty
                                        |      print -> S.pop
                                        |  end while
```

复杂度：

我们访问了每个节点一次和每条边一次。所以时间复杂度为 O(V + E) ，其中 V 是节点数，E 是边数。

第41.2节：在二维图中寻找源点的最短路径

大多数情况下，我们需要找出从单一源点到所有其他节点或特定节点的最短路径，比如：我们想知道骑士在棋盘上到达某个格子需要多少步，或者我们有一个数组，其中某些格子被阻挡，我们需要找出从一个格子到另一个格子的最短路径。我们只能水平和垂直移动，甚至也可以允许对角线移动。

对于这些情况，我们可以将格子或单元转换为节点，并使用广度优先搜索（BFS）轻松解决这些问题。现在我们的 visited、parent 和 level 都将是二维数组。对于每个节点，我们将考虑所有可能的移动。为了找到到特定节点的距离，我们还会检查是否已经到达目标。

这里会有一个额外的数组，称为方向数组。它简单地存储了我们可以移动的所有可能方向组合。比如，对于水平和垂直移动，我们的方向数组如下：

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

这里 dx 表示在x轴上的移动，dy 表示在y轴上的移动。这部分是可选的，你也可以单独写出所有可能的组合。但使用方向数组更容易处理。对于对角线移动或骑士移动，还可以有更多甚至不同的组合。

我们需要记住的额外部分是：

- 如果任何单元格被阻塞，对于每一个可能的移动，我们都会检查该单元格是否被阻塞。
- 我们还会检查是否越界，也就是是否超出了数组边界。
- 行数和列数将会被给出。

我们的伪代码将是：

```
过程 BFS2D(图，阻塞标志，行，列)：
对于 i 从 1 到 行
    对于 j 从 1 到 列
        访问标记[i][j] := 假
    结束 for
end for
访问标记[源点.x][源点.y] := 真
层级[源点.x][源点.y] := 0
队列 = 队列()
Q.入队(源点)
m := dx.大小
当 Q 不为空时
顶部 := Q.弹出
    对于 i 从 1 到 m
temp.x := top.x + dx[i]
temp.y := top.y + dy[i]
        如果 temp 在行和列内且 top 不等于 blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
```


Complexity:

We've visited every node once and every edges once. So the complexity will be **O(V + E)** where **V** is the number of nodes and **E** is the number of edges.

Section 41.2: Finding Shortest Path from Source in a 2D graph

Most of the time, we'll need to find out the shortest path from single source to all other nodes or a specific node in a 2D graph. Say for example: we want to find out how many moves are required for a knight to reach a certain square in a chessboard, or we have an array where some cells are blocked, we have to find out the shortest path from one cell to another. We can move only horizontally and vertically. Even diagonal moves can be possible too. For these cases, we can convert the squares or cells in nodes and solve these problems easily using BFS. Now our **visited**, **parent** and **level** will be 2D arrays. For each node, we'll consider all possible moves. To find the distance to a specific node, we'll also check whether we have reached our destination.

There will be one additional thing called direction array. This will simply store the all possible combinations of directions we can go to. Let's say, for horizontal and vertical moves, our direction arrays will be:

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

Here *dx* represents move in x-axis and *dy* represents move in y-axis. Again this part is optional. You can also write all the possible combinations separately. But it's easier to handle it using direction array. There can be more and even different combinations for diagonal moves or knight moves.

The additional part we need to keep in mind is:

- If any of the cell is blocked, for every possible moves, we'll check if the cell is blocked or not.
- We'll also check if we have gone out of bounds, that is we've crossed the array boundaries.
- The number of rows and columns will be given.

Our pseudo-code will be:

```
Procedure BFS2D(Graph, blocksign, row, column):
for i from 1 to row
    for j from 1 to column
        visited[i][j] := false
    end for
end for
visited[source.x][source.y] := true
level[source.x][source.y] := 0
Q = queue()
Q.push(source)
m := dx.size
while Q is not empty
    top := Q.pop
    for i from 1 to m
        temp.x := top.x + dx[i]
        temp.y := top.y + dy[i]
        if temp is inside the row and column and top doesn't equal to blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
```

```
结束 if
    结束 for
结束 while
返回 level
```

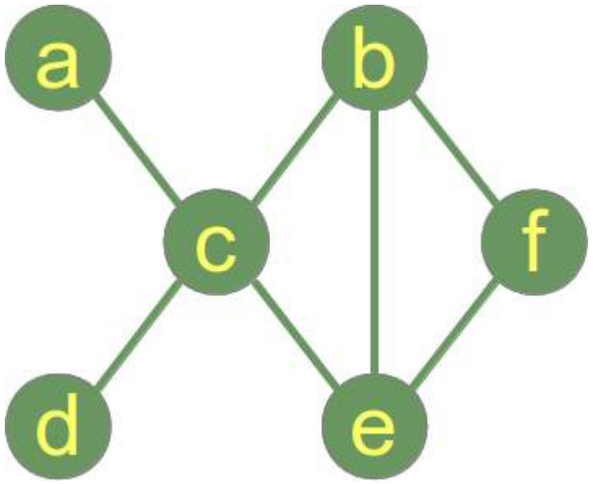
正如我们之前讨论的，BFS 只适用于无权图。对于有权图，我们需要使用 Dijkstra 算法。对于负边权环，我们需要 Bellman-Ford 算法。同样，这个算法是单源最短路径算法。如果我们需要找出每个节点到所有其他节点的距离，则需要 Floyd-Warshall 算法。

第 41.3 节：无向图的连通分量使用 BFS

BFS 可用于查找 无向图 的连通分量。我们还可以判断给定的图是否连通。以下讨论假设我们处理的是无向图。连通图的定义是：

> 如果图中每对顶点之间都有路径，则该图是连通的。

以下是一个 连通图。



以下图 不连通，且有 2 个连通分量：

1.连通分量 1: {a,b,c,d,e}
2.连通分量 2: {f}


```
        end if
    end for
end while
Return level
```

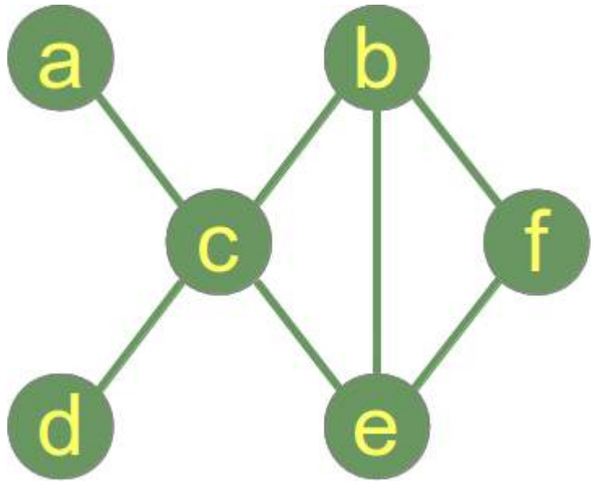
As we have discussed earlier, BFS only works for unweighted graphs. For weighted graphs, we'll need Dijkstra's algorithm. For negative edge cycles, we need Bellman-Ford's algorithm. Again this algorithm is single source shortest path algorithm. If we need to find out distance from each nodes to all other nodes, we'll need Floyd-Warshall's algorithm.

Section 41.3: Connected Components Of Undirected Graph Using BFS

BFS can be used to find the connected components of an undirected graph. We can also find if the given graph is connected or not. Our subsequent discussion assumes we are dealing with undirected graphs.The definition of a connected graph is:

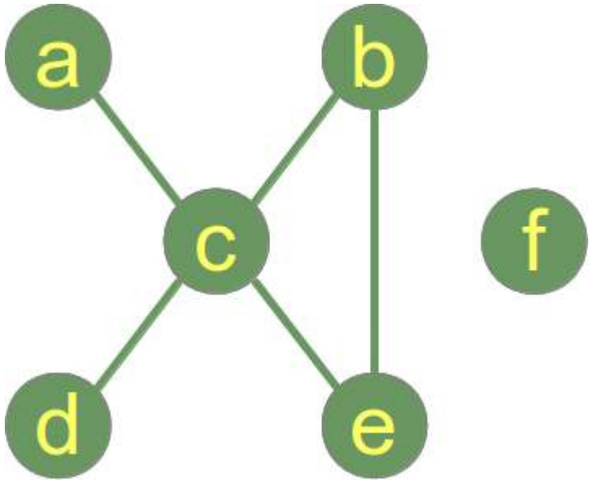
> A graph is connected if there is a path between every pair of vertices.

Following is a **connected graph**.



Following graph is **not connected** and has 2 connected components:

1. Connected Component 1: {a,b,c,d,e}
2. Connected Component 2: {f}



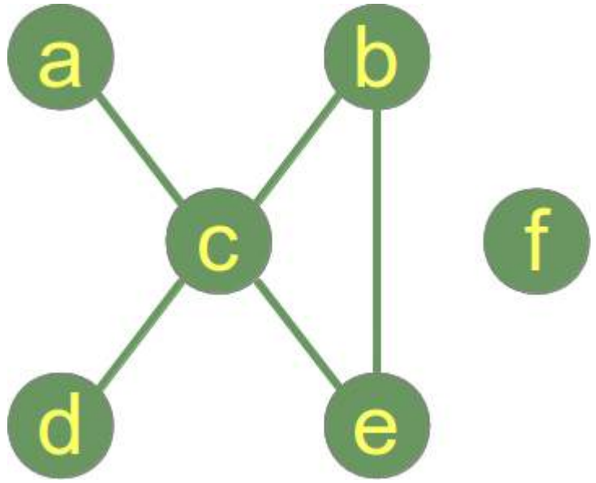
BFS 是一种图遍历算法。因此，从一个随机的源节点开始，如果算法结束时所有节点都被访问过，则该图是连通的，否则该图是不连通的。

算法的伪代码。

```
boolean isConnected(Graph g)
{
BFS(v)//v 是一个随机的源节点。
 if(allVisited(g))
 {
  return true;
 }
 else return false;
}
```

用于判断无向图是否连通的 C 语言实现：

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//队列节点表示单个队列元素
//它不是图的节点。
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//用于记录已访问顶点的数组。

int main()
```

BFS is a graph traversal algorithm. So starting from a random source node, if on termination of algorithm, all nodes are visited, then the graph is connected,otherwise it is not connected.

PseudoCode for the algorithm.

```
boolean isConnected(Graph g)
{
 BFS(v)//v is a random source node.
 if(allVisited(g))
 {
  return true;
 }
 else return false;
}
```

C implementation for finding the whether an undirected graph is connected or not:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//Queue node depicts a single Queue element
//It is NOT a graph node.
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//array that keeps track of visited vertices.

int main()
```

```
{
    int n,e;//n 是顶点数，e 是边数。
    int i,j;
    char **graph;//邻接矩阵

    printf("请输入顶点数:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "请输入1到%d之间的有效正整数",MAXVERTICES);
     return -1;
    }

graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//初始时所有顶点均未访问。
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("请输入边的数量，然后成对输入边:");
    scanf("%d",&e);

    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("图是连通的");else printf("图不连通")
    ;
}

void 入队(int 顶点)
{
    if(Qfront == NULL)
    {
Qfront = malloc(sizeof(Node));
        Qfront->v = 顶点;
Qfront->next = NULL;
        Qrear = Qfront;
    }
    否则
    {
Nodeptr 新节点 = malloc(sizeof(Node));
       新节点->v = 顶点;
新节点->next = NULL;
        Qrear->next = 新节点;
        Qrear = 新节点;
    }
}

int 出队()
{
```



```
{
    int n,e;//n is number of vertices, e is number of edges.
    int i,j;
    char **graph;//adjacency matrix

    printf("Enter number of vertices:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "Please enter a valid positive integer from 1 to %d",MAXVERTICES);
     return -1;
    }

    graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
        graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//initially all vertices are not visited.
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("enter number of edges and then enter them in pairs:");
    scanf("%d",&e);

    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("The graph is connected");
    else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if(Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int deque()
{
```

```
    if(Qfront == NULL)
    {
        printf("队列为空，返回 -1");return -1;

    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
Qfront = Qfront->next;
            Qrear = NULL;
        }
        否则
Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //假设随机源顶点为顶点0；
   BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 表示假；

    return 1;//1 表示真；
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

要查找无向图的所有连通分量，我们只需在BFS函数中添加两行代码。思路是调用BFS函数直到所有顶点都被访问。

需要添加的代码行是：

```
printf("连通分量 %d",++count);
//count 是一个全局变量，初始化为0
//将此行添加为BFS函数的第一行
```



```
    if(Qfront == NULL)
    {
        printf("Q is empty , returning -1\n");
        return -1;
    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
            Qfront = Qfront->next;
            Qrear = NULL;
        }
        else
            Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //let random source vertex be vertex 0;
    BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 implies false;

    return 1;//1 implies true;
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
    visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

For Finding all the Connected components of an undirected graph, we only need to add 2 lines of code to the BFS function. The idea is to call BFS function until all vertices are visited.

The lines to be added are:

```
printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function
```

与

```
printf("%d  ",vertex+1);
将此作为BFS中while循环的第一行添加
```

我们定义以下函数：

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```

AND

```
printf("%d ",vertex+1);
add this as first line of while loop in BFS
```

and we define the following function:

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```

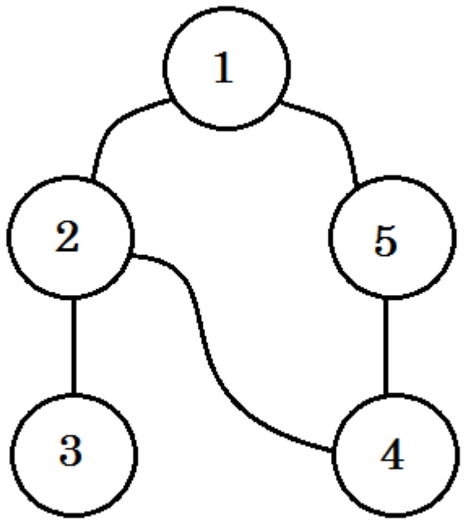
第42章：深度优先搜索

第42.1节：深度优先搜索简介

深度优先搜索是一种遍历或搜索树或图数据结构的算法。它从根节点开始，沿着每个分支尽可能深入地探索，然后回溯。19世纪法国数学家查尔斯·皮埃尔·特雷莫（Charles Pierre Trémaux）研究过深度优先搜索的一个版本，作为解决迷宫的策略。

深度优先搜索是一种系统地查找从源顶点可达的所有顶点的方法。与广度优先搜索类似，DFS遍历给定图的一个连通分量并定义一个生成树。深度优先搜索的基本思想是有条不紊地探索每条边。必要时我们从不同的顶点重新开始。一旦发现一个顶点，DFS就从该顶点开始探索（不同于BFS，它将顶点放入队列以便稍后探索）。

让我们来看一个例子。我们将遍历这张图：



我们将遵循以下规则遍历图：

- 我们将从源节点开始。
- 任何节点都不会被访问两次。
- 尚未访问的节点将被标记为白色。
- 已访问但未访问其所有子节点的节点将被标记为灰色。
- 完全遍历的节点将被标记为黑色。

让我们一步步来看：

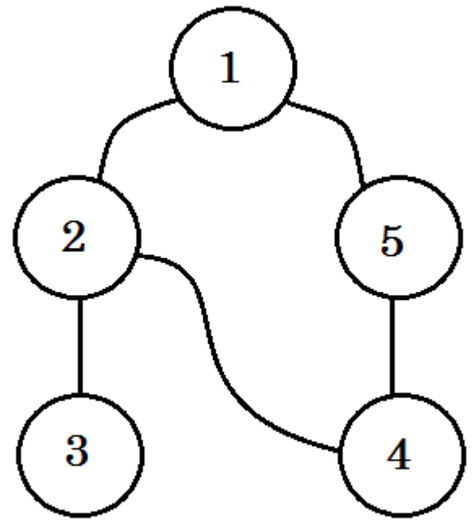
Chapter 42: Depth First Search

Section 42.1: Introduction To Depth-First Search

Depth-first search is an algorithm for traversing or searching tree or graph data structures. One starts at the root and explores as far as possible along each branch before backtracking. A version of depth-first search was investigated in the 19th century French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

Depth-first search is a systematic way to find all the vertices reachable from a source vertex. Like breadth-first search, DFS traverse a connected component of a given graph and defines a spanning tree. The basic idea of depth-first search is methodically exploring every edge. We start over from a different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it (unlike BFS, which puts a vertex on a queue so that it explores from it later).

Let's look at an example. We'll traverse this graph:

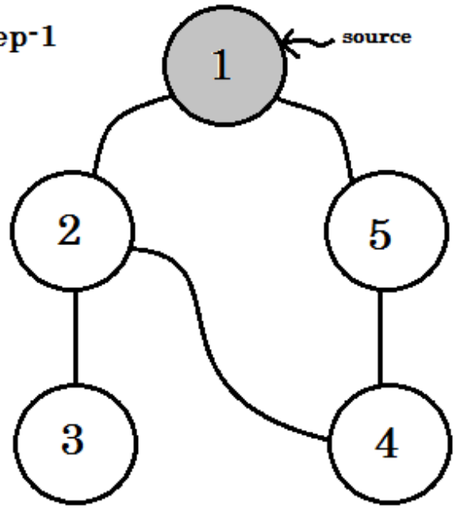


We'll traverse the graph following these rules:

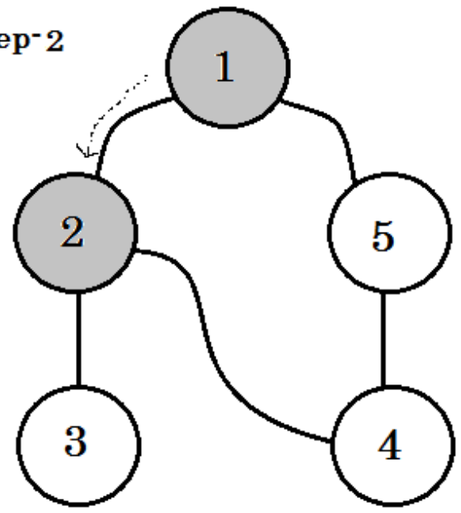
- We'll start from the source.
- No node will be visited twice.
- The nodes we didn't visit yet, will be colored white.
- The node we visited, but didn't visit all of its child nodes, will be colored grey.
- Completely traversed nodes will be colored black.

Let's look at it step by step:

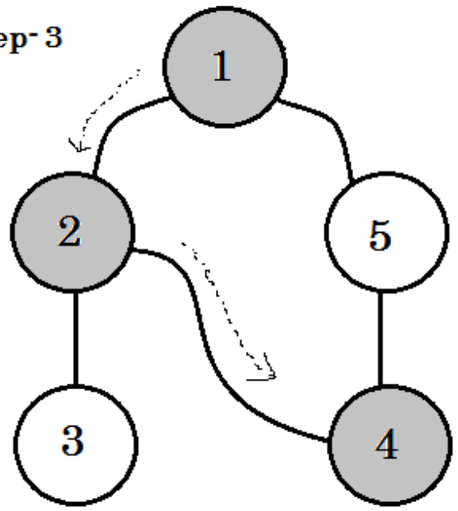
step-1



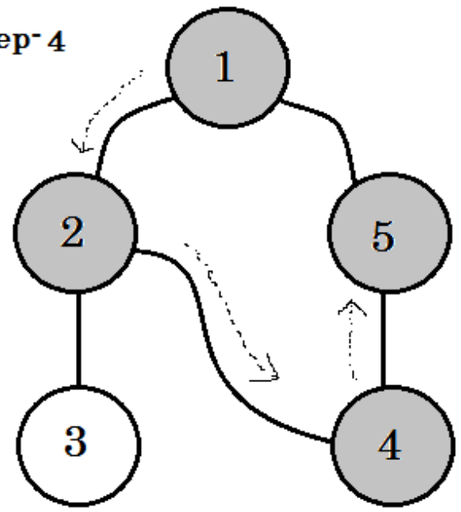
step-2



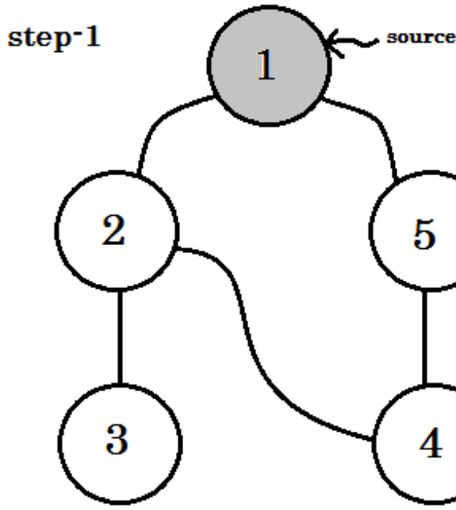
step-3



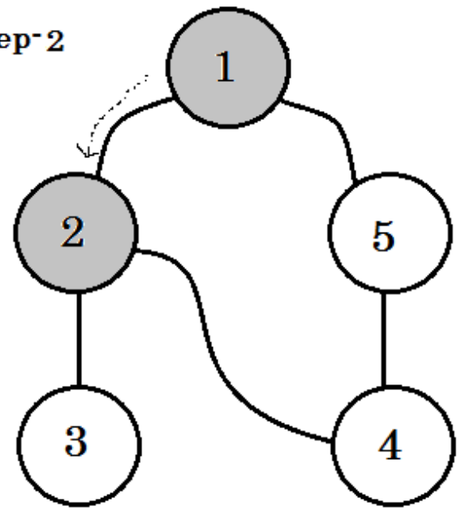
step-4



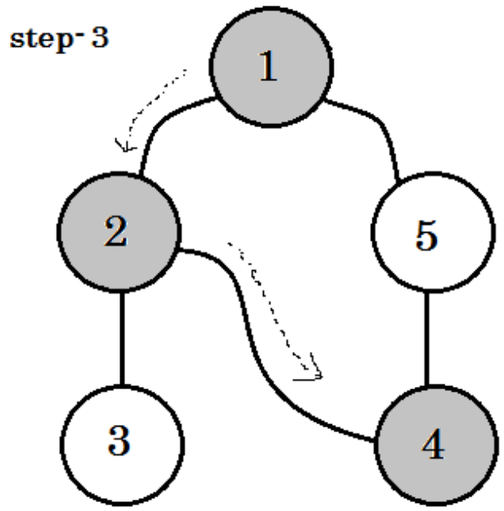
step-1



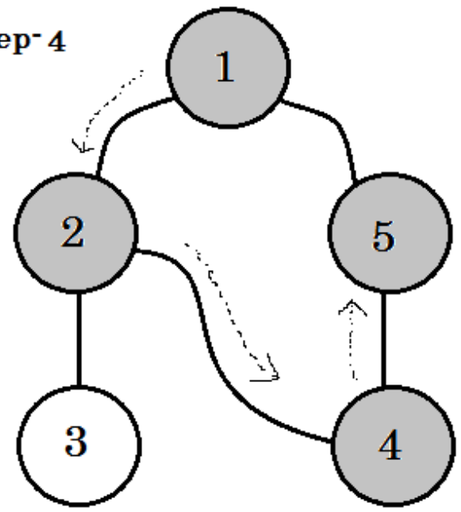
step-2



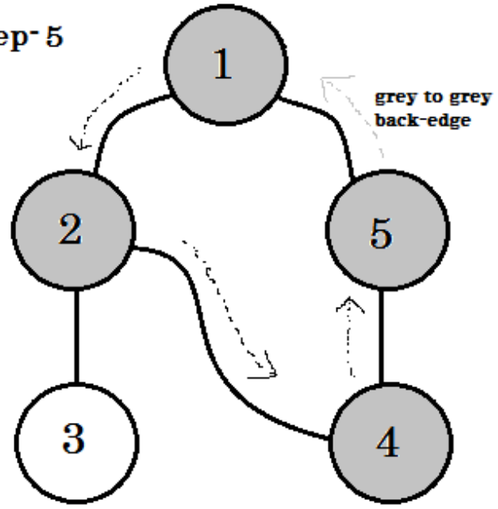
step-3



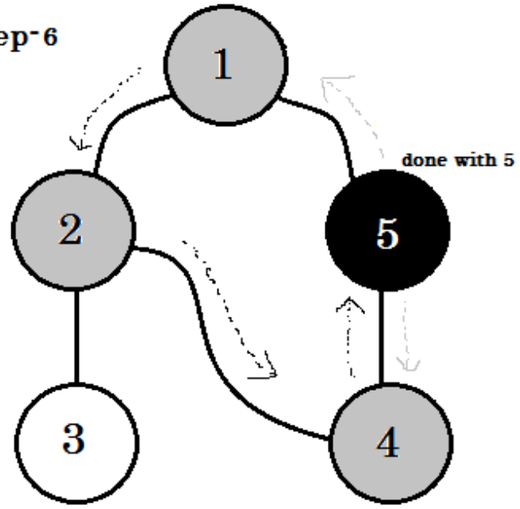
step-4



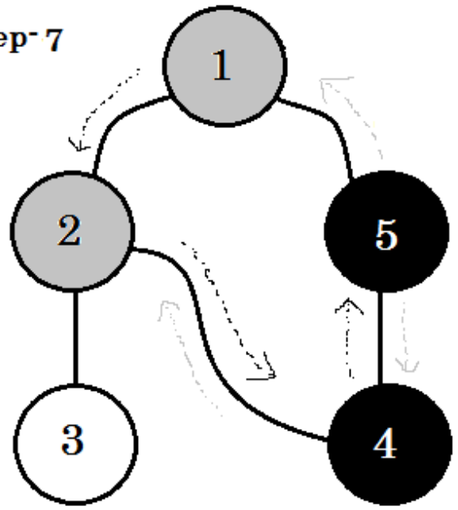
step-5



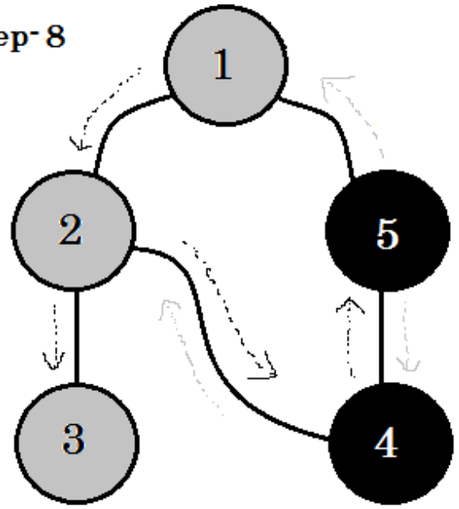
step-6



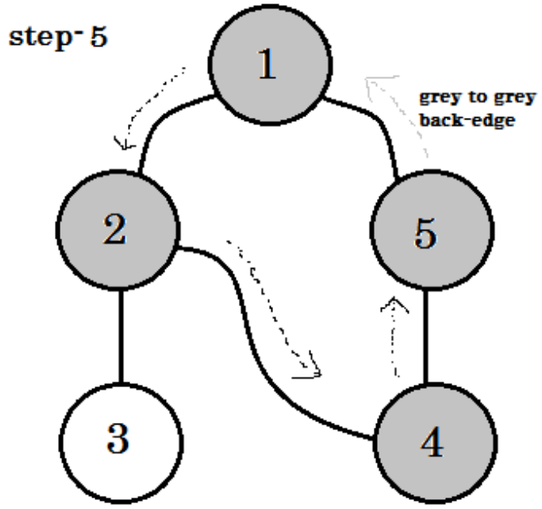
step-7



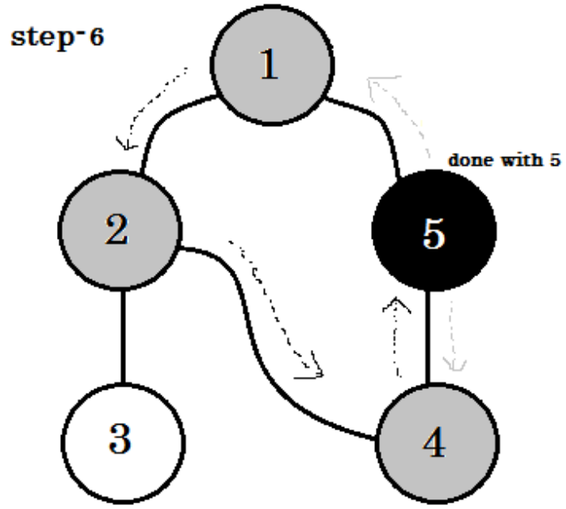
step-8



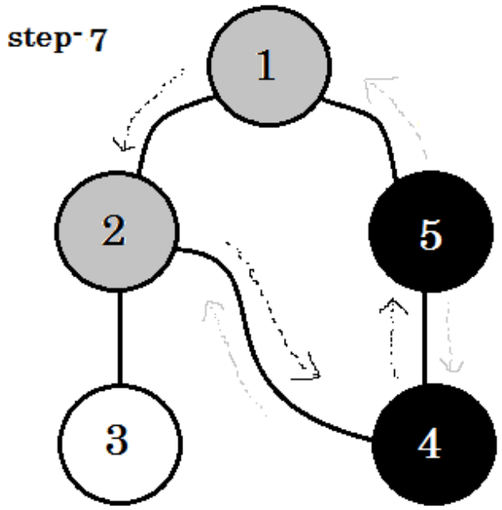
step-5



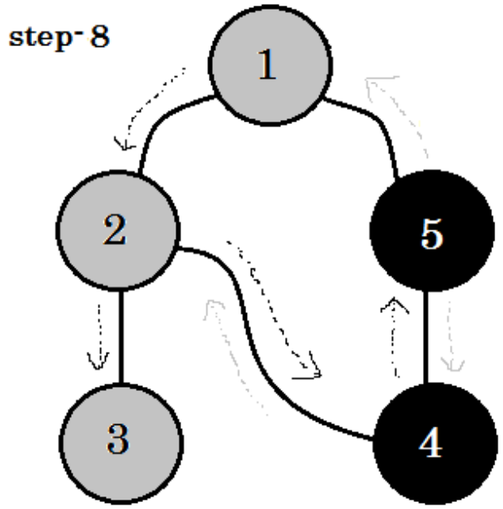
step-6

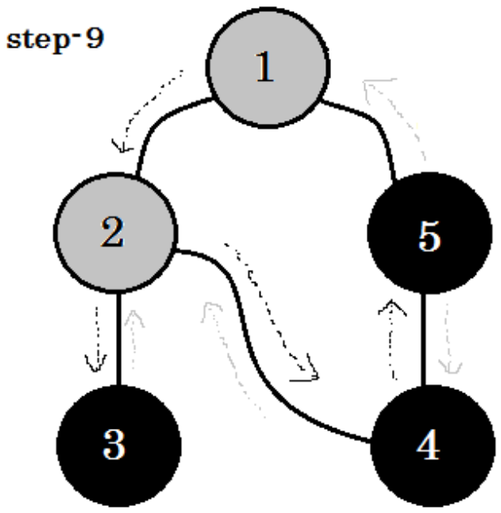


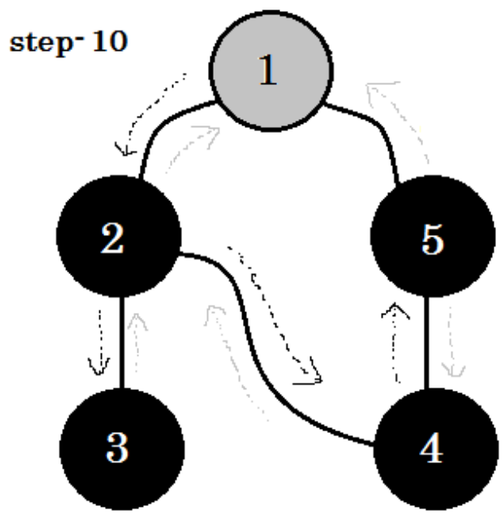
step-7

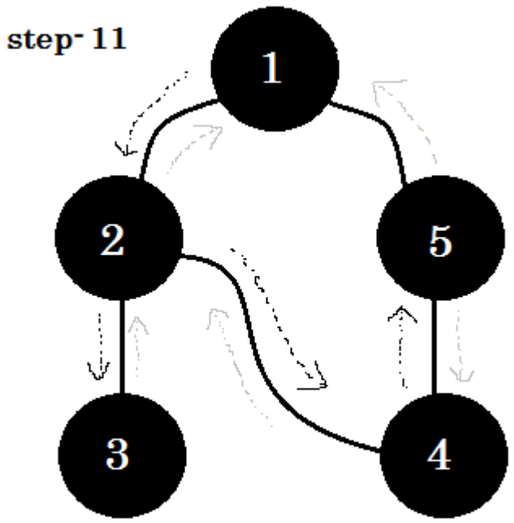


step-8









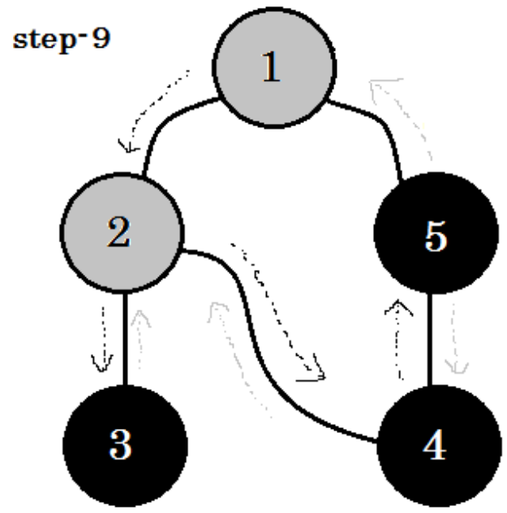
我们可以看到一个重要的关键词。那就是backedge。你可以看到，5-1被称为backedge。这是因为，我们还没有完成对node-1的处理，所以从另一个节点到node-1意味着图中存在一个环。在深度优先搜索（DFS）中，如果我们可以从一个灰色节点到另一个灰色节点，我们可以确定图中有环。这是检测图中环的一种方法。根据source节点和访问节点的顺序，我们可以将环中的任何边识别为backedge。例如：如果我们先从1访问5，我们会发现2-1是backedge。

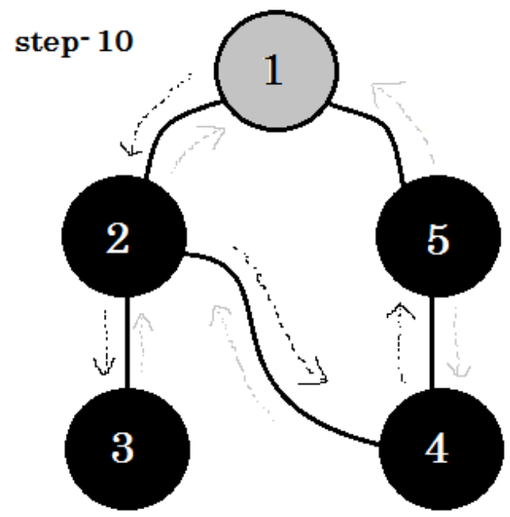
从灰色节点到白色节点的边称为**tree edge**。如果我们只保留**tree edge**并移除其他边，就会得到**DFS树**。

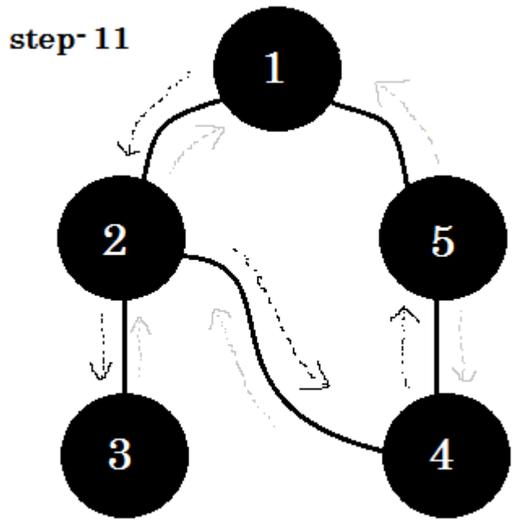
在无向图中，如果我们访问了一个已经访问过的节点，那一定是**backedge**。但对于有向图，我们必须检查颜色。*只有当我们可以从一个灰色节点到另一个灰色节点时，才称为backedge*。

在DFS中，我们还可以为每个节点记录时间戳，这可以用于多种用途（例如：拓扑排序）。

1. 当节点v从白色变为灰色时，时间记录在d[v]中。







We can see one important keyword. That is **backedge**. You can see. **5-1** is called backedge. This is because, we're not yet done with **node-1**, so going from another node to **node-1** means there's a cycle in the graph. In DFS, if we can go from one gray node to another, we can be certain that the graph has a cycle. This is one of the ways of detecting cycle in a graph. Depending on **source** node and the order of the nodes we visit, we can find out any edge in a cycle as **backedge**. For example: if we went to **5** from **1** first, we'd have found out **2-1** as backedge.

The edge that we take to go from gray node to white node are called **tree edge**. If we only keep the **tree edge**'s and remove others, we'll get **DFS tree**.

In undirected graph, if we can visit a already visited node, that must be a **backedge**. But for directed graphs, we must check the colors. *If and only if we can go from one gray node to another gray node, that is called a backedge*.

In DFS, we can also keep timestamps for each node, which can be used in many ways (e.g.: Topological Sort).

1. When a node **v** is changed from white to gray the time is recorded in **d[v]**.

2. 当节点v从灰色变为黑色时，时间记录在f[v]中。

这里d[]表示发现时间，f[]表示完成时间。我们的伪代码如下：

```
过程 DFS(G)：
对于G中每个节点 u
    color[u] := white
    parent[u] := NULL
结束for
time := 0
对于图 G 中的每个节点 u
    如果 color[u] == white
        调用 DFS-访问(u)
    结束 如果
end for

过程 DFS-访问(u):
color[u] := gray
time := time + 1
d[u] := time
对于每个与 u 相邻的节点 v
    如果 color[v] == white
parent[v] := u
        调用 DFS-访问(v)
    结束 如果
end for
color[u] := black
time := time + 1
f[u] := time
```

复杂度：

每个节点和边都被访问一次。因此，DFS 的时间复杂度是 O(V+E)，其中 V 表示节点数，E 表示边数。

深度优先搜索的应用：

- 在无向图中寻找所有节点对的最短路径。
- 检测图中的环。
- 路径查找。
- 拓扑排序。
- 测试图是否为二分图。
- 寻找强连通分量。
- 解决只有一个解的谜题。

2. When a node **v** is changed from gray to black the time is recorded in **f[v]**.

Here **d[]** means *discovery time* and **f[]** means *finishing time*. Our pesudo-code will look like:

```
Procedure DFS(G):
for each node u in V[G]
    color[u] := white
    parent[u] := NULL
end for
time := 0
for each node u in V[G]
    if color[u] == white
        DFS-Visit(u)
    end if
end for

Procedure DFS-Visit(u):
color[u] := gray
time := time + 1
d[u] := time
for each node v adjacent to u
    if color[v] == white
        parent[v] := u
        DFS-Visit(v)
    end if
end for
color[u] := black
time := time + 1
f[u] := time
```

Complexity:

Each nodes and edges are visited once. So the complexity of DFS is **O(V+E)**, where **V** denotes the number of nodes and **E** denotes the number of edges.

Applications of Depth First Search:

- Finding all pair shortest path in an undirected graph.
- Detecting cycle in a graph.
- Path finding.
- Topological Sort.
- Testing if a graph is bipartite.
- Finding Strongly Connected Component.
- Solving puzzles with one solution.

第43章：哈希函数

第43.1节：C#中常见类型的哈希码

下面显示了GetHashCode()方法为System
命名空间中的内置和常见C#类型生成的哈希码。

布尔型

如果值为true，则为1，否则为0。

字节，无符号16位整数，有符号32位整数，无符号32位整数，单精度浮点数

值（如有必要，转换为 Int32）。

SByte

```
((int)m_value ^ (int)m_value << 8);
```

字符

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64, Double

64 位数字的低 32 位和高 32 位之间的异或

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64, DateTime, TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Decimal

```
((((int *)&dbl)[0]) & 0xFFFFFFF0) ^ ((int *)&dbl)[1];
```

Object

```
RuntimeHelpers.GetHashCode(this);
```

默认实现使用同步块索引。

String

哈希码计算依赖于平台类型（Win32 或 Win64）、是否使用随机字符串哈希、调试/发布模式。对于 Win64 平台：

```
int hash1 = 5381;
int hash2 = hash1;
int c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```


Chapter 43: Hash Functions

Section 43.1: Hash codes for common types in C#

The hash codes produced by `GetHashCode()` method for built-in and common C# types from the `System` namespace are shown below.

Boolean

1 if value is true, 0 otherwise.

Byte, UInt16, Int32, UInt32, Single

Value (if necessary casted to Int32).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

Char

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64, Double

Xor between lower and upper 32 bits of 64 bit number

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64, DateTime, TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Decimal

```
((((int *)&dbl)[0]) & 0xFFFFFFF0) ^ ((int *)&dbl)[1];
```

Object

```
RuntimeHelpers.GetHashCode(this);
```

The default implementation is used sync block index.

String

Hash code computation depends on the platform type (Win32 or Win64), feature of using randomized string hashing, Debug / Release mode. In case of Win64 platform:

```
int hash1 = 5381;
int hash2 = hash1;
int c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```

```
return hash1 + (hash2 * 1566083941);
```

ValueType

查找第一个非静态字段并获取其哈希码。如果类型没有非静态字段，则返回该类型的哈希码。静态成员的哈希码无法获取，因为如果该成员与原始类型相同，计算将陷入无限循环。

Nullable<T>

```
return hasValue ? value.GetHashCode() : 0;
```

Array

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

参考文献

- GitHub .Net Core CLR

第43.2节：哈希函数简介

哈希函数 `h()` 是一个任意函数，将任意大小的数据 `x ∈ X` 映射到固定大小的值 `y ∈ Y`：`y = h(x)`。好的哈希函数满足以下限制条件：

- 哈希函数表现得像均匀分布哈希函数是确定性的。
- h(x) 对于给定的 x应始终返回相同的值计算快速（运行时间为 O(1)）
-

一般情况下，哈希函数的大小小于输入数据的大小：|y| < |x|。哈希函数不可逆，或者换句话说，可能存在冲突：∃ x 1, x2 ∈ X, x1 ≠ x2: h(x1) = h(x2)。 X 可以是有限或无限集合，Y 是有限集合。

哈希函数被广泛应用于计算机科学的许多领域，例如软件工程、密码学、数据库、网络、机器学习等。哈希函数有许多不同类型，具有不同的领域特定属性。

哈希值通常是整数。在编程语言中有专门的方法用于计算哈希值。例如，在 C# GetHashCode() 方法对所有类型返回 Int32 值（32位整数）。在 Java 中，每个类都提供 hashCode() 方法，返回 int。每种数据类型都有自己的或用户定义的实现。

哈希方法

确定哈希函数有几种方法。一般来说，设 x ∈ X = {z ∈ ℤ: z ≥ 0} 是正整数。通常 m 是素数（不太接近2的整数次幂）。

方法	哈希函数
划分方法	h(x) = x mod m
乘法法	h(x) = ⌊m (xA mod 1)⌋, A ∈ {z ∈ ℝ: 0 < z < 1}

哈希表

哈希函数用于哈希表中计算槽数组的索引。哈希表是一种数据结构，用于


```
return hash1 + (hash2 * 1566083941);
```

ValueType

The first non-static field is look for and get it's hashcode. If the type has no non-static fields, the hashcode of the type returns. The hashcode of a static member can't be taken because if that member is of the same type as the original type, the calculating ends up in an infinite loop.

Nullable<T>

```
return hasValue ? value.GetHashCode() : 0;
```

Array

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

References

- GitHub .Net Core CLR

Section 43.2: Introduction to hash functions

Hash function `h()` is an arbitrary function which mapped data `x ∈ X` of arbitrary size to value `y ∈ Y` of fixed size: `y = h(x)`. Good hash functions have follows restrictions:

- hash functions behave likes uniform distribution
- hash functions is deterministic. `h(x)` should always return the same value for a given `x`
- fast calculating (has runtime O(1))

In general case size of hash function less then size of input data: `|y| < |x|`. Hash functions are not reversible or in other words it may be collision: `∃ x1, x2 ∈ X, x1 ≠ x2: h(x1) = h(x2)`. `X` may be finite or infinite set and `Y` is finite set.

Hash functions are used in a lot of parts of computer science, for example in software engineering, cryptography, databases, networks, machine learning and so on. There are many different types of hash functions, with differing domain specific properties.

Often hash is an integer value. There are special methods in programmning languages for hash calculating. For example, in `C#` `GetHashCode()` method for all types returns `Int32` value (32 bit integer number). In `Java` every class provides `hashCode()` method which return **`int`**. Each data type has own or user defined implementations.

Hash methods

There are several approaches for determinig hash function. Without loss of generality, lets `x ∈ X = {z ∈ ℤ: z ≥ 0}` are positive integer numbers. Often `m` is prime (not too close to an exact power of 2).

Method	Hash function
Division method	h(x) = x mod m
Multiplication method	h(x) = ⌊m (xA mod 1)⌋, A ∈ {z ∈ ℝ: 0 < z < 1}

Hash table

Hash functions used in hash tables for computing index into an array of slots. Hash table is data structure for

实现字典（键值结构）。良好实现的哈希表在以下操作中具有O(1)时间复杂度：通过键插入、搜索和删除数据。多个键可能会哈希到同一个槽位。解决冲突有两种方法：

1. 链地址法：使用链表存储在同一槽中具有相同哈希值的元素
2. 开放定址法：每个槽中存储零个或一个元素

以下方法用于计算开放寻址所需的探测序列

方法	公式
线性探测	`h(x, i) = (h'(x) + i) mod m`
二次探测	`h(x, i) = (h'(x) + c1*i + c2*i^2) mod m`
双重哈希	`h(x, i) = (h1(x) + i*h2(x)) mod m`

其中 `i ∈ {0, 1, ..., m-1}`，`h'(x)`, `h1(x)`, `h2(x)` 是辅助哈希函数，`c1`, `c2` 是正的辅助常数。

示例

设 `x ∈ U{1, 1000}`，`h = x mod m`。下表显示了非质数和质数情况下的哈希值。加粗文本表示相同的哈希值。

x	m = 100（非质数）	m = 101（质数）
723	23	16
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86
549	49	44
991	91	82
757	**57**	50
920	20	11
626	26	20
557	**57**	52
831	31	23
619	19	13

链接

- 托马斯·H·科尔门，查尔斯·E·莱瑟森，罗纳德·L·里维斯特，克利福德·斯坦。《算法导论》。
- 哈希表概述
- Wolfram MathWorld - 哈希函数

implementing dictionaries (key-value structure). Good implemented hash tables have O(1) time for the next operations: insert, search and delete data by key. More than one keys may hash to the same slot. There are two ways for resolving collision:

1. Chaining: linked list is used for storing elements with the same hash value in slot
2. Open addressing: zero or one element is stored in each slot

The next methods are used to compute the probe sequences required for open addressing

Method	Formula
Linear probing	`h(x, i) = (h'(x) + i) mod m`
Quadratic probing	`h(x, i) = (h'(x) + c1*i + c2*i^2) mod m`
Double hashing	`h(x, i) = (h1(x) + i*h2(x)) mod m`

Where `i ∈ {0, 1, ..., m-1}`, `h'(x), h1(x), h2(x)` are auxiliary hash functions, `c1`, `c2` are positive auxiliary constants.

Examples

Lets `x ∈ U{1, 1000}`, `h = x mod m`. The next table shows the hash values in case of not prime and prime. Bolded text indicates the same hash values.

x	m = 100 (not prime)	m = 101 (prime)
723	23	16
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86
549	49	44
991	91	82
757	**57**	50
920	20	11
626	26	20
557	**57**	52
831	31	23
619	19	13

Links

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.
- Overview of Hash Tables
- Wolfram MathWorld - Hash Function

第44章：旅行商问题

第44.1节：暴力算法

通过每个顶点恰好一次的路径等同于以某种方式对顶点进行排序。因此，为了计算通过每个顶点恰好一次的最小旅行成本，我们可以对所有N!种排列进行暴力枚举
从1到N的数字的排列。

伪代码

```
minimum = INF
对于所有排列 P

current = 0

    对于 i 从 0 到 N-2
current = current + cost[P[i]][P[i+1]]  <- 加上从第1个顶点到下一个顶点的代价     current = current + cost[P[N-1]][P[0]]

  <- 加上从最后一个顶点到第一个顶点的代价


    if current < minimum                          <- 如有必要，更新最小值
        minimum = current

输出最小值
```

时间复杂度

共有N!种排列需要遍历，每条路径的代价计算时间为O(N)，因此该算法的时间复杂度为 O(N * N!)，用于输出精确答案。

第44.2节：动态规划算法

注意，如果我们考虑以下路径（按顺序）：

```
(1,2,3,4,6,0,5,7)
```

以及路径

```
(1,2,3,5,0,6,7,4)
```

从顶点1到顶点2再到顶点3的成本保持不变，那么为什么必须重新计算？这个结果
可以保存以备后用。

令 dp[bitmask][vertex]表示经过所有对应于bitmask中位为1的顶点，且路径终点为vertex的最小花费。例如：

```
dp[12][2]

   12   =   1 1 0 0
            ^ ^
顶点:   3 2 1 0
```

由于12在二进制中表示为1100，dp[12][2]表示经过图中顶点2和3，路径终点为顶点2。

Chapter 44: Travelling Salesman

Section 44.1: Brute Force Algorithm

A path through every vertex exactly once is the same as ordering the vertex in some way. Thus, to calculate the minimum cost of travelling through every vertex exactly once, we can brute force every single one of the `N!` permutations of the numbers from `1` to `N`.

Psuedocode

```
minimum = INF
for all permutations P

    current = 0

    for i from 0 to N-2
        current = current + cost[P[i]][P[i+1]]  <- Add the cost of going from 1 vertex to the next

    current = current + cost[P[N-1]][P[0]]      <- Add the cost of going from last vertex to the
first

    if current < minimum                        <- Update minimum if necessary
        minimum = current

output minimum
```

Time Complexity

There are `N!` permutations to go through and the cost of each path is calculated in `O(N)`, thus this algorithm takes `O(N * N!)` time to output the exact answer.

Section 44.2: Dynamic Programming Algorithm

Notice that if we consider the path (in order):

```
(1,2,3,4,6,0,5,7)
```

and the path

```
(1,2,3,5,0,6,7,4)
```

The cost of going from vertex `1` to vertex `2` to vertex `3` remains the same, so why must it be recalculated? This result can be saved for later use.

Let `dp[bitmask][vertex]` represent the minimum cost of travelling through all the vertices whose corresponding bit in `bitmask` is set to `1` ending at `vertex`. For example:

```
dp[12][2]

   12   =   1 1 0 0
            ^ ^
vertices:   3 2 1 0
```

Since `12` represents `1100` in binary, `dp[12][2]` represents going through vertices `2` and `3` in the graph with the path ending at vertex `2`.

因此我们可以有如下算法（C++实现）：

```
int cost[N][N];                         //如有需要调整N的值
int memo[1 << N][N];                    //这里全部初始化为-1
int TSP(int bitmask, int pos){
    int cost = INF;
    if (bitmask == ((1 << N) - 1)){     //所有顶点均已访问
        return cost[pos][0];            //返回到起点的花费
    }
    if (memo[bitmask][pos] != -1){      //如果这个已经被计算过了
        return memo[bitmask][pos];      //直接返回值，无需重新计算
    }
    for (int i = 0; i < N; ++i){        //遍历每个顶点
        if ((bitmask & (1 << i)) == 0){ //如果该顶点尚未被访问
            cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);   //访问该顶点
        }
    }
memo[bitmask][pos] = cost;          //保存结果
    return cost;
}
//调用 TSP(1,0)
```

这行代码可能有点难理解，我们慢慢来分析：

```
cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);
```

这里，`bitmask | (1 << i)` 将 `bitmask` 的第 `i` 位设置为 1，表示第 `i` 个顶点已被访问。逗号后的 `i` 表示该函数调用中的新 `pos`，即新的“最后”顶点。
`cost[pos][i]` 是从顶点 `pos` 到顶点 `i` 的旅行成本。

因此，这行代码的作用是将 `cost` 更新为访问所有尚未访问顶点的最小可能值。

时间复杂度

函数 TSP(bitmask,pos) 对 bitmask 有 2^N 种取值，对 pos 有 N 种取值。每个函数调用运行时间为 O(N)（即 for 循环）。因此，该实现的时间复杂度为 O(N^2 * 2^N)，能够输出精确答案。

Thus we can have the following algorithm (C++ implementation):

```
int cost[N][N];                         //Adjust the value of N if needed
int memo[1 << N][N];                    //Set everything here to -1
int TSP(int bitmask, int pos){
    int cost = INF;
    if (bitmask == ((1 << N) - 1)){     //All vertices have been explored
        return cost[pos][0];            //Cost to go back
    }
    if (memo[bitmask][pos] != -1){      //If this has already been computed
        return memo[bitmask][pos];      //Just return the value, no need to recompute
    }
    for (int i = 0; i < N; ++i){        //For every vertex
        if ((bitmask & (1 << i)) == 0){ //If the vertex has not been visited
            cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);   //Visit the vertex
        }
    }
    memo[bitmask][pos] = cost;          //Save the result
    return cost;
}
//Call TSP(1,0)
```

This line may be a little confusing, so lets go through it slowly:

```
cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);
```

Here, `bitmask | (1 << i)` sets the ith bit of `bitmask` to 1, which represents that the ith vertex has been visited. The `i` after the comma represents the new `pos` in that function call, which represents the new "last" vertex.
`cost[pos][i]` is to add the cost of travelling from vertex `pos` to vertex `i`.

Thus, this line is to update the value of `cost` to the minimum possible value of travelling to every other vertex that has not been visited yet.

Time Complexity

The function `TSP(bitmask,pos)` has `2^N` values for `bitmask` and `N` values for `pos`. Each function takes `O(N)` time to run (the **`for`** loop). Thus this implementation takes `O(N^2 * 2^N)` time to output the exact answer.

第45章：背包问题

第45.1节：背包问题基础

问题： 给定一组物品，每个物品包含重量和价值，确定每种物品的数量以使总重量小于或等于给定限制且总价值尽可能大。

背包问题伪代码

给定：

1.价值（数组 v）
2.重量（数组 w）
3.不同物品数量（n）
4.容量（W）

```
对于 j 从 0 到 W 执行：
m[0, j] := 0
对于 i 从 1 到 n 执行：
    对于 j 从 0 到 W 执行：
        如果 w[i] > j 则：
m[i, j] := m[i-1, j]
        else:
m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

上述伪代码的一个简单Python实现：

```
def knapSack(W, wt, val, n):
K = [[0 for x in range(W+1)] for x in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
K[i][w] = 0
            elif wt[i-1] <= w:
K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
K[i][w] = K[i-1][w]
    return K[n][W]
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

运行代码：将此保存为名为 knapSack.py 的文件

```
$ python knapSack.py
220
```

上述代码的时间复杂度：O(nW) 其中 `n` 是物品数量，W 是背包容量。

第45.2节：C#实现的解决方案

```
public class KnapsackProblem
{
```


Chapter 45: Knapsack Problem

Section 45.1: Knapsack Problem Basics

The Problem: Given a set of items where each item contains a weight and value, determine the number of each to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Pseudo code for Knapsack Problem

Given:

1. Values(array v)
2. Weights(array w)
3. Number of distinct items(n)
4. Capacity(W)

```
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

A simple implementation of the above pseudo code using Python:

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

Running the code: Save this in a file named knapSack.py

```
$ python knapSack.py
220
```

Time Complexity of the above code: `O(nW)` where n is the number of items and W is the capacity of knapsack.

Section 45.2: Solution Implemented in C#

```
public class KnapsackProblem
{
```

```
    private static int Knapsack(int w, int[] weight, int[] value, int n)
    {
        int i;
        int[,] k = new int[n + 1, w + 1];
        for (i = 0; i <= n; i++)
        {
            int b;
            for (b = 0; b <= w; b++)
            {
                if (i==0 || b==0)
                {
k[i, b] = 0;
                }
                else if (weight[i - 1] <= b)
                {
k[i, b] = Math.Max(value[i - 1] + k[i - 1, b - weight[i - 1]], k[i - 1, b]);
                }
                否则
                {
k[i, b] = k[i - 1, b];
                }
            }
        }
        return k[n, w];
    }

    public static int Main(int nItems, int[] weights, int[] values)
    {
        int n = values.Length;
        return Knapsack(nItems, weights, values, n);
    }
}
```



```
    private static int Knapsack(int w, int[] weight, int[] value, int n)
    {
        int i;
        int[,] k = new int[n + 1, w + 1];
        for (i = 0; i <= n; i++)
        {
            int b;
            for (b = 0; b <= w; b++)
            {
                if (i==0 || b==0)
                {
                    k[i, b] = 0;
                }
                else if (weight[i - 1] <= b)
                {
                    k[i, b] = Math.Max(value[i - 1] + k[i - 1, b - weight[i - 1]], k[i - 1, b]);
                }
                else
                {
                    k[i, b] = k[i - 1, b];
                }
            }
        }
        return k[n, w];
    }

    public static int Main(int nItems, int[] weights, int[] values)
    {
        int n = values.Length;
        return Knapsack(nItems, weights, values, n);
    }
}
```

第46章：方程求解

第46.1节：线性方程

求解线性方程组的方法有两类：

1. 直接方法：直接方法的共同特点是它们将原始方程转换
   转化为更容易求解的等价方程，意味着我们可以直接从方程中求解。
2. 迭代法：迭代法或间接法，先对解进行猜测，然后反复进行改进
   该解法直到达到某个收敛准则为止。迭代方法通常效率低于直接方法，因为需要大量运算。例如——雅可比迭代法、高斯-赛德尔迭代法。

C语言中的实现

```
//雅可比方法的实现
void JacobisMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //变量的修改形式
    int rootFound=0; //标志

    int i, j;
    while(!rootFound){
        for(i=0; i<n; i++){                 //计算
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*x[j];
            }
Nx[i] = Nx[i] / a[i][i];
        }

rootFound=1;                        //验证
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){                 //评估
            x[i]=Nx[i];
        }
    }

    return ;
}

//高斯-赛德尔方法的实现
void GaussSeidalMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //变量的修正形式
    int rootFound=0; //标志

    int i, j;
    for(i=0; i<n; i++){                     //初始化
        Nx[i]=x[i];
    }
```


Chapter 46: Equation Solving

Section 46.1: Linear Equation

There are two classes of methods for solving Linear Equations:

1. **Direct Methods**: Common characteristics of direct methods are that they transform the original equation into equivalent equations that can be solved more easily, means we get solve directly from an equation.
2. **Iterative Method**: Iterative or Indirect Methods, start with a guess of the solution and then repeatedly refine the solution until a certain convergence criterion is reached. Iterative methods are generally less efficient than direct methods because large number of operations required. Example- Jacobi's Iteration Method, Gauss-Seidal Iteration Method.

Implementation in C-

```
//Implementation of Jacobi's Method
void JacobisMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    while(!rootFound){
        for(i=0; i<n; i++){                 //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*x[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                        //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){                 //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
}

//Implementation of Gauss-Seidal Method
void GaussSeidalMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    for(i=0; i<n; i++){                     //initialization
        Nx[i]=x[i];
    }
```

```
    while(!rootFound){
        for(i=0; i<n; i++){                //计算
           Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*Nx[j];
            }
Nx[i] = Nx[i] / a[i][i];
        }

rootFound=1;                        //验证
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){                //评估
           x[i]=Nx[i];
        }
    }

    return ;
}

//打印用逗号分隔的数组
void print(int n, double x[n]){
    int i;
    for(i=0; i<n; i++){
        printf("%lf, ", x[i]);
    }
printf("");

    return ;
}

int main(){
    //方程初始化
    int n=3;    //变量个数

        double x[n];    //变量

        double b[n],    //常数
            a[n][n];    //系数

    //赋值
a[0][0]=8; a[0][1]=2; a[0][2]=-2; b[0]=8;    //8x₁+2x₂-2x₃+8=0
    a[1][0]=1; a[1][1]=-8; a[1][2]=3; b[1]=-4;    //x₁-8x₂+3x₃-4=0
    a[2][0]=2; a[2][1]=1; a[2][2]=9; b[2]=12;    //2x₁+x₂+9x₃+12=0


    int i;

    for(i=0; i<n; i++){                    //初始化
       x[i]=0;
    }
JacobisMethod(n, x, b, a);
    print(n, x);


    for(i=0; i<n; i++){                    //初始化
```



```
    while(!rootFound){
        for(i=0; i<n; i++){                //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*Nx[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                    //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){                //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
}

//Print array with comma separation
void print(int n, double x[n]){
    int i;
    for(i=0; i<n; i++){
        printf("%lf, ", x[i]);
    }
    printf("\n\n");

    return ;
}

int main(){
    //equation initialization
    int n=3;    //number of variables

    double x[n];    //variables

    double b[n],    //constants
        a[n][n];    //arguments

    //assign values
    a[0][0]=8; a[0][1]=2; a[0][2]=-2; b[0]=8;    //8x₁+2x₂-2x₃+8=0
    a[1][0]=1; a[1][1]=-8; a[1][2]=3; b[1]=-4;    //x₁-8x₂+3x₃-4=0
    a[2][0]=2; a[2][1]=1; a[2][2]=9; b[2]=12;    //2x₁+x₂+9x₃+12=0


    int i;

    for(i=0; i<n; i++){                    //initialization
        x[i]=0;
    }
    JacobisMethod(n, x, b, a);
    print(n, x);


    for(i=0; i<n; i++){                    //initialization
```

```
x[i]=0;
    }
GaussSeidalMethod(n, x, b, a);
    print(n, x);

    return 0;
}
```

第46.2节：非线性方程

形如 `f(x)=0` 的方程可以是代数方程或超越方程。这类方程可以通过
两种方法来求解-

1. 直接法：该方法在有限步内直接给出所有根的精确值。
2. 间接或迭代法：迭代法最适合用计算机程序来求解方程。
   它基于逐次逼近的概念。在迭代法中，有两种求解方程的方法-

   - **区间法**：我们取两个初始点，根位于这两点之间。例如-
     二分法，假位法。
   - **开放端法**：我们取一个或两个初始值，根可能位于任何位置。例如-
     牛顿-拉夫森法、逐次逼近法、割线法。

C语言实现：

```
/// 这里定义不同的函数以进行操作
#define f(x) ( ((x)*(x)*(x)) - (x) - 2 )
#define f2(x) ( (3*(x)*(x)) - 1 )
#define g(x) ( cbrt( (x) + 2 ) )


/**
* 取两个初始值并从两边缩小距离。
**/
double 二分法(){
    double 根=0;

    double a=1, b=2;
    double c=0;

    int 循环计数器=0;
    if(f(a)*f(b) < 0){
        while(1){
循环计数器++;
            c=(a+b)/2;

              if(f(c)<0.00001 && f(c)>-0.00001){
                根=c;
                break;
              }

              if((f(a))*(f(c)) < 0){
                  b=c;
              }else{
a=c;
              }
```



```
        x[i]=0;
    }
    GaussSeidalMethod(n, x, b, a);
    print(n, x);

    return 0;
}
```

Section 46.2: Non-Linear Equation

An equation of the type `f(x)=0` is either algebraic or transcendental. These types of equations can be solved by using two types of methods-

1. **Direct Method**: This method gives the exact value of all the roots directly in a finite number of steps.
2. **Indirect or Iterative Method**: Iterative methods are best suited for computer programs to solve an equation. It is based on the concept of successive approximation. In Iterative Method there are two ways to solve an equation-

   - **Bracketing Method**: We take two initial points where the root lies in between them. Example- Bisection Method, False Position Method.
   - **Open End Method**: We take one or two initial values where the root may be any-where. Example- Newton-Raphson Method, Successive Approximation Method, Secant Method.

Implementation in C:

```
/// Here define different functions to work with
#define f(x) ( ((x)*(x)*(x)) - (x) - 2 )
#define f2(x) ( (3*(x)*(x)) - 1 )
#define g(x) ( cbrt( (x) + 2 ) )


/**
* Takes two initial values and shortens the distance by both side.
**/
double BisectionMethod(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;
            c=(a+b)/2;

            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }
```

```
        }
    }
    printf("循环了 %d 次。", loopCounter);return root;


}

/**
* 取两个初始值，并通过单边缩短距离。
**/
double FalsePosition(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int 循环计数器=0;
    if(f(a)*f(b) < 0){
        while(1){
loopCounter++;

            c=(a*f(b) - b*f(a)) / (f(b) - f(a));

            /*/printf("%lf %lf ", c, f(c));/**////测试if(f(c)<0.00001 &&
             f(c)>-0.00001){
root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
a=c;
            }
        }
    }
    printf("循环了 %d 次。", loopCounter);return root;


}

/**
* 使用一个初始值，逐渐使该值接近真实值。
**/
double NewtonRaphson(){
    double root=0;

    double x1=1;
    double x2=0;

    int loopCounter=0;
    while(1){
loopCounter++;

        x2 = x1 - (f(x1)/f2(x1));
        /*/printf("%lf  %lf ", x2, f(x2));/**////testif(f(x2)<0.00001 &

        & f(x2)>-0.00001){
root=x2;
            break;
        }
```



```
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Takes two initial values and shortens the distance by single side.
**/
double FalsePosition(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;

            c=(a*f(b) - b*f(a)) / (f(b) - f(a));

            /*/printf("%lf\t %lf \n", c, f(c));/**////test
            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double NewtonRaphson(){
    double root=0;

    double x1=1;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        x2 = x1 - (f(x1)/f2(x1));
        /*/printf("%lf \t %lf \n", x2, f(x2));/**////test

        if(f(x2)<0.00001 && f(x2)>-0.00001){
            root=x2;
            break;
        }
```

```
x1=x2;
    }
    printf("循环了 %d 次。", loopCounter);return root;


}

/**
* 使用一个初始值，逐渐使该值接近真实值。
**/
double FixedPoint(){
    double root=0;
    double x=1;

    int loopCounter=0;
    while(1){
loopCounter++;

        if( (x-g(x)) <0.00001 && (x-g(x)) >-0.00001){
            root = x;
            break;
        }

        /*/printf("%lf   %lf ", g(x), x-(g(x)));/**////测试

x=g(x);
    }
    printf("循环了 %d 次。", loopCounter);return root;


}

/**
* 使用两个初始值，且两个值都趋近于根。
**/
double 割线法(){
    double 根=0;

    double x0=1;
    double x1=2;
    double x2=0;

    int 循环计数器=0;
    while(1){
loopCounter++;

        /*/printf("%lf   %lf   %lf ", x0, x1, f(x1));/**////测试if(f(x1)<0.00001 && f(x

        1)>-0.00001){
根=x1;
            break;
        }

x2 = ((x0*f(x1))-(x1*f(x0))) / (f(x1)-f(x0));

        x0=x1;
x1=x2;
    }
    printf("循环了 %d 次。", loopCounter);return root;


}
```



```
        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double FixedPoint(){
    double root=0;
    double x=1;

    int loopCounter=0;
    while(1){
        loopCounter++;

        if( (x-g(x)) <0.00001 && (x-g(x)) >-0.00001){
            root = x;
            break;
        }

        /*/printf("%lf \t %lf \n", g(x), x-(g(x)));/**///test

        x=g(x);
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* uses two initial values & both value approaches to the root.
**/
double Secant(){
    double root=0;

    double x0=1;
    double x1=2;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        /*/printf("%lf \t %lf \t %lf \n", x0, x1, f(x1));/**///test

        if(f(x1)<0.00001 && f(x1)>-0.00001){
            root=x1;
            break;
        }

        x2 = ((x0*f(x1))-(x1*f(x0))) / (f(x1)-f(x0));

        x0=x1;
        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}
```

```
int main(){
    double 根;

root = BisectionMethod();
    printf("使用二分法求得的根是: %lf ", root);     root = FalsePosition();


    printf("使用假位法求得的根是: %lf ", root);     root = NewtonRaphson();


    printf("使用牛顿-拉夫森法求得的根是: %lf ", root);     root = FixedPoint();


    printf("使用不动点法求得的根是: %lf ", root);     root = Secant();


    printf("使用割线法求得的根是: %lf ", root);

    return 0;
}
```



```
int main(){
    double root;

    root = BisectionMethod();
    printf("Using Bisection Method the root is: %lf \n\n", root);

    root = FalsePosition();
    printf("Using False Position Method the root is: %lf \n\n", root);

    root = NewtonRaphson();
    printf("Using Newton-Raphson Method the root is: %lf \n\n", root);

    root = FixedPoint();
    printf("Using Fixed Point Method the root is: %lf \n\n", root);

    root = Secant();
    printf("Using Secant Method the root is: %lf \n\n", root);

    return 0;
}
```

第47章：最长公共子序列

第47.1节：最长公共子序列说明

动态规划最重要的应用之一是求解最长公共子序列。首先让我们定义一些基本术语。

子序列：

子序列是通过删除另一个序列中的某些元素而不改变剩余元素顺序得到的序列。假设我们有一个字符串ABC。如果我们从该字符串中删除零个、一个或多个字符，就得到该字符串的子序列。因此，字符串ABC的子序列包括{"A", "B", "C", "AB", "AC", "BC", "ABC", " "}。即使我们删除所有字符，空字符串也算作子序列。要找出子序列，对于字符串中的每个字符，我们有两个选择——要么取该字符，要么不取。因此，如果字符串长度为n，则该字符串共有2^n个子序列。

最长公共子序列：

顾名思义，在两个字符串的所有公共子序列中，最长公共子序列（LCS）是长度最大的那个。例如：字符串"HELLOM"和"HMLD"之间的公共子序列有"H"、"HL"、"HM"等。其中"HLL"是最长公共子序列，长度为3。

暴力法：

我们可以使用回溯生成两个字符串的所有子序列，然后比较它们找出公共子序列。之后需要找出长度最大的那个。我们已经知道，长度为n的字符串有2^n个子序列。如果n超过20-25，解决这个问题将需要数年时间。

动态规划法：

让我们通过一个例子来说明方法。假设有两个字符串abcdaf和acbcf。我们用 s1和 s2表示它们。这两个字符串的最长公共子序列是"abcf"，长度为4。
再次提醒，子序列不需要在字符串中连续。构造"abcf"时，我们忽略了 s1中的"da"和 s2中的"c"。如何用动态规划找出这个结果？

我们将从一个表格（二维数组）开始，表格中 s1的所有字符放在一行， s2的所有字符放在一列。
这里表格是从0开始索引，字符从1开始放置。我们将从左到右遍历每一行。表格如下所示：

	0		1		2		3		4		5		6
	字符ʳ				a		b		c		d		a		f
0															
1	a														
2	c														
3	b														
4	c														

Chapter 47: Longest Common Subsequence

Section 47.1: Longest Common Subsequence Explanation

One of the most important implementations of Dynamic Programming is finding out the Longest Common Subsequence. Let's define some of the basic terminologies first.

Subsequence:

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Let's say we have a string **ABC**. If we erase zero or one or more than one character from this string we get the subsequence of this string. So the subsequences of string **ABC** will be {**"A"**, **"B"**, **"C"**, **"AB"**, **"AC"**, **"BC"**, **"ABC"**, **" "**}. Even if we remove all the characters, the empty string will also be a subsequence. To find out the subsequence, for each characters in a string, we have two options - either we take the character, or we don't. So if the length of the string is **n**, there are **2n** subsequences of that string.

Longest Common Subsequence:

As the name suggest, of all the common subsequencesbetween two strings, the longest common subsequence(LCS) is the one with the maximum length. For example: The common subsequences between **"HELLOM"** and **"HMLD"** are **"H"**, **"HL"**, **"HM"** etc. Here **"HLL"** is the longest common subsequence which has length 3.

Brute-Force Method:

We can generate all the subsequences of two strings using *backtracking*. Then we can compare them to find out the common subsequences. After we'll need to find out the one with the maximum length. We have already seen that, there are **2n** subsequences of a string of length **n**. It would take years to solve the problem if our **n** crosses **20-25**.

Dynamic Programming Method:

Let's approach our method with an example. Assume that, we have two strings **abcdaf** and **acbcf**. Let's denote these with **s1** and **s2**. So the longest common subsequence of these two strings will be **"abcf"**, which has length 4. Again I remind you, subsequences need not be continuous in the string. To construct **"abcf"**, we ignored **"da"** in **s1** and **"c"** in **s2**. How do we find this out using Dynamic Programming?

We'll start with a table (a 2D array) having all the characters of **s1** in a row and all the characters of **s2** in column. Here the table is 0-indexed and we put the characters from 1 to onwards. We'll traverse the table from left to right for each row. Our table will look like:

	0		1		2		3		4		5		6
	chʳ				a		b		c		d		a		f
0															
1	a														
2	c														
3	b														
4	c														

```
5  |  f  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

这里每一行和每一列表示两个字符串之间最长公共子序列的长度，前提是我们取该行和该列的字符并添加到其之前的前缀中。例如：Table[2][3]表示"ac"和"abc"之间最长公共子序列的长度。

第0列表示字符串 s1的空子序列。同样，第0行表示字符串 s2的空子序列。如果我们取一个字符串的空子序列并尝试与另一个字符串匹配，无论第二个子串的长度多长，公共子序列的长度都将是0。因此我们可以用0填充第0行和第0列。得到：

```
0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| 字符r |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

我们开始。当我们填写Table[1][1]时，我们在问自己，如果有一个字符串 a和另一个字符串 a且没有其他字符，这里的最长公共子序列会是多少？这里LCS的长度是1。现在我们看Table[1][2]。我们有字符串 ab和字符串 a。LCS的长度将是1。如你所见，第一行的其余值也将是1，因为它只考虑字符串 a与 abcd、 abcda、 abcdaf的比较。所以我们的表格将是：

```
0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| 字符r |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

对于第2行，现在包含 c。对于Table[2][1]，我们有ac在一边，另一边是 a。所以LCS的长度是1。这个1是从哪里来的？来自上方，表示两个子串之间的LCS a。因此，我们的意思是，如果 s1[2]和 s2[1]不相同，那么LCS的长度将是


```
5  |  f  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Here each row and column represent the length of the longest common subsequence between two strings if we take the characters of that row and column and add to the prefix before it. For example: **Table[2][3]** represents the length of the longest common subsequence between **"ac"** and **"abc"**.

The 0-th column represents the empty subsequence of **s1**. Similarly the 0-th row represents the empty subsequence of **s2**. If we take an empty subsequence of a string and try to match it with another string, no matter how long the length of the second substring is, the common subsequence will have 0 length. So we can fill-up the 0-th rows and 0-th columns with 0's. We get:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Let's begin. When we're filling **Table[1][1]**, we're asking ourselves, if we had a string **a** and another string **a** and nothing else, what will be the longest common subsequence here? The length of the LCS here will be 1. Now let's look at **Table[1][2]**. We have string **ab** and string **a**. The length of the LCS will be 1. As you can see, the rest of the values will be also 1 for the first row as it considers only string **a** with **abcd**, **abcda**, **abcdaf**. So our table will look like:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

For row 2, which will now include **c**. For **Table[2][1]** we have **ac** on one side and **a** on the other side. So the length of the LCS is 1. Where did we get this 1 from? From the top, which denotes the LCS **a** between two substrings. So what we are saying is, if **s1[2]** and **s2[1]** are not same, then the length of the LCS will be the maximum of the length of

顶部或左侧的LCS长度的最大值。取顶部的LCS长度表示我们不取当前 s2的字符。同样，取左侧的LCS长度表示我们不取当前 s1的字符来构造LCS。我们得到：

	0	1	2	3	4	5	6
	字符r		a	b	c	d	a	f
0		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	c	0	1					
3	b	0						
4	c	0						
5	f	0						

所以我们的第一个公式是：

```
如果 s2[i] 不等于 s1[j]
   表格[i][j] = max(表格[i-1][j], 表格[i][j-1]
endif
```

接下来，对于 表格[2][2] 我们有字符串 ab 和 ac。由于 c 和 b 不相同，我们在这里取上方或左方的最大值。在这种情况下，仍然是 1。之后，对于 表格[2][3] 我们有字符串 abc 和 ac。这次当前行和列的值相同。现在最长公共子序列（LCS）的长度将等于目前为止最长LCS的最大长度加1。
我们如何得到目前为止最长LCS的最大长度？我们检查对角线的值，它代表了 ab 和 a 之间的最佳匹配。从这个状态出发，对于当前的值，我们在 s1 和 s2 中又添加了一个相同的字符。所以LCS的长度当然会增加。我们将在 表格[2][3] 中放入 1 + 1 = 2。结果是，

	0	1	2	3	4	5	6
	字符r		a	b	c	d	a	f
0		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	c	0	1	1	2			
3	b	0						
4	c	0						
5	f	0						

所以我们的第二个公式是：

```
如果 s2[i] 等于 s1[j]
   表格[i][j] = 表格[i-1][j-1] + 1
endif
```


LCS at the **top**, or at the **left**. Taking the length of the LCS at the top denotes that, we don't take the current character from **s2**. Similarly, Taking the length of the LCS at the left denotes that, we don't take the current character from **s1** to create the LCS. We get:

	0	1	2	3	4	5	6
	chr		a	b	c	d	a	f
0		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	c	0	1					
3	b	0						
4	c	0						
5	f	0						

So our first formula will be:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1]
endif
```

Moving on, for **Table[2][2]** we have string **ab** and **ac**. Since **c** and **b** are not same, we put the maximum of the top or left here. In this case, it's again 1. After that, for **Table[2][3]** we have string **abc** and **ac**. This time current values of both row and column are same. Now the length of the LCS will be equal to the maximum length of LCS so far + 1. How do we get the maximum length of LCS so far? We check the diagonal value, which represents the best match between **ab** and **a**. From this state, for the current values, we added one more character to **s1** and **s2** which happened to be the same. So the length of LCS will of course increase. We'll put **1 + 1 = 2** in **Table[2][3]**. We get,

	0	1	2	3	4	5	6
	chr		a	b	c	d	a	f
0		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	c	0	1	1	2			
3	b	0						
4	c	0						
5	f	0						

So our second formula will be:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

我们已经定义了这两种情况。使用这两个公式，我们可以填充整个表格。填充完表格后，它将如下所示：

```
             0     1     2     3     4     5     6

| chr |     | a  | b  | c  | d  | a  | f  |

0   |     |  0 |  0 |  0 |  0 |  0 |  0 |  0 |

1   |  a  |  0 |  1 |  1 |  1 |  1 |  1 |  1 |

2   |  c  |  0 |  1 |  1 |  2 |  2 |  2 |  2 |

3   |  b  |  0 |  1 |  2 |  2 |  2 |  2 |  2 |

4   |  c  |  0 |  1 |  2 |  3 |  3 |  3 |  3 |

5   |  f  |  0 |  1 |  2 |  3 |  3 |  3 |  4 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

字符串 s1 和 s2 之间最长公共子序列（LCS）的长度为 Table[5][6] = 4。这里，5 和 6 分别是 s2 和 s1 的长度。我们的伪代码如下：

```
过程 LCSlength(s1, s2):
Table[0][0] = 0
for i 从 1 到 s1.length
    Table[0][i] = 0
endfor
for i 从 1 到 s2.length
    Table[i][0] = 0
结束循环
for i 从 1 到 s2.length
    for j 从 1 到 s1.length
        if s2[i] 等于 s1[j]
          Table[i][j] = Table[i-1][j-1] + 1
        否则
Table[i][j] = max(Table[i-1][j], Table[i][j-1])
       endif

endfor
返回 Table[s2.length][s1.length]
```

该算法的时间复杂度为：O(mn)，其中 m 和 n 分别表示两个字符串的长度。

我们如何找到最长公共子序列？我们将从右下角开始。我们会检查
该值是从哪里来的。如果该值来自对角线，也就是说**Table[i-1][j-1]**等于
Table[i][j] - 1，我们将推入 **s2[i]**或 **s1[j]**（两者相同）并沿对角线移动。如果该值来自上方，
也就是说，如果**Table[i-1][j]**等于**Table[i][j]**，我们就向上移动。如果该值来自左边，也就是说，如果
Table[i][j-1]等于**Table[i][j]**，我们就向左移动。当我们到达最左边或最上边的列时，搜索
结束。然后我们从栈中弹出值并打印。伪代码如下：

```
过程 PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
当 i 不等于 0 且 j 不等于 0
     如果 Table[i-1][j-1] == Table[i][j] - 1 且 s1[j]==s2[i]
```


We have defined both the cases. Using these two formulas, we can populate the whole table. After filling up the table, it will look like this:

```
             0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
+-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The length of the LCS between **s1** and **s2** will be **Table[5][6] = 4**. Here, 5 and 6 are the length of **s2** and **s1** respectively. Our pseudo-code will be:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

The time complexity for this algorithm is: **O(mn)** where **m** and **n** denotes the length of each strings.

How do we find out the longest common subsequence? We'll start from the bottom-right corner. We will check from where the value is coming. If the value is coming from the diagonal, that is if **Table[i-1][j-1]** is equal to **Table[i][j] - 1**, we push either **s2[i]** or **s1[j]** (both are the same) and move diagonally. If the value is coming from top, that means, if **Table[i-1][j]** is equal to **Table[i][j]**, we move to the top. If the value is coming from left, that means, if **Table[i][j-1]** is equal to **Table[i][j]**, we move to the left. When we reach the leftmost or topmost column, our search ends. Then we pop the values from the stack and print them. The pseudo-code:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
     if Table[i-1][j-1] == Table[i][j] - 1 and s1[j]==s2[i]
```

```
S.push(s1[j])    //或 S.push(s2[i])
        i := i - 1
j := j - 1
     否则如果 Table[i-1][j] == Table[i][j]
        i := i-1
     否则
j := j-1
    结束条件
endwhile
while S 不为空
    print(S.pop)
endwhile
```

注意事项：如果 Table[i-1][j] 和 Table[i][j-1] 都等于 Table[i][j]，且 Table[i-1][j-1] 不等于Table[i][j] - 1，则此时可能存在两个最长公共子序列（LCS）。该伪代码未考虑这种情况。你需要递归解决此问题以找到多个LCS。

该算法的时间复杂度为：O(max(m, n))。


```
        S.push(s1[j])   //or S.push(s2[i])
        i := i - 1
        j := j - 1
    else if Table[i-1][j] == Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
while S is not empty
    print(S.pop)
endwhile
```

Point to be noted: if both **Table[i-1][j]** and **Table[i][j-1]** is equal to **Table[i][j]** and **Table[i-1][j-1]** is not equal to **Table[i][j] - 1**, there can be two LCS for that moment. This pseudo-code doesn't consider this situation. You'll have to solve this recursively to find multiple LCSs.

The time complexity for this algorithm is: **O(max(m, n))**.

第48章：最长递增子序列

第48.1节：最长递增子序列基础信息

最长递增子序列问题是从给定的输入序列中找到一个子序列，使得该子序列的元素按从小到大的顺序排列。所有子序列不要求连续或唯一。

最长递增子序列的应用：

像最长递增子序列、最长公共子序列这样的算法被用于版本控制系统，如Git等。

算法的简单形式：

1. 找出两个文档中共有的唯一行。
2. 从第一个文档中取出所有这样的行，并根据它们在第二个文档中出现的顺序进行排序。

3. 计算所得序列的最长递增子序列（通过耐心排序），得到最长匹配序列的行，即两个文档行之间的对应关系。
4. 对已匹配行之间的每个区间递归执行该算法。

现在让我们考虑一个更简单的最长公共子序列（LCS）问题的例子。这里，输入仅为一组不同的整数序列 a1,a2,...,an。，我们想要找到其中的最长递增子序列。例如，如果输入为**7,3,8,4,2,6** 那么最长递增子序列是**3,4,6**。

最简单的方法是将输入元素按递增顺序排序，并将最长公共子序列算法应用于原始序列和排序序列。然而，如果你观察结果数组，会发现许多值是相同的，数组看起来非常重复。这表明最长递增子序列（LIS）问题可以用仅使用一维数组的动态规划算法来解决。

伪代码：

1. 描述我们想要计算的值的数组。
   对于1 <= i <= n，令A(i)为输入中最长递增序列的长度。注意，我们最终感兴趣的长度是max{A(i)|1 ≤ i ≤ n}。
2. 给出一个递推关系。
   对于 1 <= i <= n，A(i) = 1 + max{A(j)|1 ≤ j < i 且 input(j) < input(i)}。
3. 计算 A 的值。
4. 找到最优解。

以下程序使用 A 来计算最优解。第一部分计算一个值 m，使得 A(m) 是输入的最优递增子序列的长度。第二部分计算一个最优递增子序列，但为了方便，我们以逆序打印它。该程序的运行时间为 O(n)，因此整个算法的运行时间为 O(n^2)。

第一部分：

```
m ← 1
对于 i : 2..n
    如果 A(i) > A(m) 则
```


Chapter 48: Longest Increasing Subsequence

Section 48.1: Longest Increasing Subsequence Basic Information

The Longest Increasing Subsequence problem is to find subsequence from the give input sequence in which subsequence's elements are sorted in lowest to highest order. All subsequence are not contiguous or unique.

Application of Longest Increasing Subsequence:

Algorithms like Longest Increasing Subsequence, Longest Common Subsequence are used in version control systems like Git and etc.

Simple form of Algorithm:

1. Find unique lines which are common to both documents.
2. Take all such lines from the first document and order them according to their appearance in the second document.
3. Compute the LIS of the resulting sequence (by doing a Patience Sort), getting the longest matching sequence of lines, a correspondence between the lines of two documents.
4. Recurse the algorithm on each range of lines between already matched ones.

Now let us consider a simpler example of the LCS problem. Here, input is only one sequence of distinct integers $a1, a2, \ldots, an$., and we want to find the longest increasing subsequence in it. For example, if input is **7,3,8,4,2,6** then the longest increasing subsequence is **3,4,6**.

The easiest approach is to sort input elements in increasing order, and apply the LCS algorithm to the original and sorted sequences. However, if you look at the resulting array you would notice that many values are the same, and the array looks very repetitive. This suggest that the LIS (longest increasing subsequence) problem can be done with dynamic programming algorithm using only one-dimensional array.

Pseudo Code:

1. Describe an array of values we want to compute.
   For $1 <= i <= n$, let **A(i)** be the length of a longest increasing sequence of input. Note that the length we are ultimately interested in is $\max\{A(i)|1 \leq i \leq n\}$.
2. Give a recurrence.
   For $1 <= i <= n$, $A(i) = 1 + \max\{A(j)|1 \leq j < i$ and $input(j) < input(i)\}$.
3. Compute the values of A.
4. Find the optimal solution.

The following program uses A to compute an optimal solution. The first part computes a value m such that **A(m)** is the length of an optimal increasing subsequence of input. The second part computes an optimal increasing subsequence, but for convenience we print it out in reverse order. This program runs in time O(n), so the entire algorithm runs in time O(n^2).

Part 1:

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
```

```
m ← i
    结束 如果
结束 循环
```

第二部分：

```
放置 a
当 A(m) > 1 时执行
    i ← m−1
     当 不(ai < am 且 A(i) = A(m)−1) 时执行
        i ← i−1
结束 循环
m ← i
放置 a
结束 循环
```

递归解法：

方法一：

```
LIS(A[1..n]):
     如果(n = 0)则返回0
m = LIS(A[1..(n − 1)])
B 是 A[1..(n − 1)] 的子序列，且仅包含小于 a[n]的元素(* 设 h 为 B 的大小，h ≤ n-1 *)

m = max(m, 1 + LIS(B[1..h]))
    输出 m
```

方法一的时间复杂度：$O(n*2^n)$

方法二：

```
LIS(A[1..n], x):
     如果 (n = 0) 则 返回 0
    m = LIS(A[1..(n − 1)], x)
     如果 (A[n] < x) 则
m = max(m, 1 + LIS(A[1..(n − 1)], A[n]))
    输出 m

MAIN(A[1..n]):
     返回 LIS(A[1..n], ∞)
```

方法二的时间复杂度：$O(n^2)$

方法三：

```
LIS(A[1..n]):
     if (n = 0) return 0
m = 1
     for i = 1 to n − 1 do
         if (A[i] < A[n]) then
             m = max(m, 1 + LIS(A[1..i]))
     return m

MAIN(A[1..n]):
     return LIS(A[1..i])
```



```
        m ← i
    end if
end for
```

Part 2:

```
put a
while A(m) > 1 do
    i ← m−1
    while not(ai < am and A(i) = A(m)−1) do
        i ← i−1
    end while
    m ← i
    put a
 end while
```

Recursive Solution:

Approach 1:

```
LIS(A[1..n]):
    if (n = 0) then return 0
    m = LIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with only elements less than a[n]
    (* let h be size of B, h ≤ n-1 *)
    m = max(m, 1 + LIS(B[1..h]))
    Output m
```

Time complexity in Approach 1 : $O(n*2^n)$

Approach 2:

```
LIS(A[1..n], x):
    if (n = 0) then return 0
    m = LIS(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS(A[1..(n − 1)], A[n]))
    Output m

MAIN(A[1..n]):
    return LIS(A[1..n], ∞)
```

Time Complexity in Approach 2: $O(n^2)$

Approach 3:

```
LIS(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS(A[1..i]))
    return m

MAIN(A[1..n]):
    return LIS(A[1..i])
```

方法3的时间复杂度：O(n^2)

迭代算法：

以自底向上的方式迭代计算值。

```
LIS(A[1..n]):
    数组 L[1..n]
    (* L[i] = 以(A[1..i]) 结尾的LIS的值 *)
    for i = 1 to n do
L[i] = 1
        for j = 1 到 i − 1 执行
            if (A[j] < A[i]) do
L[i] = max(L[i], 1 + L[j])
    return L

MAIN(A[1..n]):
L = LIS(A[1..n])
        return L 中的最大值
```

迭代方法的时间复杂度：O(n^2)

辅助空间：O(n)

以 {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15} 作为输入。给定输入的最长递增子序列是 {0, 2, 6, 9, 11, 15}。

Time Complexity in Approach 3: O(n^2)

Iterative Algorithm:

Computes the values iteratively in bottom up fashion.

```
LIS(A[1..n]):
    Array L[1..n]
    (* L[i] = value of LIS ending(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i - 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L

MAIN(A[1..n]):
    L = LIS(A[1..n])
        return the maximum value in L
```

Time complexity in Iterative approach: O(n^2)

Auxiliary Space: O(n)

Lets take **{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}** as input. So, Longest Increasing Subsequence for the given input is **{0, 2, 6, 9, 11, 15}**.

第49章：检查两个字符串是否为变位词

两个具有相同字符集合的字符串称为变位词。我这里使用了JavaScript。

我们将创建 str1 的哈希表并将计数加1。然后遍历第二个字符串，检查所有字符是否都在哈希表中，并将哈希键的值减1。检查所有哈希键的值是否为零，若是则为变位词。

第49.1节：示例输入和输出

示例1：

```
让 str1 = 'stackoverflow';
让 str2 = 'flowerovstack';
```

这些字符串是变位词（字母异位词）。

// 从 str1 创建哈希表并增加计数。

```
hashMap = {
    s : 1,
    t : 1,
    a : 1,
    c : 1,
    k : 1,
    o : 2,
    v : 1,
    e : 1,
    r : 1,
    f : 1,
    l : 1,
    w : 1
}
```

你可以看到哈希键 'o' 的值是 2，因为字符串中 'o' 出现了两次。

现在遍历 str2，检查每个字符是否存在于 hashMap 中，如果存在，则减少 hashMap 键的值，否则返回 false（表示它不是变位词）。

```
hashMap = {
    s : 0,
    t : 0,
    a : 0,
    c : 0,
    k : 0,
    o : 0,
    v : 0,
    e : 0,
    r : 0,
    f : 0,
    l : 0,
    w : 0
}
```

现在，遍历 hashMap 对象，检查所有键的值是否都为零。

Chapter 49: Check two strings are anagrams

Two string with same set of character is called anagram. I have used javascript here.

We will create an hash of str1 and increase count +1. We will loop on 2nd string and check all characters are there in hash and decrease value of hash key. Check all value of hash key are zero will be anagram.

Section 49.1: Sample input and output

Ex1:

```
let str1 = 'stackoverflow';
let str2 = 'flowerovstack';
```

These strings are anagrams.

// Create Hash from str1 and increase one count.

```
hashMap = {
    s : 1,
    t : 1,
    a : 1,
    c : 1,
    k : 1,
    o : 2,
    v : 1,
    e : 1,
    r : 1,
    f : 1,
    l : 1,
    w : 1
}
```

You can see hashKey 'o' is containing value 2 because o is 2 times in string.

Now loop over str2 and check for each character are present in hashMap, if yes, decrease value of hashMap Key, else return false (which indicate it's not anagram).

```
hashMap = {
    s : 0,
    t : 0,
    a : 0,
    c : 0,
    k : 0,
    o : 0,
    v : 0,
    e : 0,
    r : 0,
    f : 0,
    l : 0,
    w : 0
}
```

Now, loop over hashMap object and check all values are zero in the key of hashMap.

在我们的例子中，所有值都是零，所以它是一个变位词。

第49.2节：通用变位词代码

```
(function(){

var hashMap = {};

    function isAnagram (str1, str2) {

        if(str1.length !== str2.length){
            return false;
        }

        // 创建 str1 字符的哈希映射并将值加一 (+1)。
createStr1HashMap(str1);

        // 检查 str2 字符是否为哈希映射中的键，并将值减一 (-1);
var valueExist = createStr2HashMap(str2);

        // 检查哈希映射所有键的值是否为零，如果是，则为变位词。
        return isStringsAnagram(valueExist);
    }

function createStr1HashMap (str1) {
        [].map.call(str1, function(value, index, array){
            hashMap[value] = value in hashMap ?  (hashMap[value] + 1) : 1;
            return value;
        });
    }

function createStr2HashMap (str2) {
        var valueExist = [].every.call(str2, function(value, index, array){
            if(value in hashMap) {
hashMap[value] = hashMap[value] - 1;
            }
            return value in hashMap;
        });
        return valueExist;
    }

function isStringsAnagram (valueExist) {
        if(!valueExist) {
            return valueExist;
        } else {
var isAnagram;
            for(var i in hashMap) {
                if(hashMap[i] !== 0) {
                    isAnagram = false;
                    break;
                } else {
isAnagram = true;
                }
            }

            return isAnagram;
        }
    }

isAnagram('stackoverflow', 'flowerovstack'); // true
    isAnagram('stackoverflow', 'flowervvstack'); // false
```


In our case all values are zero so its a anagram.

Section 49.2: Generic Code for Anagrams

```
(function(){

    var hashMap = {};

    function isAnagram (str1, str2) {

        if(str1.length !== str2.length){
            return false;
        }

        // Create hash map of str1 character and increase value one (+1).
        createStr1HashMap(str1);

        // Check str2 character are key in hash map and decrease value by one(-1);
        var valueExist = createStr2HashMap(str2);

        // Check all value of hashMap keys are zero, so it will be anagram.
        return isStringsAnagram(valueExist);
    }

    function createStr1HashMap (str1) {
        [].map.call(str1, function(value, index, array){
            hashMap[value] = value in hashMap ?  (hashMap[value] + 1) : 1;
            return value;
        });
    }

    function createStr2HashMap (str2) {
        var valueExist = [].every.call(str2, function(value, index, array){
            if(value in hashMap) {
                hashMap[value] = hashMap[value] - 1;
            }
            return value in hashMap;
        });
        return valueExist;
    }

    function isStringsAnagram (valueExist) {
        if(!valueExist) {
            return valueExist;
        } else {
            var isAnagram;
            for(var i in hashMap) {
                if(hashMap[i] !== 0) {
                    isAnagram = false;
                    break;
                } else {
                    isAnagram = true;
                }
            }

            return isAnagram;
        }
    }

    isAnagram('stackoverflow', 'flowerovstack'); // true
    isAnagram('stackoverflow', 'flowervvstack'); // false
```

```
})();
```

时间复杂度：3n 即 O(n)。


```
})();
```

Time complexity: 3n i.e O(n).

第50章：帕斯卡三角形

第50.1节：C语言中的帕斯卡三角形

```
int i, space, rows, k=0, count = 0, count1 = 0;
row=5;
for(i=1; i<=rows; ++i)
{
    for(space=1; space <= rows-i; ++space)
    {
printf("  ");
        ++count;
    }

    while(k != 2*i-1)
    {
        if (count <= rows-1)
        {
printf("%d ", i+k);
            ++count;
        }
        否则
        {
            ++count1;
printf("%d ", (i+k-2*count1));
        }
        ++k;
    }
count1 = count = k = 0;     prin

tf("");
}
```

输出

```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```


Chapter 50: Pascal's Triangle

Section 50.1: Pascal triangle in C

```
int i, space, rows, k=0, count = 0, count1 = 0;
row=5;
for(i=1; i<=rows; ++i)
{
    for(space=1; space <= rows-i; ++space)
    {
        printf("  ");
        ++count;
    }

    while(k != 2*i-1)
    {
        if (count <= rows-1)
        {
            printf("%d ", i+k);
            ++count;
        }
        else
        {
            ++count1;
            printf("%d ", (i+k-2*count1));
        }
        ++k;
    }
    count1 = count = k = 0;

    printf("\n");
}
```

Output

```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

第51章：算法：按方块顺序打印一个m*n矩阵

请查看下面的示例输入和输出。

第51.1节：示例

```
输入：

14 15 16 17 18 21
19 10 20 11 54 36
64 55 44 23 80 39
91 92 93 94 95 42

输出：
打印索引处的值
14 15 16 17 18 21 36 39 42 95 94 93 92 91 64 19 10 20 11 54 80 23 44 55

或打印索引
00 01 02 03 04 05 15 25 35 34 33 32 31 30 20 10 11 12 13 14 24 23 22 21
```

第51.2节：编写通用代码

```
function noOfLooping(m,n) {
     if(m > n) {
smallestValue = n;
     } else {
smallestValue = m;
    }

    if(smallestValue % 2 == 0) {
            return smallestValue/2;
    } else {
            return (smallestValue+1)/2;
    }
}

function squarePrint(m,n) {
    var looping = noOfLooping(m,n);
    for(var i = 0; i < looping; i++) {
        for(var j = i; j < m - 1 - i; j++) {
                console.log(i+''+j);
        }
        for(var k = i; k < n - 1 - i; k++) {
                console.log(k+''+j);
        }
        for(var l = j; l > i; l--) {
                console.log(k+''+l);
        }
        for(var x = k; x > i; x--) {
                console.log(x+''+l);
        }
    }
}

squarePrint(6,4);
```


Chapter 51: Algo:- Print a m*n matrix in square wise

Check sample input and output below.

Section 51.1: Sample Example

```
Input:

14 15 16 17 18 21
19 10 20 11 54 36
64 55 44 23 80 39
91 92 93 94 95 42

Output:
print value in index
14 15 16 17 18 21 36 39 42 95 94 93 92 91 64 19 10 20 11 54 80 23 44 55

or print index
00 01 02 03 04 05 15 25 35 34 33 32 31 30 20 10 11 12 13 14 24 23 22 21
```

Section 51.2: Write the generic code

```
function noOfLooping(m,n) {
    if(m > n) {
        smallestValue = n;
    } else {
        smallestValue = m;
    }

    if(smallestValue % 2 == 0) {
            return smallestValue/2;
    } else {
            return (smallestValue+1)/2;
    }
}

function squarePrint(m,n) {
    var looping = noOfLooping(m,n);
    for(var i = 0; i < looping; i++) {
        for(var j = i; j < m - 1 - i; j++) {
                console.log(i+''+j);
        }
        for(var k = i; k < n - 1 - i; k++) {
                console.log(k+''+j);
        }
        for(var l = j; l > i; l--) {
                console.log(k+''+l);
        }
        for(var x = k; x > i; x--) {
                console.log(x+''+l);
        }
    }
}

squarePrint(6,4);
```

第52章：矩阵快速幂

第52.1节：利用矩阵快速幂解决示例问题

求f(n)：第n个斐波那契数。当n相对较小时，问题相当简单。我们可以使用简单的递归，
f(n) = f(n-1) + f(n-2)，或者我们可以使用动态规划方法来避免重复计算相同的函数。但如果题目说，给定0 < n < 10^9，求f(n) mod 999983？动态规划将失效，那么我们该如何解决这个问题？

首先让我们看看矩阵快速幂如何帮助表示递推关系。

前提条件：

- 已知两个矩阵，知道如何求它们的乘积。进一步地，已知两个矩阵的乘积矩阵和其中一个矩阵，知道如何求另一个矩阵。
- 给定一个大小为 d X d 的矩阵，知道如何在 O(d3log(n)) 时间内求其第 n 次幂。

模式：

首先我们需要一个递推关系，并且我们想找到一个矩阵 M，能够从一组已知状态导出所需的状态。假设我们已知给定递推关系的 k 个状态，想要找到第 (k+1)th 个状态。令 M 为一个 k X k 矩阵，并且我们从已知的递推关系状态构建一个矩阵 A:[k X 1]，现在我们想得到一个矩阵 B:[k X 1]，表示下一组状态，即 M X A = B，如下所示：

```
      |  f(n)  |     | f(n+1) |
      | f(n-1) |     |  f(n)  |
  M X | f(n-2) |  =  | f(n-1) |
      | ...... |     | ...... |
      | f(n-k) |     |f(n-k+1)|
```

所以，如果我们能相应地设计 M，任务就完成了！该矩阵将用于表示递推关系。

类型 1：
让我们从最简单的开始，`f(n) = f(n-1) + f(n-2)`
我们得到，`f(n+1) = f(n) + f(n-1)`。
假设我们已知 f(n) 和 f(n-1)；我们想求 f(n+1)。
根据上述情况，可以构造矩阵 A 和矩阵 B，如下所示：

```
矩阵 A          矩阵 B

|  f(n)  |        | f(n+1) |
| f(n-1) |        |  f(n)  |
```

[注：矩阵A将始终设计成，使得f(n+1)所依赖的每个状态都包含在内]
现在，我们需要设计一个**2X2**矩阵**M**，使其满足上述的**M X A = B**。
矩阵**B**的第一个元素是`f(n+1)`，实际上是`f(n) + f(n-1)`。为了得到这个，从矩阵**A**中，我们需要**1 X f(n)**和**1 X f(n-1)**。所以矩阵**M**的第一行是**[1 1]**。

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| ----- |     | f(n-1) |     | ------ |
```


Chapter 52: Matrix Exponentiation

Section 52.1: Matrix Exponentiation to Solve Example Problems

Find f(n): nth Fibonacci number. The problem is quite easy when **n** is relatively small. We can use simple recursion, `f(n) = f(n-1) + f(n-2)`, or we can use dynamic programming approach to avoid the calculation of same function over and over again. But what will you do if the problem says, **Given 0 < n < 10^9, find f(n) mod 999983?** Dynamic programming will fail, so how do we tackle this problem?

First let's see how matrix exponentiation can help to represent recursive relation.

Prerequisites:

- Given two matrices, know how to find their product. Further, given the product matrix of two matrices, and one of them, know how to find the other matrix.
- Given a matrix of size **d X d**, know how to find its nth power in **O(d3log(n))**.

Patterns:

At first we need a recursive relation and we want to find a matrix **M** which can lead us to the desired state from a set of already known states. Let's assume that, we know the **k** states of a given recurrence relation and we want to find the **(k+1)th** state. Let **M** be a **k X k** matrix, and we build a matrix **A:[k X 1]** from the known states of the recurrence relation, now we want to get a matrix **B:[k X 1]** which will represent the set of next states, i. e. **M X A = B** as shown below:

```
      |  f(n)  |     | f(n+1) |
      | f(n-1) |     |  f(n)  |
  M X | f(n-2) |  =  | f(n-1) |
      | ...... |     | ...... |
      | f(n-k) |     |f(n-k+1)|
```

So, if we can design **M** accordingly, our job will be done! The matrix will then be used to represent the recurrence relation.

Type 1:
Let's start with the simplest one, `f(n) = f(n-1) + f(n-2)`
We get, `f(n+1) = f(n) + f(n-1)`.
Let's assume, we know `f(n)` and `f(n-1)`; We want to find out `f(n+1)`.
From the situation stated above, matrix **A** and matrix **B** can be formed as shown below:

```
 Matrix A          Matrix B

|  f(n)  |        | f(n+1) |
| f(n-1) |        |  f(n)  |
```

[Note: Matrix **A** will be always designed in such a way that, every state on which `f(n+1)` depends, will be present]
Now, we need to design a **2X2** matrix **M** such that, it satisfies **M X A = B** as stated above.
The first element of **B** is `f(n+1)` which is actually `f(n) + f(n-1)`. To get this, from matrix **A**, we need, **1 X f(n)** and **1 X f(n-1)**. So the first row of **M** will be **[1 1]**.

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| ----- |     | f(n-1) |     | ------ |
```

[注：----- 表示我们不关心该值。]
同样，矩阵**B**的第二个元素是f(n)，可以通过简单地从**A**中取**1 X f(n)**得到，所以矩阵**M**的第二行是[1 0]。

```
| ----- |  X  |  f(n)  |  =  | ------ |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

然后我们得到了所需的2 X 2矩阵M。

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

这些矩阵是通过矩阵乘法简单推导出来的。

类型2：

让我们稍微复杂一点：求f(n) = a X f(n-1) + b X f(n-2)，其中 a和 b是常数。
这告诉我们，f(n+1) = a X f(n) + b X f(n-1)。
到目前为止，应该很清楚矩阵的维度将等于依赖项的数量，即
在这个特定的例子中，同样是2。因此对于A和B，我们可以构建两个大小为2 X 1的矩阵：

```
矩阵A            矩阵B
|  f(n)  |       | f(n+1) |
| f(n-1) |       |  f(n)  |
```

现在对于f(n+1) = a X f(n) + b X f(n-1)，我们需要将[a, b]放在目标矩阵M的第一行。对于B中的第二个元素，即f(n)，我们已经在矩阵A中有了，所以直接取用，这使得矩阵M的第二行为[1 0]。这次我们得到：

```
| a   b |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

相当简单，是吧？

类型3：

如果你坚持到这一步，你已经成长了许多，现在让我们面对一个稍微复杂的关系：求f(n) = a X f(n-1) + c X f(n-3)？
哎呀！几分钟前，我们看到的都是连续状态，但这里，状态f(n-2)缺失了。现在怎么办？

实际上这已经不再是问题了，我们可以将关系转换如下：f(n) = a X f(n-1) + 0 X f(n-2) +c X f(n-3)，推导出 f(n+1) = a X f(n) + 0 X f(n-1) + c X f(n-2)。现在，我们看到，这实际上是类型2中描述的一种形式。因此这里的目标矩阵 M 将是 3 X 3，元素为：

```
| a 0 c |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 1 0 |     | f(n-2) |     | f(n-1) |
```

这些计算方法与类型2相同，如果觉得困难，可以用笔和纸试一试。

类型4：

生活变得异常复杂，问题先生现在要求你找到 f(n) = f(n-1) + f(n-2) + c 其中 c 是任意常数。

这是一个新的问题，我们过去见过的都是在乘法之后，A 中的每个状态转换到它在 B 中的下一个状态。

[Note: ----- means we are not concerned about this value.]
Similarly, 2nd item of **B** is f(n) which can be got by simply taking **1 X f(n)** from **A**, so the 2nd row of **M** is [1 0].

```
| ----- |  X  |  f(n)  |  =  | ------ |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Then we get our desired **2 X 2** matrix **M**.

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

These matrices are simply derived using matrix multiplication.

Type 2:

Let's make it a little complex: find f(n) = a X f(n-1) + b X f(n-2), where **a** and **b** are constants.
This tells us, f(n+1) = a X f(n) + b X f(n-1).
By this far, this should be clear that the dimension of the matrices will be equal to the number of dependencies, i.e. in this particular example, again 2. So for **A** and **B**, we can build two matrices of size **2 X 1**:

```
Matrix A            Matrix B
|  f(n)  |         | f(n+1) |
| f(n-1) |         |  f(n)  |
```

Now for f(n+1) = a X f(n) + b X f(n-1), we need [a, b] in the first row of objective matrix **M**. And for the 2nd item in **B**, i.e. f(n) we already have that in matrix **A**, so we just take that, which leads, the 2nd row of the matrix M to [1 0]. This time we get:

```
| a   b |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Pretty simple, eh?

Type 3:

If you've survived through to this stage, you've grown much older, now let's face a bit complex relation: find f(n) = a X f(n-1) + c X f(n-3)?
Ooops! A few minutes ago, all we saw were contiguous states, but here, the state **f(n-2)** is missing. Now?

Actually this is not a problem anymore, we can convert the relation as follows: f(n) = a X f(n-1) + 0 X f(n-2) + c X f(n-3), deducing f(n+1) = a X f(n) + 0 X f(n-1) + c X f(n-2). Now, we see that, this is actually a form described in Type 2. So here the objective matrix **M** will be **3 X 3**, and the elements are:

```
| a 0 c |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 1 0 |     | f(n-2) |     | f(n-1) |
```

These are calculated in the same way as type 2, if you find it difficult, try it on pen and paper.

Type 4:

Life is getting complex as hell, and Mr, Problem now asks you to find f(n) = f(n-1) + f(n-2) + c where **c** is any constant.
Now this is a new one and all we have seen in past, after the multiplication, each state in **A** transforms to its next

状态在 B 中。

```
f(n) = f(n-1) + f(n-2) + c
f(n+1) = f(n) + f(n-1) + c
f(n+2) = f(n+1) + f(n) + c
.....如此类推
```

所以，通常我们无法用之前的方法得到它，但如果我们把 c 作为一个状态加入呢：

```
    |  f(n)  |   | f(n+1) |
M X | f(n-1) | = |  f(n)  |
    |    c   |   |    c   |
```

现在，设计M并不难。下面是具体做法，但别忘了验证：

```
| 1 1 1 |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 0 1 |     |    c   |     |    c   |
```

类型5：

我们把它们合起来：求解f(n) = a X f(n-1) + c X f(n-3) + d X f(n-4) + e。这个作为练习留给你。首先尝试找出状态和矩阵M。并检查是否与你的解法一致。同时求出矩阵A和B。

```
| a 0 c d 1 |
| 1 0 0 0 0 |
| 0 1 0 0 0 |
| 0 0 1 0 0 |
| 0 0 0 0 1 |
```

类型6：

有时递推关系是这样给出的：

```
f(n) = f(n-1)   -> 如果 n 是奇数
f(n) = f(n-2)   -> 如果 n 是偶数
```

简而言之：

```
f(n) = (n&1) X f(n-1) + (!(n&1)) X f(n-2)
```

这里，我们可以根据奇偶性将函数分开，并为它们分别保留两个不同的矩阵，单独计算。

类型7：

感觉有点过于自信？那很好。有时我们可能需要维护多个递推关系，只要它们是相关的。例如，设一个递推关系为：

```
g(n) = 2g(n-1) + 2g(n-2) + f(n)
```

这里，递推关系 g(n) 依赖于 f(n)，可以在同一个矩阵中计算，但维度会增加。首先让我们设计矩阵 A 和 B。

state in **B**.

```
f(n) = f(n-1) + f(n-2) + c
f(n+1) = f(n) + f(n-1) + c
f(n+2) = f(n+1) + f(n) + c
..................... so on
```

So , normally we can't get it through previous fashion, but how about we add **c** as a state:

```
    |  f(n)  |   | f(n+1) |
M X | f(n-1) | = |  f(n)  |
    |    c   |   |    c   |
```

Now, its not much hard to design **M**. Here's how its done, but don't forget to verify:

```
| 1 1 1 |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 0 1 |     |    c   |     |    c   |
```

Type 5:

Let's put it altogether: find `f(n) = a X f(n-1) + c X f(n-3) + d X f(n-4) + e`. Let's leave it as an exercise for you. First try to find out the states and matrix **M**. And check if it matches with your solution. Also find matrix **A** and **B**.

```
| a 0 c d 1 |
| 1 0 0 0 0 |
| 0 1 0 0 0 |
| 0 0 1 0 0 |
| 0 0 0 0 1 |
```

Type 6:

Sometimes the recurrence is given like this:

```
f(n) = f(n-1)   -> if n is odd
f(n) = f(n-2)   -> if n is even
```

In short:

```
f(n) = (n&1) X f(n-1) + (!(n&1)) X f(n-2)
```

Here, we can split the functions in the basis of odd even and keep 2 different matrix for both of them and calculate them separately.

Type 7:

Feeling little too confident? Good for you. Sometimes we may need to maintain more than one recurrence, where they are interested. For example, let a recurrence re;atopm be:

```
g(n) = 2g(n-1) + 2g(n-2) + f(n)
```

Here, recurrence *g(n)* is dependent upon `f(n)` and this can be calculated in the same matrix but of increased dimensions. From these let's at first design the matrices **A** and **B**.

```
矩阵 A          矩阵 B
|  g(n)  |        | g(n+1) |
| g(n-1) |        |  g(n)  |
| f(n+1) |        | f(n+2) |
|  f(n)  |        | f(n+1) |
```

这里，`g(n+1) = 2g(n-1) + f(n+1)` 且 `f(n+2) = 2f(n+1) + 2f(n)`。现在，使用上述过程，我们可以求得目标矩阵 **M** 为：

```
| 2 2 1 0 |
| 1 0 0 0 |
| 0 0 2 2 |
| 0 0 1 0 |
```

所以，这些是通过这种简单技术解决的基本递推关系类别。


```
 Matrix A          Matrix B
|  g(n)  |        | g(n+1) |
| g(n-1) |        |  g(n)  |
| f(n+1) |        | f(n+2) |
|  f(n)  |        | f(n+1) |
```

Here, `g(n+1) = 2g(n-1) + f(n+1)` and `f(n+2) = 2f(n+1) + 2f(n)`. Now, using the processes stated above, we can find the objective matrix **M** to be:

```
| 2 2 1 0 |
| 1 0 0 0 |
| 0 0 2 2 |
| 0 0 1 0 |
```

So, these are the basic categories of recurrence relations which are used to solveby this simple technique.

第53章：最小顶点覆盖的多项式时间有界算法

变量	含义
G	输入的连通无向图
X	顶点集合
C	最终顶点集合

这是一个用于获取连通无向图最小顶点覆盖的多项式算法。该算法的时间复杂度为 O(n2)

第53.1节：算法伪代码

算法 PMinVertexCover（图 G）
输入 连通图 G
输出最小顶点覆盖集 C

```
集合 C <- 新建 集合<顶点>()集合

 X <- 新建 集合<顶点>()X <- G.

按度数降序排列获取所有顶点()


列表<顶点> adjacentVertices1 <- G.获取相邻顶点(v)

    如果 !C 包含 adjacentVertices1 中的任意顶点，则

C.添加(v)

遍历 C 中的顶点 执行

列表<顶点> adjacentVertices2 <- G.相邻顶点(vertex)

    如果 C 包含 adjacentVertices2 中的任意顶点，则

C.移除(vertex)


返回 C
```

> C 是图 G 的最小顶点覆盖集

> 我们可以使用桶排序对顶点按度数进行排序，因为度数的最大值是 (n-1)，其中 n 是顶点数，因此排序的时间复杂度为 O(n)

Chapter 53: polynomial-time bounded algorithm for Minimum Vertex Cover

Variable	Meaning
G	Input connected un-directed graph
X	Set of vertices
C	Final set of vertices

This is a polynomial algorithm for getting the minimum vertex cover of connected undirected graph. The time complexity of this algorithm is O(n2)

Section 53.1: Algorithm Pseudo Code

Algorithm PMinVertexCover (graph G)
Input connected graph G
Output Minimum Vertex Cover Set C

```
Set C <- new Set<Vertex>()

Set X <- new Set<Vertex>()

X <- G.getAllVerticiesArrangedDescendinglyByDegree()

for v in X do
    List<Vertex> adjacentVertices1 <- G.getAdjacent(v)

    if !C contains any of adjacentVertices1 then

        C.add(v)

for vertex in C do

    List<vertex> adjacentVertices2 <- G.adjacentVertecies(vertex)

    if C contains any of adjacentVertices2 then

        C.remove(vertex)


return C
```

> C is the minimum vertex cover of graph G

> we can use bucket sort for sorting the vertices according to its degree because the maximum value of degrees is (n-1) where n is the number of vertices then the time complexity of the sorting will be O(n)

第54章：动态时间规整

第54.1节：动态时间规整简介

动态时间规整（DTW）是一种用于测量两个时间序列相似度的算法，这两个序列的速度可能不同。例如，即使一个人走得比另一个人快，或者在观察过程中有加速和减速，DTW仍能检测出走路的相似性。它可以用来匹配一个语音命令样本与其他命令，即使说话者说得比预录的样本语音快或慢。DTW可以应用于视频、音频和图形数据的时间序列——实际上，任何可以转换成线性序列的数据都可以用DTW进行分析。

一般来说，DTW是一种计算两个给定序列之间最优匹配的方法，带有一定的限制。但这里我们先讲简单的部分。假设我们有两个语音序列"样本"（Sample）和"测试"（Test），我们想检查这两个序列是否匹配。这里的语音序列指的是你声音转换成的数字信号，可能是表示你说的词的声音幅度或频率。假设：

```
样本 = {1, 2, 3, 5, 5, 5, 6}
测试 = {1, 1, 2, 2, 3, 5}
```

我们想找出这两个序列之间的最优匹配。

首先，我们定义两个点之间的距离，d(x, y)，其中 x 和 y 表示两个点。设，

```
d(x, y) = |x - y|     //绝对差值
```

让我们用这两个序列创建一个二维矩阵 Table。我们将计算每个点之间的距离示例包含每个测试点，并找到它们之间的最佳匹配。

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |  0  |  1  |  1  |  2  |  2  |  3  |  5  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  0  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  1  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  2  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  3  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  5  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  5  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  5  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  6  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

这里，Table[i][j] 表示如果我们考虑序列到 Sample[i] 和 Test[j] 为止，结合之前观察到的所有最优距离，两个序列之间的最优距离。

对于第一行，如果我们不取 Sample 中的任何值，那么该行与 Test 之间的距离将是 infinity。因此我们在第一行填入 infinity。第一列同理。如果我们不取 Test 中的任何值，那么该列与 Sample 之间的距离也将是 infinity。并且 0 和 0 之间的距离就是 0。我们得到，

Chapter 54: Dynamic Time Warping

Section 54.1: Introduction To Dynamic Time Warping

Dynamic Time Warping (DTW) is an algorithm for measuring similarity between two temporal sequences which may vary in speed. For instance, similarities in walking could be detected using DTW, even if one person was walking faster than the other, or if there were accelerations and decelerations during the course of an observation. It can be used to match a sample voice command with others command, even if the person talks faster or slower than the prerecorded sample voice. DTW can be applied to temporal sequences of video, audio and graphics data-indeed, any data which can be turned into a linear sequence can be analyzed with DTW.

In general, DTW is a method that calculates an optimal match between two given sequences with certain restrictions. But let's stick to the simpler points here. Let's say, we have two voice sequences **Sample** and **Test**, and we want to check if these two sequences match or not. Here voice sequence refers to the converted digital signal of your voice. It might be the amplitude or frequency of your voice that denotes the words you say. Let's assume:

```
Sample = {1, 2, 3, 5, 5, 5, 6}
Test   = {1, 1, 2, 2, 3, 5}
```

We want to find out the optimal match between these two sequences.

At first, we define the distance between two points, *d(x, y)* where **x** and **y** represent the two points. Let,

```
d(x, y) = |x - y|     //absolute difference
```

Let's create a 2D matrix **Table** using these two sequences. We'll calculate the distances between each point of **Sample** with every points of **Test** and find the optimal match between them.

```
+------+------+------+------+------+------+------+------+------+
|      |   0  |   1  |   1  |   2  |   2  |   3  |   5  |
+------+------+------+------+------+------+------+------+------+
|   0  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   1  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   2  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   3  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
|   6  |      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+------+
```

Here, **Table[i][j]** represents the optimal distance between two sequences if we consider the sequence up to **Sample[i]** and **Test[j]**, considering all the optimal distances we observed before.

For the first row, if we take no values from **Sample**, the distance between this and **Test** will be *infinity*. So we put *infinity* on the first row. Same goes for the first column. If we take no values from **Test**, the distance between this one and **Sample** will also be infinity. And the distance between **0** and **0** will simply be **0**. We get,

		0	1	1	2	2	3	5
	0	0	inf	inf	inf	inf	inf	inf
1	inf							
2	inf							
3	inf							
5	inf							
5	inf							
5	inf							
6	inf							

现在对于每一步，我们将考虑相关点之间的距离，并加上迄今为止找到的最小距离。这将给出两个序列到该位置的最优距离。我们的公式是，

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

对于第一个，d(1, 1) = 0，Table[0][0] 表示最小值。所以 Table[1][1] 的值将是 0 + 0 = 0。对于第二个，d(1, 2) = 0，Table[1][1] 表示最小值。值为：Table[1][2] = 0 + 0 = 0。如果我们继续这样计算，完成后表格将如下所示：

		0	1	1	2	2	3	5
	0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	1

表格中 Table[7][6] 的值表示这两个给定序列之间的最大距离。这里 1 表示 Sample 和 Test 之间的最大距离是 1。

现在，如果我们从最后一点回溯，一直回到起始点(0, 0)，我们会得到一条长线，沿水平方向、垂直方向和对角线方向移动。我们的回溯过程将是：

```
如果 Table[i-1][j-1] <= Table[i-1][j] 且 Table[i-1][j-1] <= Table[i][j-1]
```


		0	1	1	2	2	3	5
	0	0	inf	inf	inf	inf	inf	inf
1	inf							
2	inf							
3	inf							
5	inf							
5	inf							
5	inf							
6	inf							

Now for each step, we'll consider the distance between each points in concern and add it with the minimum distance we found so far. This will give us the optimal distance of two sequences up to that position. Our formula will be,

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

For the first one, **d(1, 1)** = **0**, **Table[0][0]** represents the minimum. So the value of **Table[1][1]** will be **0** + **0** = **0**. For the second one, **d(1, 2)** = **0**. **Table[1][1]** represents the minimum. The value will be: **Table[1][2]** = **0** + **0** = **0**. If we continue this way, after finishing, the table will look like:

		0	1	1	2	2	3	5
	0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	1

The value at **Table[7][6]** represents the maximum distance between these two given sequences. Here **1** represents the maximum distance between **Sample** and **Test** is **1**.

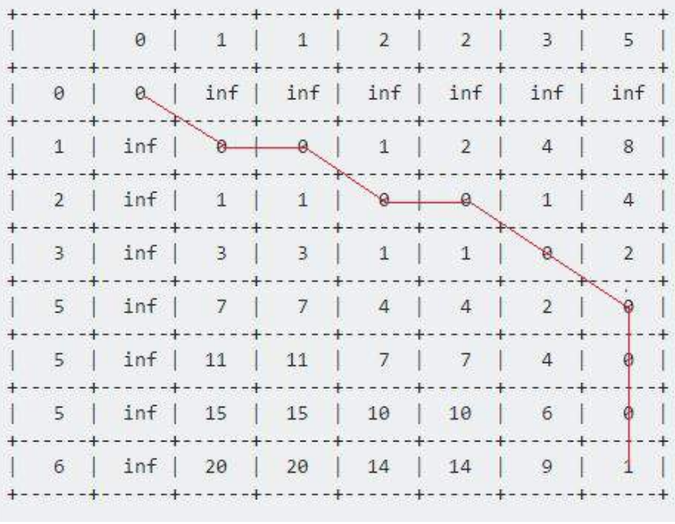
Now if we backtrack from the last point, all the way back towards the starting **(0, 0)** point, we get a long line that moves horizontally, vertically and diagonally. Our backtracking procedure will be:

```
if Table[i-1][j-1] <= Table[i-1][j] and Table[i-1][j-1] <= Table[i][j-1]
```

```
i := i - 1
    j := j - 1
否则如果 Table[i-1][j] <= Table[i-1][j-1] 且 Table[i-1][j] <= Table[i][j-1]
   i := i - 1
否则
j := j - 1
结束if
```

我们将继续此过程，直到达到(0，0)。每一步移动都有其含义：

- 水平移动表示删除。这意味着我们的Test序列在此时间段内加速了。
- 垂直移动表示插入。这意味着我们的Test序列在此时间段内减速了。
- 对角线移动表示匹配。在此期间，Test和Sample是相同的。



我们的伪代码将是：

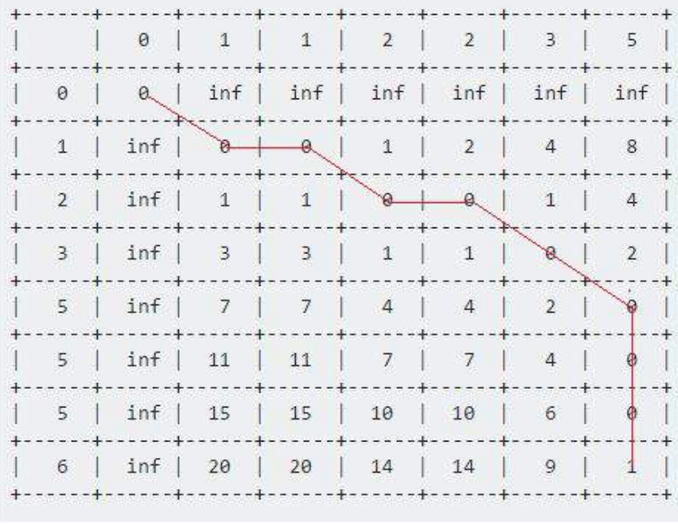
```
过程 DTW(样本，测试)：
n := Sample.length
m := Test.length
创建表[n + 1][m + 1]
for i 从 1 到 n
    表[i][0] := 无穷大
end for
for i 从 1 到 m
    表[0][i] := 无穷大
end for
表[0][0] := 0
for i 从 1 到 n
     for j 从 1 到 m
表[i][j] := d(Sample[i], Test[j])
                          +最小值(表[i-1][j-1],      //匹配
                               表[i][j-1],        //插入
                               表[i-1][j])        //删除
    end for
end for
返回 表[n + 1][m + 1]
```

我们还可以添加局部约束。也就是说，我们要求如果Sample[i]与Test[j]匹配，那么|i - j|不大于w，一个窗口参数。


```
    i := i - 1
    j := j - 1
else if Table[i-1][j] <= Table[i-1][j-1] and Table[i-1][j] <= Table[i][j-1]
    i := i - 1
else
    j := j - 1
end if
```

We'll continue this till we reach **(0, 0)**. Each move has its own meaning:

- A horizontal move represents deletion. That means our **Test** sequence accelerated during this interval.
- A vertical move represents insertion. That means out **Test** sequence decelerated during this interval.
- A diagonal move represents match. During this period **Test** and **Sample** were same.



Our pseudo-code will be:

```
Procedure DTW(Sample, Test):
n := Sample.length
m := Test.length
Create Table[n + 1][m + 1]
for i from 1 to n
    Table[i][0] := infinity
end for
for i from 1 to m
    Table[0][i] := infinity
end for
Table[0][0] := 0
for i from 1 to n
    for j from 1 to m
        Table[i][j] := d(Sample[i], Test[j])
                       + minimum(Table[i-1][j-1],      //match
                                 Table[i][j-1],        //insertion
                                 Table[i-1][j])        //deletion
    end for
end for
Return Table[n + 1][m + 1]
```

We can also add a locality constraint. That is, we require that if `Sample[i]` is matched with `Test[j]`, then `|i - j|` is no larger than **w**, a window parameter.

复杂度：

计算DTW的复杂度是O(m * n)，其中m和n表示每个序列的长度。更快的计算DTW技术包括PrunedDTW、Sparse DTW和FastDTW。

应用：

- 语音识别
- 相关功率分析

Complexity:

The complexity of computing DTW is **O(m * n)** where **m** and **n** represent the length of each sequence. Faster techniques for computing DTW include PrunedDTW, SparseDTW and FastDTW.

Applications:

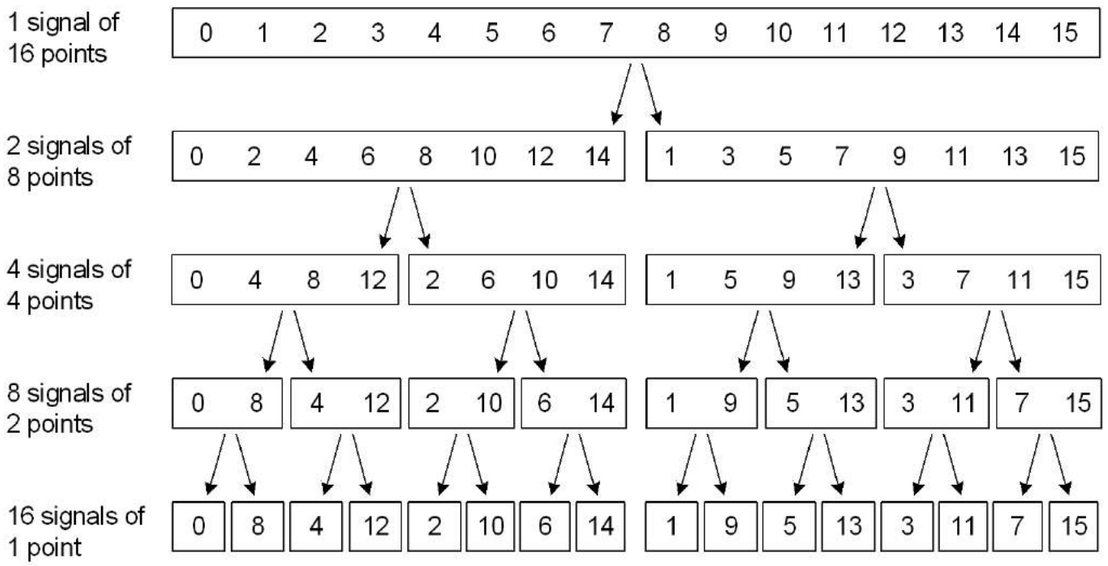
- Spoken word recognition
- Correlation Power Analysis

第55章：快速傅里叶变换

DFT（离散傅里叶变换）的实数和复数形式可用于对任何离散且周期信号进行频率分析或合成。FFT（快速傅里叶变换）是DFT的一种实现，可以在现代CPU上快速执行。

第55.1节：基2快速傅里叶变换

计算FFT最简单且可能最著名的方法是基2时域抽取算法。
基2快速傅里叶变换通过将N点时域信号分解为N个仅包含单点的时域信号来工作



.

信号分解，或称为‘时域抽取’，是通过对时域数据数组的索引进行位反转实现的。因此，对于一个16点信号，样本1（二进制0001）与样本8（1000）交换，样本2（0010）与样本4（0100）交换，依此类推。使用位反转技术进行样本交换可以简单地通过软件实现，但这限制了基2快速傅里叶变换仅适用于长度为N = 2^M的信号。

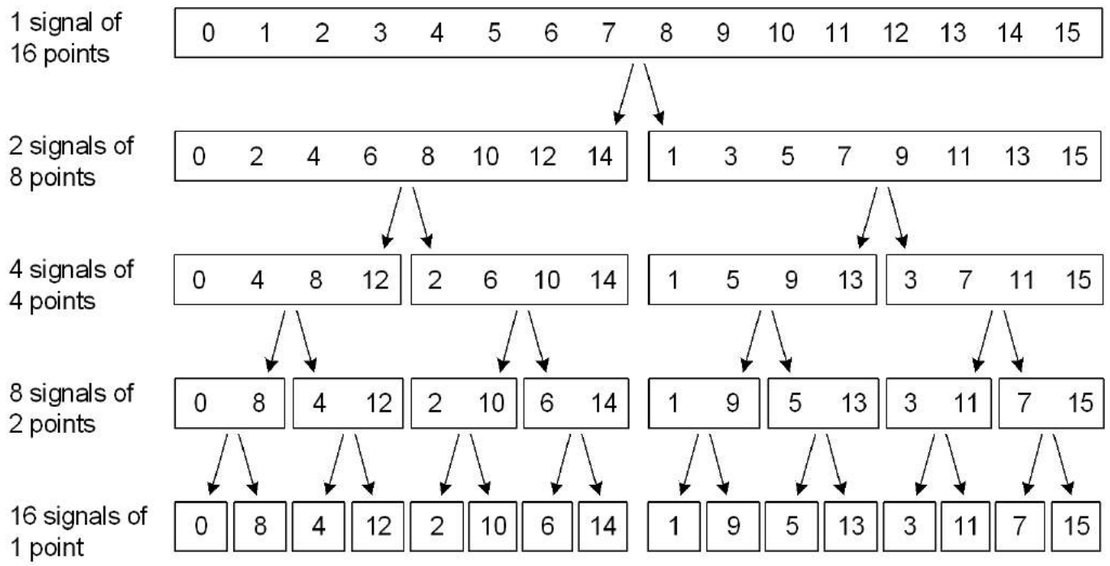
时域中一个点信号的值等于其在频域中的值，因此这组分解后的单个时域点数组无需变换即可成为频域点数组。然而，这N个单点需要重构为一个N点的频谱。完整频谱的最优重构是通过蝶形计算完成的。Radix-2 FFT中的每个重构阶段都会执行若干个两点蝶形运算，使用一组类似的指数加权函数Wn^R。

Chapter 55: Fast Fourier Transform

The Real and Complex form of DFT (**D**iscrete **F**ourier **T**ransforms) can be used to perform frequency analysis or synthesis for any discrete and periodic signals. The FFT (**F**ast **F**ourier **T**ransform) is an implementation of the DFT which may be performed quickly on modern CPUs.

Section 55.1: Radix 2 FFT

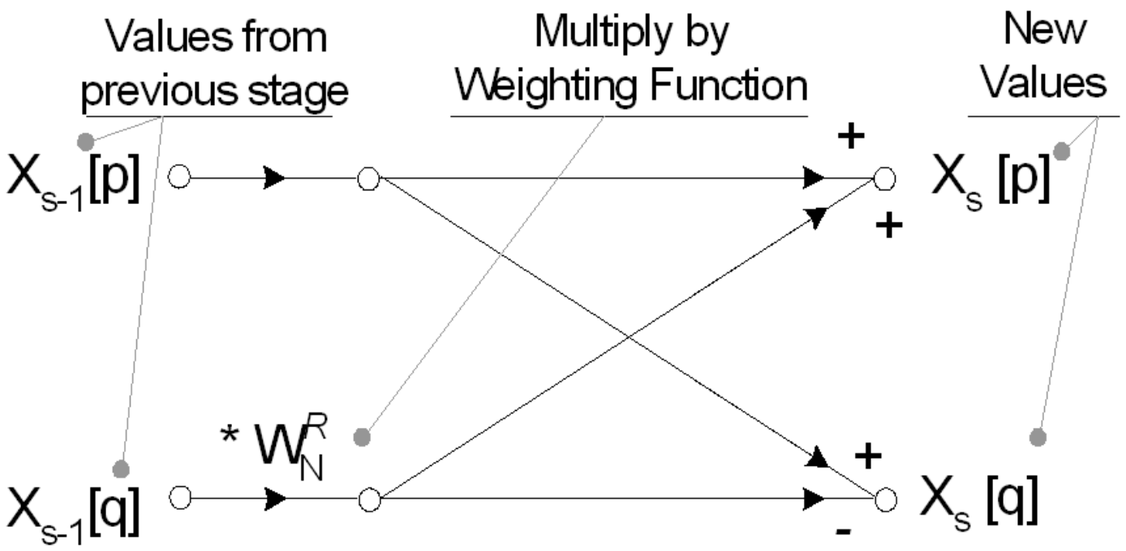
The simplest and perhaps best-known method for computing the FFT is the Radix-2 Decimation in Time algorithm. The Radix-2 FFT works by decomposing an N point time domain signal into N time domain signals each composed of a single point



.

Signal decomposition, or ‘decimation in time’ is achieved by bit reversing the indices for the array of time domain data. Thus, for a sixteen-point signal, sample 1 (Binary 0001) is swapped with sample 8 (1000), sample 2 (0010) is swapped with 4 (0100) and so on. Sample swapping using the bit reverse technique can be achieved simply in software, but limits the use of the Radix 2 FFT to signals of length N = 2^M.

The value of a 1-point signal in the time domain is equal to its value in the frequency domain, thus this array of decomposed single time-domain points requires no transformation to become an array of frequency domain points. The N single points; however, need to be reconstructed into one N-point frequency spectra. Optimal reconstruction of the complete frequency spectrum is performed using butterfly calculations. Each reconstruction stage in the Radix-2 FFT performs a number of two point butterflies, using a similar set of exponential weighting functions, Wn^R.



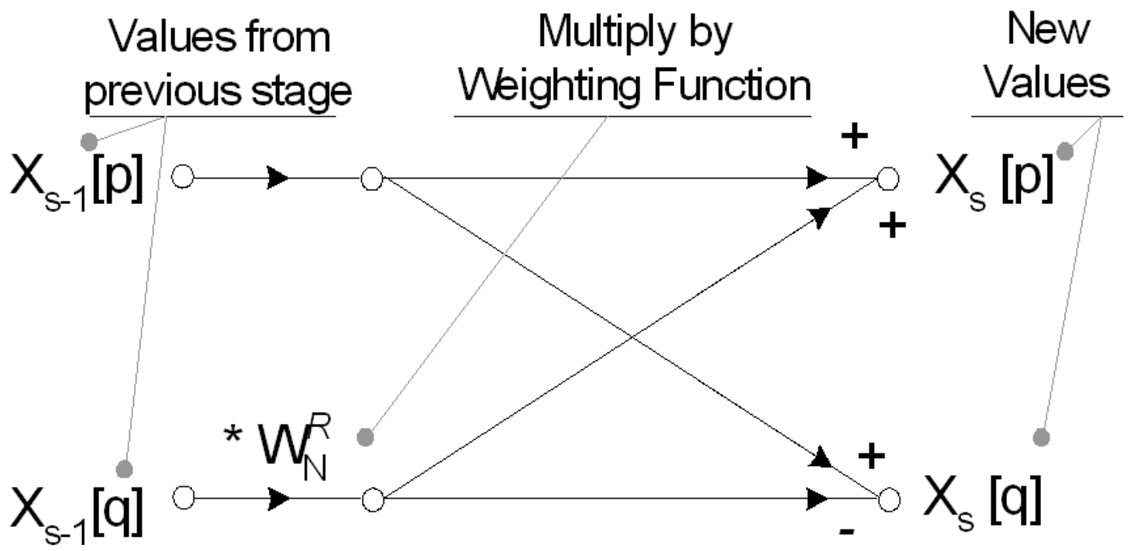
FFT通过利用Wn^R的周期性，消除了离散傅里叶变换中的冗余计算。
频谱重建在log2(N)个蝴蝶计算阶段完成，得到X[K]；即矩形形式的实部和虚部频域数据。要转换为幅度和相位（极坐标），需要计算绝对值√(Re² + Im²)和幅角tan⁻¹(Im/Re)。

$$\textit{Exponential Weighting Factor:} \qquad W_N^R = e^{-j(2\pi R/N)} = \cos(2\pi R/N) - j\sin(2\pi R/N)$$

N: *Number of points in the FFT*

R: *Current WN Factor: depends on N, current FFT stage and separation of butterflies in that stage*

下面显示了一个八点基2 FFT的完整蝴蝶流程图。注意输入信号已根据前面介绍的时域抽取程序重新排序。



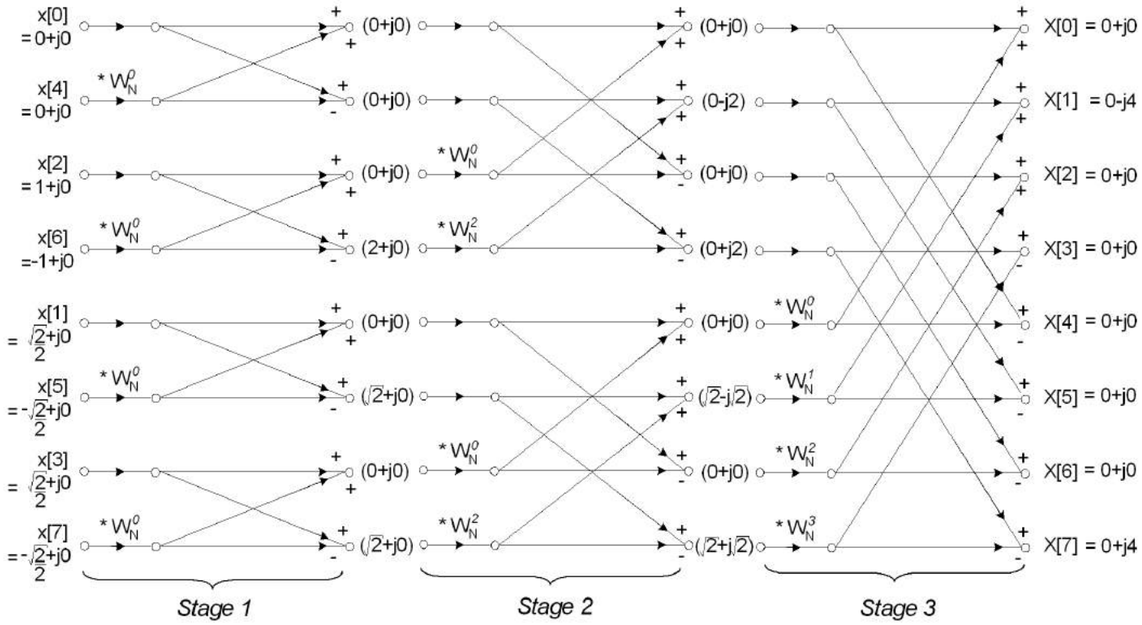
The FFT removes redundant calculations in the Discrete Fourier Transform by exploiting the periodicity of Wn^R. Spectral reconstruction is completed in log2(N) stages of butterfly calculations giving X[K]; the real and imaginary frequency domain data in rectangular form. To convert to magnitude and phase (polar coordinates) requires finding the absolute value, √(Re2 + Im2), and argument, tan-1(Im/Re).

$$\textit{Exponential Weighting Factor:} \qquad W_N^R = e^{-j(2\pi R/N)} = \cos(2\pi R/N) - j\sin(2\pi R/N)$$

N: *Number of points in the FFT*

R: *Current WN Factor: depends on N, current FFT stage and separation of butterflies in that stage*

The complete butterfly flow diagram for an eight point Radix 2 FFT is shown below. Note the input signals have previously been reordered according to the decimation in time procedure outlined previously.



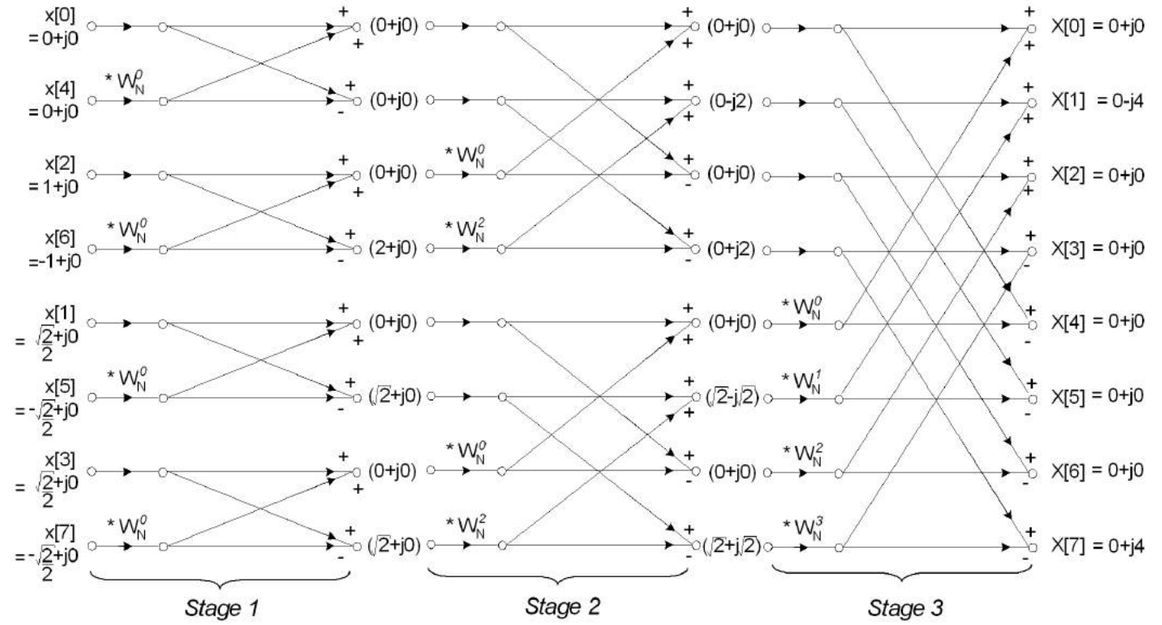
FFT通常对复数输入进行操作并产生复数输出。对于实数信号，虚部可设为零，实部设为输入信号x[n]，但针对仅实数数据的变换存在许多优化。重建过程中使用的Wn^R值可通过指数加权公式确定。

指数加权幂R的值由频谱重建的当前阶段和特定蝴蝶中的当前计算决定。

代码示例（C/C++）

下面是一个计算基2 FFT的C/C++代码示例。这是一个简单实现，适用于任何N为2的幂的大小。它大约比最快的FFTw实现慢3倍，但仍是未来优化或学习该算法工作原理的良好基础。

```
#include <math.h>

#define PI       3.14159265358979323846264338327950     // 用于正弦/余弦计算的圆周率PI
#define TWOPI    6.28318530717958647692528676655900      // 用于正弦/余弦计算的2倍圆周率2*PI
#define Deg2Rad  0.01745329251994329576923690768488 6   // 角度转弧度的转换因子
#define Rad2Deg  57.29577951308232087679815481410 5     // 弧度转角度的转换因子
#define log10_2  0.30102999566398119521373889472449    // 2的以10为底的对数
#define log10_2_INV 3.32192809488736234787031942949 48 // 1/Log10(2)

// 复数变量结构（双精度）
struct complex
{
public:
    double  Re, Im;         // 其实并不复杂
};

// 如果N是2的幂则返回true
bool isPwrTwo(int N, int *M)
{
    *M = (int)ceil(log10((double)N) * log10_2_INV);// M是执行的阶段数。2^M = N
    int NN = (int)pow(2.0, *M);
```

The FFT typically operates on complex inputs and produces a complex output. For real signals, the imaginary part may be set to zero and real part set to the input signal, x[n], however many optimisations are possible involving the transformation of real-only data. Values of Wn^R used throughout the reconstruction can be determined using the exponential weighting equation.

The value of R (the exponential weighting power) is determined the current stage in the spectral reconstruction and the current calculation within a particular butterfly.

Code Example (C/C++)

A C/C++ code sample for computing the Radix 2 FFT can be found below. This is a simple implementation which works for any size N where N is a power of 2. It is approx 3x slower than the fastest FFTw implementation, but still a very good basis for future optimisation or for learning about how this algorithm works.

```
#include <math.h>

#define PI       3.14159265358979323846264338327950     // PI for sine/cos calculations
#define TWOPI    6.28318530717958647692528676655900      // 2*PI for sine/cos calculations
#define Deg2Rad  0.01745329251994329576923690768488 6   // Degrees to Radians factor
#define Rad2Deg  57.29577951308232087679815481410 5     // Radians to Degrees factor
#define log10_2  0.30102999566398119521373889472449    // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942949 48 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;         // Not so complicated after all
};

// Returns true if N is a power of 2
bool isPwrTwo(int N, int *M)
{
    *M = (int)ceil(log10((double)N) * log10_2_INV);// M is number of stages to perform. 2^M = N
    int NN = (int)pow(2.0, *M);
```

```
    if ((NN != N) || (NN == 0)) // 检查N是否为2的幂。
        return false;

    return true;
}

void rad2FFT(int N, complex *x, complex *DFT)
{
    int M = 0;

    // 检查是否为2的幂。如果不是，则退出
    if (!isPwrTwo(N, &M))
        throw "Rad2FFT(): N 必须是基数FFT的2的幂";

    // 整型变量

    int BSep;                   // BSep 是蝴蝶操作之间的内存间距
    int BWidth;                 // BWidth 是蝴蝶两端的内存间距
    int P;                      // P 是该阶段使用的相似 Wn 的数量
    int j;                      // j 用于循环执行每个阶段的所有计算
    int stage = 1;              // stage 是FFT的阶段编号。总共有 M 个阶段
(1 到 M)。
    int HiIndex;              // HiIndex 是DFT数组中每个蝴蝶计算顶部值的索引

    unsigned int iaddr;         // 用于位反转的位掩码
    int ii;                     // 用于位反转（时域抽取）的整型位域
    int MM1 = M - 1;

    unsigned int i;
    int l;
    unsigned int nMax = (unsigned int)N;

    // 双精度变量
    double TwoPi_N = TWOPI / (double)N;     // 常量以节省计算时间。= 2*PI / N
    double TwoPi_NP;

    // 复数变量（参见 'struct complex'）
复数 WN;                    // Wn 是指数加权函数，形式为 a + jb
    复数 TEMP;                  // TEMP 用于保存蝶形计算中的中间结果以节省计算量
复数 *pDFT = DFT;           // 指向 DFT 数组第一个元素的指针
    复数 *pLo;                  // 指向蝶形计算中低值的指针
    复数 *pHi;
复数 *pX;                // 指向 x[n] 的指针


    // 时间抽取法 - x[n] 采样排序
    for (i = 0; i < nMax; i++, DFT++)
    {
pX = x + i;             // 根据基地址 *x 和索引 i 计算当前的 x[n]
ii = 0;                     // 重置 DFT[n] 的新地址
        iaddr = i;                  // 复制 i 以便操作
        for (l = 0; l < M; l++) // 对 i 进行位反转并存储到 ii 中...
        {
            if (iaddr & 0x01)       // 判断最低有效位                  ii += (1 <<
(MM1 - l));     // 如果最低位为 1，则将 ii 增加 2^(M-1-l)              iaddr >>= 1;                // 右移
 iaddr 以测试下一位。使用逻辑运算以提高速度

            if (!iaddr)
                break;
        }
DFT = pDFT + ii;        // 根据基地址 *pDFT 和位反转索引 ii 计算当前的 DFT[n]
```



```
    if ((NN != N) || (NN == 0)) // Check N is a power of 2.
        return false;

    return true;
}

void rad2FFT(int N, complex *x, complex *DFT)
{
    int M = 0;

    // Check if power of two. If not, exit
    if (!isPwrTwo(N, &M))
        throw "Rad2FFT(): N must be a power of 2 for Radix FFT";

    // Integer Variables

    int BSep;                   // BSep is memory spacing between butterflies
    int BWidth;                 // BWidth is memory spacing of opposite ends of the butterfly
    int P;                      // P is number of similar Wn's to be used in that stage
    int j;                      // j is used in a loop to perform all calculations in each stage
    int stage = 1;              // stage is the stage number of the FFT. There are M stages in total
(1 to M).
    int HiIndex;                // HiIndex is the index of the DFT array for the top value of each
butterfly calc
    unsigned int iaddr;         // bitmask for bit reversal
    int ii;                     // Integer bitfield for bit reversal (Decimation in Time)
    int MM1 = M - 1;

    unsigned int i;
    int l;
    unsigned int nMax = (unsigned int)N;

    // Double Precision Variables
    double TwoPi_N = TWOPI / (double)N;     // constant to save computational time.  = 2*PI / N
    double TwoPi_NP;

    // complex Variables (See 'struct complex')
    complex WN;                 // Wn is the exponential weighting function in the form a + jb
    complex TEMP;               // TEMP is used to save computation in the butterfly calc
    complex *pDFT = DFT;        // Pointer to first elements in DFT array
    complex *pLo;               // Pointer for lo / hi value of butterfly calcs
    complex *pHi;
    complex *pX;                // Pointer to x[n]


    // Decimation In Time - x[n] sample sorting
    for (i = 0; i < nMax; i++, DFT++)
    {
        pX = x + i;             // Calculate current x[n] from base address *x and index i.
        ii = 0;                 // Reset new address for DFT[n]
        iaddr = i;              // Copy i for manipulations
        for (l = 0; l < M; l++) // Bit reverse i and store in ii...
        {
            if (iaddr & 0x01)     // Detemine least significant bit
                ii += (1 << (MM1 - l));     // Increment ii by 2^(M-1-l) if lsb was 1
            iaddr >>= 1;                // right shift iaddr to test next bit. Use logical
operations for speed increase
            if (!iaddr)
                break;
        }
        DFT = pDFT + ii;        // Calculate current DFT[n] from base address *pDFT and bit
reversed index ii
```

```
DFT->Re = pX->Re;       // 使用地址排序的时域信号更新复数数组
x[n]
DFT->Im = pX->Im;       // 注意：虚部始终为零
    }

    // 通过蝶形计算进行FFT计算
    for (stage = 1; stage <= M; stage++) // M阶段循环，其中2^M = N
    {
BSep = (int)(pow(2, stage)); // 蝶形间隔 = 2^stage
        P = N / BSep;               // 本阶段相似的Wn数量 = N/BSep
                        BWidth = BSep / 2;     // 蝶形宽度（相对点间距）= 间隔 / 2
2.

TwoPi_NP = TwoPi_N*P;

        for (j = 0; j < BWidth; j++) // 每个蝶形的j计算循环
        {
            if (j != 0)             // 当R=0时节省计算，因为WN^0 = (1 + j0)
            {
                //WN.Re = cos(TwoPi_NP*j)
WN.Re = cos(TwoPi_N*P*j);      // 计算Wn（实部和虚部）
                 WN.Im = -sin(TwoPi_N*P*j);
            }

            for (HiIndex = j; HiIndex < N; HiIndex += BSep) // HiIndex 以步长 BSep 循环
每级蝴蝶数
            {
pHi = pDFT + HiIndex;                   // 指向较高值                pLo = pHi + BWidth;
             // 指向较低值（注意 VC++ 调整了元素间距）


                if (j != 0)                         // 如果指数幂不为零...
                {
                    //CMult(pLo, &WN, &TEMP);          // 对低值与 Wn 进行复数乘法

TEMP.Re = (pLo->Re * WN.Re) - (pLo->Im * WN.Im);
                    TEMP.Im = (pLo->Re * WN.Im) + (pLo->Im * WN.Re);

                    //CSub (pHi, &TEMP, pLo);
pLo->Re = pHi->Re - TEMP.Re;       // 计算新的低值（复数减法）
                 pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);          // 计算新的高值（复数加法）
                 pHi->Re = (pHi->Re + TEMP.Re);
                 pHi->Im = (pHi->Im + TEMP.Im);
                }
                否则
                {
TEMP.Re = pLo->Re;
                    TEMP.Im = pLo->Im;

                    //CSub (pHi, &TEMP, pLo);
pLo->Re = pHi->Re - TEMP.Re;       // 计算新的低值（复数减法）
                 pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);          // 计算新的高值（复数加法）
                 pHi->Re = (pHi->Re + TEMP.Re);
                 pHi->Im = (pHi->Im + TEMP.Im);
                }
            }
        }
    }
```



```
        DFT->Re = pX->Re;       // Update the complex array with address sorted time domain signal
x[n]
        DFT->Im = pX->Im;       // NB: Imaginary is always zero
    }

    // FFT Computation by butterfly calculation
    for (stage = 1; stage <= M; stage++) // Loop for M stages, where 2^M = N
    {
        BSep = (int)(pow(2, stage)); // Separation between butterflies = 2^stage
        P = N / BSep;               // Similar Wn's in this stage = N/Bsep
        BWidth = BSep / 2;     // Butterfly width (spacing between opposite points) = Separation /
2.

        TwoPi_NP = TwoPi_N*P;

        for (j = 0; j < BWidth; j++) // Loop for j calculations per butterfly
        {
            if (j != 0)             // Save on calculation if R = 0, as WN^0 = (1 + j0)
            {
                //WN.Re = cos(TwoPi_NP*j)
                WN.Re = cos(TwoPi_N*P*j);     // Calculate Wn (Real and Imaginary)
                WN.Im = -sin(TwoPi_N*P*j);
            }

            for (HiIndex = j; HiIndex < N; HiIndex += BSep) // Loop for HiIndex Step BSep
butterflies per stage
            {
                pHi = pDFT + HiIndex;                   // Point to higher value
                pLo = pHi + BWidth;                     // Point to lower value (Note VC++ adjusts
for spacing between elements)

                if (j != 0)                             // If exponential power is not zero...
                {
                    //CMult(pLo, &WN, &TEMP);          // Perform complex multiplication of Lovalue
with Wn
                    TEMP.Re = (pLo->Re * WN.Re) - (pLo->Im * WN.Im);
                    TEMP.Im = (pLo->Re * WN.Im) + (pLo->Im * WN.Re);

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;       // Find new Lovalue (complex subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);          // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
                else
                {
                    TEMP.Re = pLo->Re;
                    TEMP.Im = pLo->Im;

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;       // Find new Lovalue (complex subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);          // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
            }
        }
    }
```

```
pLo = 0;    // 将所有指针置空
    pHi = 0;
pDFT = 0;
    DFT = 0;
    pX = 0;
}
```

第55.2节：基2逆FFT

由于傅里叶变换的强对偶性，调整正向变换的输出可以得到逆FFT。频域数据可以通过以下方法转换到时域：

1. 通过对所有K的虚部取反，找到频域数据的复共轭。

2. 对共轭的频域数据执行正向FFT。
3. 将该FFT结果的每个输出除以N，以得到真实的时域值。
4. 通过对所有n的时域数据虚部取反，找到输出的复共轭。

注意：频域和时域数据均为复变量。通常逆FFT后时域信号的虚部要么为零，要么被视为舍入误差而忽略。将变量精度从32位浮点提升到64位双精度或128位长双精度，可以显著减少多次连续FFT操作产生的舍入误差。

代码示例（C/C++）

```
#include <math.h>

#define PI      3.14159265358979323846264338327950    // 用于正弦/余弦计算的圆周率PI
#define TWOPI    6.28318530717958647692528676655900     // 用于正弦/余弦计算的2倍圆周率2*PI
#define Deg2Rad  0.01745329251994329576923690768488   // 角度转弧度的转换因子
#define Rad2Deg  57.29577951308232087679815481410     // 弧度转角度的转换因子
#define log10_2  0.30102999566398119521373889472449   // 2的以10为底的对数
#define log10_2_INV 3.32192809488736234787031942949480 // 1/Log10(2)

// 复数变量结构（双精度）
struct complex
{
public:
    double  Re, Im;        // 其实并不复杂
};

void rad2InverseFFT(int N, complex *x, complex *DFT)
{
    // M是执行的阶段数。2^M = N
    double Mx = (log10((double)N) / log10((double)2));
    int a = (int)(ceil(pow(2.0, Mx)));
    int status = 0;
    if (a != N) // 检查 N 是否为 2 的幂
    {
x = 0;
DFT = 0;
        throw "rad2InverseFFT(): N 必须是 Radix 2 逆 FFT 的 2 的幂";
    }

complex *pDFT = DFT;         // 重置 DFT 指针的向量
    complex *pX = x;             // 重置 x[n] 指针的向量
    double NN = 1 / (double)N;  // 逆 FFT 的缩放因子
```



```
    pLo = 0;    // Null all pointers
    pHi = 0;
    pDFT = 0;
    DFT = 0;
    pX = 0;
}
```

Section 55.2: Radix 2 Inverse FFT

Due to the strong duality of the Fourier Transform, adjusting the output of a forward transform can produce the inverse FFT. Data in the frequency domain can be converted to the time domain by the following method:

1. Find the complex conjugate of the frequency domain data by inverting the imaginary component for all instances of K.
2. Perform the forward FFT on the conjugated frequency domain data.
3. Divide each output of the result of this FFT by N to give the true time domain value.
4. Find the complex conjugate of the output by inverting the imaginary component of the time domain data for all instances of n.

***Note**: both frequency and time domain data are complex variables. Typically the imaginary component of the time domain signal following an inverse FFT is either zero, or ignored as rounding error. Increasing the precision of variables from 32-bit float to 64-bit double, or 128-bit long double significantly reduces rounding errors produced by several consecutive FFT operations.*

Code Example (C/C++)

```
#include <math.h>

#define PI      3.14159265358979323846264338327950    // PI for sine/cos calculations
#define TWOPI    6.28318530717958647692528676655900     // 2*PI for sine/cos calculations
#define Deg2Rad  0.01745329251994329576923690768488   // Degrees to Radians factor
#define Rad2Deg  57.29577951308232087679815481410     // Radians to Degrees factor
#define log10_2  0.30102999566398119521373889472449   // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942949480 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;        // Not so complicated after all
};

void rad2InverseFFT(int N, complex *x, complex *DFT)
{
    // M is number of stages to perform. 2^M = N
    double Mx = (log10((double)N) / log10((double)2));
    int a = (int)(ceil(pow(2.0, Mx)));
    int status = 0;
    if (a != N) // Check N is a power of 2
    {
        x = 0;
        DFT = 0;
        throw "rad2InverseFFT(): N must be a power of 2 for Radix 2 Inverse FFT";
    }

    complex *pDFT = DFT;        // Reset vector for DFT pointers
    complex *pX = x;            // Reset vector for x[n] pointer
    double NN = 1 / (double)N;  // Scaling factor for the inverse FFT
```

```
    for (int i = 0; i < N; i++, DFT++)
DFT->Im *= -1;            // 求频谱的复共轭

    DFT = pDFT;                    // 重置频域指针
rad2FFT(N, DFT, x); // 计算前向 FFT，变量（时间和频率）互换

    int i;
complex* x;
    for ( i = 0, x = pX; i < N; i++, x++){
x->Re *= NN;     // 将时间域除以N以获得正确的幅度缩放
        x->Im *= -1;     // 改变ImX的符号
    }
}
```



```
    for (int i = 0; i < N; i++, DFT++)
        DFT->Im *= -1;          // Find the complex conjugate of the Frequency Spectrum

    DFT = pDFT;                 // Reset Freq Domain Pointer
    rad2FFT(N, DFT, x); // Calculate the forward FFT with variables switched (time & freq)

    int i;
    complex* x;
    for ( i = 0, x = pX; i < N; i++, x++){
        x->Re *= NN;    // Divide time domain by N for correct amplitude scaling
        x->Im *= -1;    // Change the sign of ImX
    }
}
```

附录A：伪代码

A.1节：变量赋值

你可以用不同的方式描述变量赋值。

有类型

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

无类型

```
a = 1
a := 1
let a = 1
a <- 1
```

A.2节：函数

只要函数名、返回语句和参数清晰，就没问题。

```
def incr n
    return n + 1
```

或者

```
let incr(n) = n + 1
```

或者

```
function incr (n)
    return n + 1
```

都非常清楚，所以你可以使用它们。尽量避免变量赋值时产生歧义

Appendix A: Pseudocode

Section A.1: Variable affectations

You could describe variable affectation in different ways.

Typed

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

No type

```
a = 1
a := 1
let a = 1
a <- 1
```

Section A.2: Functions

As long as the function name, return statement and parameters are clear, you're fine.

```
def incr n
    return n + 1
```

or

```
let incr(n) = n + 1
```

or

```
function incr (n)
    return n + 1
```

are all quite clear, so you may use them. Try not to be ambiguous with a variable affectation

致谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

阿卜杜勒·卡里姆	第一章
长石	第43章
艾哈迈德·法亚兹	第28章
阿尔伯·塔德罗斯	第53章
阿纳格·赫格德	第29章和第39章
安德烈·阿尔塔莫诺夫	第27章
AnukuL	第40章
巴赫蒂亚尔·哈桑	第9、11、14、17、19、20、22、40、41、42、47、52和54章
本森·林	第14、39和44章
brijs	第39章
克里斯	第15章
创意约翰	第49章和第51章
电巴克提	第10章
迪吉里杜管	第2章和第43章
迪佩什·普德尔	第21章
ABT博士	第55章
EsmaeelE	第2、29、30、39和50章
菲利普·奥尔贝格	第1和9章
ghilesZ	第17章
goeddek	第18和27章
greatwolf	第5章
伊贾兹·汗	第29章
invisal	第31章
伊莎·阿加瓦尔	第4、5、6、7和8章
伊希特·梅塔	第5章
弗拉德	第16和28章
伊万	第30章
贾纳基·穆尔西	第6章
JJTO	第9章
朱利安·鲁塞	第24章
朱欣·梅塔伊	第2章和第30章
凯尤尔·拉莫利亚	第23、29、30、31、32、33、34、35、36、37、38、45、47、48和50章
Khaled.K	第39章
kiner_shah	第12章
lambda	第38章
卢夫·阿加瓦尔	第30章
淋巴状	第31章
M S 霍赛因	第17章
马拉夫	第33章
马尔科姆·麦克莱恩	第4章和第39章
马丁·弗兰克	第21章
梅赫迪·哈桑	第5章
米尔延·米基奇	第2章、第28章和第39章
米纳斯·卡马尔	第12章和第46章
mnoronha	第23章、第29章、第31章、第32章、第33章、第34章、第35章、第36章和第45章
msohng	第39章
尼克·拉尔森	第二章

Credits

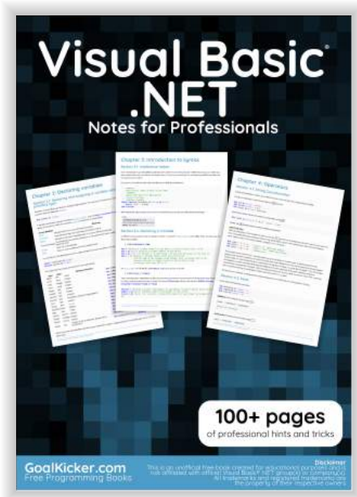
Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Abdul Karim	Chapter 1
afeldspar	Chapter 43
Ahmad Faiyaz	Chapter 28
Alber Tadrous	Chapter 53
Anagh Hegde	Chapters 29 and 39
Andrii Artamonov	Chapter 27
AnukuL	Chapter 40
Bakhtiar Hasan	Chapters 9, 11, 14, 17, 19, 20, 22, 40, 41, 42, 47, 52 and 54
Benson Lin	Chapters 14, 39 and 44
brijs	Chapter 39
Chris	Chapter 15
Creative John	Chapters 49 and 51
Dian Bakti	Chapter 10
Didgeridoo	Chapters 2 and 43
Dipesh Poudel	Chapter 21
Dr. ABT	Chapter 55
EsmaeelE	Chapters 2, 29, 30, 39 and 50
Filip Allberg	Chapters 1 and 9
ghilesZ	Chapter 17
goeddek	Chapters 18 and 27
greatwolf	Chapter 5
Ijaz Khan	Chapter 29
invisal	Chapter 31
Isha Agarwal	Chapters 4, 5, 6, 7 and 8
Ishit Mehta	Chapter 5
IVlad	Chapters 16 and 28
Iwan	Chapter 30
Janaky Murthy	Chapter 6
JJTO	Chapter 9
Julien Rousé	Chapter 24
Juxhin Metaj	Chapters 2 and 30
Keyur Ramoliya	Chapters 23, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 45, 47, 48 and 50
Khaled.K	Chapter 39
kiner_shah	Chapter 12
lambda	Chapter 38
Luv Agarwal	Chapter 30
Lymphatus	Chapter 31
M S Hossain	Chapter 17
Malav	Chapter 33
Malcolm McLean	Chapters 4 and 39
Martin Frank	Chapter 21
Mehedi Hasan	Chapter 5
Miljen Mikic	Chapters 2, 28 and 39
Minhas Kamal	Chapters 12 and 46
mnoronha	Chapters 23, 29, 31, 32, 33, 34, 35, 36 and 45
msohng	Chapter 39
Nick Larsen	Chapter 2

程序员尼克	第三章
乐观的阿努普	第29章和第33章
彼得·K	第二章
拉希克·哈斯纳特	第40章
罗伯托·费尔南德斯	第12章
三角	第29章
塞缪尔·彼得	第三章
圣地亚哥·吉尔	第30章
Sayakiss	第9章和第14章
沙尔马	第30章
ShreePool	第39章
舒布哈姆	第16章
苏米特·辛格	第20章和第41章
TajyMany	第12章和第13章
特朱斯·普拉萨德	第2、5、9、11、18、19和45章
theJollySin	第17章
umop apisdn	第39章
User0911	第29章
user23013	第9章
VermillionAzure	第4和9章
维什瓦斯	第14、25和26章
WitVault	第三章
xenteros	第17章、第29章和第39章
亚伊尔·特维托	第二章
yd1	第4章
耶尔肯	第16章和第20章
年轻的霍比特人	第29章

Nick the coder	Chapter 3
optimistanoop	Chapters 29 and 33
Peter K	Chapter 2
Rashik Hasnat	Chapter 40
Roberto Fernandez	Chapter 12
samgak	Chapter 29
Samuel Peter	Chapter 3
Santiago Gil	Chapter 30
Sayakiss	Chapters 9 and 14
SHARMA	Chapter 30
ShreePool	Chapter 39
Shubham	Chapter 16
Sumeet Singh	Chapters 20 and 41
TajyMany	Chapters 12 and 13
Tejus Prasad	Chapters 2, 5, 9, 11, 18, 19 and 45
theJollySin	Chapter 17
umop apisdn	Chapter 39
User0911	Chapter 29
user23013	Chapter 9
VermillionAzure	Chapters 4 and 9
Vishwas	Chapters 14, 25 and 26
WitVault	Chapter 3
xenteros	Chapters 17, 29 and 39
Yair Twito	Chapter 2
yd1	Chapter 4
Yerken	Chapters 16 and 20
YoungHobbit	Chapter 29

你可能也喜欢



You may also like