Notes for Introduction to Algorithms

算法导论笔记

- The notes are sorted according to the teaching order of NEU CS 5800 **Prof. Virgil Pavlu**, some chapters of the original book are skimmed.
- 笔记按照NEU CS 5800 **Prof. Virgil Pavlu** 的教授顺序来排序，有掠过原书的一些章节。
- 如果不支持markdown可以下载pdf。
- 持续更新中，有一些图片还未上传到图床。
- notes by `Yiqiu Huang`

3. Growth of Funcitons

3.1 Asymptotic notation: 算法的渐进表示

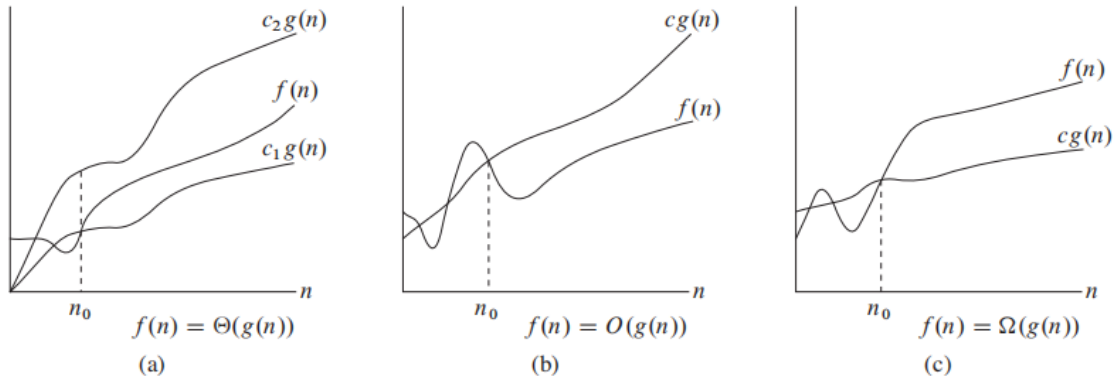
根据算法导论的原文：

> T(n) = number of computational steps required to run the algorithm/program for input of size n

也就是, $T(n)$ 代表了给定input size: n 以后的计算步骤 (computational steps)；

- 但是我们在意的是算法伴随 n 的增长，而不是具体的步骤数量；
  - 比如 $T(n) = \Theta(n^2)$ 代表了quadratic running time
  - $T(n) = O(nlogn)$ 说明 $T(n)$ **至多**是n的 $nlog(n)$倍数；

渐进分析 Asymptotic notation:



图：Θ、O、Ω 的含义

最简单的理解他们的方法就是,:

Θ :代表了**等于号**，比如$\Theta(f(n))$代表函数 f 的"运行时间"等于 $T(n)$；

O:代表了**小于号**，比如$O(f(n))$代表函数 f 的"运行时间"**至多是** $T(n)$；

Ω: 代表了**大于号**，比如$\Theta(f(n))$代表函数 f 的"运行时间"**至少是** $T(n)$；

以O 为例，看图(b) ，当 input size n 到达一定规模 n_0 以后, c是一个常数

3.1.1 Θ Notation

原文定义如下：

> For a given function g(n), we denote by $\Theta(g(n))$ the **set of functions:**
>
> $\Theta(g(n))$ = {$f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all n $\geq$ n0 }

> 如果满足如上条件，我们说 g(n) is an **asymptotically tight bound** for f(n)

简单来说就是 n 到一定大小以后 ($n \geq n_0$,n_0是个常数), **在常数范围内**, $f(n) = g(n)$,($0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$), 看上图(a)。

注意，$\Theta(g(n))$ 本身**描述的是一个集合**，用来表示所有$\Theta(n)$运行时间的函数集合，所以你可以这么写：

$$f(n) \in \Theta(n)$$

不过习惯上这么写：

$$f(n) = \Theta(n)$$

这样写有自己独特的优势。

Θ Notation 是 **asymptotically tight bound**，最简单的说，Θ notation 就像是 等于号

$$\Theta :=$$

3.1.2 O Notation

原文定义如下：

> For a given function g(n), we denote by O(g(n)) the set of functions:
>
> $O(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n $\geq$ n0 }
>
> 如果满足如上条件，我们说 g(n) is an **asymptotically upper bound** for f(n)

简单来说就是n到一定大小以后($n \geq n_0$,n_0是个常数), **在常数范围内**, $f(n) \leq g(n)$, 看图(b)。

最简单的说，O notation 就像是小于号，它描述的是上界。

$$O :\leq$$

> 平时我们习惯用大O表示法来表示runtime，比如寻找一个数组的最大值，我们说runtime 是O(n)；
>
> 这么说不完全严谨，因此O(n)描述的是上界(upper bound)，无法说明how tight that bound is, 你说寻找最大值的算法是$O(n^2)$,$O(n^3)$也没问题。

3.1.3 Ω Notation

原文定义如下：

> For a given function g(n), we denote by Ω(g(n)) the set of functions:
>
> $\Omega(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }
>
> 如果满足如上条件，我们说 g(n) is an **asymptotically lower bound** for f(n)

简单来说就是n到一定大小以后($n \geq n_0$,n_0是个常数), **在常数范围内**, $f(n) \geq g(n)$, 看图(b)。

$$\Omega :\geq$$

3.2 例题

1.

3.1-1

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

答案:

To prove this, we have to show that there exists constants $c_1, c_2, n_0 > 0$ such that for all $n \geq n_0$,

$$0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$$

As the functions are asymptotically non-negative, we can assume that for some $n_0 > 0$, $f(n) \geq 0$ and $g(n) \geq 0$. Therefore, $n \geq n_0$,

$$f(n) + g(n) \geq \max(f(n), g(n))$$

Also note that, $f(n) \leq \max(f(n), g(n))$ and $g(n) \leq \max(f(n), g(n))$

$$f(n) + g(n) \leq 2\max(f(n), g(n))$$
$$\frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n))$$

Therefore, we can combine the above two inequalities as follows:

$$0 \leq \frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq (f(n) + g(n)) \text{ for } n \geq n_0$$

So, $\max(f(n), g(n)) = \Theta(f(n) + g(n))$ because there exists $c_1 = 0.5$ and $c_2 = 1$.

本质上，你要证明常数存在，来证明公式正确。

2.

3.1-4

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

类似的技巧：

a. Is $2^{n+1} = O(2^n)$?

Yes.
$O(2^n) \Rightarrow 0 \leq f(n) \leq C_1 \cdot 2^n$
There exists constants like $c_1 = 4$ such that
$c_1 \cdot 2^n \geq 2^n \cdot 2 \geq 0$

b. Is $2^{2 \cdot n} = O(2^n)$?

No.
suppose $2^n \cdot 2^n = O(2^2)$
Then there is a constant c_1, such that
$c_1 \cdot 2^n \geq 2^n \cdot 2^n \geq 0$
Since 2^n is unbounded, such constant c_1 does not exist.

2.3.1 Divide-and-conquer

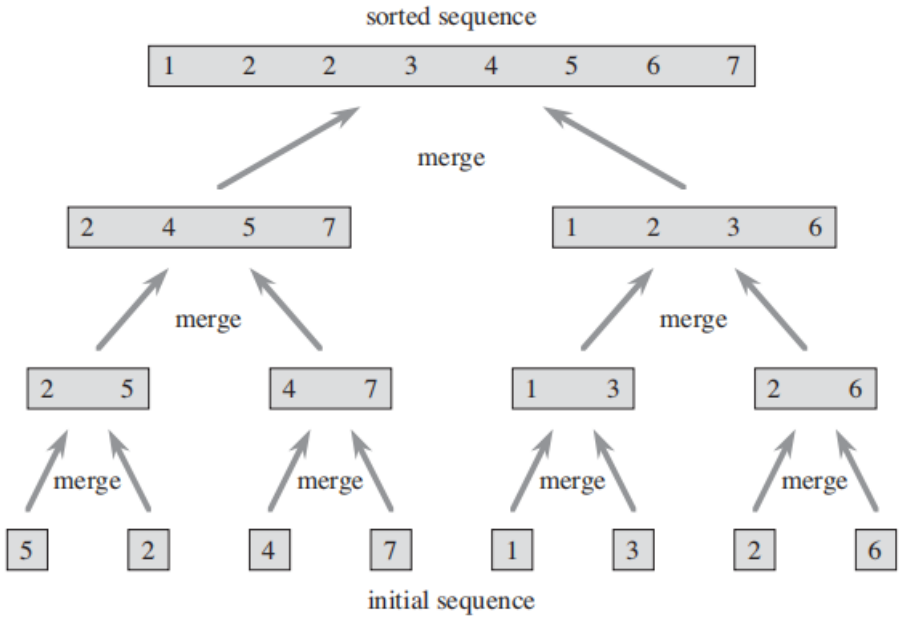
分治法：

> **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
>
> **Conquer** the subproblems by solving them recursively.
>
> **Combine** the solutions to the subproblems into the solution for the original problem.

与我们熟悉的 **Mergesort** 一一对应：

> **Divide**: 把长度为 n 的序列切分为两个长度为 $\frac{n}{2}$ 的序列
>
> **Conquer**: 递归的调用mergesort来sort sequence。(原文：Sort the two subsequences recursively using merge sort)
>
> **Combine:** 合并两个sorted sequence

2.3.2 Analyzing divide-and-conquer algorithms



Suppose that our division of the problem yields **a** subproblems, each of which is **1/b** the size of the original.

- **注意，Mergesort的a = b = 2**, 很多分治法a 并不等于 b，a 更不等于 2

- 注意后面这句话: each of which is **1/b** the size of the original 他的意思是每个子问题的size都是原来的1/b，那么随着递归的进行，子问题的size就是:

$$\frac{n}{b} -> \frac{n}{b^2} -> \frac{n}{b^3} ->\cdots \frac{n}{b^?}$$

用白话来说就是，mergesort产生了2个size是(n/2)的subproblem。

假设mergesort的runtime 是**T(n)**,我们有 **a** 个size 为**b/n** 的子问题，因此我们需要 **aT(n/b)** 来解决他们。

除此之外，我们需要 D(n) 来divide the problem into subproblem, 以及 C(n) 来combine我们的 solution from all subproblem, 我们的总时间为:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \ . \end{cases}$$

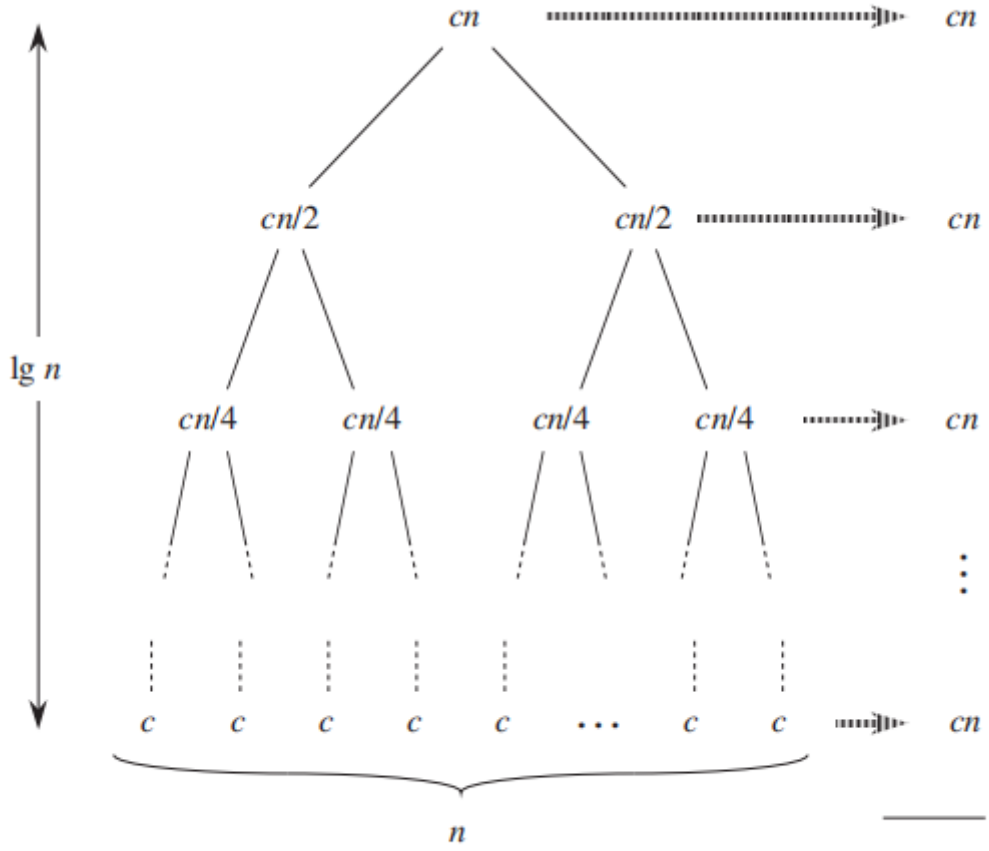
这段比较重要，还是原汁原味吧:

> **Divide:** The divide step just computes the middle of the subarray, which takes
>
> constant time. Thus, D.n/ D ,.1/.
>
> **Conquer:** We recursively solve two subproblems, each of size n=2, which contributes 2T(n/2) to the running time.
>
> **Combine:** We have already noted that the MERGE procedure on an n-element
>
> subarray takes time ,.n/, and so C.n/ D ,.n/.

> **Divide:** mergesort 在 diide 时只计算出中位数，所以是contant time, D(n) = 1
>
> **Conquer:** 我们递归的解决两个子问题，每一个子问题的size是 n/2, 因此是 2T(n/2)
>
> **Combine:** combine的本质就是遍历，谁小谁先进sorted array, 因此 $C(n) = \Theta(n)$

因此:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \ , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \ . \end{cases}$$



2.3.3 求解分治法的时间复杂度

递归法往往能写成以下的格式：

Recurrence examples

- T(n) = 2T(n/2) + O(1)
- T(n)= 2T(n/2) + O(n)
  - 2 subproblems of size n/2 each, plus O(n) steps to combine results
- T(n) = 4T(n/3) + n
  - 4 subproblems of size n/3 each, plus n steps to combine results
- T(n/4) + T(n/2) +n^2
  - a subproblem of size n/3, another of size n/2; n^2 to combine
- want to solve such recurrences, to obtain the order of growth of function T

主要有三种方法求解分治法的时间复杂度

> In the **substitution method**, we guess a bound and then use mathematical in□
> duction to prove our guess correct.
>
> The **recursion-tree method** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
>
> The **master method** provides bounds for recurrences of the form
> $T(n) = aT(\frac{n}{b}) + f(n)$

1. subtitution method

Substitution method

- T(n) = 4T(n/2) + n
- STEP1 : guess solution, order of growth T(n)= $O(n^3)$
  - that means there is a constant C and a starting value n_0, such that $T(n) \le Cn^3$, for any $n \ge n_0$
- STEP2: verify by induction
  - assume $T(k) \le k^3$, for k<n
  - induction step: prove that $T(n) \le Cn^3$, using $T(k) \le Ck^3$, for k<n

$$
\begin{aligned}
T(n) &= 4T(\tfrac{n}{2}) + n && (1)\\
&\le 4c\left(\frac{n}{2}\right)^3 + n && (2)\\
&= \frac{c}{2}n^3 + n && (3)\\
&= cn^3 - (\frac{c}{2}n^3 - n) && (4)\\
&\le cn^3; \quad if\ \frac{c}{2}n^3 - n > 0, choose \quad c \ge 2 && (5)
\end{aligned}
$$

- STEP 3 : identify constants, in our case c=2 works
- so we proved $T(n)=O(n^3)$
- thats correct, but the result is too weak
  - technically we say the bound $O(n^3)$ "cubic" is too lose
  - can prove better bounds like T(n) "quadratic" $T(n)=O(n^2)$
  - Our guess was wrong ! (too big)
- lets try again : STEP1: guess $T(n)=O(n^2)$
- STEP2: verify by induction
  - assume $T(k) \le Ck^2$, for k<n
  - induction step: prove that $T(n) \le Cn^2$, using $T(k) \le Ck^2$, for k<n

- Fallacious argument

$$
\begin{aligned}
T(n) &= 4T(\tfrac{n}{2}) + n && (1)\\
&\leq 4c\left(\tfrac{n}{2}\right)^2 + n && (2)\\
&= cn^2 + n && (3)\\
&= O(n^2) && (4)\\
&\leq cn^2 && (5)
\end{aligned}
$$

  - cant prove T(n)=O(n²) this way: need same constant steps 3-4-5
  - maybe its not true? Guess O(n²) was too low?
  - or maybe we dont have the right proof idea

- common trick: if math doesnt work out, make a stronger assumption (subtract a lower degree term)
  - assume instead $T(k) \leq C_1k^2 - C_2k$, for k<n

$$
\begin{aligned}
T(n) &= 4T(\frac{n}{2}) + n\\
&\leq 4\left(c_1\left(\frac{n}{2}\right)^2 - c_2\frac{n}{2}\right) + n\\
&= c_1n^2 - 2c_2n + n\\
&= c_1n^2 - c_2n - (c_2n - n)\\
&\leq c_1n^2 - c_2n \quad for \quad c_2 > 1
\end{aligned}
$$

- So we can prove T(n)=O(n²), but is that asymptotically correct ?
  - maybe we can prove a lower upper bound, like O(nlogn)? NOPE
- to make sure its the asymptote, prove its also the lower bound
  - T(n) =Ω(n²) or there is a different constant d s.t. T(n) ≥ dn²

- induction step $T(n) = 4T(\frac{n}{2}) + n$

$$\geq 4d\left(\frac{n}{2}\right)^2 + n$$

$$= dn^2 + n \geq dn^2$$

- now we know its asymptotically close, $T(n)=\Theta(n^2)$
- hard to make the initial guess $\Theta(n^2)$
  - need another method to educate our guess

2. recursion-tree method

Iteration method

$$T(n) = n + 4T(\frac{n}{2})$$

$$= n + 4(\frac{n}{2} + 4T(\frac{n}{4})) = n + 2n + 4^2T(\frac{n}{2^2})$$

$$= n + 2n + 4^2(\frac{n}{2^2} + 4T(\frac{n}{2^3})) = n + 2n + 2^2n + 4^3T(\frac{n}{2^3})$$

$$= ....$$

$$= n + 2n + 2^2n + ... + 2^{k-1}n + 4^kT(\frac{n}{2^k})$$

$$= \sum_{i=0}^{k-1} 2^in + 4^kT(\frac{n}{2^k});$$

$want\ k = \log(n) \Leftrightarrow \frac{n}{2^k} = 1$

$$= n\sum_{i=0}^{\log(n)-1} 2^i + 4^{log(n)}T(1)$$

$$= n\frac{2^{log(n)} - 1}{2 - 1} + n^2T(1)$$

$$= n(n-1) + n^2T(1) = \Theta(n^2)$$

- math can be messy
  - recap sum, product, series, logarithms
  - iteration method good for guess, but usually unreliable for an exact result
  - use iteration for guess, and substitution for proofs
- stopping condition
  - T(...) = T(1), solve for k

3. Master Theorum

使用**Master Theorum**可以快速求出常见递归的时间复杂度。

比较重要，比较实用，比较递归参数的大小得到时间复杂度：

Master Theorem - simple

- simple general case T(n) = aT(n/b) + $\Theta(n^c)$
- R=a/b^c, compare R with 1, or c with $\log_b(a)$

Case 1:	$c < \log_b a$	$T(n) = \Theta(n^{\log_b a})$
Case 2:	$c = \log_b a$	$T(n) = \Theta(n^c \log n) = \Theta(n^{\log_b a} \log n)$
Case 3:	$c > \log_b a$	$T(n) = \Theta(n^c)$

- MergeSort T(n) = 2T(n/2) + $\Theta(n)$; a=2 b=2 c=1
  case 2 ; T(n) = $\Theta(n\log n)$
- Strassen's T(n) =7T(n/2) + $\Theta(n^2)$; a=7 b=2 c=2
  case 1, T(n)= $\Theta(n\log_2(7))$
- Binary Search T(n)=T(n/2) + $\Theta(1)$; a=1 b=2 c=0
  case 2, T(n)= $\Theta(\log n)$

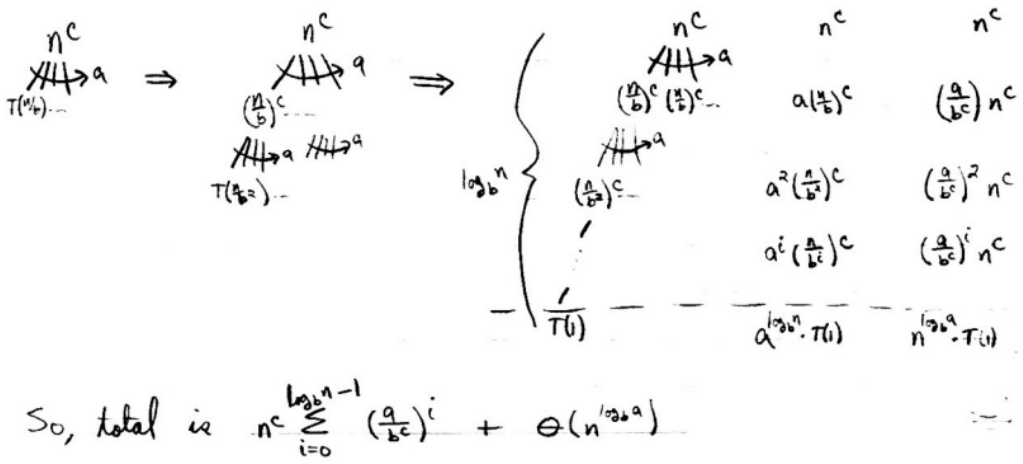
Binary search, Mergesort 的runtime 可以轻松的求出。

3.1 Why MT have 3 cases

首先求出递归的数学表达：

$$T(n) = aT(n/b) + n^c \quad \text{(for simplicity, eliminate } \Theta\text{)}$$

Recursion tree:



So, total is $n^c \sum_{i=0}^{\log_b n - 1} (\frac{a}{b^c})^i + \Theta(n^{\log_b a})$

对于关键项

$$\left(\frac{a}{b^c}\right)^i$$

来说，**有三种表达 <1,=1,>1,so three**

Case 1 $c < \log_b a \iff \frac{a}{b^c} > 1$ - work increases geometrically

$$\text{sum} = n^c \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i + \Theta(n^{\log_b a})$$

$$= n^c \frac{\left(\frac{a}{b^c}\right)^{\log_b n} - 1}{\left(\frac{a}{b^c}\right) - 1} + \Theta(n^{\log_b a})$$

$$(b^c)^{\log_b n} \quad = \quad \Theta\left(n^c \frac{a^{\log_b n}}{(b^c)^{\log_b n}}\right) + \Theta(n^{\log_b a})$$

$$\| \quad (b^{\log_b n})^c$$

$$b^{c \cdot \log_b n} \quad = \quad \Theta\left(n^c \frac{n^{\log_b a}}{n^c}\right) + \Theta(n^{\log_b a})$$

$$n^c \quad = \quad \Theta(n^{\log_b a})$$

∴ work at each level increases geometrically;
constant fraction of work is in leaves...

Case 2 $c = \log_b a \iff a/b^c = 1$ - work constant at each level

$$\text{sum} = n^c \sum_{i=0}^{\log_b n - 1} (1)^i + \Theta(n^{\log_b a}) = n^c \log_b n + \Theta(n^{\log_b a}) = \Theta(n^c \log_b n)$$

∴ work at each level is n^c $(= n^{\log_b a})$; $\log_b n$ levels;
answer is $\Theta(n^c \log_b n)$

Case 3 $c > \log_b a \iff \frac{a}{b^c} < 1$ - work decreases geometrically

$$\text{perm} = n^c \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i + \Theta\left(n^{\log_b a}\right)$$

$$= n^c \, \Theta(1) + \Theta\left(n^{\log_b a}\right)$$

$$= \Theta(n^c) + \Theta(n^{\log_b a})$$

$$= \Theta(n^c)$$

Note:

① $\sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i \geq \left(\frac{a}{b^c}\right)^0 = 1$

$\Rightarrow \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i = \Omega(1)$

② $\sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i < \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i$

$= \frac{1}{1 - a/b^c}$ (constant)

$\Rightarrow \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i = O(1)$

$\therefore \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i = \Theta(1)$

$\therefore$ work at each level decreases geometrically;
constant fraction of work is at root...

6. Sorting: 排序算法

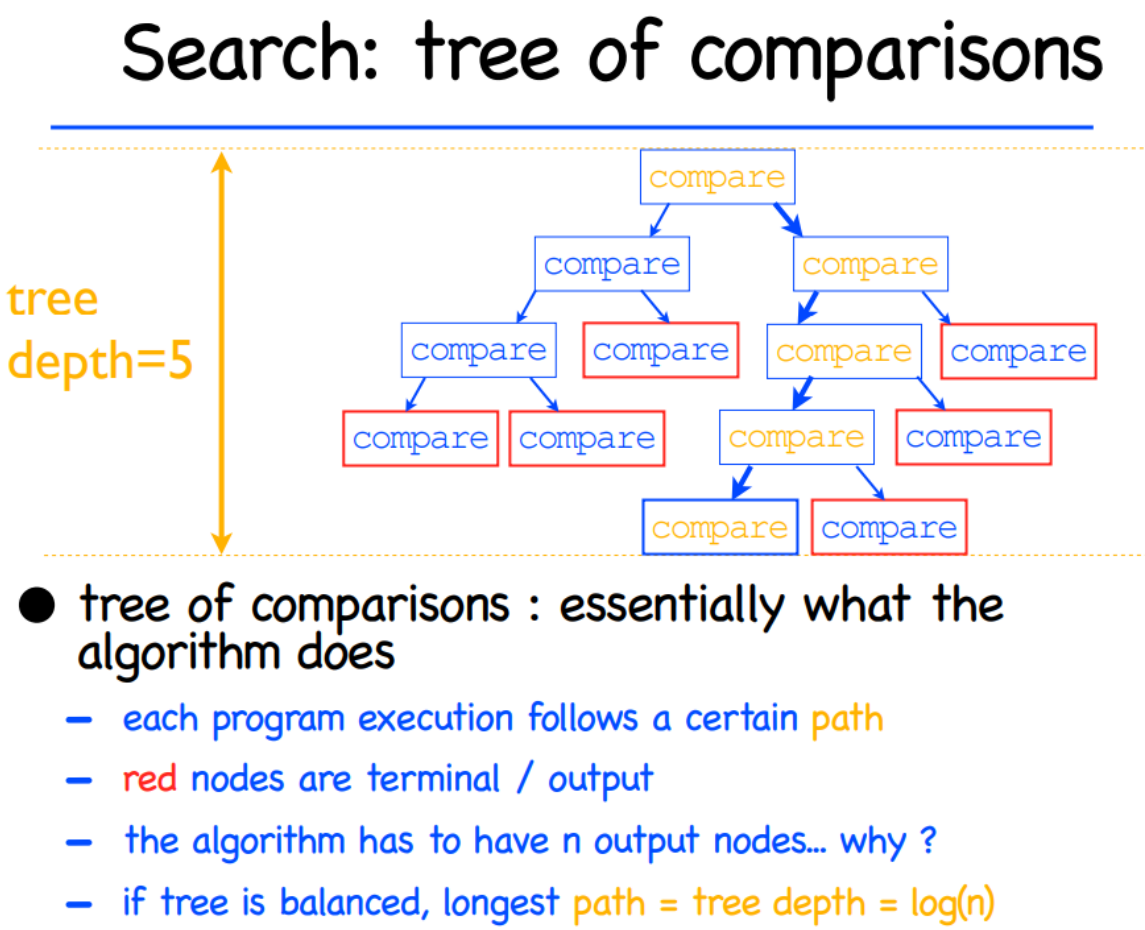
	time	in-place	stable
Bubble	n^2	✔	✔
Insertion	n^2	✔	✔
Selection	n^2	✘	?
QuickSort	n*log(n)	✔	?
MergeSort	n*log(n)	✘	✔

6.1. Binary Search

算法本身不做介绍。

worst running time is O(log n)

二叉树的本质是一种**比较算法**, 在这里先引入这个概念:



6.2. Selection Sort/Bubble sort/Insertion sort

Insertion sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

6.3 Heap sort: 堆排序

6.3.1 MAX-HEAPIFY

最大堆性质：

In a **max-heap**, the **max-heap property** is that for every node i other than the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

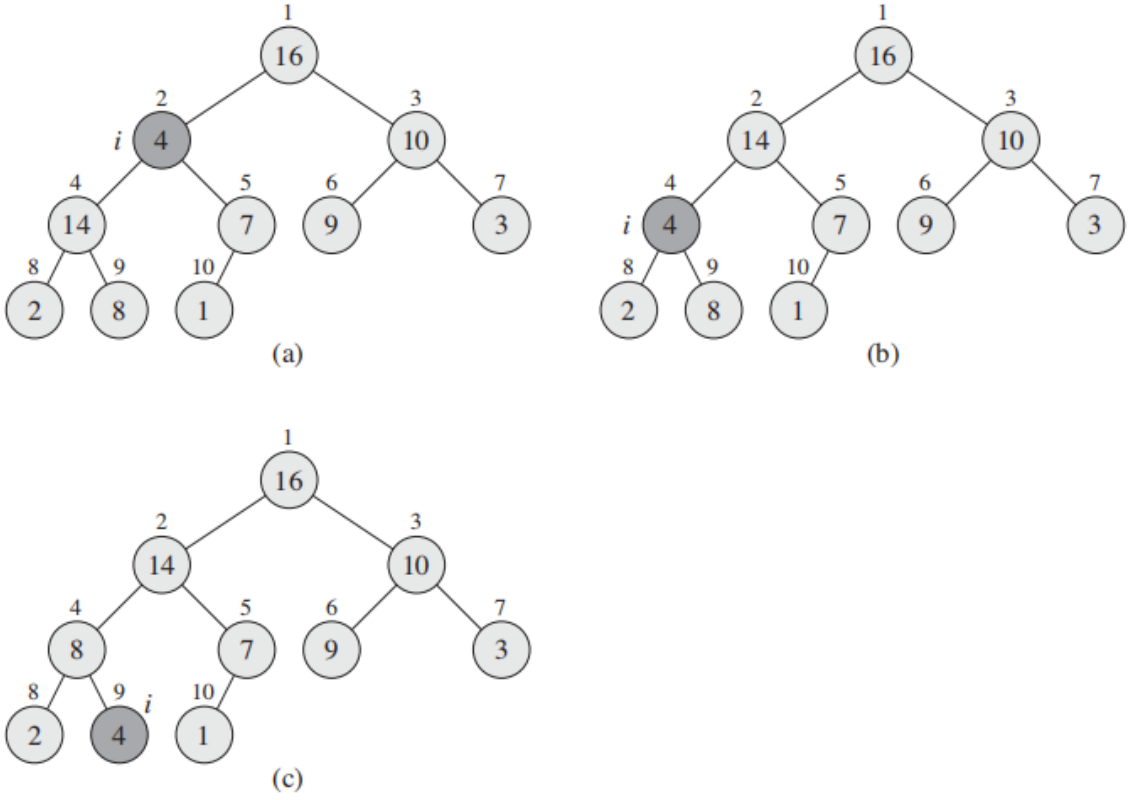
最小堆性质与之相反；

为了满足最大堆性质，你需要调用:

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

这段代码的逻辑就是找出 $A[i], A[LEFT(i)], A[RIGHT(i)]$的最大值；

- 如果最大值是i，那么左右子树小于i, 满足max-heap property,无事发生。
- 如果不是，将i与largest 交换；交换后当前node满足max-heap, 但是子树不一定满足；因此对子树继续进行max-heapify



MAX-HEAPIFY 复杂度分析：

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \le T(2n/3) + \Theta(1) .$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

6.3.2 Build-max-heap

根据heap(二叉树)的性质，$A[(\frac{n}{2}+1)\ldots n]$都是叶节点；因此从$\frac{n}{2}$ downto 1进行heapify

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

这样就完成了BUILD-MAX-HEAP。

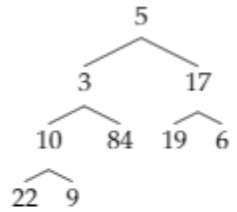
下面来自例题：

6.3-1

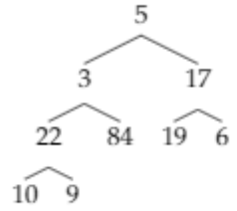
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

答案：

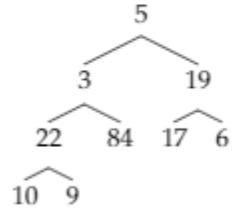
Visualize the array with heap:



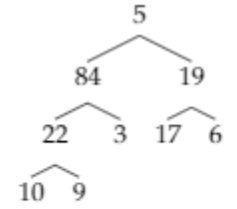
The heapify process start at the last non-leaf node, which is 10.We swap 10 and 22 this time. After that:



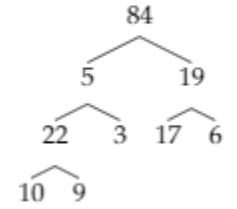
Then, heaify will move to node with value 17, and swap 17 and 19



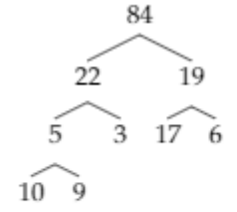
Then, heaify will move to node with value 3, and swap 3 and 84



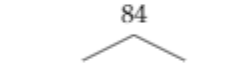
Then, heaify will move to root node 5, and swap 5 and 84

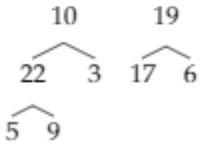


Then, heaify will move to root node 5, and swap 5 and 22



Then, heaify will move to root node 5, and swap 5 and 22

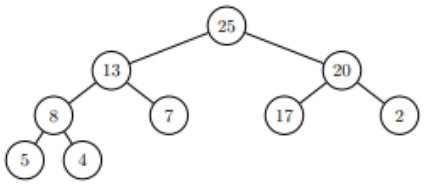




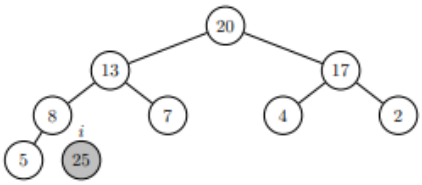
6.3.3

Heapify -> 得到最大值 ->换到最后一位，不再管他，size - 1 -> 循环

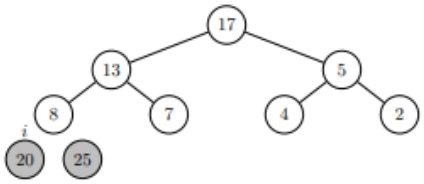
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i = A.length$ **downto** 2
3 exchange $A[1]$ with $A[i]$
4 $A.heap\text{-}size = A.heap\text{-}size - 1$
5 MAX-HEAPIFY(A, 1)



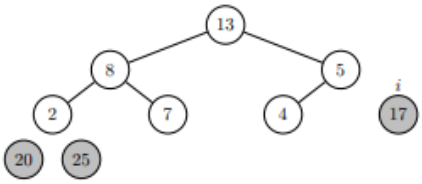
Step 0



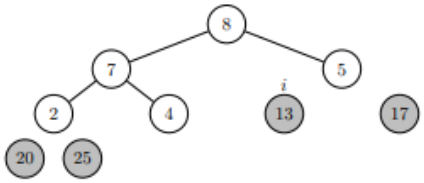
Step 1



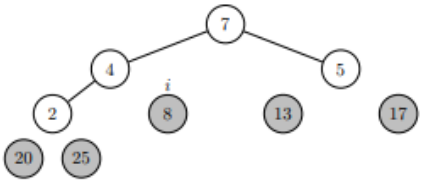
Step 2



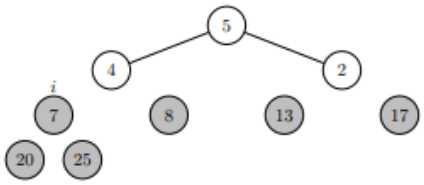
Step 3



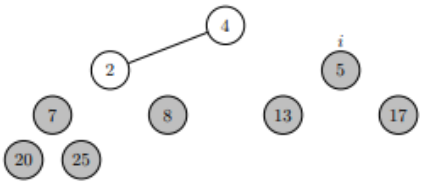
Step 4



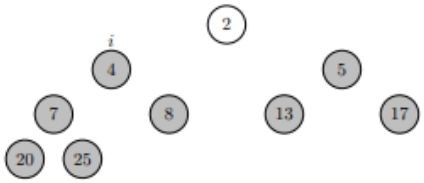
Step 5



Step 6



Step 7



Step 8

6.4 Quicksort

A divide-and-conquer algorithm with worst-case running time of $\Theta(n^2)$,expected in $O(n \cdot lgn)$.

Divide: Partition (rearrange) the array $A[p\,..\,r]$ into two (possibly empty) subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ such that each element of $A[p\,..\,q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1\,..\,r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

6.4.1 Partition

Partition 是 quicksort 最重要的机制。

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Partition 会选择 x = A[r] 作为 pivot element。你要理解的是在Partition的过程中，有四个区域，先理解这个思路，伪码就能看懂了：

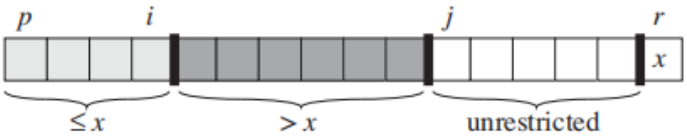


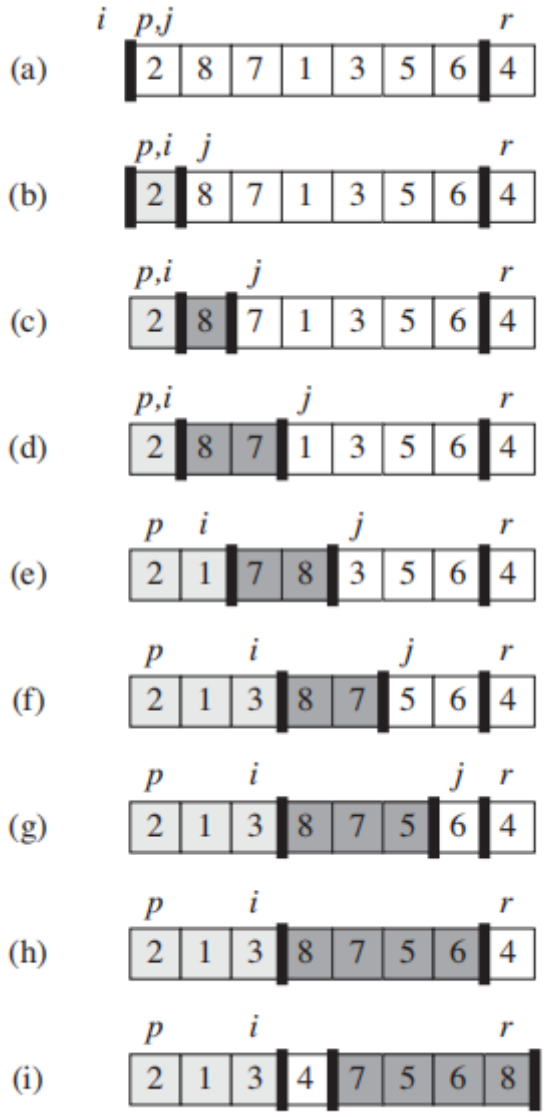
Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.

[j,r-1] 还没看到的区域

[p, i] 小于pivot 的区域

[i,j] 大于pivot的区域

[r] **pivot element**



注意，partition返回的是i + 1.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements.

7. Linear Sort

7.1. Counting Sort

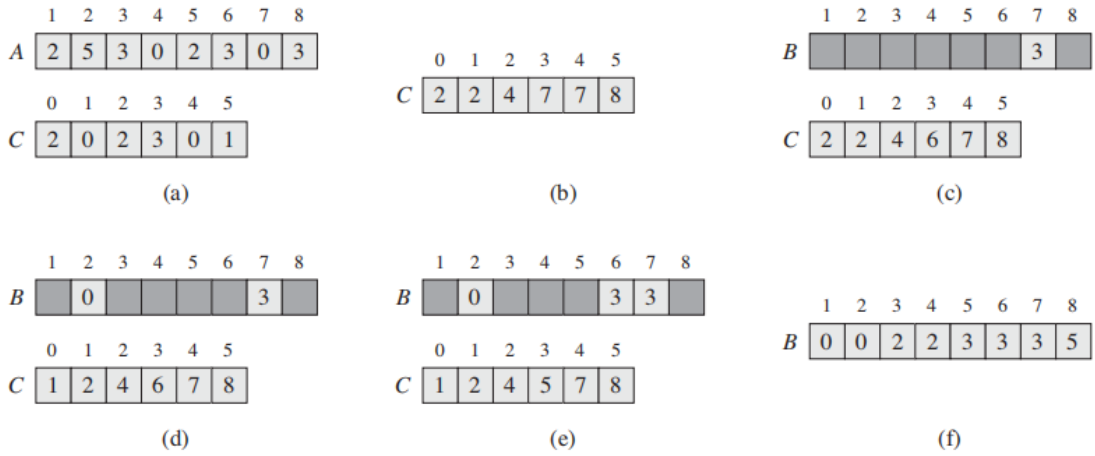
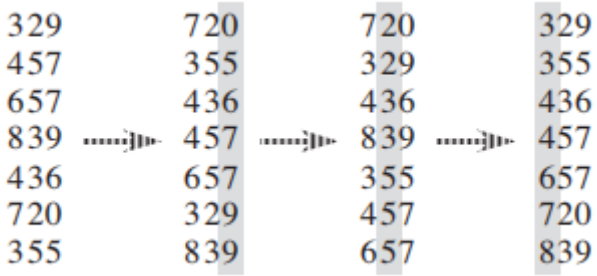


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

7.2. Radix sort



```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

7.3. Bucket Sort

老师讲的很少，这块不复习了。

8.3-3

> Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Loop invariant: At the beginning of the **for** loop, the array is sorted on the last $i-1$ digits.

Initialization: The array is trivially sorted on the last 0 digits.

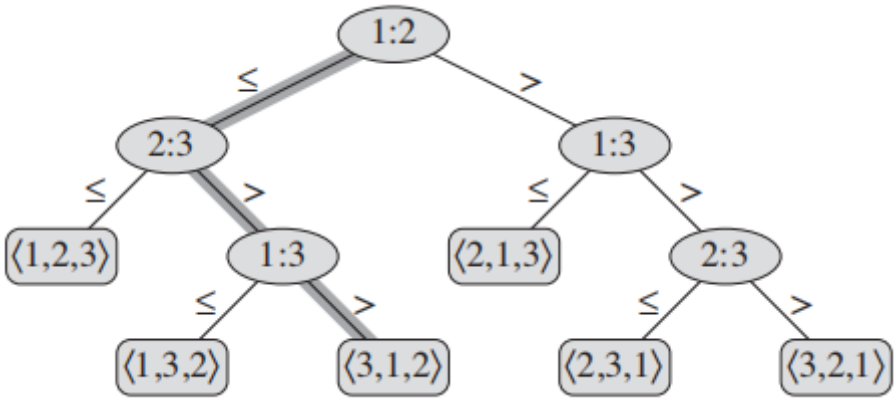
Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the ith digit, the array will be sorted on the last i digits. It is obvious that elements with different digit in the ith position are ordered accordingly; in the case of the same ith digit, we still get a correct order, because we're using a stable sort and the elements were already sorted on the last $i-1$ digits.

Termination: The loop terminates when $i = d + 1$. Since the invariant holds, we have the numbers sorted on d digits.

8.1 Lower bounds for sorting

为什么主流comparison sort(比较算法,如快排，mergesort,heapsort)的time complexity 为nlogn?

常见的算法主要是比较算法(**comparison sort**)，比如mergesort在merge的部分需要比较两个数字的大小来排序。comparison sort 可以用下图的例子来类比：



求出树的高度h**(i.e., 抵达leaf的worst case时间)**,就是排序的worst case时间。

输入n的permutation 为n!, 参考下图的决策树，运行时间即为h，可以得出

$$h = \Omega(nlgn)$$

摘一段原文：

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$

■

为什么 $lg(n!) = \Omega(n \cdot lgn)$? 下面给出书中的证明。（看看就得了）

公式3.19: Factorial 的时间复杂度

$$
\begin{aligned}
n! &= o(n^n)\,, \\
n! &= \omega(2^n)\,, \\
\lg(n!) &= \Theta(n \lg n)
\end{aligned}
$$

Some intuition about why:

$$lg(n!) = \Theta(n \cdot lgn)$$

你需要记住

where Stirling's approximation is helpful in proving equation (3.19). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.20}$$

and

$$
\begin{aligned}
\lg(n!) &\approx \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\
&= \lg\left(\sqrt{2\pi n}\right) + \lg\left(\frac{n}{e}\right)^n \\
&= \lg\sqrt{2\pi} + \lg\sqrt{n} + n\lg\left(\frac{n}{e}\right) \\
&= \lg\sqrt{2\pi} + \frac{1}{2}\lg n + n\lg n - n\lg e \\
&= \Theta(1) + \Theta(\lg n) + \Theta(n\lg n) - \Theta(n) \\
&= \Theta(n\lg n)
\end{aligned}
$$

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

9 Medians and Order Statistics

这一章节主要find i_th smallest number的问题

在选择最小值的算法上，我们都知道:

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

遍历数组，经过n-1次的比较我们可以找到最小值。***This is the best we could do.***

9.2 Selection in expected linear time

The general selection problem appears more diffificult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same:

$$\Theta(n)$$

RANDOMIZED-SELECT类似quickSort,也是一种**divide-and-conquer** alg.

The following code for **RANDOMIZED-SELECT** returns the **i** smallest element of array A[p...r]

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

- Line 1 & Line 2: 如果array长度为1，那直接返回array中的值
- Line3: 类似quickSort的partition, 选择pivot，让A[p...q-1]的小于pivot, 让A[q+1,r]的值大于pivot, **A[q]**就是pivot number
- Line4: 计算k(即A[p...q-1]的长度 + 1)
- Line5: 如果 i == k,即你要的i th 等于 k,找到了答案，返回A[q]

- Line7, 8,9: 否则，根据 i与k的关系继续调用**RANDOMIZED-SELECT**(if i>k, the desired element will lies in high partition part)

The worst-case running time for RANDOMIZED-SELECT is

$$\Theta(n^2)$$

9.3 Select algorithm with worst case runtime O(n)

select algorithm 应该就是median-of-medians,但是没有在wiki上得证。

大致思路:

- Divide the n elements of the input array into n/5 groups of 5 elements each gourp
- Find the median of each of the n/5 groups by insertion sorting

通常找第 i^{th} 小的数字比找最小的数字更难，而The Median-of-medians Algorithm的时间复杂度:**O(n)。**

The Median-of-medians/Select Algorithm(找第 i^{th} 小的数字)

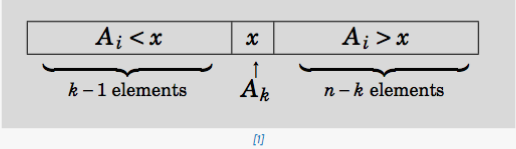
> Use a **divide and conquer** strategy to efficiently compute **the i^th smallest number** in an unsorted list of size n.

好文：https://brilliant.org/wiki/median-finding-algorithm/

Median-of-medians Algorithm[1]

The algorithm takes in a list and an index— `median-of-medians(A, i)` . Assume that all elements of A are distinct (though the algorithm can be further generalized to allow for duplicate elements).

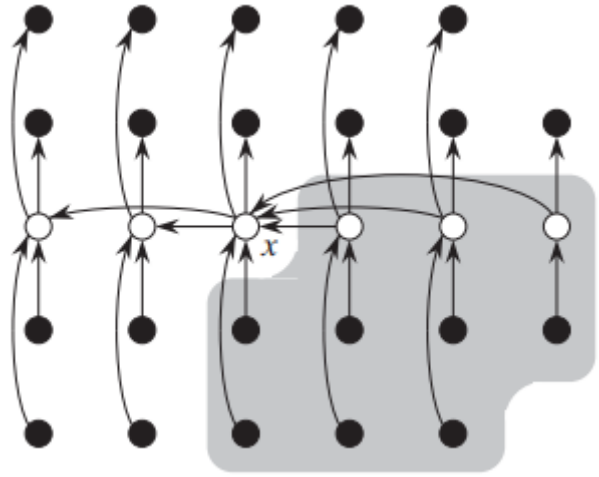
1. Divide the list into sublists each of length five (if there are fewer than five elements available for the last list, that is fine).
2. Sort each sublist and determine the median. Sorting very small lists takes linear time since these sublists have five elements, and this takes $O(n)$ time. In the algorithm described on this page, if the list has an even number of elements, take the floor of the length of the list divided by 2 to find the index of the median.
3. Use the median-of-median algorithm to recursively determine the median of the set of all the medians.
4. Use this median as the pivot element, x. The pivot is an approximate median of the whole list and then each recursive step hones in on the true median.
5. Reorder A such that all elements less than x are to the left of x, and all elements of A that are greater than x are to the right. This is called partitioning. The elements are in no particular order once they are placed on either side of x. For example, if x is 5, the list to the right of x maybe look like $[8, 7, 12, 6]$ (i.e. not in sorted order). This takes linear time since $O(n)$ comparisons occur—each element in A is compared against x only.



[1]

6. Let k be the "rank" of x, meaning, for a set of numbers S, x is the k^{th} smallest number in S.

7a. If $i = k$, then return x.
7b. If $i < k$, then recurse using median-of-medians on $(A[1, \ldots, k-1], i)$.
7c. If $i > k$, recurse using the median-of-medians algorithm on $(A[k+1, \ldots, i], i-k)$.



EXAMPLE of Median-Median Algorithm：来跟一遍例子就懂了：

给定A的，找到4th smallest element:

$$A = [25, 21, 98, 100, 76, 22, 43, 60, 89, 87]$$

把A分成**n/5**份，保证每个subgroup有5个元素

$$A_1 = [25, 21, 98, 100, 76] \quad \text{and} \quad A_2 = [22, 43, 60, 89, 87].$$

分别求中位数：

$$M = [76, 60].$$

Sort M:

$$M = [60, 76].$$

求出M 中的中位数 len(M)/2 = 1, which is **76** (这一步就是所谓median-of-medians)

使用**76**作为pivot, **partition(A)**，左边元素小于pivot index(76的idx是5)，右边大于pivot index;

$$A' = [25, 22, 43, 60, 21, 76, 100, 89, 87, 98].$$

76的index是5, **5 > 3**, 所以继续，在左半边(p, q-1)继续partition，也就是

$$A', \text{which is } [25, 22, 43, 60, 21].$$

This list is only five elements long, so we can sort it and find what is at index 3: $[21, 22, 25, 43, 60]$ and 43 is at index three.

So, 43 is the fourth smallest number of A. □

最后只剩5个元素的时候会直接排序，书中说小于5个的元素排序时间复杂度是n，idk y。

注：*sort小于5的array时间为O(n)? WTF?* 这点我也没搞懂

算法分析 The Median-of-medians

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n).$$

详细分析参考上方的原文链接。以下为intuition：

- **T(n/5)** + O(n):

  我们把n分成了n/5的subproblem, partition需要上述的时间，参考quickSort。

- **T(7n/10)**

  在M(median list, 参考example M)中，M的长度为**n/5** ,因此M中有一半的元素小于p(M的中位数，也就是中位数的中位数)，也就是n/10，这一半的元素本身又有2个元素小于自己，因为这些元素本身是median of 5 element，因此有n/10 + 2* n/10 = 3n/10 的元素小于p。

  因此在worst case情况下，算法每次都会recurse on the remaining 7n/10的元素。

根据master thoerum, 得出time complexity O(n)

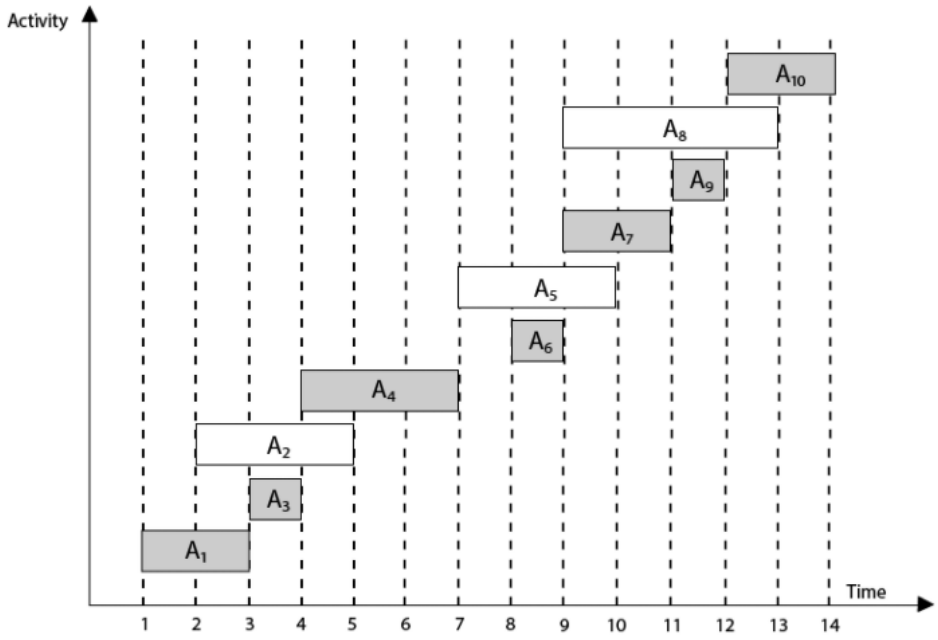
16 Greedy Algorithms

Intro: why greedy algorithm?

> For many optimization problems, **using dynamic programming to determine the best choices is overkill**;
>
> A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a **locally optimal** choice **in the hope that** this choice will lead to a **globally optimal** solution.
>
> **Greedy algorithms do not always yield optimal solutions**, but for many problems they do

16.1 经典例题1: An activity-selection problem

目标：我们要选择最大activities数量的集合，activities时间不能重合。（maximum-size set of activities.）



我们有n个activities, $S = \{a_1, a_2, \ldots, a_n\}$ 每个activity **a $_i$** has a **start time f $_i$** and a **finish time s $_i$**

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

你选择的集合中，**a $_i$** 和**a $_j$** 需要是兼容的(compatible), 也就是他们**时间区间 [si,fi] 不能有重合。**

比如 set :{**a $_1$**, **a $_2$**} 是一个不合格的set, 因为 **a $_1$** 的区间[1,4] 和 **a $_2$** 的区间[3,5] 就在[3,4]上有重合，我们要避免这样的overlap,选出最大的子集。

贪心算法的核心在于，我们要找出最好的**greedy choice**。在这个问题中，我们每一步都需要:

> we should choose an activity that leaves the resource available
>
> for as many other activities as possible.

这句话的意思在后来的另一个例子找硬币中也能体现出来，现在先往下走。

> **Greedy Algorithm的核心**:
>
> - **Greedy choice**: 每次都寻找**最早结束**的activity, Let's call it **a** $_{\textbf{earliest}}$
> - **Subproblem**: only consider activity start after **a** $_{\textbf{earliest}}$ have finished. (排除有overlap的activity)
>
> **剩下的，交给递归*recursive***
>
> RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.
>
> RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
> 1 $m = k + 1$
> 2 **while** $m \le n$ and $s[m] < f[k]$ // find the first activity in S_k to finish
> 3 $m = m + 1$
> 4 **if** $m \le n$
> 5 **return** $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
> 6 **else return** $\emptyset$
>
> (#注意，**书中假设activity list: n已经按照finish time list: f 进行升序的排序**），因此time complexity O(n)):
>
> > *It also assumes that the input activities are ordered by monotonically increasing finish time.*

看看就行:

经过经典的递归greedy algorithm解法，经典的下一步就是**把rucursive变成iterative**的解法。

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

16.1 经典例题2: An activity-selection problem

16.2 总结：Elements of the greedy strategy

贪心算法的核心性质：

Greedy-choice property

The first key ingredient is the **greedy-choice property**: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

这个性质比较好理解，这也是greedy和DP的主要区别。

Optimal substructure

A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

这个性质可能需要时间消化，以activity-selection问题为例：

> if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}

下方是书中给出的步骤：

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

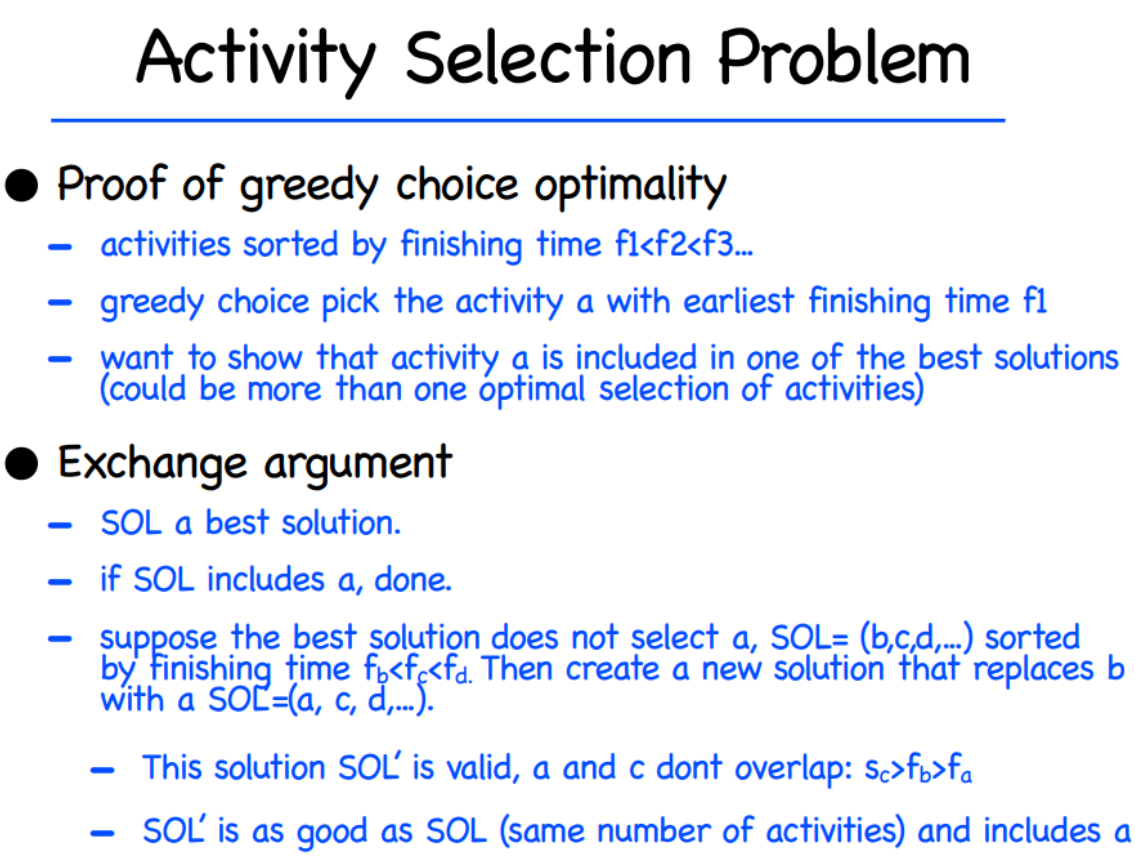
总结

- 贪心算法only make **locally optimal choiceœ**, 部分的贪心算法不能保证走向 global-optimal solution, 但我们只关心那些能保证走向global-optimal solution的算法。
- 贪心算法的核心在于寻找strategy, 你需要证明你的贪心策略是正确的。

16.3 [optional] Corectness of greedy algorithm

证明贪心算法需要证明以下两个性质

> **Greedy-choice property**
>
> The first key ingredient is the **greedy-choice property**: we can assemble a globally optimal solution by making locally optimal (greedy) choices.
>
> **Optimal substructure**
>
> A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.



Activity Selection - Induction Argument

- s(a)= start time; f(a)=finish time
- SOL=$\{a_1,a_2,\ldots,a_k\}$ greedy solution
  - chosen by earliest finishing time
- OPT = $\{b_1,b_2,\ldots,b_m\}$ optimal solution, sorted by finishing time; optimal means m max possible
- prove by induction that $f(a_i)\leq f(b_i)$ for all i=1:k
  - base case $f(a_1)\leq f(b_1)$ because $f(a_1)$ smallest in the whole set
  - inductive step: assume $f(a_{n-1})\leq f(b_{n-1})$. Then b_n is a valid choice for greedy at step n because $f(a_{n-1})\leq f(b_{n-1})\leq s(b_n)$. Since greedy picked a_n over b_n, it must be because an fits the greedy criteria $f(a_n)\leq f(b_n)$
- so $f(a_k)\leq f(b_k)$. If m>k then any b_{k+1} item would also fit into greedy solution (CONTRADICTION) thus m=k

Fractional Knapsack - greedy proof

- proving now that the greedy choice is optimal
  - meaning that the solution includes the greedy choice.
- greedy choice: take as much as possible form best quality (below item with quality q1)
  - items available sorted by quality: q1>q2>q3>..., greedy choice is to take as much as possible of item 1, that is quantity w1
- contradiction/exchange argument
  - suppose that best solution doesnt include the greedy choice: SOL=(r1,r2,r3,...) quantities chosen of these items, and that r1 is not the max quantity available (of max quality item), r1<w1
  - create a new solution SOL' from SOL by taking more of item 1 and less of the others
    - e=min(r2,w1-r1); SOL'=(r1+e,r2-e,r3,r4...)
    - value(SOL') - value(SOL) = e(q1-q2)>0 which means SOL' is better than SOL: CONTRADICTING that SOL is best solution

15 Dynamic Programming

> - **Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.**
> - **In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.:**
>
> > - **divide-and-conquer** algorithm **does more work than necessary**, repeatedly solving the common subsubproblems.
> >
> > - **dynamic-programming** algorithm solves each subsubproblem just once and then **saves its answer in a table**, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

15.1 EX: Rod cutting

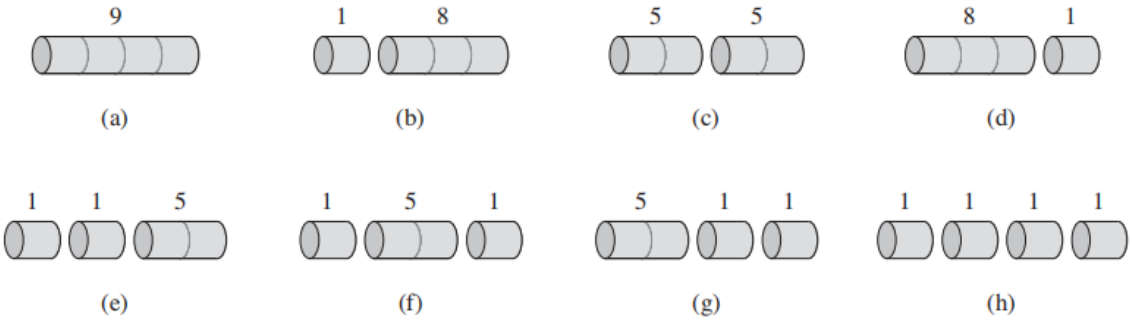
切绳子问题定义如下：

给定一个总长度为n的绳子，给定价格 p_i 与 长度 i 的对应表，如下图所示。

Task: **如何切分绳子，使得总价值 r_n 最大化?**

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

比如给定长度为4的绳子，绳子可以切分的**长度**和根据上图计算得出的**总价值**分别如下：



可以看出 (c)的总价值为10，是最大的。

切绳子问题展现出了**optimal substructure**:

原始问题是 problem of **size(n)**, 当我们第一次cut之后，我们就在解决两个独立的子问题。

> optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.
>
> i.e. 子问题的最优解被包含在了全局问题的最优解中。

Method 1: Brute force

长度为n的绳子共有 2^{n-1} 种切法，算出最大值。

Method 2: Recursive top-down implementation

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

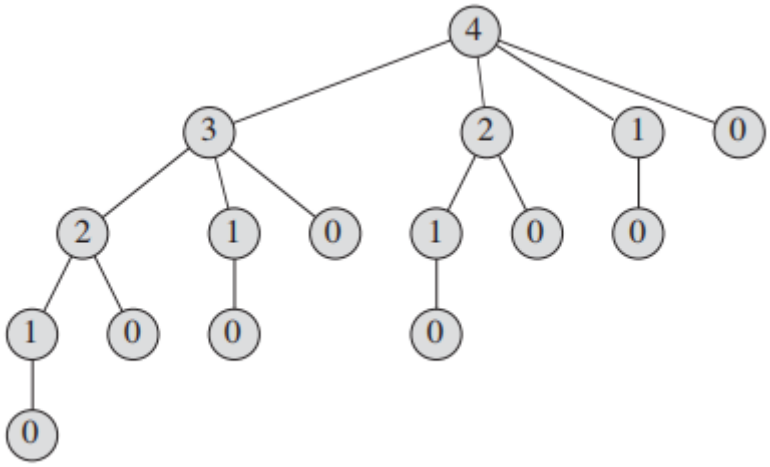
```
@param:
**p** : array p[1..n] of price

n: 长度n

#comment
3: init max value to -infinity
4-5: compute max value q
```

CUT-ROD is very inefficient. it solves the same subproblems repeatedly.

下图展示了 CUT-ROD(p, 4)的递归，可以看到**相同的子问题被反复的计算。**



时间复杂度T(n) = 2^n

Method 3 dynamic programming for optimal rod cutting

动态规划出现了!

核心思想：每个子问题只解决一次，使用额外的空间来保存子问题的solution。

We just look it up, rather than re-compute it.

It is a **time-memory trade-off**.

> There are usually two equivalent ways to implement a dynamic-programming

> approach.
>
> The fifirst approach is **top-down with memoization**.
>
> ▶ 详细信息
>
> The second approach is the **bottom-up method**.
>
> ▶ 详细信息

> 简单来说，二者差别如下：
>
> **top-down**方法先检索是否有该subproblem的答案；
>
> > 如果有，使用该答案。
> >
> > **否则，计算该答案，进入递归。**
>
> 而**bottom-up**方法使用问题大小的**自然顺序**(natural notion of the size of problem)，**从最小的问题开始， 自底向上 的逐一解决**，因此解决到大问题时，之前的小问题已经解决完了。
>
> 原文的详细定义点开上方折叠。

Pseudocode for the top-down CUT-ROD procedure:

top-down:

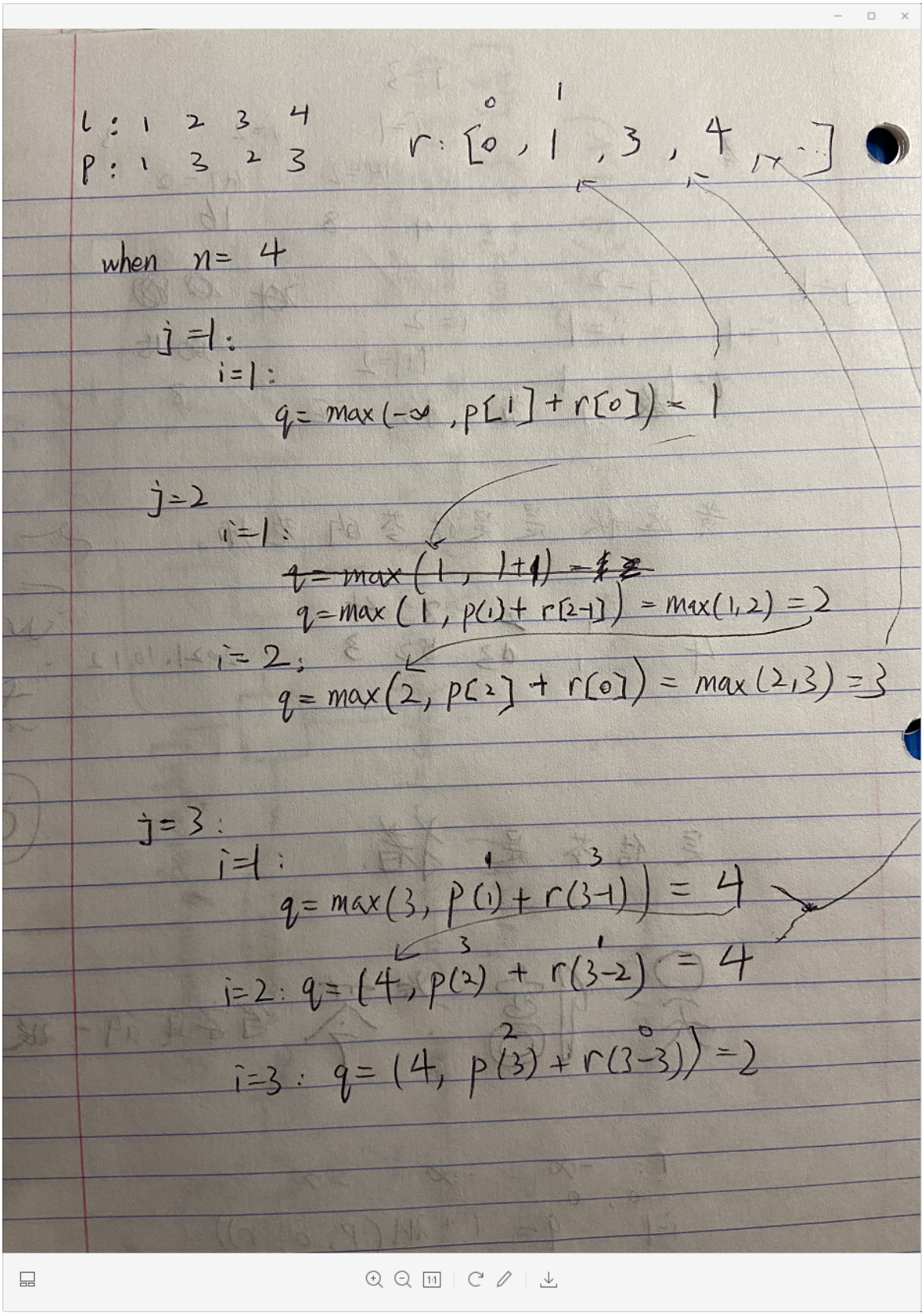
```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

bottom-up:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

下图为**bottom-up** 方法我个人的**部分**演算：



15.2 EX: Matrix-chain multiplication 矩阵连乘问题

我们熟知的矩阵乘法的伪码如下：

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

假设我们要连乘矩阵 <A_1, A_2, A_3, A_4>, 我们有以下5种方法, i,e, we can fuuly parenthesize the product in 5 distinct ways:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

现在我们来分析假设我们要连乘矩阵 <A_1, A_2, A_3> ，他们的维度分别为**<10 x 100, 100 x 5, 5 x 50>**。

parenthesization <$((A_1, A_2), A_3)$> 中，第一次括号内运算会有10 * 100 * 5 = 5000 次运算，然后再有 10 * 5 * 50 = 2500 次运算，总共有**7500**次的运算。

而<$(A_1, (A_2, A_3))$>,第一个括号 100 * 5 * 50 = 25000次运算，之后有10 * 100 * 50 = **50000** 的运算，总共有**75000**次运算，比第一个快**十倍**。

如下，我们引出**矩阵连乘问题**:

> **Our goal is to determine an order for multiplying matrices that has the lowest cost.**

即，找出最快矩阵连乘的顺序。

假设我们有下方的矩阵:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

注意，其中，矩阵A_i的维度是 $p_{i-1} \cdot p_i$.

Method 1 brute-force 穷举法

根据矩阵数量，这是inefficient的递归公式:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 . \end{cases}$$

**

Method 1 Applying dynamic programming

记得之前的四个步骤：

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

现在大概跟一遍：

Step 1: The structure of an optimal parenthesization

假设我们有如下**optimal** 的待乘矩阵 $A_i, A_{i+1}, \ldots, A_j$, 我们在$A_k$ (i < k < j) 将待乘矩阵分开，那么 $A_i, A_{i+1}, \ldots, A_k$ 也一定也是子问题 $A_i, A_{i+1}, \ldots, A_k$的optimal solution。

因为如果$A_i, A_{i+1}, \ldots, A_k$ 有更好的方法，那么 $A_i, A_{i+1}, \ldots, A_j$ 也会有更好的方法，这就形成了contradiction。

上面的方法是反证法（contradiction）。

Step 2: A recursive solution

如果我们在A_k (i < k < j) 将待乘矩阵分开，那么我们会得到两个子问题$A_i, A_{i+1}, \ldots, A_k$ 和 $A_k, A_{k+1}, \ldots, A_j$;

我们让 **m[i, j]** 来表示全局问题的最优解（即最小的multiplication 次数），两个子问题即为 m[i,k], m[k+1,j];

合并两个子问题的product $A_{i..k} \cdot A_{k+1..j}$ 我们需要$p_{i-1} \cdot p_j \cdot p_j$。（$A_i$是$p_{I-1} \cdot p_i$）因此

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_j \cdot p_j$$

这样求optimal solution需要知道最优切分点 k 的位置，但是我们不知道。因此我们遍历**k from i to j**!

这样，我们的问题变成了

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases} \qquad (15.7)$$

Step 3: Computing the optimal costs

回顾一下斐波那契数列，斐波那契数列也可以用动态规划来表示；

如果只用递归的方法调用斐波那契数列，递归的过程中，F(5)和F(8)都会重新计算F(4),F(3)....

动态规划的标志就是他储存了已经计算过的答案来防止re-compute。

假设我们有下方的矩阵：

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

注意，其中，矩阵A_i的维度是 $p_{i-1} \cdot p_i$.

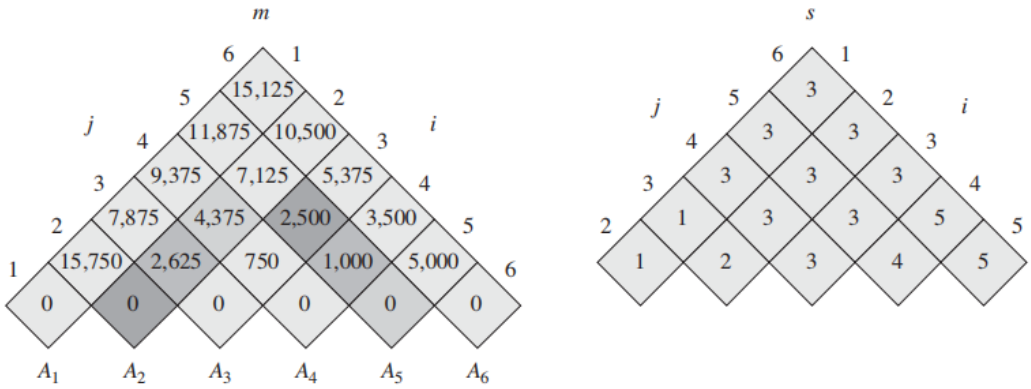
bottom-up approach：

- set $m[i,i]$ = 0 ($m[i,i]$代表矩阵乘以矩阵本身，连子问题都不算，只是trivial), 这也是为什么源码的n-1，n=6的问题我们只需要计算五个就好了
- compute $m[i,i+1]$ for $i = 1,2,\ldots,n-1$
- then compute $m[i,i+2]$ for $i = 1,2,\ldots,n-1$, and so forth

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                  // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

作为一个bottom-up的DP, 我们每次都在尝试想上计算；比如当算到$m[2,5]$时， 我们其实在做以下事情

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$



上图左侧的table即为伪码中的 **m** table， 右边的是 **s** table。

m表只给出了m[i,...,j] 的对应最优计算次数，并没有告诉我们 m[1,6] 的过程应该怎么进行切分；因此我们需要右边的 s table 来告诉我们。

$s[i,j]$ **记录了每一次的最优切分k。**通过s表递归的寻找k,我们就能知道最优切分。

如果上面你看懂了，下面这三张图你可以跳过。

这代码 for 的我人傻了，因此打印一下结果看看运行顺序：

好一个bottom-up：

```
l: 2
   i: 1
   j is  2
       k: 1
       m[1,1]+m[2,2]+ppp
       尝试计算m[1,2]
   i: 2
   j is  3
       k: 2
       m[2,2]+m[3,3]+ppp
       尝试计算m[2,3]
   i: 3
   j is  4
       k: 3
       m[3,3]+m[4,4]+ppp
       尝试计算m[3,4]
   i: 4
   j is  5
       k: 4
       m[4,4]+m[5,5]+ppp
       尝试计算m[4,5]
```

```
l: 3
    i: 1
    j is  3
        k: 1
        m[1,1]+m[2,3]+ppp
        尝试计算m[1,3]
        k: 2
        m[1,2]+m[3,3]+ppp
        尝试计算m[1,3]
    i: 2
    j is  4
        k: 2
        m[2,2]+m[3,4]+ppp
        尝试计算m[2,4]
        k: 3
        m[2,3]+m[4,4]+ppp
        尝试计算m[2,4]
    i: 3
    j is  5
        k: 3
        m[3,3]+m[4,5]+ppp
        尝试计算m[3,5]
        k: 4
        m[3,4]+m[5,5]+ppp
        尝试计算m[3,5]
```

```
l: 4
   i: 1
   j is  4
       k: 1
       m[1,1]+m[2,4]+ppp
       尝试计算m[1,4]
       k: 2
       m[1,2]+m[3,4]+ppp
       尝试计算m[1,4]
       k: 3
       m[1,3]+m[4,4]+ppp
       尝试计算m[1,4]
   i: 2
   j is  5
       k: 2
       m[2,2]+m[3,5]+ppp
       尝试计算m[2,5]
       k: 3
       m[2,3]+m[4,5]+ppp
       尝试计算m[2,5]
       k: 4
       m[2,4]+m[5,5]+ppp
       尝试计算m[2,5]


l: 5
   i: 1
   j is  5
       k: 1
       m[1,1]+m[2,5]+ppp
       尝试计算m[1,5]
       k: 2
       m[1,2]+m[3,5]+ppp
       尝试计算m[1,5]
       k: 3
       m[1,3]+m[4,5]+ppp
       尝试计算m[1,5]
       k: 4
       m[1,4]+m[5,5]+ppp
       尝试计算m[1,5]
```

15.3 Elements of dynamic programming

Optimal substructure

> A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

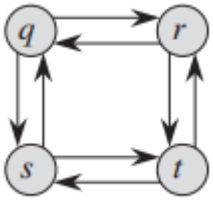
有些问题没有Optimal substructure性质，比如给定一个没有权值的**有向图**(directed graph)：

$$G = (V, E) \text{ and vertices } u, v \in V.$$

现在有两个问题，定义如下：

1. 无权值的最短路径问题（**Unweighted shortest path**）：找到某顶点u到某顶点v的 最少边的数量。

   （英文比较好看懂：Find a path from u to v consisting of the **fewest edges**.）

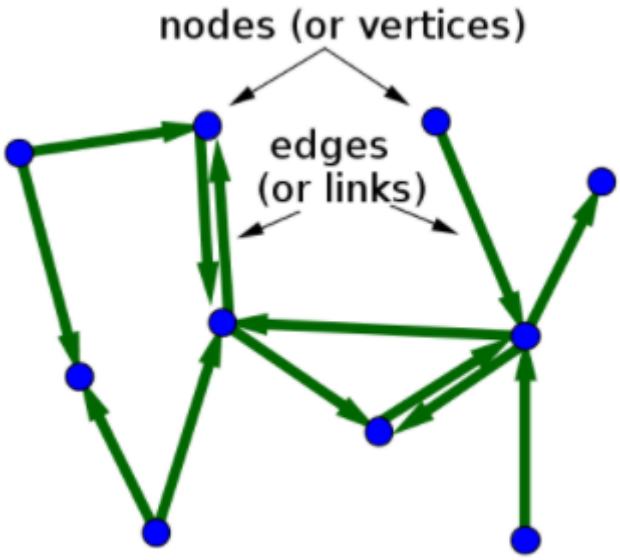


2. 无权值的最长路径问题(**Unweighted longest simple path:**): 找到某顶点u到某顶点v的 最多边的数量。

   哪个问题有optimal structure呢?

问题1有。

> 查缺补漏：图的定义如下：
>
> A directed graph (or digraph) is a set of vertices and a collection of directed edges**(边)** that each connects an ordered pair of vertices**(顶点)**.
>
> 一句话说，由**顶点 set:V** 和 **边 set :E**组成



Unweighted shortest path:3

Find a path from u to consisting of the fewest

edges. Such a path must be simple, since removing a cycle from a path pro□

duces a path with fewer edges.

作者很懒还没写完

15.4 本章节书本/作业包含的Leetcode题目

作业以及课本包含一些经典DP的leetcode题目，如下：

> 516. Longest Palindromic Subsequence
>
> 72. Edit Distance
>
> 300. Longest Increasing Subsequence
>
> 1143. Longest Common Subsequence

> 887. Super Egg Drop (took me a long while...for real...)
>
> 得从brute force开始：
>
> https://leetcode.com/problems/super-egg-drop/discuss/159079/Python-DP-from-kn2-to-knlogn-to-kn
>
> 到这位大神的：
>
> https://leetcode.com/problems/super-egg-drop/discuss/158974/C%2B%2BJavaPython-2D-and-1D-DP-O(KlogN)

10. 基础数据结构

1. **stack:**

last-in,first-out

Methods: push , pop

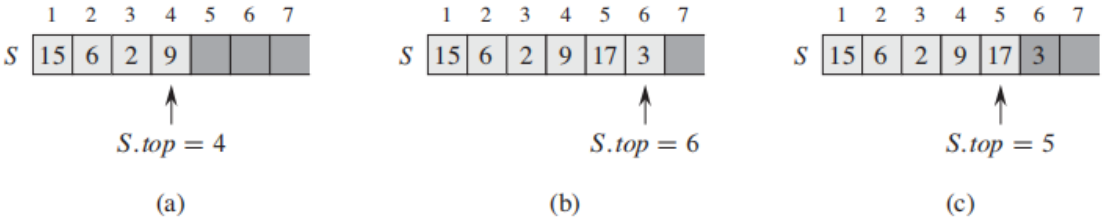


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

2. **queue**

queue: first-in, first-out

Methods: deque, enque

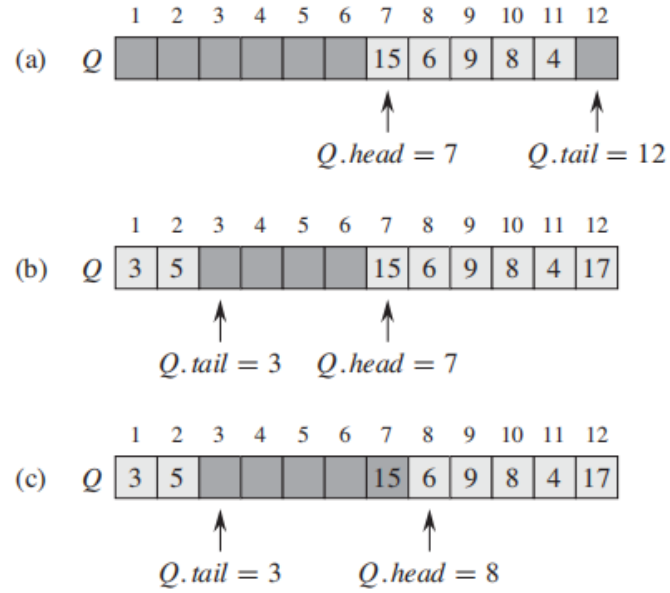


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

1. Linked List

The order in alinked list is determined by a pointer.

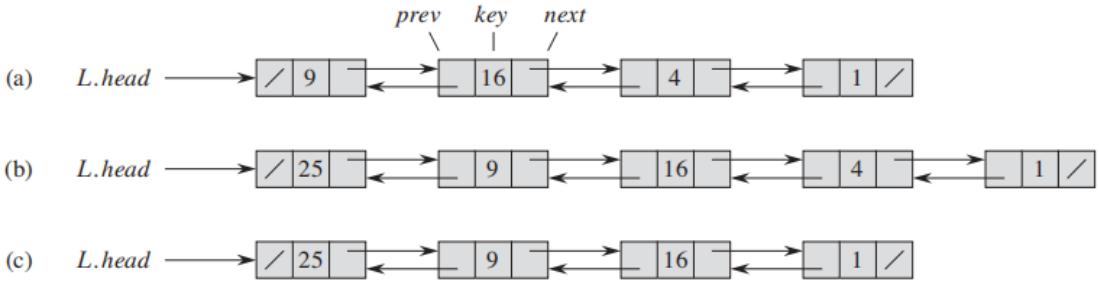


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Methods: search, insert, delete

Search, Insertion, deletion

搜索：

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

时间复杂度：$O(n)$

插入：

注意，这个是从**前面插入**；

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

不用遍历，**因此时间复杂度是**$O(1)$

删除：

需要先使用Search功能，因此worst case时间复杂度是$O(n)$

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

2. Binary Search Tree:

好像没啥说的

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

inorder tree walk. This algorithm is so named because it **prints the key of the root of a subtree**

between printing the values in its left subtree and printing those in its right subtree.

0. Min,Max

Min

最小值就是找到最左的节点

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

Max

最大值就是找到最右的节点

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

0.5. Successor and Predecessor

Successor and Predecessor是个啥?

Successor

> If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key.

即，x的successor是刚好大于x的那个节点；

为了找到这个节点：

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

我们的目标是找到刚好比x大的元素，所以：

- 如果x的右子树不为空，那么找到右子树中的最小元素即可；

  要找到右子树中的最小元素，对着x.right调用 `min` 方法即可；

- 如果x的右子树是空的，那么sucessor只能在x的头上；此时又有两种情况：
  - 假设y是x的parent, 若x是y的左子树，说明x<y,那么直接返回y;
  - 否则, x是y的右子树，x>y, 再往上走；(如果走到根节点，就会返回NIL)

Predecessor

与Successor 相反，Predecessor是刚好小于 x 的节点；

假设我们要找的节点x是存在于树中的。

因此:

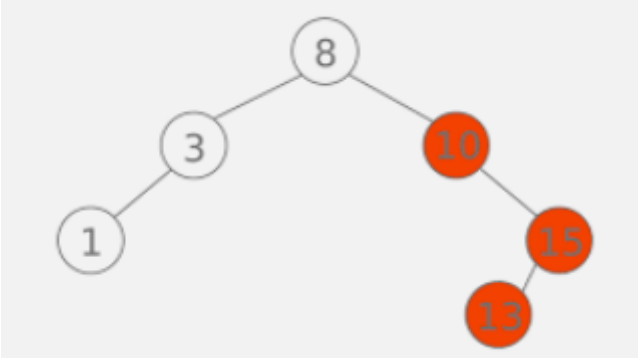
- 如果x.left $\neq$ NIL, 那么Predecessor就是x.left中的最大值;
- 否则，Predecessor不存在；

书中BST涉及的一些leetcode:

1. Check Balance

> 110.Balanced Binary Tree：https://leetcode.com/problems/balanced-binary-tree/

比较tricky地方在于，检查BST是否平衡，不只是统计根节点的左右子树最大高度差，因为可能发生这种情况：



对于根节点来说，height(左子树) - height(右子树)的高度差小于2；

但是对于节点10来说，高度差是2，破坏了平衡；

因此，你需要对**每一个节点**都check balance。

2. Insertion

> 701. Insert into a Binary Search Tree: https://leetcode.com/problems/insert-into-a-binary-search-tree/

找到位置插入即可，优雅的代码来自：

https://leetcode.com/problems/insert-into-a-binary-search-tree/discuss/180244/Python-4-line-clean-recursive-solution

这哥们老想写成一行，为了可读性我给他展开了：

```
class Solution(object):
    def insertIntoBST(self, root, val):
        if root == None:
            return TreeNode(val)
        if root.val > val:
            root.left = self.insertIntoBST(root.left,val)
        else:
            root.right = self.insertIntoBST(root.right,val)

        return root
```

3. Deletion

> 450. Delete Node in a BST： https://leetcode.com/problems/delete-node-in-a-bst/

原文写的太复杂了，用到了transplant; 因此使用leetcode大哥的；

删除操作有3个case;

- **如果节点没有children, 那么直接删除；**
- **如果节点没有左children, 那么右children直接顶上来；**
- **否则，找到左children中的最大值，并且顶上去；**

代码来自：

https://leetcode.com/problems/delete-node-in-a-bst/discuss/213685/Clean-Python-3-with-comments-in-details

```
class Solution:
```

```
    def deleteNode(self, root, key):
        """
        :type root: TreeNode
        :type key: int
        :rtype: TreeNode
        """
        if not root:
            return

        # we always want to delete the node when it is the root of a subtree,
        # so we handle left or right according to the val.
        # if the node does not exist, we will hit the very first if statement
and return None.
        if key > root.val:
            root.right = self.deleteNode(root.right, key)

        elif key < root.val:
            root.left = self.deleteNode(root.left, key)

        # now the key is the root of a subtree
        else:
            # if the subtree does not have a left child, we just return its
right child
            # to its father, and they will be connected on the higher level
recursion.
            if not root.left:
                return root.right

            # if it has a left child, we want to find the max val on the left
subtree to
            # replace the node we want to delete.
            else:
                # try to find the max value on the left subtree
                tmp = root.left
                while tmp.right:
                    tmp = tmp.right

                # replace
                root.val = tmp.val

                # since we have replaced the node we want to delete with the tmp,
now we don't
                # want to keep the tmp on this tree, so we just use our function
to delete it.
                # pass the val of tmp to the left subtree and repeat the whole
approach.
                root.left = self.deleteNode(root.left, tmp.val)

        return root
```

so fking clean**!

见过的最优雅代码之一了!

11 Hash Table

https://docs.python.org/2/library/functions.html#hash

Python 官方文档:

hash(*object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

用来for loop 比较 key 值是否相等的

使用pyhon试试 $hash$ 函数:

```
C:\Users\Administrator>python3
Python 3.9.6 (default, Jul 12 2021, 05:46:11) [GCC 10.3.0 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> hash('asd')
8023943316209056078
>>> hash('asdasd')
8207874031295978013
>>> hash('asdasdasdd')
2903334087122411768
>>>
```

11.1 Direct-address tables

> Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

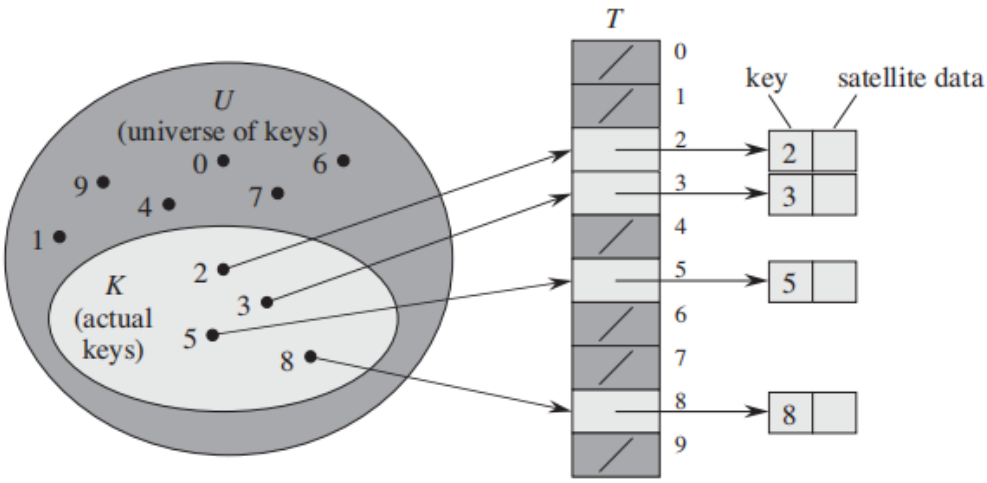
直接寻址法 适用于key比较少的时候。

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.



这样的方法同时记住了 key 和 data（而不是data的address），非常占用空间；

11.2 Hash tables

Python用久了让人有 dict 很简单的错觉，所以这个内容让我思考了很久。

建议先看一下下文：

https://zhuanlan.zhihu.com/p/74003719

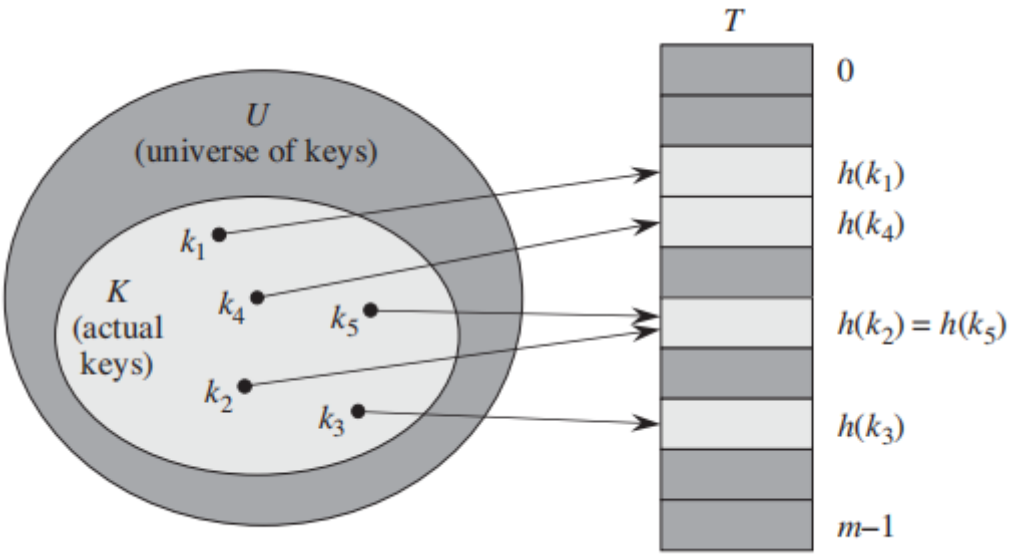
> With direct addressing, an element with key k is stored in slot k.
>
> With hashing, this element is stored in slot **h(k)**; that is, we use a **hash function** h to compute the
>
> slot from the key k.
>
> Here, h maps the universe U of keys into the slots of a **hash**
>
> **table** T[0 .. m-1]:

> h: U ---> {0,1, ... , m-1}

简而言之：

> **An element with key k hashes to slot h(k);**

hash function减少了indices的范围；



但是有时候两个key会被match到一个slot里面；这就是一个 **collision**; 参考上图。

比如我们的 `hash(x) = x%10`, 这样的话25 和 35就会被match到同一个slot上，这就是一个冲突；

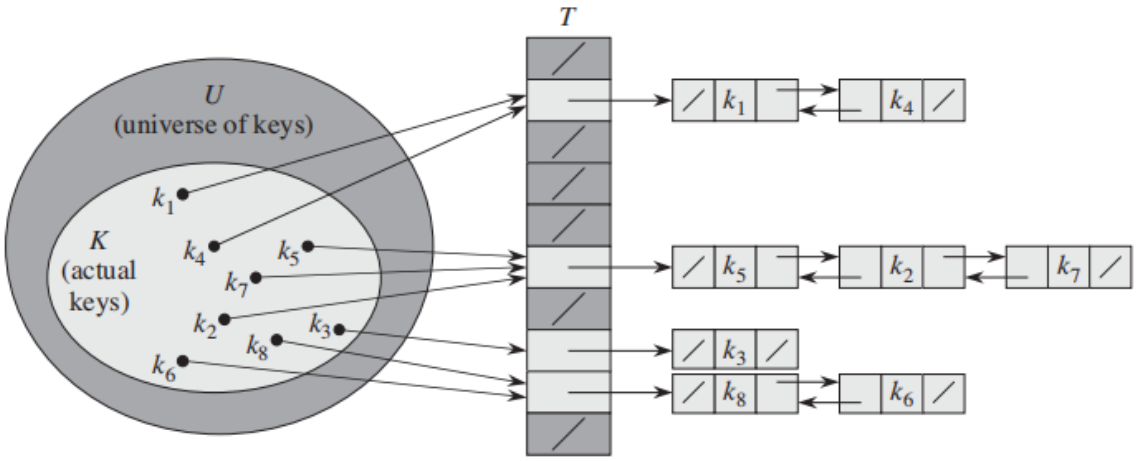
发生冲突并不可怕，有一些有效的方法能化解冲突带来的结果；（冲突本身是不可能解决的）

哈希算法最重要的特点就是：

- 相同的输入一定得到相同的输出；
- 不同的输入大概率得到不同的输出。

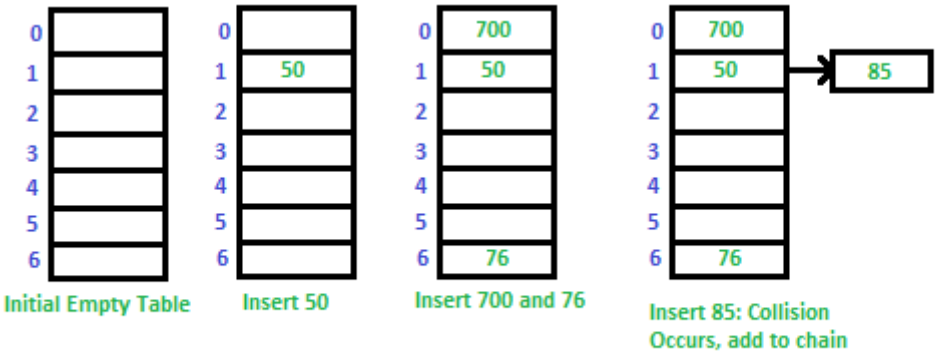
下面介绍原书中提供的最简单的冲突解决方法；

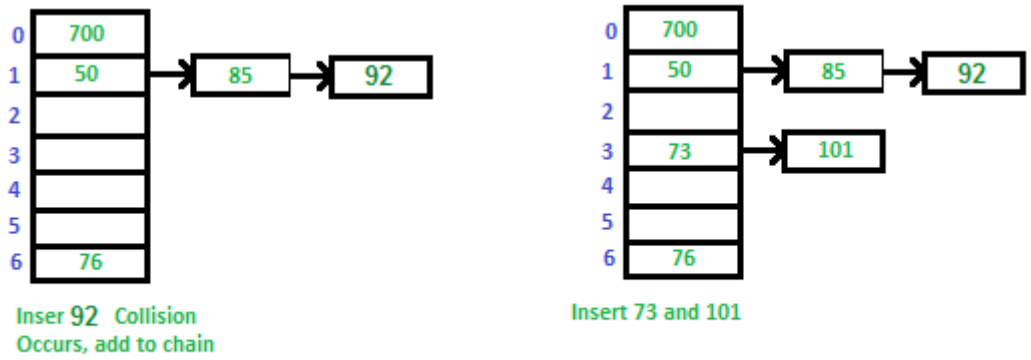
1. Chaining:



让每一个key都对应一个linked list;

一个例子:





2. 开放寻址法(open addressing):

11.3 Hash functions

什么是好的 hash function?

- **simple uniform hashing:** each key is equally likely to hash to any of the m slots

如果完全随机的插入slot, 就可以满足这个条件;

-

Most hash functions assume that the universe of keys is the set N= {0,1,2, ……}

of natural numbers. Thus, if the keys are not natural numbers, we find a way to

interpret them as natural numbers.

比如最简单的**division method**:

$$h(k) = k \mod m$$

11.5 Perfect hashing

> A hashing technique **perfect hashing** if $O(1)$memory accesses are required to perform a search in the **worst** case.

为了做到完美哈希，第一个因素和之前的方法是一样的：选择一个好的 hash function h 把 n 个 $keys$ hash into m slots。

之后，除了使用chaining 搭建新 linked list的方法之外，转而使用一个小的第二哈希表 **secondary hash table** S_j ; 可以保证在**第二哈希level没有冲突。**

> **Theorem:**
>
> Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions.
>
> Then, the probability is less than $1/2$ that there are any collisions.
>
> 翻译:
>
> 使用随意一个哈希函数 h ,把 n 个keys存到 $m = n^2$ 个slot里，有冲突的概率小于$1/2$

Proof There are $\binom{n}{2}$ pairs of keys that may collide; each pair collides with probability $1/m$ if h is chosen at random from a universal family $\mathcal{H}$ of hash functions. Let X be a random variable that counts the number of collisions. When $m = n^2$, the expected number of collisions is

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2 . \end{aligned}$$

11.6 自己实现hashmap for string/text

题目来自Homework 8

> **Implement a hash for text.**

> Given a string as input, construct a hash with words as keys, and wordcounts as values. Your implementation should include:
>
> - a hash function that has good properties for text
> - storage and collision management using linked lists
> - operations: insert(key,value), delete(key), increase(key), find(key), list-all-keys

先来找一个适合string的hash func，google了以后：

http://www.cse.yorku.ca/~oz/hash.html

实施 $djb2$.

复习一下：

> Python
>
> `ord`:
>
> 输入一个字符，返回ASCII数值
>
> `hex`:
>
> 输入整数，返回16进制

实施hash_djb2:

忘记位运算的先复习一下；

```
def hash_djb2(s):
    hash = 5381
    for x in s:
        hash = (( hash << 5) + hash) + ord(x)
    return hash & 0xFFFFFFFF
```

这个函数的magic在于他的两个魔法数字：33和5381，这里用的是5381；这是一个经验主义得到的magic number,不用纠结太多；

贴一点test case:

```
print(hash_djb2(u'hello world')) # '0xa6bd702fL'
print(hash_djb2('a'))
print(hash_djb2('b'))

#输出：
894552257
177670
177671
```

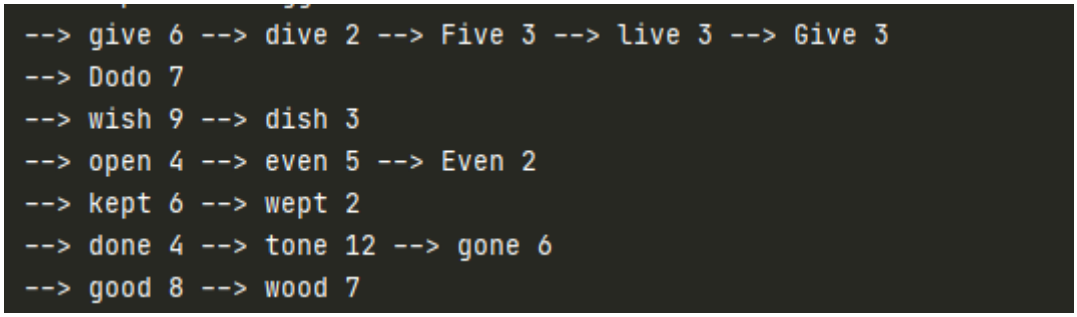
Hash table python implementation

python自己实现哈希表

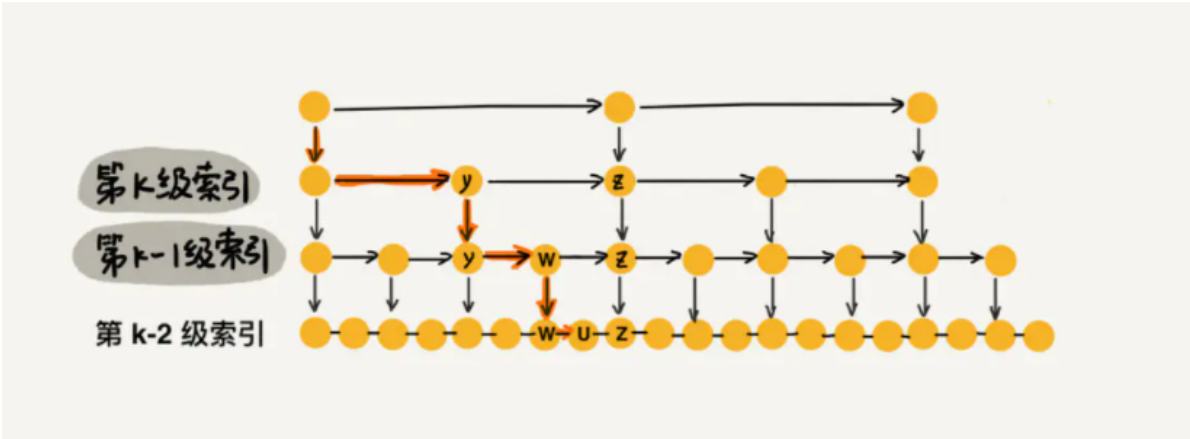
目标如下：

- 给定一段长文本txt文件，我的hashmap读入文件，以word作为key，word count作为value；
- 使用chaining 方法来handle collision

结果是这样的：



12 SkipList



跳表在工业中也会被经常用到，墙裂建议阅读下文：

https://www.jianshu.com/p/9d8296562806

简单概括重点：

跳表的索引高度 $h = log_2 n$，且每层索引最多遍历 3 个元素。所以跳表中查找一个元素的时间复杂度为 $O(3 * logn)$，即：$O(logn)$。

假如原始链表包含 n 个元素，则一级索引元素个数为 n/2、二级索引元素个数为 n/4、三级索引元素个数为 n/8 以此类推。所以，索引节点的总和是：n/2 + n/4 + n/8 + … + 8 + 4 + 2 = n-2，**空间复杂度是 O(n)。**

为什么Redis选择使用跳表而不是红黑树来实现有序集合?

Redis 中的有序集合(zset) 支持的操作：

1. 插入一个元素
2. 删除一个元素
3. 查找一个元素
4. 有序输出所有元素
5. 按照范围区间查找元素（比如查找值在 [100, 356] 之间的数据）

其中，前四个操作红黑树也可以完成，且时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。按照区间查找数据时，跳表可以做到 O(logn) 的时间复杂度定位区间的起点，然后在原始链表中顺序往后遍历就可以了，非常高效。

skiplists实现详解

跳表Skiplist的Python实现。

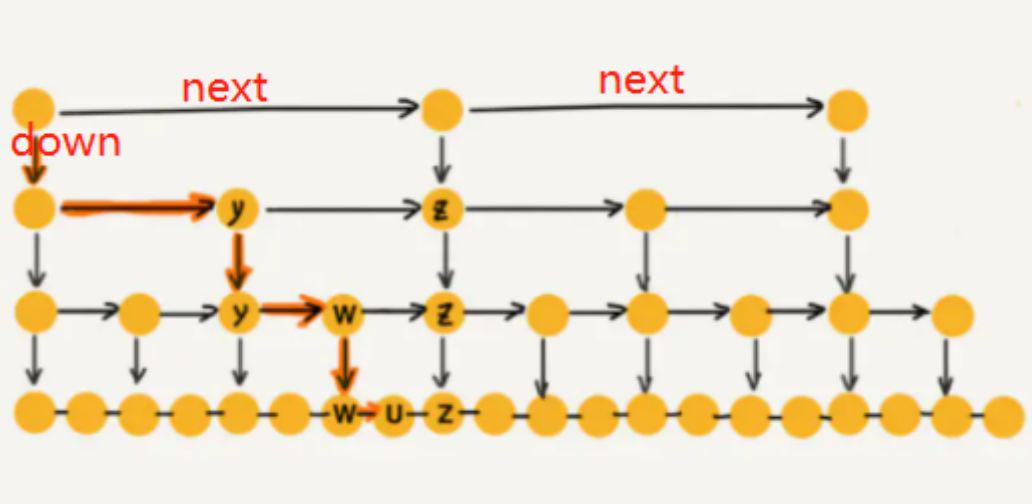
https://leetcode.com/problems/design-skiplist/discuss/?currentPage=1&orderBy=most_votes&query=

Python:

这个实现是我从leetcode某个大哥那抄过来的，有做改动；

It really took me one day to understand....

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None
        self.down = None
```



`next` 指的是**相同级别**的下一个元素，就是往右走；

`down` 指的是**下一个级别**的相同元素，就是往下走；

初始化：

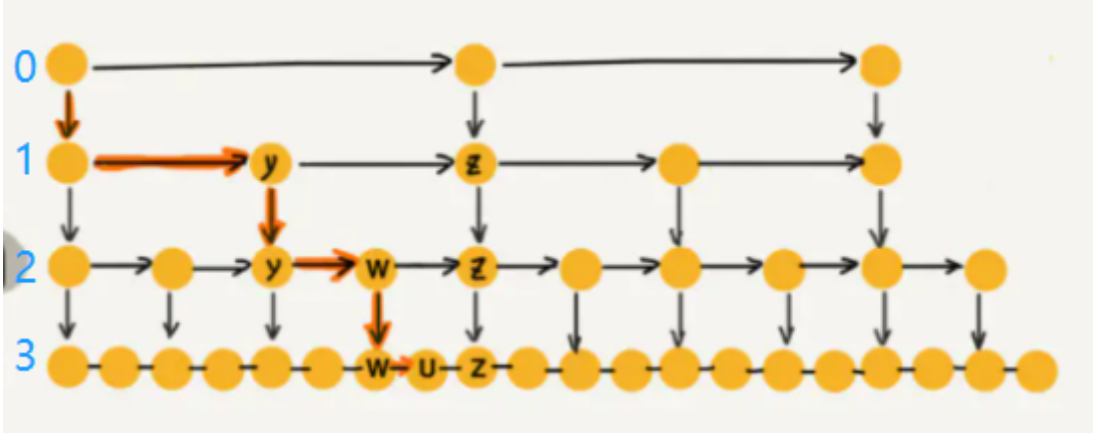
```
class Skiplist:
    def __init__(self):
        self.levels = []
        self.max_level = 4
        prev = None
        for i in range(self.max_level):
            node = Node(-math.inf)
            self.levels.append(node)
            if prev:
                prev.down = node
            prev = node
```

在这次的实现中，levels储存的是每个级别的单链表。

index = 0 的位置存的是**最高级**的链表，我们在这一级别实现skip 操作；

index = 3 存的是最基础的链表，也就是长度为 n 的链表，如下图：

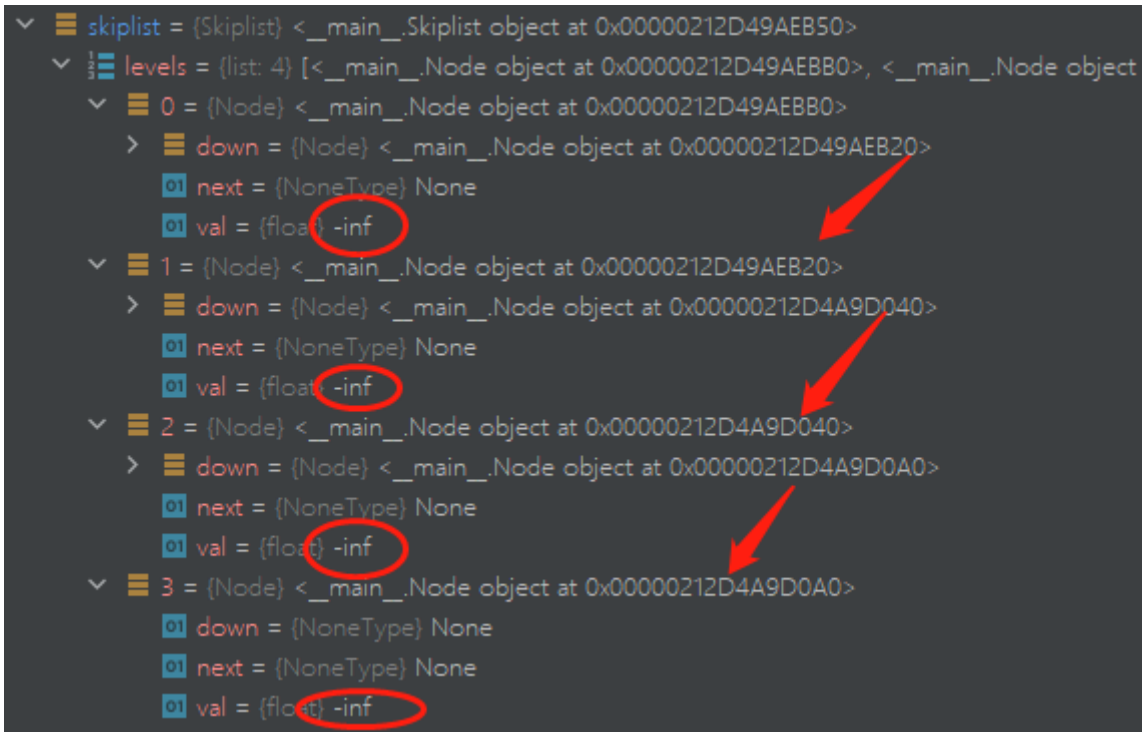
> 后面统一一下口径，**最高级**的链表指 **index = 0**, 元素最少的那一条链表；



通过上述代码，你实现了如下的操作：

1. `self.level` 中储存了每个级别的对应的链表, 每个级别链表的初始化都为负无穷：`-math.inf`
2. 让每个更高级的链表对象指向更基础的级别；

如图：



实现iter

```
    def _iter(self, val):
        res = []
        l = self.levels[0]
        while l:
            while l.next and l.next.val < val:
                l = l.next
            res.append(l)
            l = l.down
        return res
```

这个函数是很核心的函数，后面的操作都会用到它；

这个函数**实现了skip**的作用：

- **输入一个`val`，只要下一个元素比`val`小，就往右走(`next`)；**
- **否则，就往下走(`down`)；**

为什么是`while 1`,因为他只能往右或者往下走，就一定有走到None的时候；

所以，这个函数中返回的结果`res`是**每个级别**中刚好小于`val`的那个node节点对象；

即使下一个元素能等于val, 也停留在之前一个，方便后续delete操作；

这个函数实现了，理解了返回的res是什么，后续就简单了；

search 搜索操作：

```
    def search(self, target: int) -> bool:        last = self._iter(target)[-1]
       return last.next and last.next.val == target
```

上一个`_iter`函数停留在输入值的前一个数，所以直接检查下一个元素就好了；

add/insert操作

```
    def add(self, num: int) -> None:        res = self._iter(num)        prev =
None        for i in range(len(res) - 1, -1, -1):            node = Node(num)
        #res[i]是刚好比val小的元素，那么next就比val大咯            node.next =
res[i].next            #指向低级链表            node.down = prev            #res[i]
是刚好比val小的元素            res[i].next = node            prev = node
rand = random.random()            if rand > 0.5:                break
```

- 这个`for`是从低级走到高级的
- 在保证了基础级别存在插入的数值以后，每个更高级的节点都`random`一次，大于0.5就在更高级的节点添加该节点；

erase/delete删除操作

有了`_iter`操作后很简单，不用说了

13 红黑树：Red-Black Trees

先仔细阅读下原文：

> A **red-black tree** is a binary search tree with one extra bit of storage per node: its **color**, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.
>
> Each node of the tree now contains the attributes `color`, `key`, `left`, `right`, and `p`.

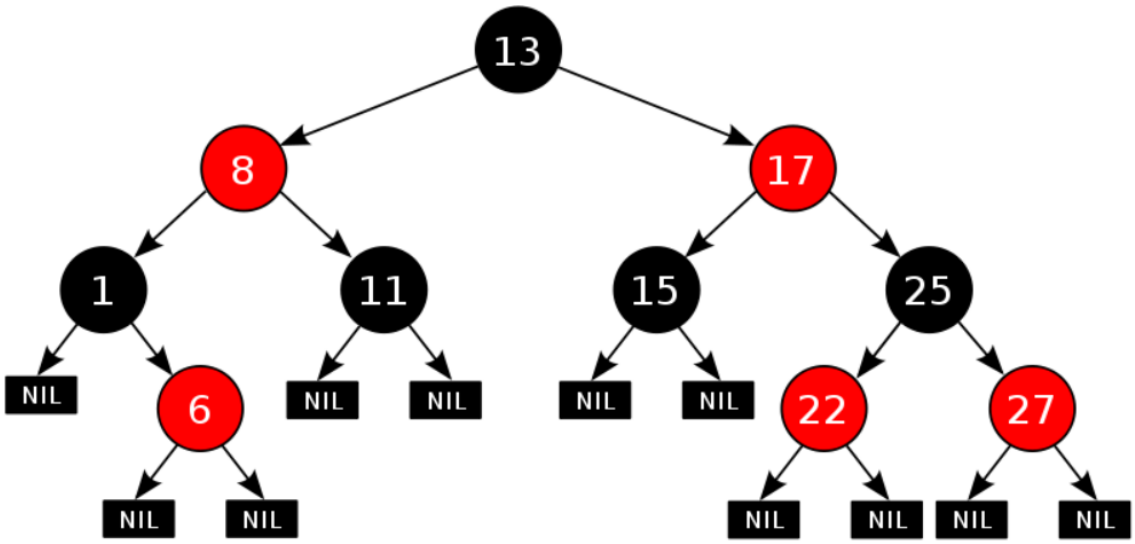
A red-black tree is a binary tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

所以红黑树本质就是自平衡的BST，但是他多了颜色属性，并且服从红黑树性质:

1）每个结点要么是红的，要么是黑的。 2）根结点是黑的。 3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。 4）如果一个结点是红的，那么它的俩个儿子都是黑的。 5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。



如果忘记了BST的性质，先去上面复习一下 Insertion 和 Deletion 的操作。

13.1. Rotation

> When we do a left rotation on a node x, we assume that its right child y is not T:nil; x may be any node
>
> in the tree whose right child is not T:nil. The left rotation “pivots” around the link
>
> from x to y. It makes y the new root of the subtree, with x as y’s left child and y’s
>
> left child as x’s right child.

我们要**左**旋节点某子树的根节点： `x`;

假设`x` 的右child `y`的 不为空:

旋转围绕着 `x`和`y`的连接，我们让:

```
        - ``y``成为该子树的root,        - ``x``成为 ``y`` 的左child, ``y`` 原来的左child成为 ``x``的右child。
```

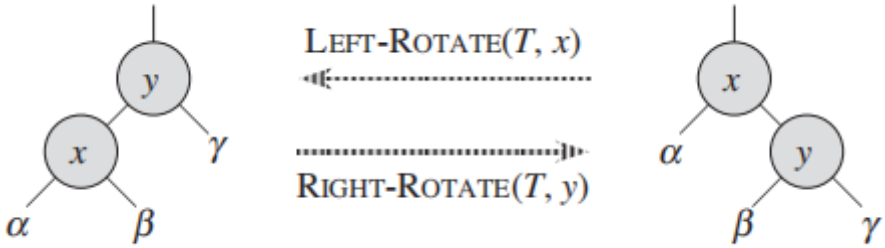


figure 来自原文**13.2** *p313*

伪码：

```
LEFT-ROTATE(T, x)
1   y = x.right              // set y
2   x.right = y.left         // turn y's left subtree into x's right subtree
3   if y.left ≠ T.nil
4       y.left.p = x
5   y.p = x.p                // link x's parent to y
6   if x.p == T.nil
7       T.root = y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

左旋、右旋的时间复杂度为$O(1)$。

13.2. Insertion

插入的时间复杂度为 $O(lgn)$

```
RB-INSERT(T, z)
1   y = T.nil
2   x = T.root
3   while x ≠ T.nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

为了保证红黑树性质，需要一个额外的`fix`函数来 recolor 以及 rotate

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

今天被旋晕了，下次再来吧

Case 1: `z's` uncle `y` **is red**

Case 2: z's uncle y **is black and** z **is a right child**

Case 3: z's uncle y **is black and** z **is a left child**

总共有6个cases;

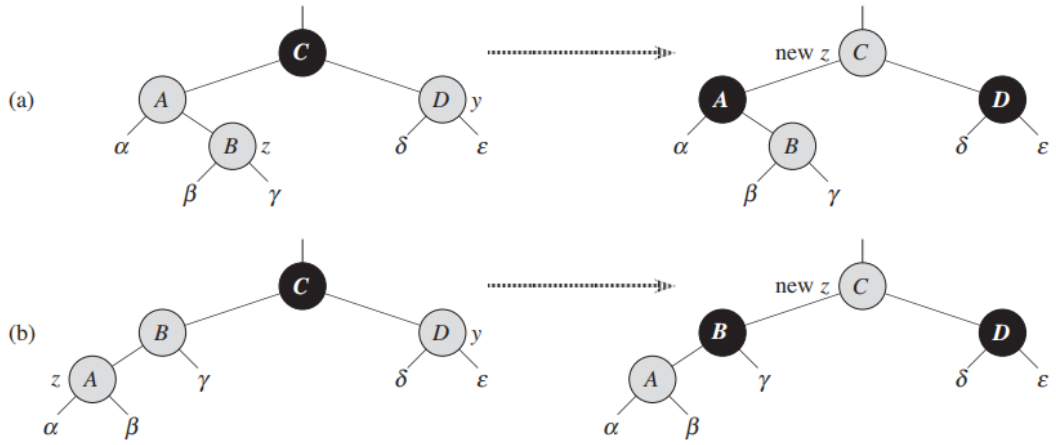
因为3个3个cases之间是对称的，因此

我们关注三个case:

> **Case 1:** `z's` uncle `y` **is red**
>
> **Case 2: z's uncle** y **is black and** z **is a right child**
>
> **Case 3: z's uncle** y **is black and** z **is a left child**

Case 1: `z's` uncle `y` **is red**

z的舅舅是红色的，此时违反了性质4: 即一个red的儿子必须是两个black;



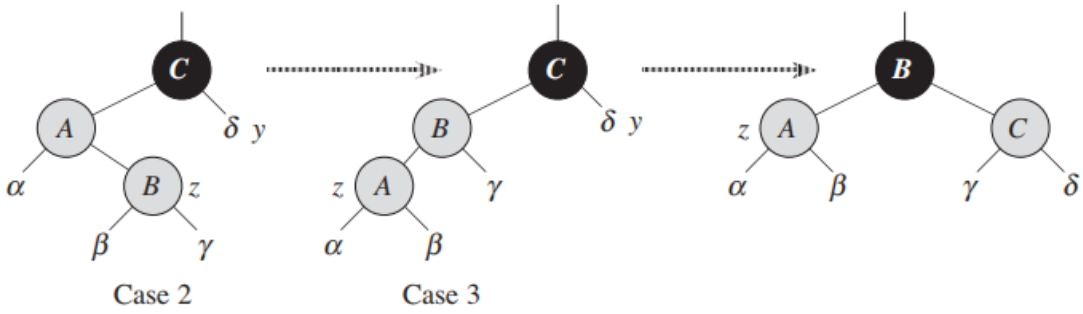
此时的操作：

```
1. 把uncle由红变黑；2. 把parent(A)变黑3. 把爷爷(c)变红；4. 把指针从z移到爷爷
```

Case 2: z's uncle y **is black and** z **is a right child**

case 2和 case 3是相互交织的；

case 2还是违反了性质4；此时用一个**左旋/右旋**直接进入case 3;




```
1.移动指针到parent2.旋转，进入case3
```

Case 3: z's uncle y **is black and** z **is a left child**

```
1.翻转parent和爷爷的color;2.对爷爷调用旋转；
```

插入以后会导致RBT的那些性质会被violated?

- Property 2: 根节点是黑

当树为空时，插入的节点是红色的，这时违反；

- Property 4: 红色节点不能有红色节点，只能有两个黑色节点；（NIL也算黑色）

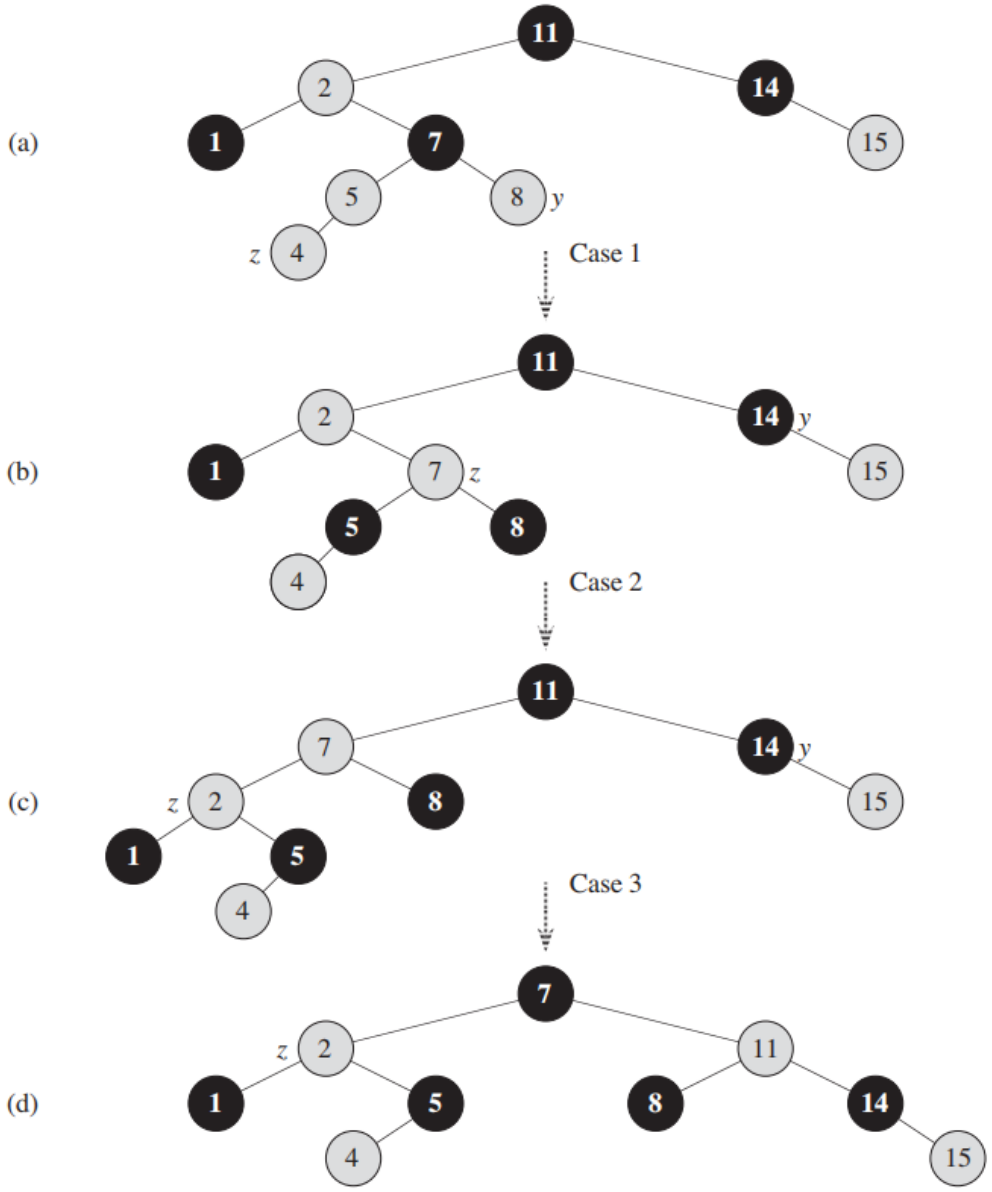


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

13.3. Deletion

13.4. Why Red-Black-Tree?

https://www.quora.com/Difference-between-binary-search-tree-and-red-black-tree

一开始没理解R-B-Tree 比 BST强的地方，谷歌了一下总结如下：

常规的BST不是self-balancing的，因此你的**插入顺序**会导致其他操作的时间复杂度发生变化；

比如：

- if you inserted in order {2, 3, 1}, the BST will be O(log(N))
- however if you inserted {1,2,3}, the BST will be O(N), like a linked list.

而RB-Tree总能自平衡，来确保你的操作总会是$O(lgn)$。

Red Black Tree : best case O(logN), worst case O(logN)
Binary Search Tree: best case O(logN), worst case O(N)

17 Amortized Analysis: 平摊分析

在平摊分析中，执行一系列数据结构操作所需要的时间是通过执行的所有操作求平均而得出的。

17.1. Aggregate analysis

如果堆栈添加一个新的操作`MULTIPOP`来一次性弹出栈顶的 n 个元素：

```
PUSH(S, x):将x压入S
POP（S）：弹出栈顶
MULTIPOP（S, k）：弹出栈顶k个对象
```

在最坏情况下，MULTIPOP操作的时间复杂度为O(n)。

现在开始分析由*n*个*PUSH*， *POP*和*MULTIPOP*操作序列，其作用于一个初始为空的栈：

每个操作的最坏情况是$O(n)$, 因此n个操作序列的代价是$O(n^2)$;

这一分析虽然正确，但是这个bound不够紧凑；

一个对象在每次被压入栈后，至多被弹出一次。所以，调用POP（包括MULTIPOP）的次数至多等于PUSH的次数，即至多为n.。对任意的n，包含n个PUSH, POP, MULTIPOP操作的序列的总时间为O(n). 每个操作的平均代价为O(n) / n = O(1).。

聚集分析中，将每个操作的平摊代价指派为平均代价。所以三个栈操作的平摊代价都是O(1)。

17.2. Accounting methoed: 记账法

直觉是这样的：Accounting Method，要求你先计算出每个操作要"存"多少钱，然后给别的操作消费。

在平摊分析的记帐方法中，对不同的操作赋予不同的费用，某些操作的费用比它们的实际代价或多或少。

我们对一个操作的收费的数量称为平摊代价。当一个操作的平摊代价超过了它的实际代价时，两者的差值就被当作存款(credit)，并赋予数据结构中的一些特定对象，可以用来补偿那些平摊代价低于其实际代价的操作。

数据结构中存储的总存款等于总的平摊代价和总的实际代价之差。注意：总存款不能是负的。

核心思想是PAY IN ADVANCE：提前支付费用

Stack Operations

操作	真实花费	平摊花费
Push	1	2
Pop	1	0
MultiPop	min(k,s)	0

可以看到对于Push操作，平摊花费比真实花费要多1，这个1即使`credit`。即栈中的每个元素都有一个值为1的credit，之后的Pop操作和Multipop操作平摊花费都为0，相当于是使用一个`credit`。因为在Pop前都必须调用了相应数量的Push，所以总和的`credit`永远不会小于0

因此对于一系列n个操作而言，其总共的平摊花费也是O(n)。

17.3. Potential Method: 势能法

原文：

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with data structure D_i. The ***amortized cost*** $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \qquad (17.2)$$

我们假设数据结构$Di-1$在进行了第i个操作后，变为了数据结构Di，其中第i个操作的真实花费为ci，数据Di的势能为$\Phi(Di)$，数据$Di-1$的势能为$\Phi(Di-1)$。在使用势能法时，平摊开销(amortized cost)就是：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

平摊开销 = 真实开销 + 势能差 （核心思想和上一节的acounting 方法一样，把"提前支付"变成了储存势能 -> 释放势能）

经过 n次操作以后, 平摊开销就是：

$$\begin{aligned}\sum_{i=1}^{n}\hat{c}_i &= \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\ .\end{aligned}$$

继续使用stack作为例子：

我们定义potential function Φ 是stack的数据数量；

那么对于空栈D_0来，他的$\Phi(D_0)$就是0；

如果第i次的操作是对一个 长度为 s的堆栈 D_0 进行**PUSH**操作，那么势能差：

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1\ .\end{aligned}$$

求出平摊开销：

By equation (17.2), the amortized cost of this PUSH operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2\ .\end{aligned}$$

如果第 i 次的操作是**MULTIPOP**操作，我们要将 $k' = \min(k, s)$ 个数据对象全部pop出去，那么势能差为：

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'\ .$$

平摊开销：

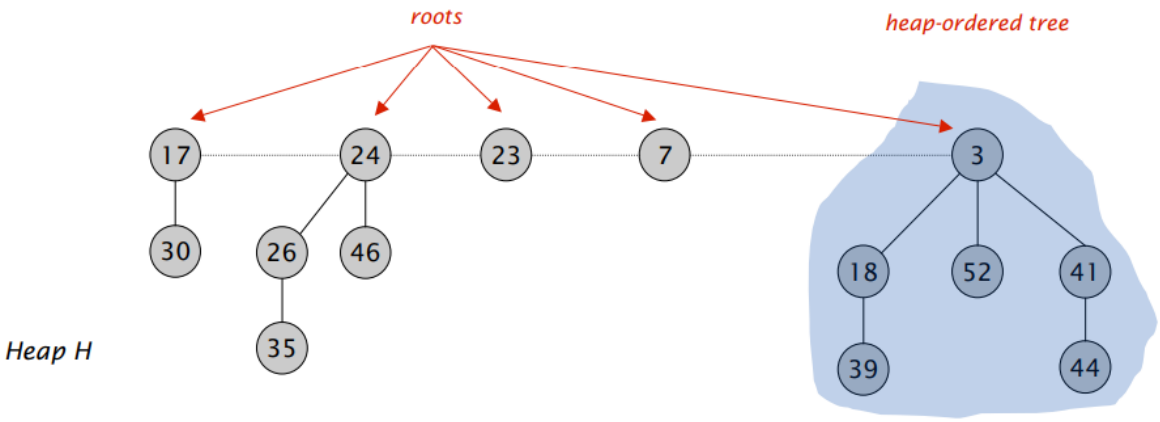
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0\ .\end{aligned}$$

好文：帮助理解

https://www.zhihu.com/question/40156083

19 Fibonacci Heaps: 斐波那契堆

这算法看的比红黑树还眩晕，细节参考原文吧，网上教程也很少；



- 一组满足堆性质（一般是**最小堆**性质）的**树**的集合
- 始终保持一个指针指向最小的元素的对象
- 使用`marked`node属性，来保持heaps的扁平

Mergeable Heaps: 可和并堆:

定义：

> 可合并堆支持以下五个操作，并且每一个元素都有一个`key`：
>
> MAKE-HEAP() creates and returns a new heap containing no elements.
>
> INSERT(H, x) inserts element x, whose *key* has already been filled in, into heap H.
>
> MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
>
> EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.
>
> UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.
>
> Fibonacci 堆额外支持以下两个操作：
>
> DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.[1]
>
> DELETE(H, x) deletes element x from heap H.
>
> 操作的时间复杂度如下：

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

19.1. Structure of Fibonacci heaps

最重要的第一句话就是：

> A **Fibonacci heap** is a collection of rooted trees that are **min-heap ordered**. That is, each tree obeys the **min-heap property**:

即：

1. **斐波那契堆是树的集合，每一棵树都满足最小堆性质；**

就像是$[root1, root2, \ldots, rootN]$, 每个root都是一个树；他们彼此之间用**Double Linked List**串起来;

2. **Fib Heap**的特点是始终维护一个指针指向最小值的节点；

3. root的每一个node x 都有 `x.p` 指向parent, `x.child` 指向一个children;

x的children用**双向链表** 像环一样的连在一起，我们叫他 `child list of x`

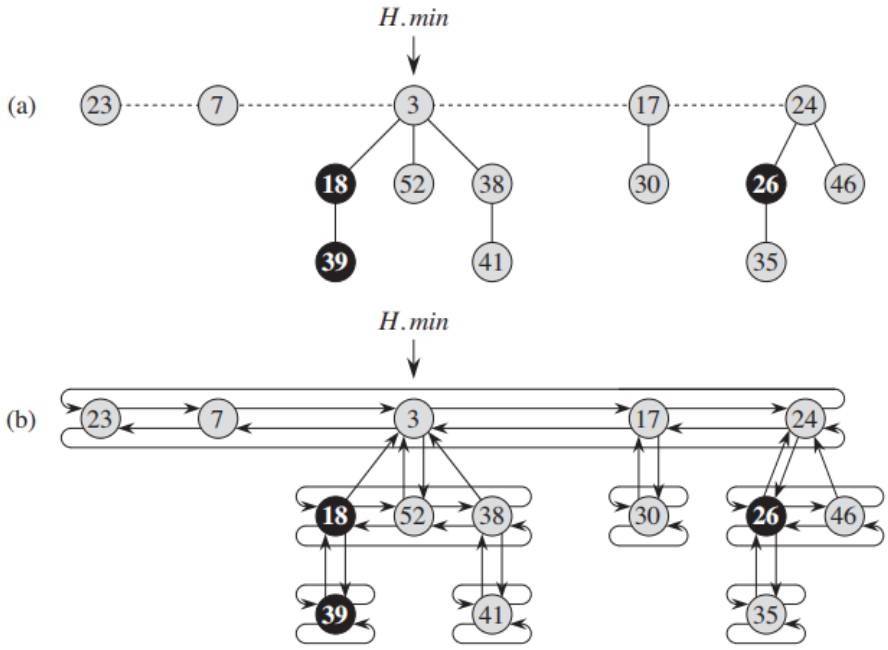


Figure 19.2 **(a)** A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5+2\cdot 3=11$. **(b)** A more complete representation showing pointers p (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). The remaining figures in this chapter omit these details, since all the information shown here can be determined from what appears in part (a).

看伪码会看傻逼的，我总结一下几个操作：

`insert`：斐波那契堆的插入操作

```
1  x.degree = 0
2  x.p = NIL
3  x.child = NIL
4  x.mark = FALSE
5  if H.min == NIL
6       create a root list for H containing just x
7       H.min = x
8  else insert x into H's root list
9       if x.key < H.min.key
10           H.min = x
11 H.n = H.n + 1
```

1-4是初始化，现在你不用管；

若min指针判断堆为空，那就创建rootlist = [x] 这一个元素

不为空，那就和min指针比较一下，然后直接插入root list

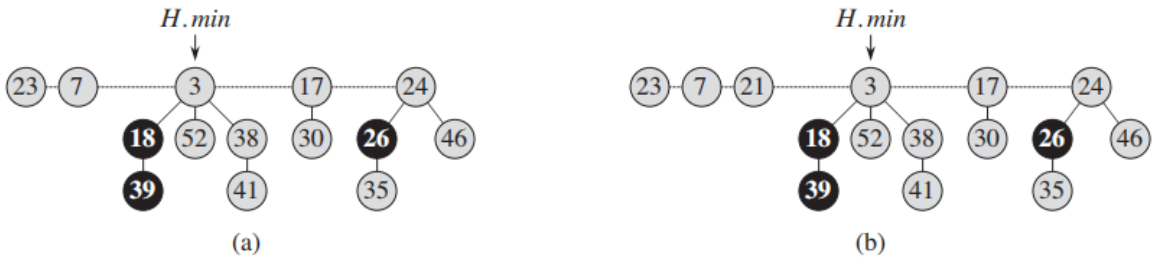


Figure 19.3 Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

注意：

- 我们直接插入root list而不是某个节点的child list

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5      H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

合并两个堆，这个操作比较简单，核心就是直接concatenate两个堆的root list, 更新min指针；

`extract_min()`：最复杂的操作

这个操作会像pop出当前最小的节点，然后调用`consolidate`来确保自己的结构不被破坏；

这是最核心的操作，也是最眩晕的操作：

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3      for each child x of z
 4          add x to the root list of H
 5          x.p = NIL
 6      remove z from the root list of H
 7      if z == z.right
 8          H.min = NIL
 9      else H.min = z.right
10          CONSOLIDATE(H)
11      H.n = H.n − 1
12  return z
```

- 根据`min`指针取出最小对象，say `z`；
- 如果`z`有child, 把所有child先**升级到root list**当中，然后将parent设置为**None**；
- 设置好各种left 和 right指针，把`z`移除；
- 判断：
  - 如果`z`是唯一节点，那么成了空堆，把min指针和root list都设置为None
  - 很可惜他往往不是唯一节点，那么就要调用`consolidate`方法
- 最后将树的节点数-1

`consolidate`：

`degree`的作用在这个函数体现，**我们要确定每个node的`degree`都是不同的**;

- 遍历root list的节点：
  - 如果degree相同，把值小的节点从从root list中拿掉，连接到值大的节点的child方法上；这个操作是用`heap-link`函数来实现的；之后，把值更大的节点的degree + 1

经过这个操作以后，堆结构会变得更加扁平；

关于decrease key操作，降低某个节点的值，所以可能破坏最小堆性质；

因此又是一堆伪码；上张图理解一下intuition吧：

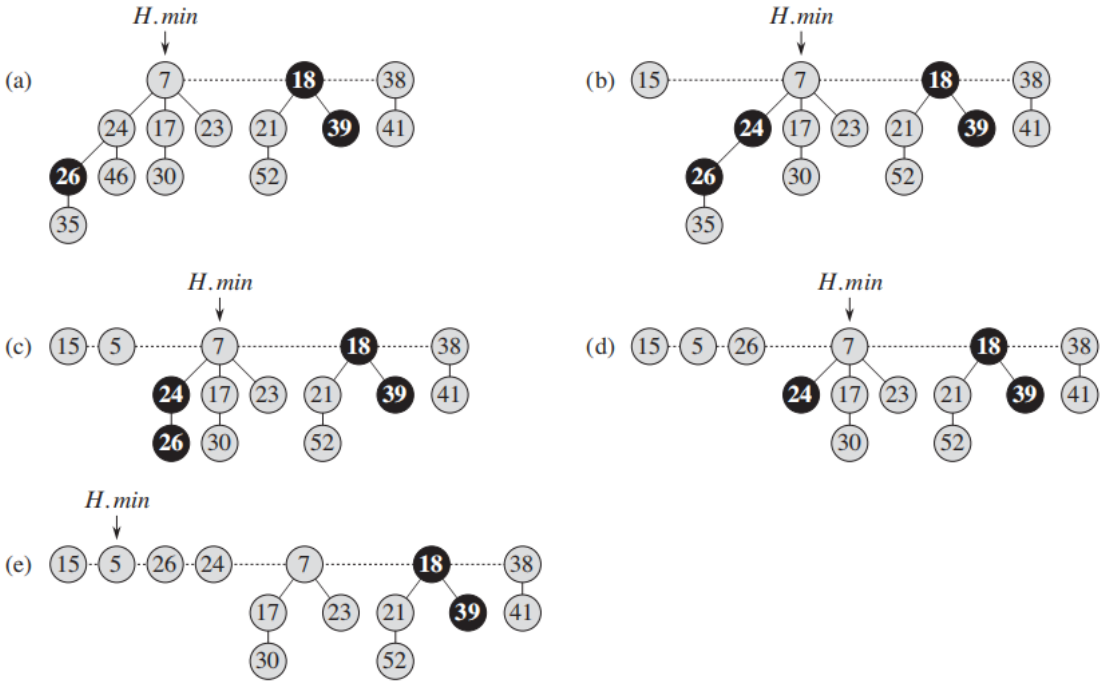


Figure 19.5 Two calls of FIB-HEAP-DECREASE-KEY. **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)–(e)** The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) Part (e) shows the result of the FIB-HEAP-DECREASE-KEY operation, with $H.min$ pointing to the new minimum node.

22 Graph: 图

BFS/DFS: $O(E+V)$

Topological sort: $O(V+E)$

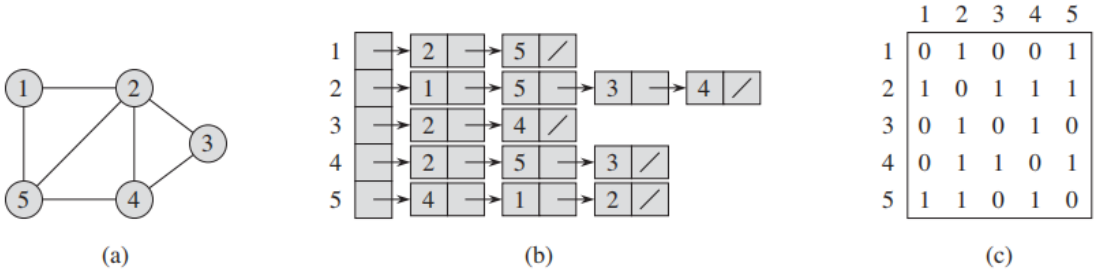
22.1 图的表示：

表示图的标准方法：

$$G=(V,E)$$

Graph是由Vertices定点和Edges边组成的；

无向图有两种表示法:



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

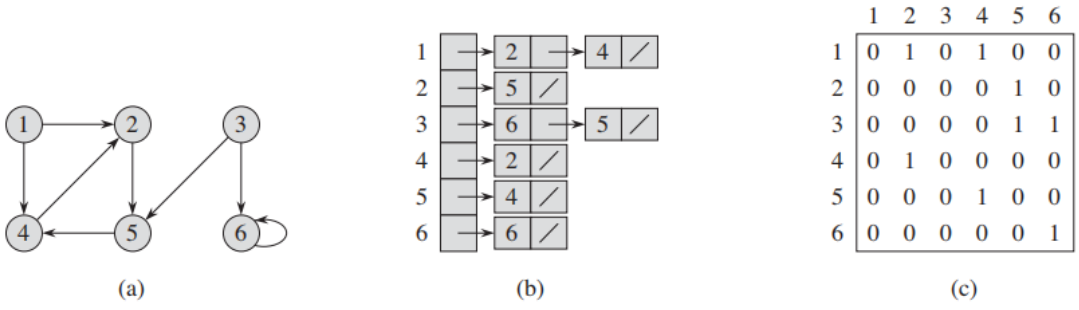
(a): 由5个顶点，7个边组成的无向图

(b): list表示法(adjaceny-list representation) of G

(c): matrix表示法(The adjacency-matrix representation) of G

再无向图中，(c)是对称的;

有向图(directed graph)表示法类似：



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

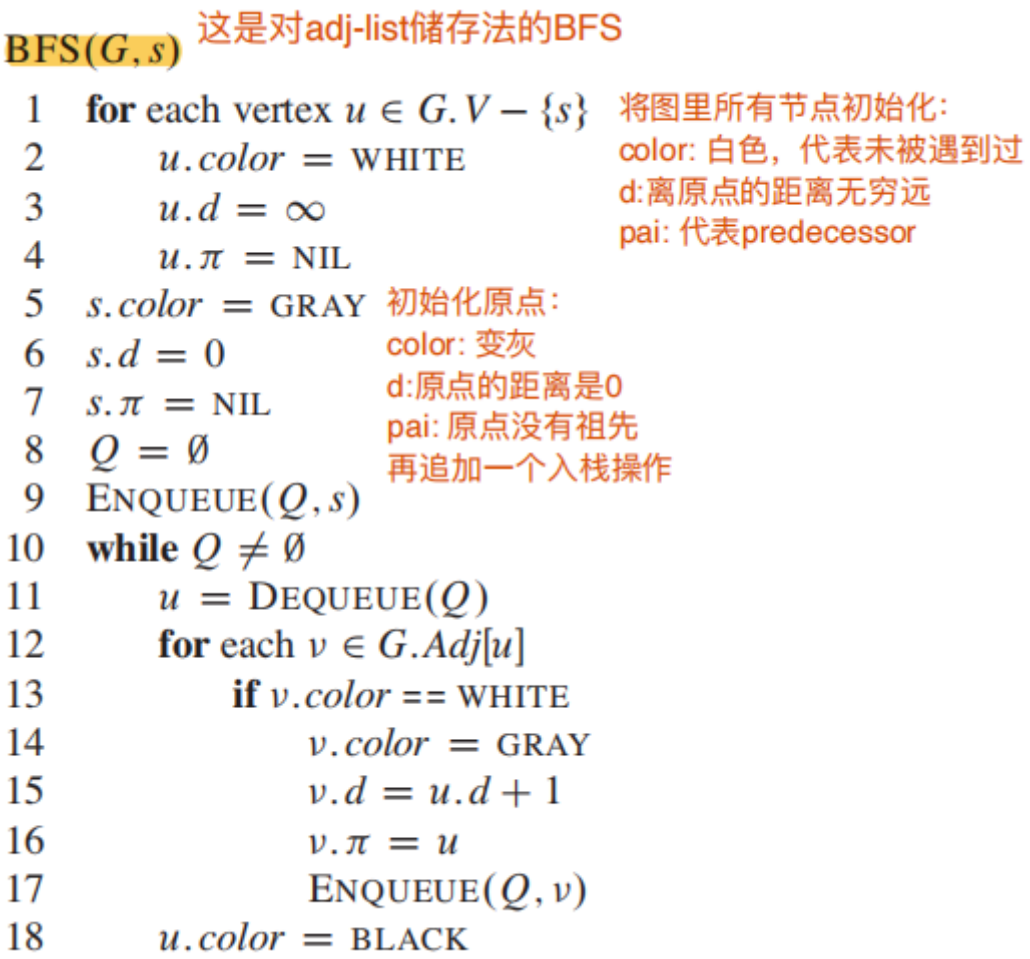
(a):由6个顶点，8个边组成的无向图

(b): list表示法(adjaceny-list representation) of G

(c): matrix表示法(The adjacency-matrix representation) of G

22.2 BFS：广度优先搜索

广度优先搜索，没啥说的了；



这里color = white代表没遇到过的节点；

d表示距离；

π : 代表parent;

这里使用堆栈的方法来储存接下来要开始BFS的节点；

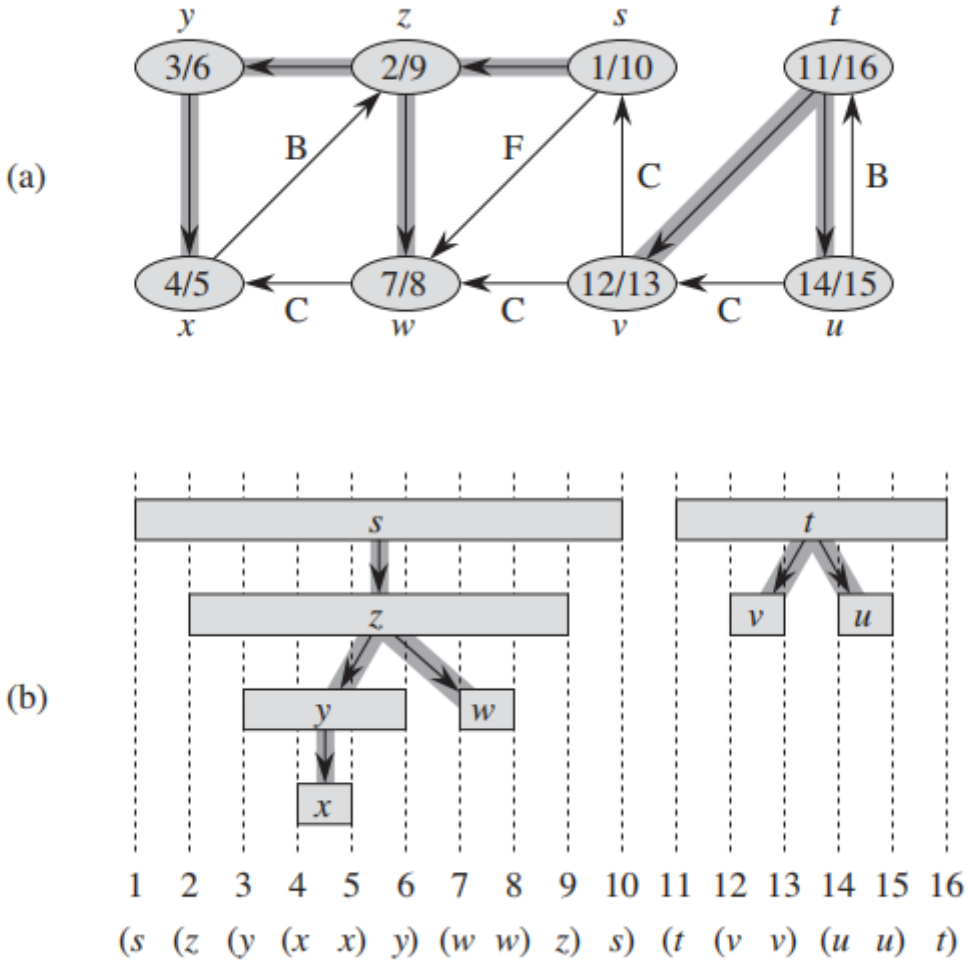
RUNTIME: $O(V+E)$

22.3 DFS： 深度优先搜索

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

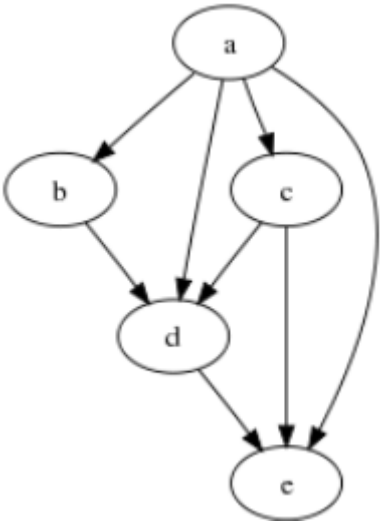
请注意开始和结束的时间，使用DFS来计算终止时间的方法会在后续算法被用到。



22.4 DAG & Topological Sort: 有向无环图 与 拓扑排序

DAG（Directed acyclic graph）有向无环图:

无法形成cycle就是DAG，最后一定会终止在某个点；



严格的定义如下：

因为这样的特性，才能被拓扑排序;

拓扑排序 (Topological Sort)：

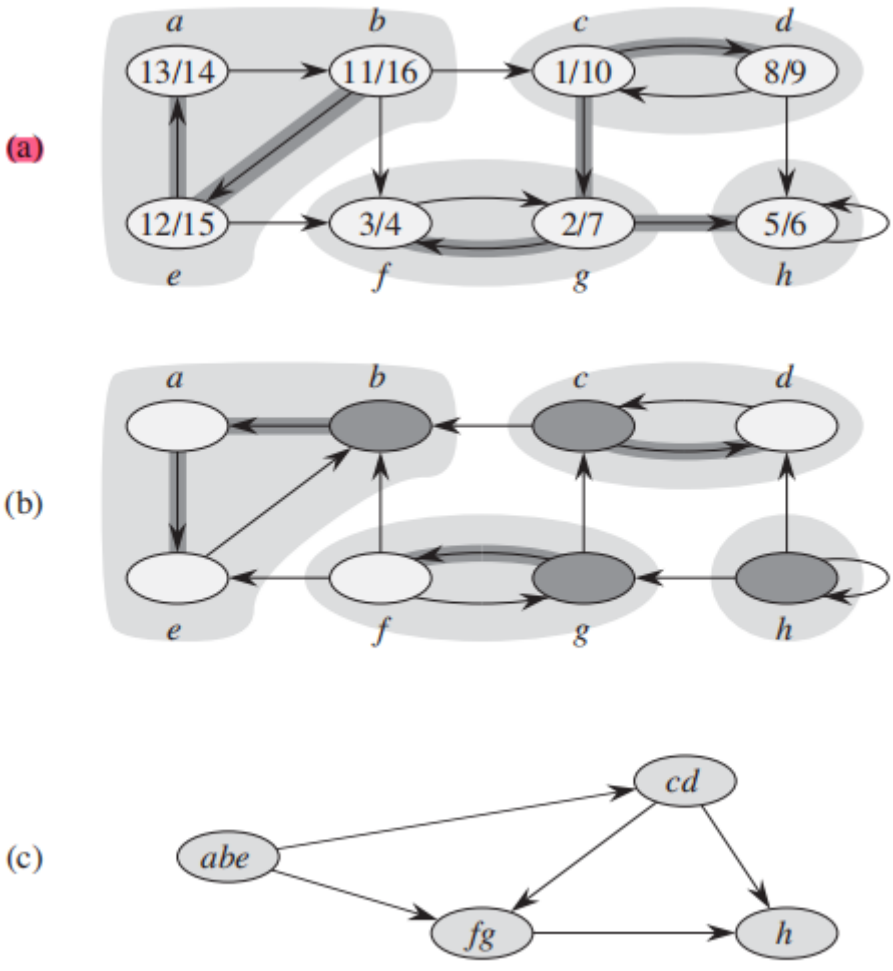
TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

RUNTIME: $O(V+E)$

22.5 Strongly connected components

啥是紧密连接组件？看图就懂了：



严格定义如下：

Strongly connected component of a directed graph $G=(V,E)$ is a maximal set of vertices $C \in V$ such that for every pair of vertices u and v in C, **we have both u~>v and v~>u;** that is, **vertices u and**

v are reachable from each other.

使用DFS来发现紧密组件：

STRONGLY-CONNECTED-COMPONENTS(G)

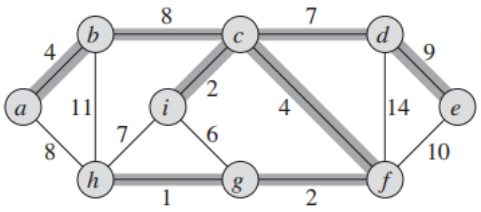
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

23. 最小生成树: Minimum Spanning Trees

Spanning Trees:

A set of edges A that "span" or "touch" all vertices, and forms no cycles

最小生成树往往是在**无向有权图**上来讨论。



阴影部分就是一颗最小生成树

Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

最小生成树不一定是唯一的；

简单的定义：你希望找到一组边：

- **1.连接了所有点**
- **2. 总权重最小**

严格定义如下：

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v. We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

消耗最少的权重，最少的边把所有顶点连接起来

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree,

找到最小生成树的算法的大致**模糊思路**如下，详细的会在下一节展，这里看看就好：

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

A是一组边的集合，一开始设为空集，最终会成为一颗MST（最小生成树）；

如图，在A成为完整MST前，每一步我们都：

- 找到一条 "safe edge", 并加入A

知道A成为完整MST。

因此，核心就是如何判断edge是不是safe的。

有以下的算法能实现上述的思路：

- Prim algorithm
- Kruskal algorithm

在展开算法前，对一些术语下定义：

先给原文，再给我的简单理解：

We first need some definitions. A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. Figure 23.2 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$. We say that a cut ***respects*** a set A of edges if no edge in A crosses the cut. An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. Note that there can be more than one light edge crossing a cut in the case of ties. More generally, we say that an edge is a ***light edge*** satisfying a given property if its weight is the minimum of any edge satisfying the property.

Our rule for recognizing safe edges is given by the following theorem.

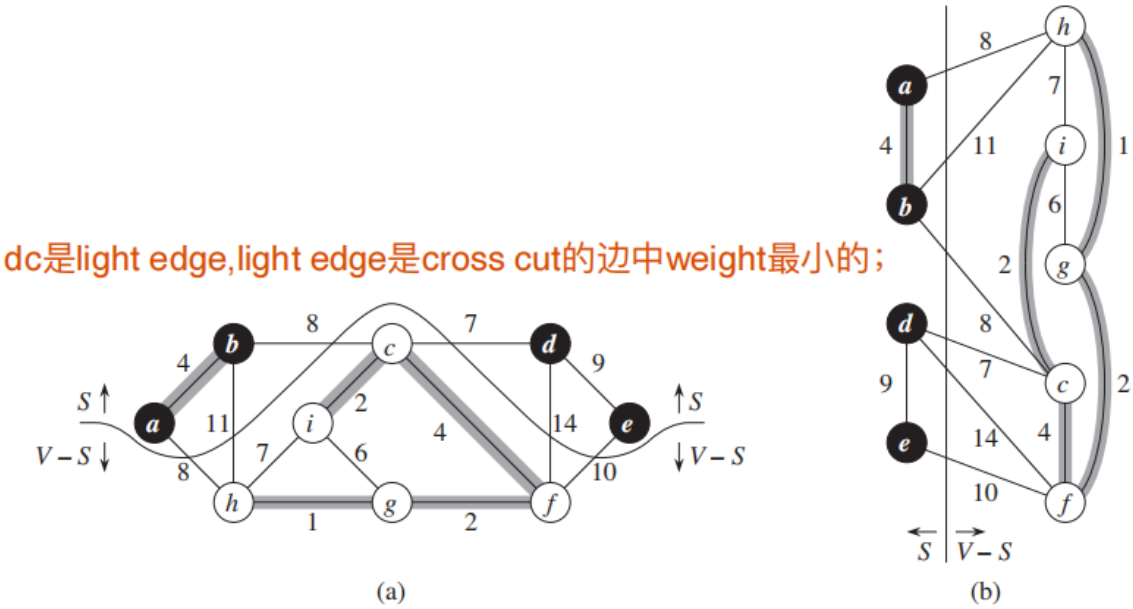
1. **图的切分：cut(S, V - S)**

cut(S, V - S)；

很直观，切分后S代表黑点；V-S是白点，下方；

2. 如果一条边在S和V-S各有一个顶点，那我们说这条边**cross** cut(S, V - S)

3. 在cross的边中，weight最小的边叫做**light edge**.



23.2 最小生成树算法: The algorithms of Kruskal and Prim

两个算法都是贪心算法。

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

> **1.** *Sort all the edges in non-decreasing order of their weight.*
> **2.** *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
> **3.** *Repeat step#2 until there are (V-1) edges in the spanning tree.*

第六行的 `find-set` 操作其实就是检查图是否有形成cycle。

所以，核心就是将边先按照**升序**排序，然后进行遍历；

因此每次都是当前最小权重的edge。

对当前遍历到的边：

- 如果加入这条边后，形成了cycle, 那么跳过这条边；
- 反之，没有形成cycle，那么将当前的边加入A

最后返回。

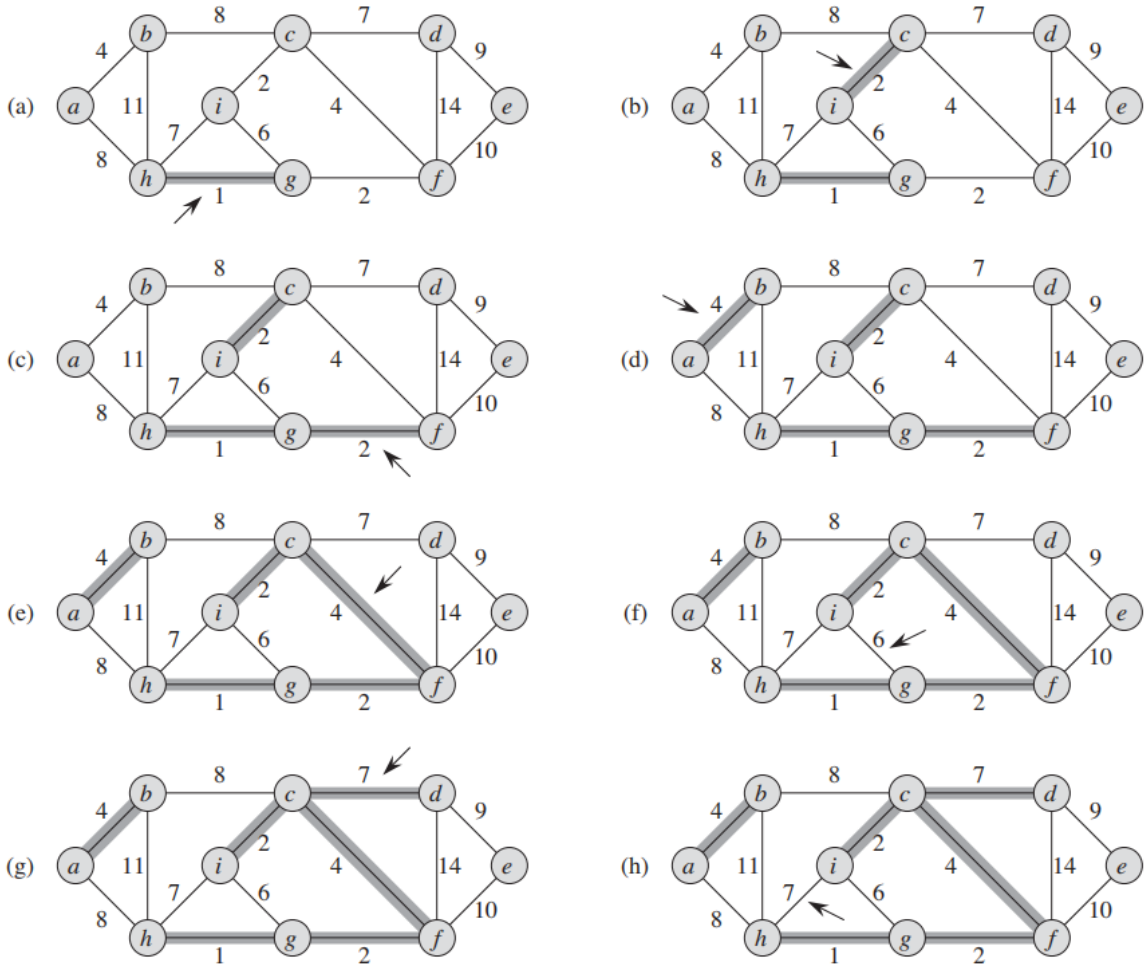
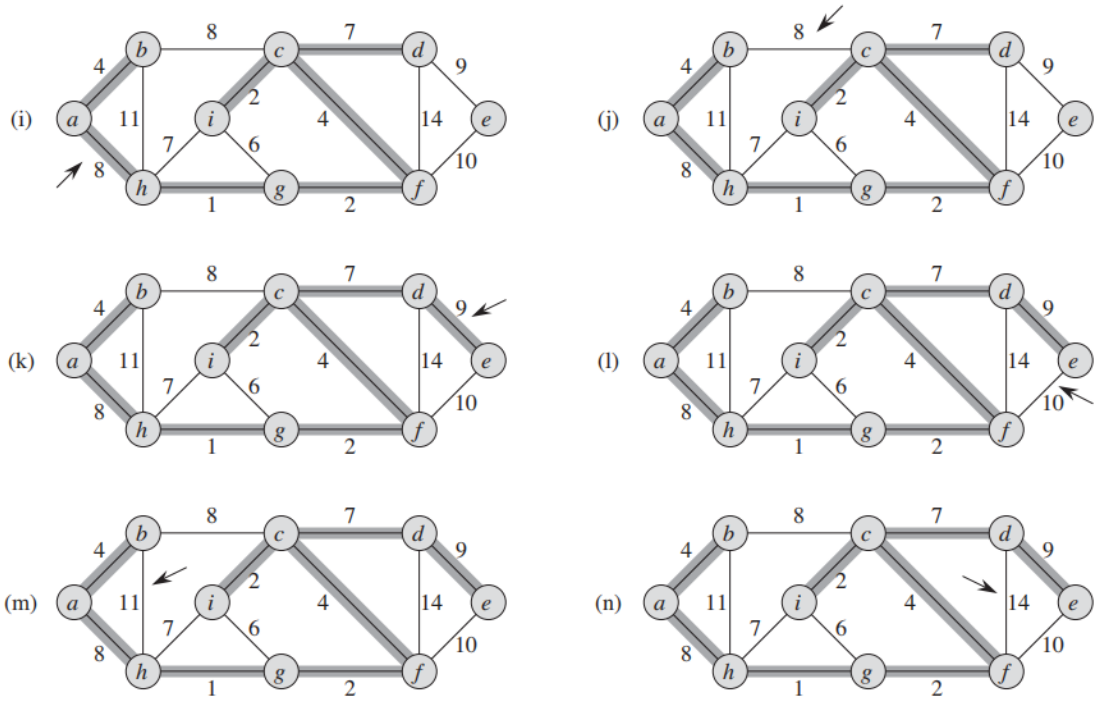


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.



一个简单的并查集算法:

```
# Python Program for union-find algorithm to detect cycle in a undirected graph
# we have one egde for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict


# This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self, num_of_vertices):
        self.V = num_of_vertices
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A utility function to find the subset of an element i
    def find_parent(self, parent, i):
        if parent[i] == -1:
            return i
        if parent[i] != -1:
            return self.find_parent(parent, parent[i])

    # A utility function to do union of two subsets
    def union(self, parent, x, y):
        parent[x] = y

    # The main function to check whether a given graph
    # contains cycle or not
    def isCyclic(self):
```

```
        parent = [-1] * (self.V)

        # Iterate through all edges of graph, find subset of both
        # vertices of every edge, if both subsets are same, then
        # there is cycle in graph.
        for i in self.graph:
            for j in self.graph[i]:
                x = self.find_parent(parent, i)
                y = self.find_parent(parent, j)
                if x == y:
                    return True
                self.union(parent, x, y)


# Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

if g.isCyclic():
    print("Graph contains cycle")
else:
    print("Graph does not contain cycle ")

# This code is contributed by Neelam Yadav
```

Prim's Algorithm

24. Single Source Shortest Paths: 最短路径问题

In this chapter, we shall focus on the ***single-source shortest-paths problem***: given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to each vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants.

Big name Algorithm:

- Dijkstra algorithm
- Floyd Warshall algorithm

定义问题：

本章节关注**单源头**(single source)最短路径问题：给定图 $G(V, E)$,找到一个**给定的点** s 到**图中所有点**的最短路径。

负权重边的影响

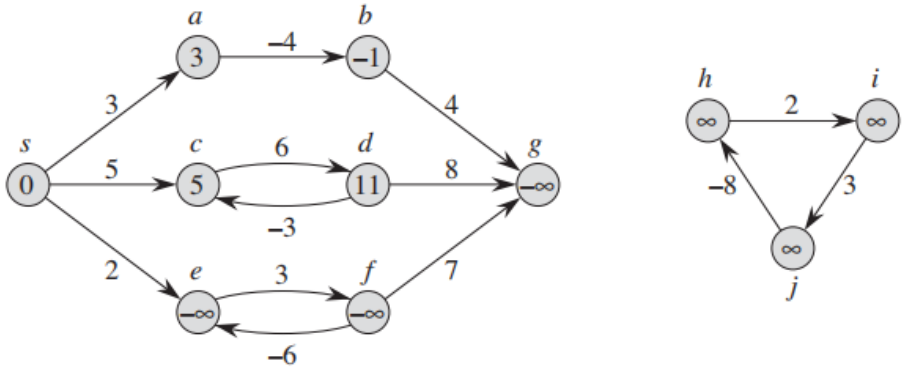


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

- $s-b$ 只有一条路径，此时负权重边无影响；
- $s-d$ 有无限条路径：$<s,c>,<s,c,d,c>,<s,c,d,c,d,c>$, 但是最小权值路径还是 $<s,c>$,因此也不受影响；
- 关于 $s-e$, 同样有无限路径：$<s,e>,<s,e,f,e>,<s,e,f,e,f,e>$, 但是 $<e,f,e>$ 的权重是$3+(-6)=-3<0$, 因此 $<s,e>$ **没有最短路径；**因此我们表示$<s,e>=-\infty$；
- $<s,g>$,**因为g和f是相连的**，因此$<s,g>$**也没有最短路径**；$<s,e>=-\infty$；
- s 永远无法抵达 h,i,j ,所以$<s,h>=<s,i>=<s,j>=\infty$

Dijkstra算法 假设所有边都是**非负数**；

Bellman-Fordrm 算法 没有这种假设；

Relaxation: 更新最短路径的机制

good explanation from :

https://stackoverflow.com/questions/2592769/what-is-the-relaxation-condition-in-graph-theory

- You have two nodes, `u` and `v`

- For every node, you have a *tentative distance* from the source node (for all nodes except for the source, it starts at positive infinity and it only decreases up to reaching its minimum).

你使用relaxation来检测是否能improve到达某个节点的shortest path。(每个节点初始值默认为无穷大)

举个例子:

```
s ~~~~~~~> v
 \         ^
  \        |
   \~~~~~> u
```

比如上图，s是源点，那么:

- 目前**已知**从s出发能到达v, 我们表示为distance(s,v)
- 你也知道s能到u, 表示为distance(s,u)

在使用Relaxation的某个算法的某个时刻遍历到$< u, v >$这条边，就会判断：If `dist[u] + weight(u, v) < dist[v]`, 那么 `s~>u->v` is shorter than `s~>v`, 所以我们应该更新s - v的最短路径!

理解这个很重要，后面直接用Relax来表示这个机制;

24.1 BELLMAN-FORD算法

初始化一个长度是 V 的矩阵 $[0, \infty, \infty, \dots]$，代表源头到其他点的距离，第一项设为0因为是自己到自己的距离;

之后就是Relaxation: 像DP 一样不断更新s 到其他点的最短路径;

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1                使用Relaxation,像DP一样不断更新最短路径
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E            检查是否有负循环，如果有直接Return False
6      if v.d > u.d + w(u, v)
7          return FALSE                  如果没有负循环，则返回最小路径和权重;
8  return TRUE                           Return True if 有负循环(negative weight cycle)
```

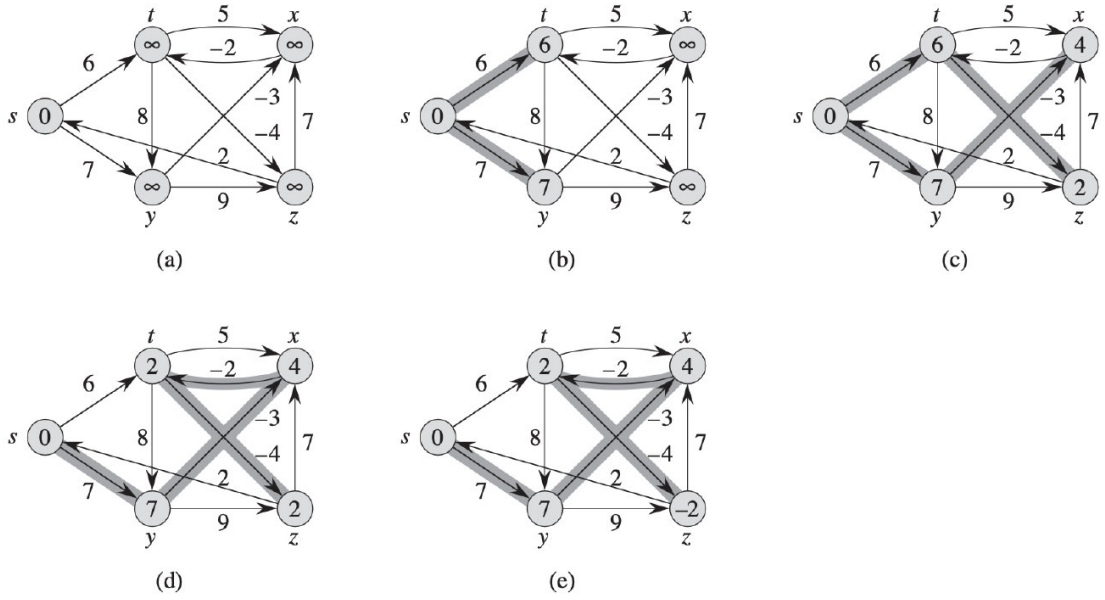


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

```
      2
V0-----------V1
 \             \
4 \             \ 5
   \             \
   V2------------V3
          2
```

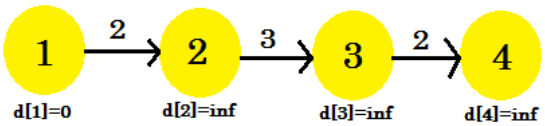
书上例子不太好，用这个走一遍：

- 1. 初始化$dist = [0, inf, inf, inf]$
- 2. 走到边$< v0 - v1 >$, $dist[0] + 2 < dist[1]$ 成立, 更新$dist = [0, 2, inf, inf]$
- 3. 走到边$< v0 - v2 >$, $dist[0] + 4 < dist[2]$ 成立, 更新$dist = [0, 2, 4, inf]$
- 4. 走到边$< v1 - v3 >$, $dist[1] + 5 < dist[3]$ 成立, 更新$dist = [0, 2, 4, 7]$
- 5. 走到边$< v2 - v3 >$, $dist[2] + 2 < dist[3] = 7$ 成立, 更新$dist = [0, 2, 4, 6]$

这是第一遍，**我们总共要走$V - 1$遍才能结束**，但是后面的两遍都不会有更新了；

为啥要走$V - 1$遍呢？目前我发现和**储存顺序有关**；

再来一个例子：



如果你按照顺序 1-2, 2-3, 3-4来储存，可以一遍过；

但是如果是3-4, 2-3, 1-2，你需要$V-1$ 也就是三次才能走对；

因为如果当前节点之前没遇到过，$dist[\text{当前节点}]$就是∞, 无法被更新；

在程序里会判断**每一步**的出发点是否是NIL; 可以去代码试一下；

24.2 DAG shortest path: DAG的最短路径

657

对拓扑排序好的DAG，按照顺序用一遍RELAX即可：

DAG-SHORTEST-PATHS(G, w, s)
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** each vertex u, taken in topologically sorted order
4 **for** each vertex $v \in G.Adj[u]$
5 RELAX(u, v, w)

RUNTIME: $O(V+E)$

24.3 Dijkstra 算法

Dijkstra算法比Bellman-ford更快，但是需要图中不存在负循环；

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

- 2,3行的 S 代表最短路径集合，而 Q 是没探索的点，也就是 $V-S$，一开始就是所有点$G.V$; Q是用最小优先队列(堆算法)实现的，有 `EXTRACT-MEAN` 方法；还需要 `MIN-HEAPIFY` 方法；

只要 Q 不空就循环：

- 从Q

25 All-Pairs Shortest Paths

> In this chapter, we consider the problem of fifinding shortest paths between all pairs of vertices in a graph.

上一张是sinle source shortest path (SSSP)，求出源点到其他所有点的最短距离;

本章节关注**All-Pairs Shortest Paths** (APSP), 求出所有点之间的最短距离。

25.1 Matrix Multiplication算法

使用DP的方法不断更新最小路径；

回顾一下矩阵A 乘以 矩阵B的是matrix multiplycation算法：

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

APSP图算法和这个极为相似：

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

时间复杂度是$O(n^3)$

解释一下 l_{ij}^m :

l_{ij}^m 代表了： 在最多m 条边的情况下, 点i 到 点j 的最小weight. 因此：

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \neq j \ . \end{cases}$$

如果m大于1:

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \ . \end{aligned}$$

:

这里是DP的思想，遍历所有中介点k来尝试更新 i 到 j 能不能有更短的距离；*(原书的版本写的实在太绕了，希望我的总结能帮你get到 intuition :))*

总体的APSP算法如下：

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  for m = 2 to n − 1
4       let L^(m) be a new n × n matrix
5       L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  return L^(n−1)
```

总耗时$O(n^4)$.

总体思想如下：

$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &&= W\,, \\
L^{(2)} &= L^{(1)} \cdot W &&= W^2\,, \\
L^{(3)} &= L^{(2)} \cdot W &&= W^3\,, \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}\,.
\end{aligned}
$$

这是一个Bottom up的算法，随着m的增加，我们不断更新L矩阵；到最后m = n-1,就得到了全局的最小权重路径；

这个算法又绕又慢，不看也罢；

25.2 The Floyd-Warshall algorithm

时间复杂度：$O(V^3)$

简单粗暴多了(可以直接看下面的改进版，忽略这个)

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for i = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
8  return D^(n)
```

改进版：

W是矩阵表示法的图，输入进去：

这个算法我们只需要维护一个矩阵，减少了空间占用；

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              d_ij = min(d_ij, d_ik + d_kj)
7  return D
```

一句话概括就是在所有i 到 j 的中间再遍历一层 k, 使用DP方法动态更新 i 到 j 的最短路径；

顺便贴一下leetcode:[https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/]

26 Maximum Flow：最大流问题

Flow networks

> A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.

Flow networks是有向图，每一条边多了一个运载能力(capacity),代表通过这条边的上线；(想象网络负载的场景)

在这一章我们主要关注 Maximum flow problem:

> In maximum flow problem, we are given a flow network G with **source s and sink t**, and we wish to find a flflow of maximum value.

这里又引入了两个概念：source 和 Sink

26.2 The Ford-Fulkerson method

注意，这里叫Method的原因是这个算法没有准确的runtime,会根据你的实现方式/数据输入发生很大的变化；

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

第2,3句包含的信息量太大了，第一次看容易懵逼。

为了理解这个算法，引入残差网络的概念：

Residual networks

这个ResNet不是CV里的那个resnet。

注意，普通的network flow只允许单向边，单向边代表了flow的大小；

Residual network只是在原有基础上，借助原有的capacity属性浓缩了更多信息，并且变成了双向的(也可能是单向)。

统一一下数学说法：

- $c(u,v)$ 来表示节点的capacity；
- $f(u,v)$ 表示当前的flow的大小；

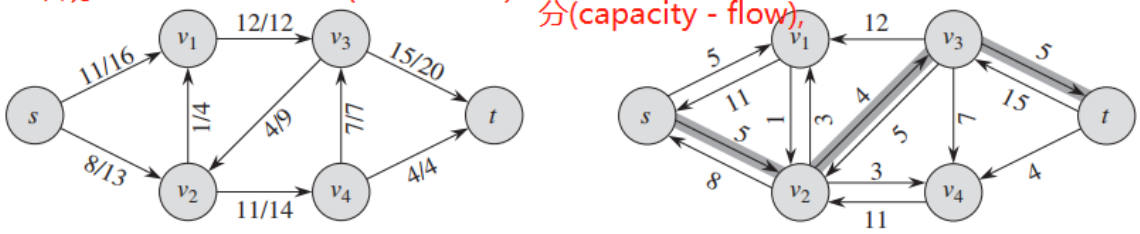
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{26.2}$$

augmentation

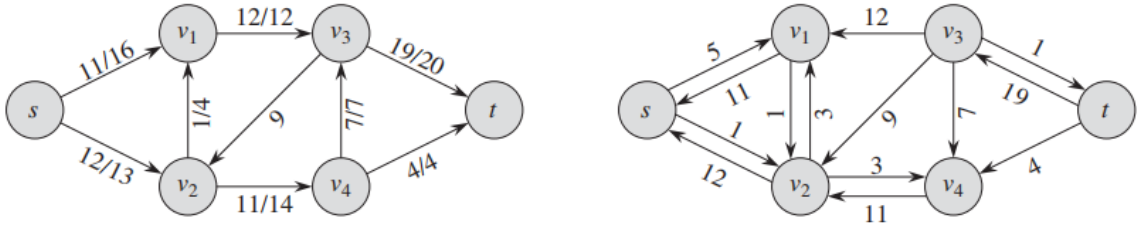
$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

常规的flow network G (边是单向的)

G的残差网络Gf，正向代表了还可以增加的部分(capacity - flow)，



反向代表了当前的flow;
增加阴影部分可以增加maximum flow



将e(v3,v2)原本的阴影部分的4加满，当前的flow = capacity,
残差网络不显示没有augmentation 能力的边

Figure 26.4 **(a)** The flow network G and flow f of Figure 26.1(b). **(b)** The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

Theorem 26.6 (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

FORD-FULKERSON(G, s, t)

```
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

26.3 Bipartite Matching

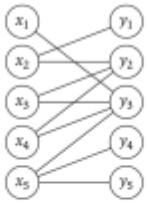


Figure 7.1 A bipartite graph.

- *Bipartite Graph*: a graph $G(V, E)$ where
  1. $V = X \cup Y$, X and Y are disjoint and
  2. $E \subseteq X \times Y$.

Algorithm for Bipartite Graph Matching

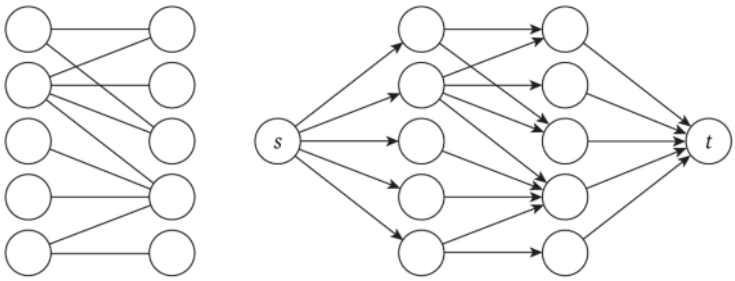


Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

- Convert G to a flow network G': direct edges from X to Y, add nodes s and t, connect s to each node in X, connect each node in Y to t, set all edge capacities to 1.
- Compute the maximum flow in G'.
- Claim: the value of the maximum flow is the size of the maximum matching.

26.4 Push-Relabel Algorithm

Push-Relabel Algorithm 比 Ford-Fulkerson algorithm 更快;

push-relabel approach based algorithm that works in $O(V^3)$

和Ford-Fulkerson algorithm 一样，Push-Relabel Algorithm也是使用残差网络;

Push-Relabel的机制更加注重当前的点位而不是全局;

```
Preflow()
1) Initialize height and flow of every vertex as 0.
2) Initialize height of source vertex equal to total
   number of vertices in graph.
3) Initialize flow of every edge as 0.
4) 对于源点的临近点， flow and excess flow is equal to capacity initially.
```

1. **Push()** is used to make the flow from a node which has excess flow. If a vertex has excess flow and there is an adjacent with smaller height (in residual graph), we push the flow from the vertex to the adjacent with lower height. The amount of pushed flow through the pipe (edge) is equal to the minimum of excess flow and capacity of edge.

1. **Relabel()** operation is used when a vertex has excess flow and none of its adjacent is at lower height. We basically increase height of the vertex so that we can perform push(). To increase height, we pick the minimum height adjacent (in residual graph, i.e., an adjacent to whom we can add flow) and add 1 to it.

这个算法的代码几乎搜不到，花了几天看懂以后真觉得是个蛮神奇的算法，第一次有算法能给我"立体"的感觉。

算法的Intuition：

先看一下算法导论原文的伪码：

GENERIC-PUSH-RELABEL(G)
1 INITIALIZE-PREFLOW(G, s)
2 **while** there exists an applicable push or relabel operation
3 select an applicable push or relabel operation and perform it

```
1. 初始化
2. while 有任何一条边能进行push或者relabel操作:
        执行 push 或者 relabel
```

一开始看的一头雾水。

- 首先，之前说了，算法允许一个点的inflow大于outflow; 因此，有一个**蓄水池**的机制，用来储存这个多出来的**excess flow**；
- 同时, 我们赋予了每个点新属性：`height` (代表高度)

现在，把图想象成水管(edge)和连接点(vertex)。

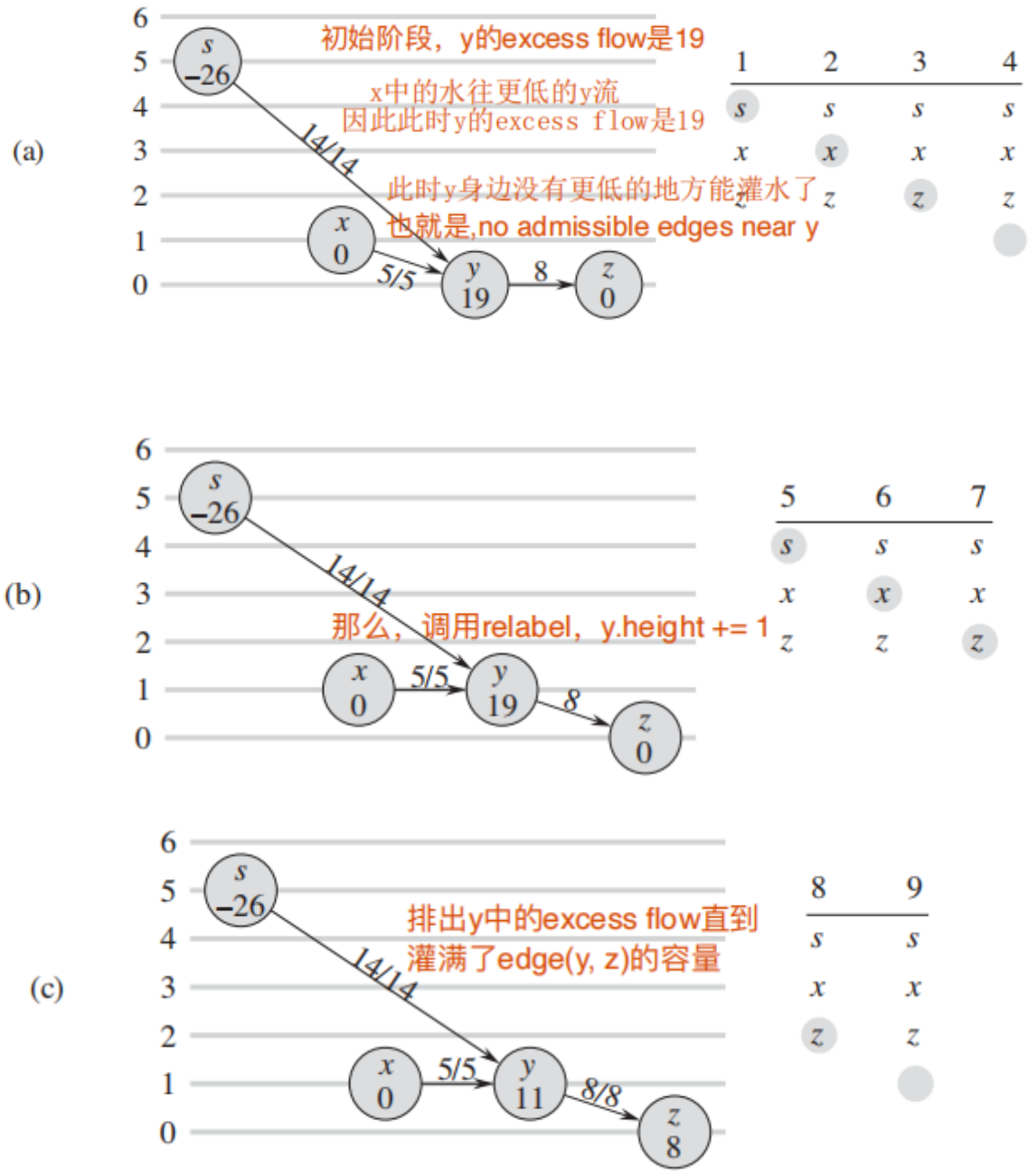
1. 起初，source高度是最高的，其他点的高度都是 0；**水往低处流**，因此source向邻近点注射水流(flow)，根据source的邻边的capacity总和，直接有多少capacity就发射多少的flow；(因此通常会"超发"，多出来的水就是excess flow，就会触发蓄水池)
2. 接着就进入了while：
    - 只要周围有高度更低的边，我们就让**水往低处流**，使用`push`
    - 否则，使用`relabel`来增加$height$, 确保自己比周围一个点高
    - 如果蓄水池中所有点 **(除了source)** 的 $excess flow == 0$，跳出while

这个算法精髓的地方在于，一开始我们"超发"水流，就会导致有$excess flow$;

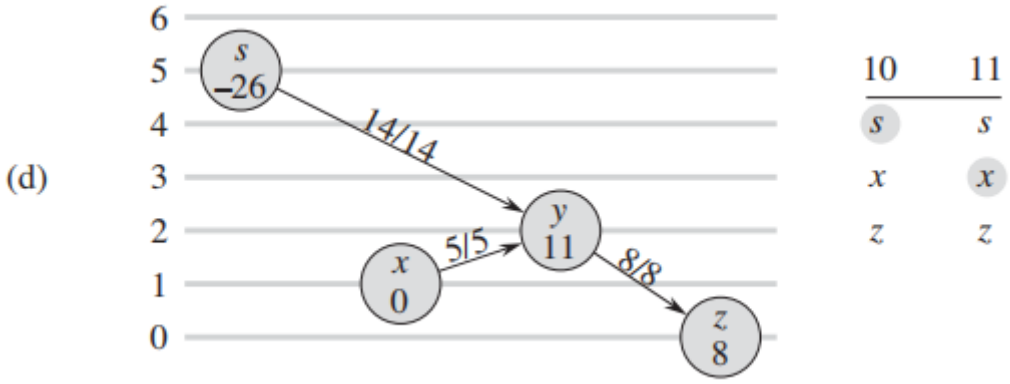
但是到最后，`relabel`会把点的 $height$ 升高到 $source.height$ 以上, 让那些一开始超出capacity的流最终从蓄水池流回source，这样最终的流全部都是legal的， 我们也就得到了正确的最大流。

这句话理解了，你就get到了核心的思想!

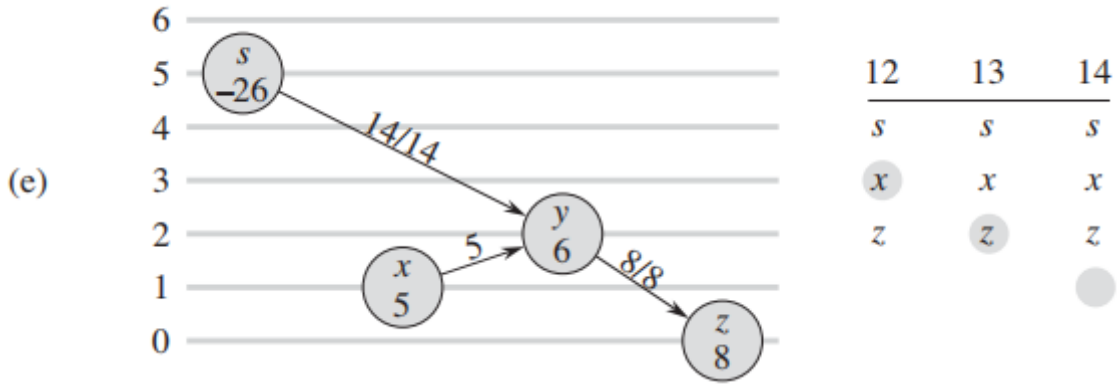
下图来自算法导论原文p752, 纵轴代表了点的height属性



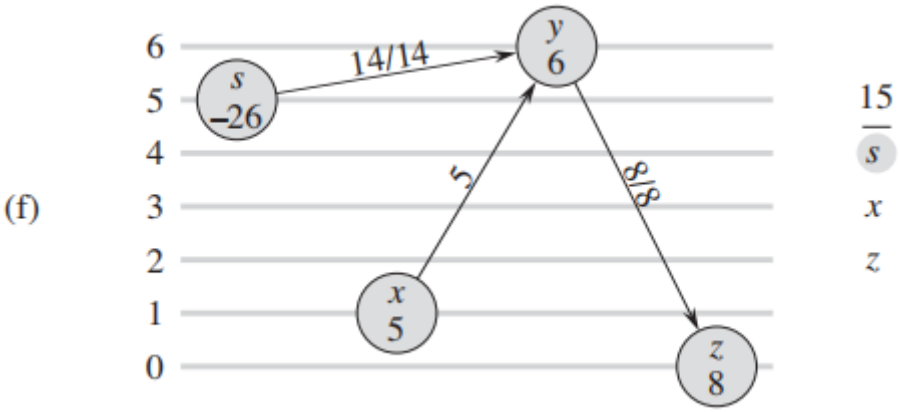
d. 由于周围没有admissive edge, 继续调用 `relabel` 升高高度:



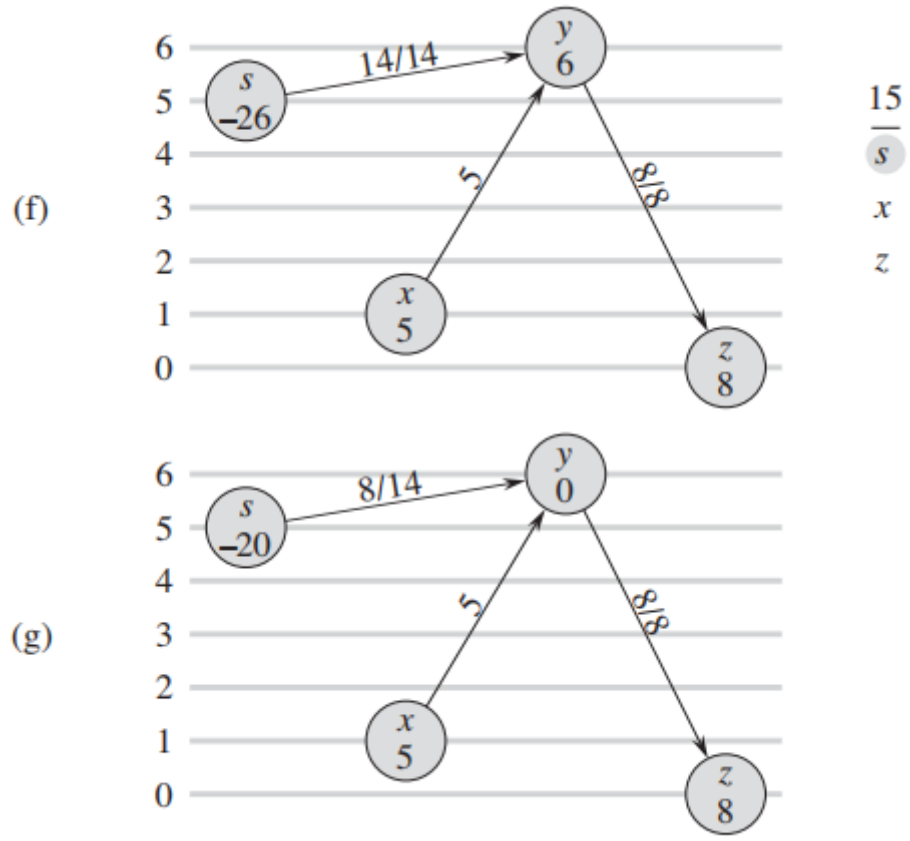
e: 调用 `relabel` 以后，y 中的excess flow灌回了 x:



f: 没有admissive edge继续升高 y:



f,g: y的flow灌回了source以后，没有了excess flow,算法终止。



具体实现稍微和原文有些出入，但是思想是一样的；

PUSH(u, v)

1 // **Applies when**: u is overflowing, $c_f(u, v) > 0$, and $u.h = v.h + 1$.
2 // **Action:** Push $\Delta_f(u, v) = \min(u.e, c_f(u, v))$ units of flow from u to v.
3 $\Delta_f(u, v) = \min(u.e, c_f(u, v))$
4 **if** $(u, v) \in E$
5 $\quad (u, v).f = (u, v).f + \Delta_f(u, v)$
6 **else** $(v, u).f = (v, u).f - \Delta_f(u, v)$
7 $u.e = u.e - \Delta_f(u, v)$
8 $v.e = v.e + \Delta_f(u, v)$

RELABEL(u)

1 // **Applies when:** u is overflowing and for all $v \in V$ such that $(u, v) \in E_f$,
$\quad$ we have $u.h \leq v.h$.
2 // **Action:** Increase the height of u.
3 $u.h = 1 + \min\{v.h : (u, v) \in E_f\}$

```
INITIALIZE-PREFLOW(G, s)
1   for each vertex v ∈ G.V
2       v.h = 0
3       v.e = 0
4   for each edge (u, v) ∈ G.E
5       (u, v).f = 0
6   s.h = |G.V|
7   for each vertex v ∈ s.Adj
8       (s, v).f = c(s, v)
9       v.e = c(s, v)
10      s.e = s.e − c(s, v)
```

唯一要注意的是初始化操作`preflow`的7-10行将source周围的点根据capacity属性直接灌满了；

这个算法和ford-fulkerson注重全局的思路不一样，Push-Relabel Algorithm更加localized, 也就会不停的更新当前的点，直到这个点没有 $excess flow$ 为止。这个操作就是`Discharge`完成的。

```
DISCHARGE(u)
1   while u.e > 0
2       v = u.current
3       if v == NIL
4           RELABEL(u)
5           u.current = u.N.head
6       elseif c_f(u, v) > 0 and u.h == v.h + 1
7           PUSH(u, v)
8       else u.current = v.next-neighbor
```

Python Implementation:

```python
# push-relabel algorithm

def MaxFlow(C, s, t):
    n = len(C)  # C is the capacity matrix
    F = [[0] * n for i in range(n)]

    # the residual capacity from u to v is C[u][v] - F[u][v]
    height = [0] * n  # height of node
    excess = [0] * n  # flow into node minus flow from node
    seen = [0] * n  # neighbours seen since last relabel
    # node list other than s and t
    nodelist = [i for i in range(n) if i != s and i != t]

    def push(u, v):
        send = min(excess[u], C[u][v] - F[u][v])
        F[u][v] += send
        F[v][u] -= send
```

```
        excess[u] -= send
        excess[v] += send

    def relabel(u):
        '''
        find smallest new height from neighbor for making a push possible
        '''
        min_height = float('inf')
        for v in range(n):
            if C[u][v] - F[u][v] > 0 and v != u:
                min_height = min(min_height, height[v])
                height[u] = min_height + 1

    def discharge(u):
        '''
        An overflowing vertex u is discharged by pushing all of its excess flow
through
        admissible edges to neighboring vertices. Perform relabel if nessesary.
        '''
        while excess[u] > 0:
            if seen[u] < n and seen[u] != u:  # check next neighbour
                v = seen[u]
                #if admissive and height greater than nerghbor
                if C[u][v] - F[u][v] > 0 and height[u] > height[v]:
                    push(u, v)
                else:
                    seen[u] += 1
            else:  # we have checked all neighbours. must relabel
                relabel(u)
                seen[u] = 0

    height[s] = n  # longest path from source to sink is less than n long
    excess[s] = float("inf")
    # send as much flow as possible to neighbours of source
    for v in range(1,n):
         push(s, v)

    p = 0
    while p < len(nodelist):
        u = nodelist[p]
        old_height = height[u]
        discharge(u)
        if height[u] > old_height:
            nodelist.insert(0, nodelist.pop(p))  # move to front of list
            p = 0  # start from front of list
        else:
            p += 1
    return sum(F[s])


# C = [[0, 3, 3, 0, 0, 0],
#      [0, 0, 2, 3, 0, 0],
#      [0, 0, 0, 0, 2, 0],
#      [0, 0, 0, 0, 4, 2],
#      [0, 0, 0, 0, 0, 2],
#      [0, 0, 0, 0, 0, 3]]

#Example given in CLRS p752
```

```
C = [[0,0,14,0],
     [0,0,5,0],
     [0,0,0,8],
     [0,0,0,0]]

source = 0  # source
sink = 3  # sink

max_flow_value = MaxFlow(C, source, sink)
print("Push-Relabeled(Preflow-push) algorithm")
print("max_flow_value is: ", max_flow_value)
```

Bit Manipulation: 位运算算法

原码

是一种计算机中对数字的二进制定点表示方法。原码表示法在数值前面增加了一位符号位（即最高位为符号位）：正数该位为0，负数该位为1（0有两种表示：+0和-0），其余位表示数值的大小。

比如说 `int` 类型的 `3` 的原码是 `11B` （B表示二进制位，这里你可以多了解一些进制之间的转换），在32为机器上占4个字节，所以，高位补0就是：

```
00000000 00000000 00000000 00000011 # 一个字节8个bit位
```

那么 `int` 类型的 `-3` 的绝对值的二进制位就是 `11B` 展开后最高位补1就是：

```
10000000 00000000 00000000 00000011
```

缺点：

- 原码中的0分为 `+0` 和 `-0`
- 在进行不同符号的加法运算或者同符号的减法运算时，不能直接判断出结果的正负，我们必须要将两个值的绝对值进行比较。然后再进行加减操作。

反码

正数的反码就是原码：

那 `int` 类型的 `-3` 的反码是，让我们默念公式：负数的反码等于原码除符号位以外所有位取反！

```
10000000 00000000 00000000 00000011  # -3的原码
11111111 11111111 11111111 11111100  # 最高位为符号位，不变，其余取反
```

补码

在计算机系统中，数值一律用补码来表示和存储。

原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理。此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路支持。

记住口诀：正数的补码与原码相同，负数的补码为其原码除符号位外所有位取反（这就是反码了），然后最低位加1。

还是那个 `int` 类型的 `3` 的补码是：

```
00000000 00000000 00000000 00000011  # 正数的补码与原码一致
```

那么 `int` 类型的 `-3` 的补码就是，让我们手掐口诀：

```
10000000 00000000 00000000 00000011  # -3的原码
11111111 11111111 11111111 11111100  # 负数的补码为其原码除符号位外所有位取反
11111111 11111111 11111111 11111101  # 然后最低位加1，完美！
```

如何check if 一个数是2的次方?

方法1:

如果数字能不停被2整除，并且最终等于1，那么是2的次方;

否则不是。

这个方法的时间复杂度是$O(lgn)$

```
def isPowerOfTwo(n):    if (n == 0):        return False    while (n != 1):
       if (n % 2 != 0):                return False            n = n // 2
          return True
```

方法2:

使用位运算。

如果一个数是2的次方，那么位运算: `n & (n - 1)` 结果必定是0。 (只要n > 0)

为啥呢?

2, 4, 8, 16 .... 这样的2的次方的数字的二进制数都只有第一位是1;

只要把他们减一，比如:

把4 - 1 = 3 –> 011
 16 - 1 = 15 –> 01111

而4 = 100

16 = 10000,

这样如果进行 `&` 运算，就不会有任何一位相等; 4&3 == 0， 16&15 == 0;

所以我们的函数:

```
def isPowerOfTwo(x):    return (x and (not(x & (x - 1))))
```

二进制数中 1 的数量

首先, python中有内置函数，来count1:

```
>>> bin(5)'0b101'>>> bin(5).count('1')2
```

但是这不是我们想要的。

玩的就是old school:

继续上一个算法 ， 每次一个数字n被减去 1, **最右边的1 和 再往右的数字就会被翻转;**

因此这个神奇的操作:

```
n & (n - 1)
```

每次会让原来的数字中少一个1；(每次转掉一个1)

就这样用循环来count 1的数量即可;

以23 和 22 为例子:

23： 10111

22： 10110

23 & 22 = 10110 = 22

21: 10101

22 & 21 = 10100 = 20

19: 10011

20 & 19 = 10000 = 16

16 & 15 = 0

来一题LC：汉明距离:

https://leetcode.com/problems/hamming-distance/

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        x = x^y
        res = 0
        while x:
            x = x & (x - 1)
            res += 1
           return res
```

先进性异位运算，在count("1")即可。

另外，leetcode大神常使用

```
if num & 1:
    then num is odd
else:
    num is even
```

来判断一个输入`num`是奇数还是偶数；

附录

Logarithms

We shall use the following notations:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} ,\\
\ln n &= \log_e n && \text{(natural logarithm)} ,\\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} ,\\
\lg\lg n &= \lg(\lg n) && \text{(composition)} .
\end{aligned}$$

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$\begin{aligned}
a &= b^{\log_b a} ,\\
\log_c(ab) &= \log_c a + \log_c b ,\\
\log_b a^n &= n \log_b a ,\\
\log_b a &= \frac{\log_c a}{\log_c b} , && (3.15)\\
\log_b(1/a) &= -\log_b a ,\\
\log_b a &= \frac{1}{\log_a b} ,\\
a^{\log_b c} &= c^{\log_b a} , && (3.16)
\end{aligned}$$

egg_drop

cd /d