1. 食用心得

- 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。
- notes by `Yiqiu Huang`

3. Growth of Funcitons

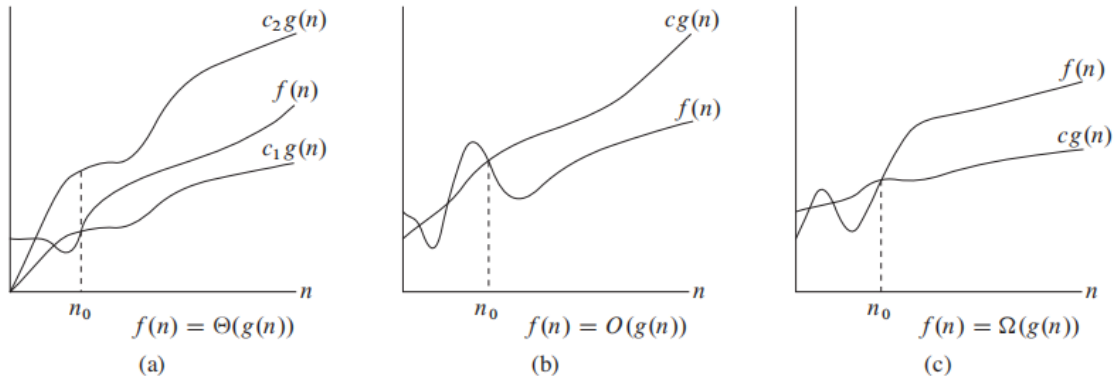
3.1 Asymptotic notation

What is $T(n)$?

- we call it T(n) = number of computational steps required to run the algorithm/program for input of size n
- we are interested in order of growth, not exact values
  - for example T(n) = Θ(n2) means quadratic running time
  - T(n) = O(n logn) means T(n) grows not faster than CONST*n*log(n)

Why **Asymptotic notation**?

> Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean.



3.1.1 Θ Notation

原文定义如下：

> For a given function g(n), we denote by Θ(g(n)) the set of functions:
>
> $\Theta(g(n))$ = {$f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ 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**, $f(n) = g(n)$,($0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$), 看下图(a)。

注意，$\Theta(g(n))$ 本身**描述的是一个集合**，所以你可以这么写：

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

不过我们习惯这么写：

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

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 Θ(g(n)) the set of functions:
>
> $\Theta(g(n))$ = {$f(n)$: 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 Θ(g(n)) the set of functions:

> $\Theta(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n $\geq$ n0 }
>
> 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

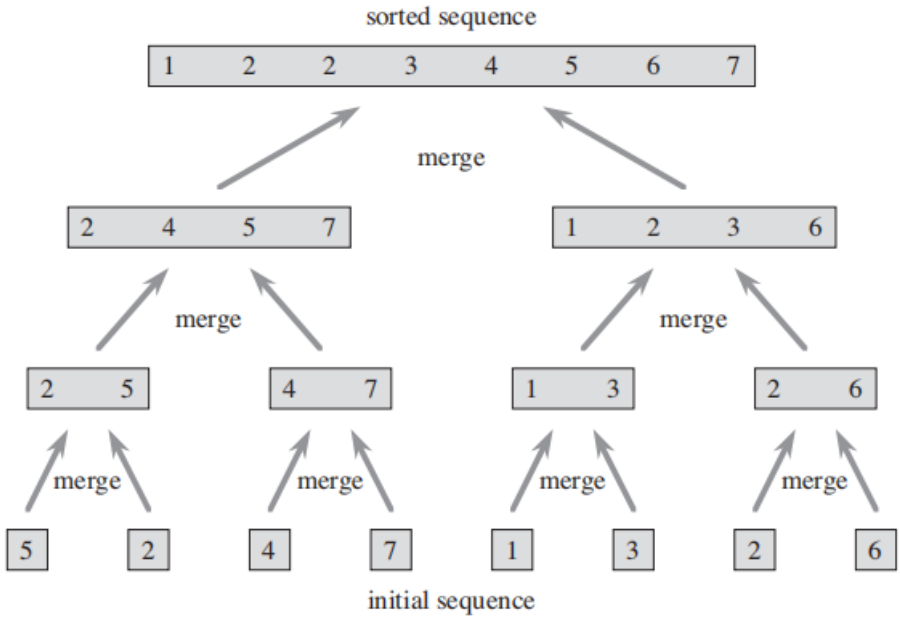
分治法：

> **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 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:

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

可以思考一下这个 **?** 应该是什么

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:

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

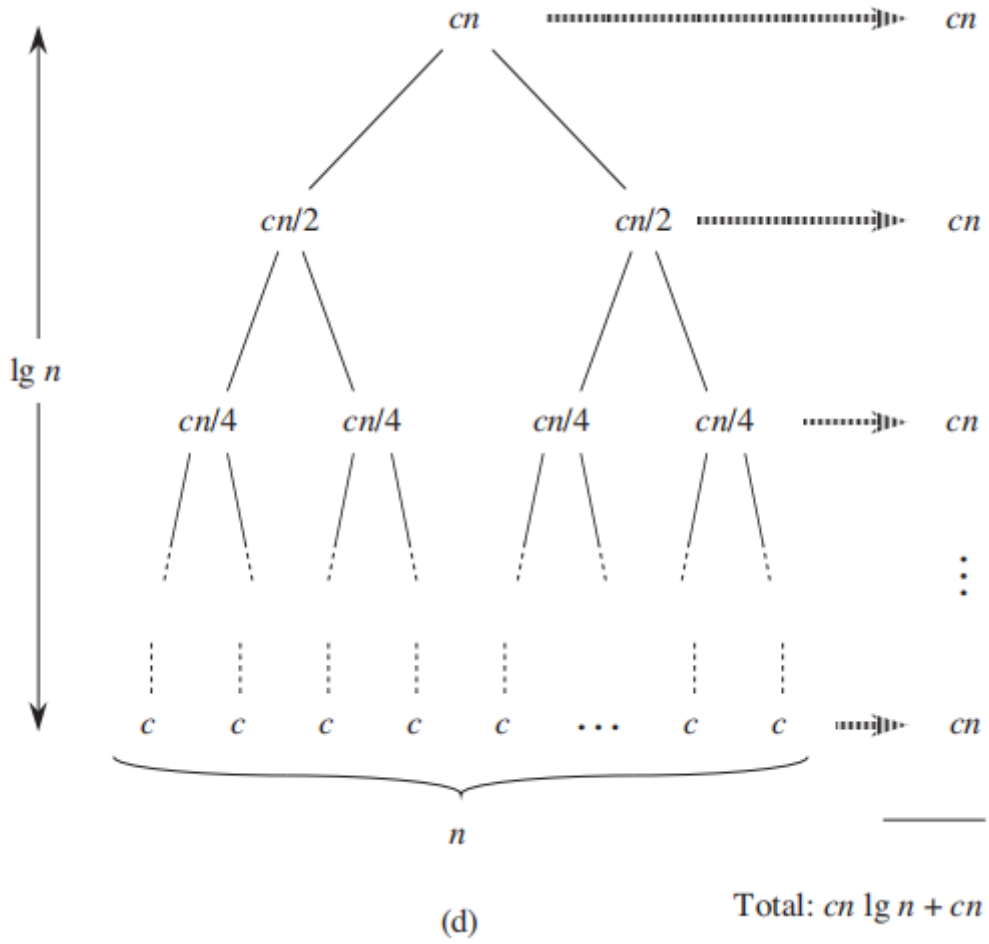
这段比较重要，还是原汁原味吧:

> **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)= $\Theta(\log n)$

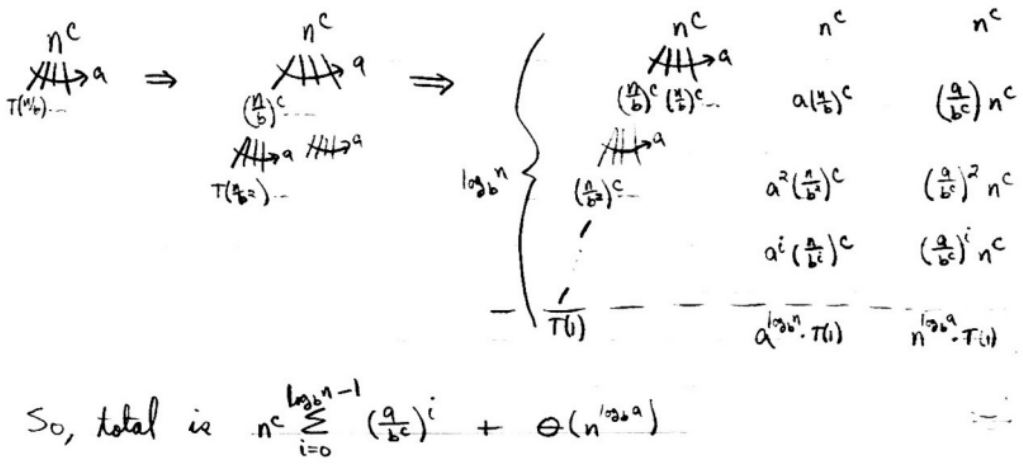
Binary search, Mergesort 的runtime 可以轻松的求出。

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

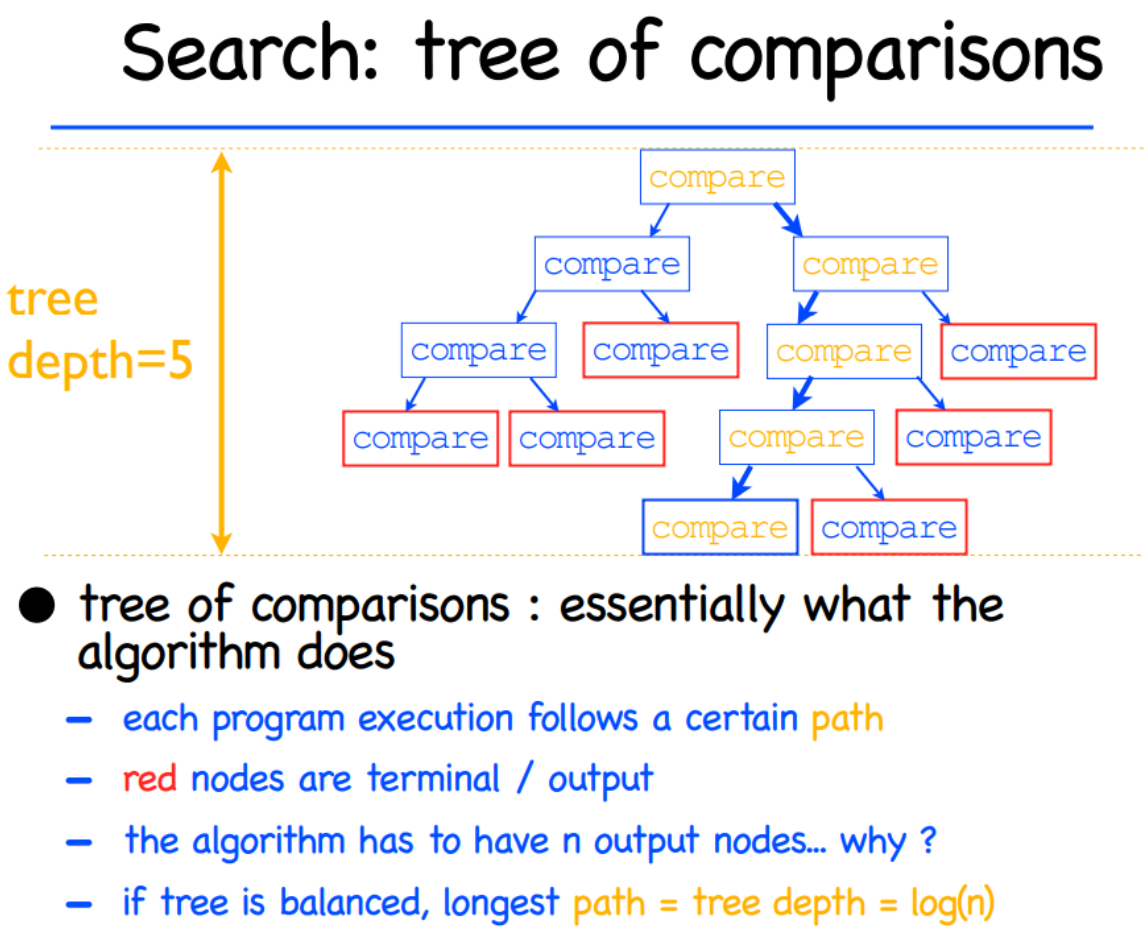
	time	in-place	stable
Bubble	n^2	✔	✔
Insertion	n^2	✔	✔
Selection	n^2	✘	?
QuickSort	n*log(n)	✔	?
MergeSort	n*log(n)	✘	✔

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

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:

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

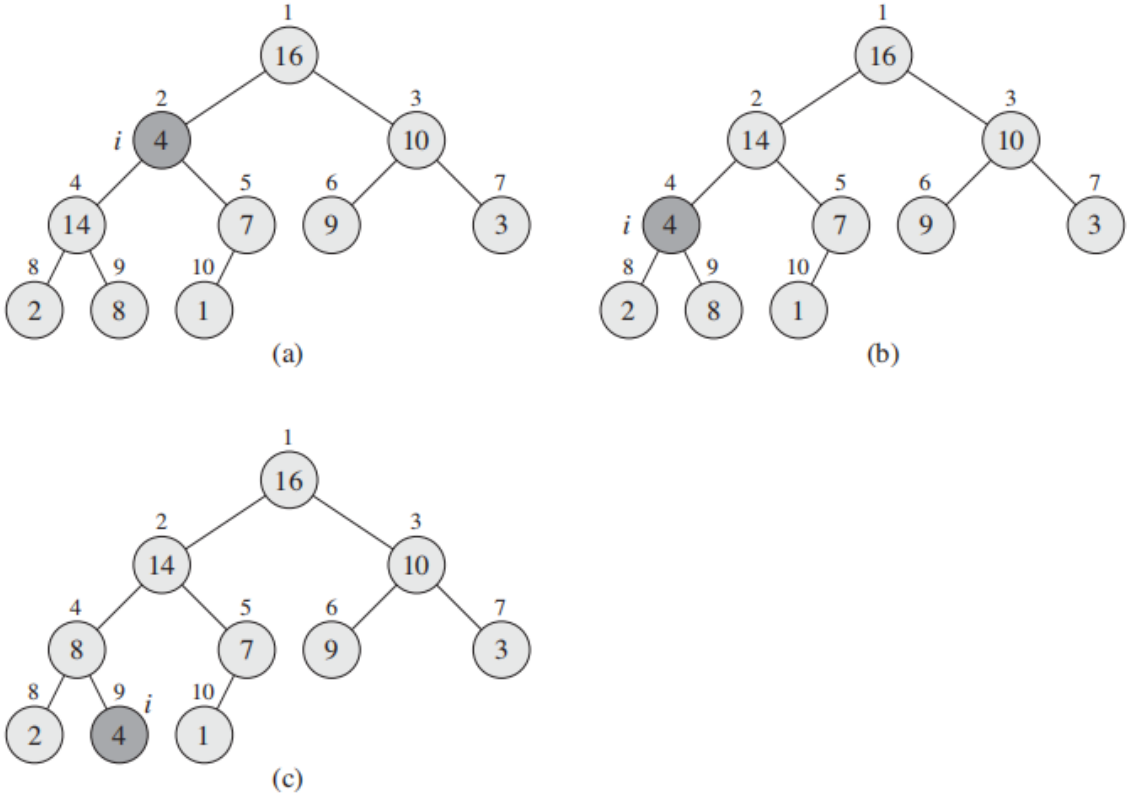
为了满足最大堆性质，你需要调用:

:

```
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。

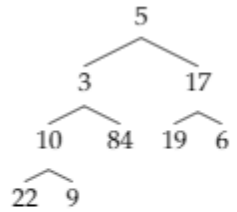
下面来自例题：

6.3-1

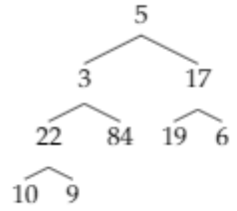
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

答案：

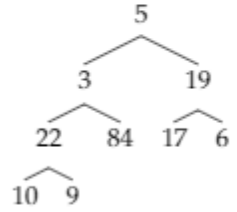
Visualize the array with heap:



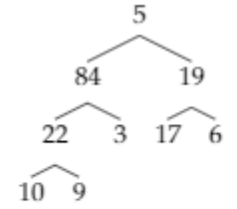
The heapify process start at the last non-leaf node, which is 10.We swap 10 and 22 this time. After that:



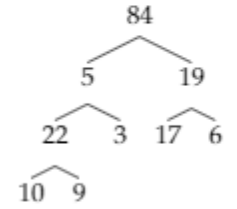
Then, heaify will move to node with value 17, and swap 17 and 19



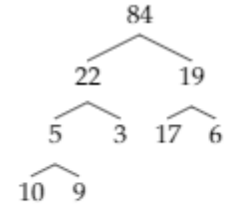
Then, heaify will move to node with value 3, and swap 3 and 84



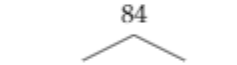
Then, heaify will move to root node 5, and swap 5 and 84

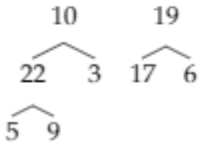


Then, heaify will move to root node 5, and swap 5 and 22



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

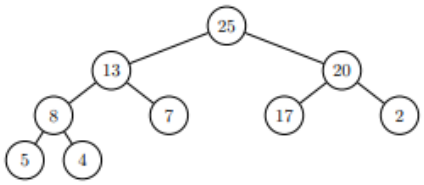




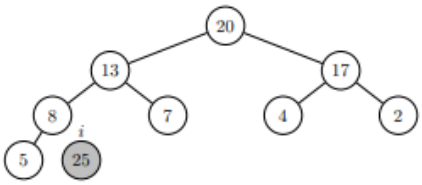
6.3.3

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

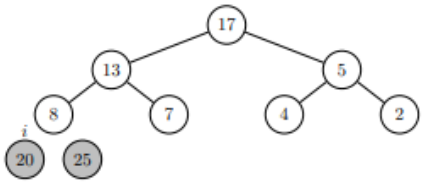
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i = A.length$ **downto** 2
3 exchange $A[1]$ with $A[i]$
4 $A.heap\text{-}size = A.heap\text{-}size - 1$
5 MAX-HEAPIFY(A, 1)



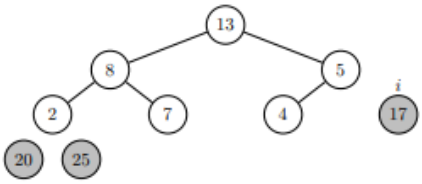
Step 0



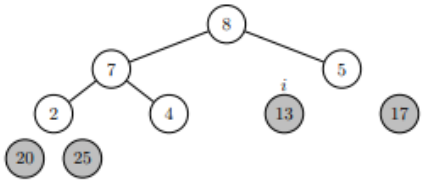
Step 1



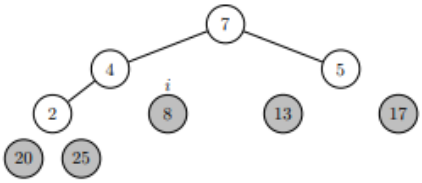
Step 2



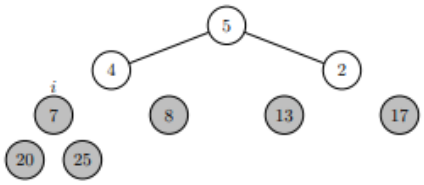
Step 3



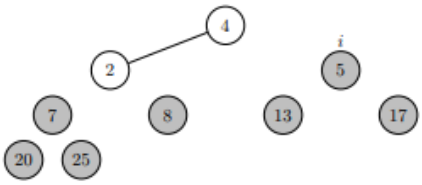
Step 4



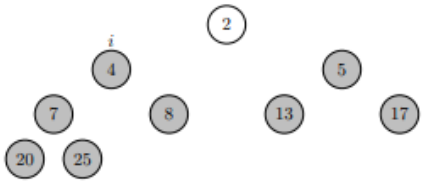
Step 5



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：

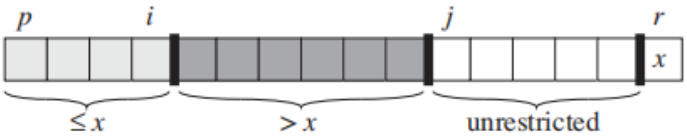


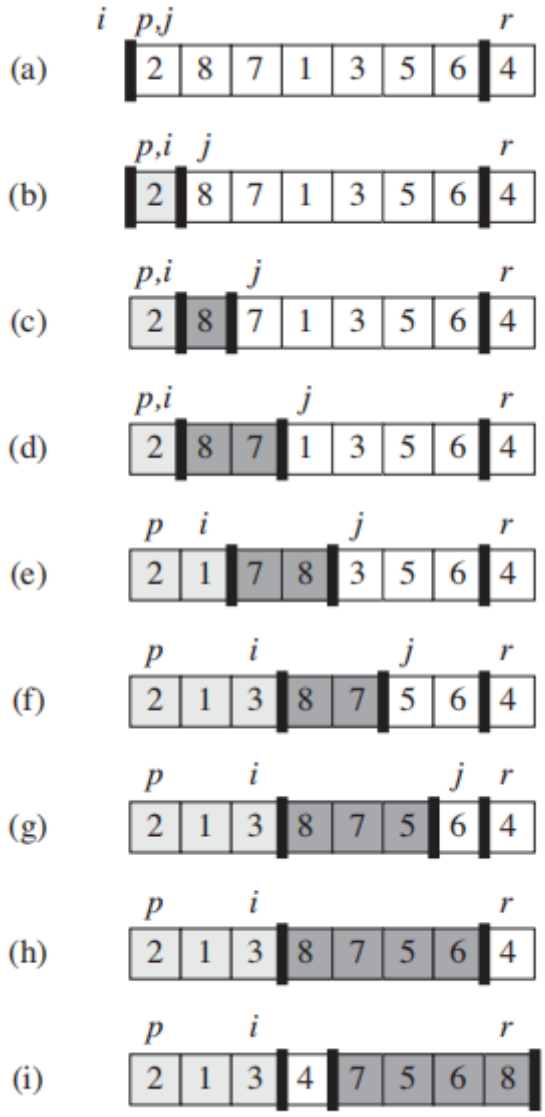
Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.

[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

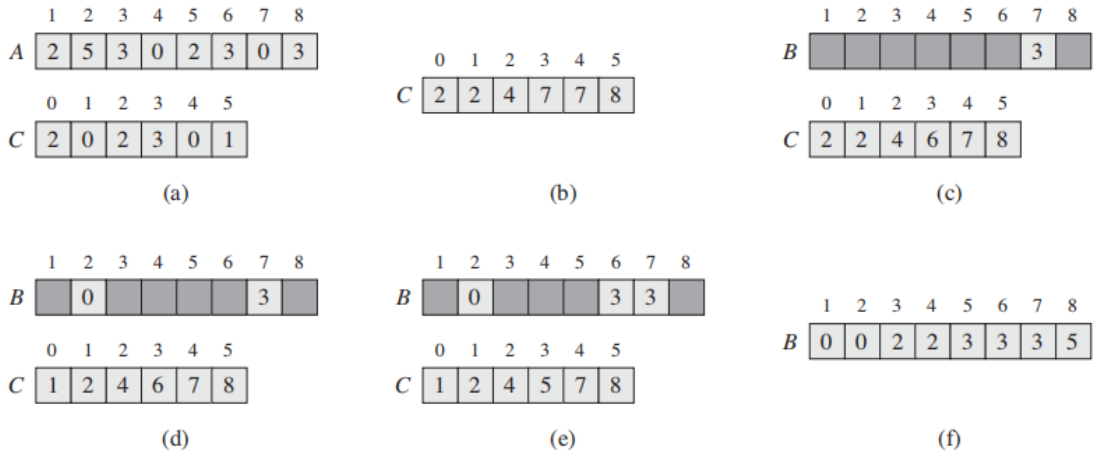
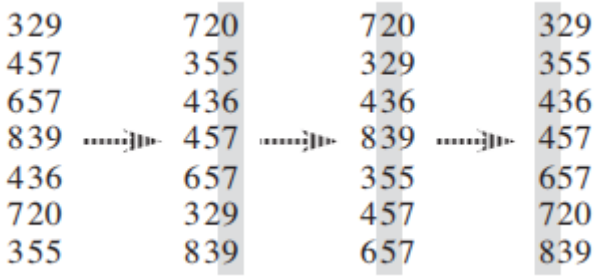


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after 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.

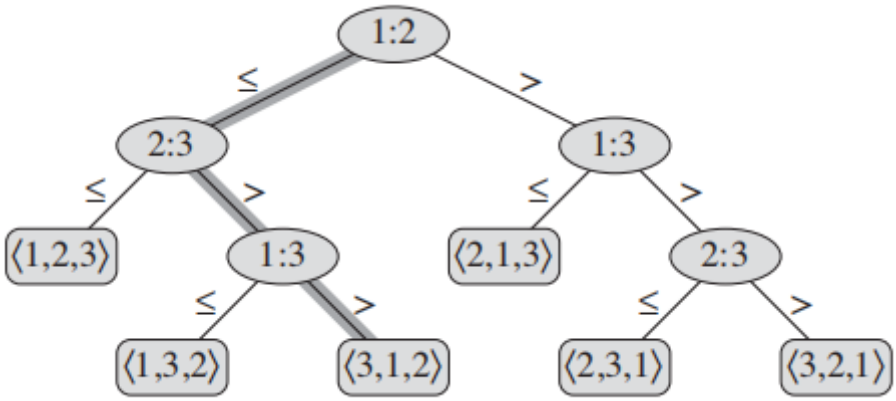
Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the ith digit, the array will be sorted on the last i digits. It is obvious that elements with different digit in the ith position are ordered accordingly; in the case of the same ith digit, we still get a correct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

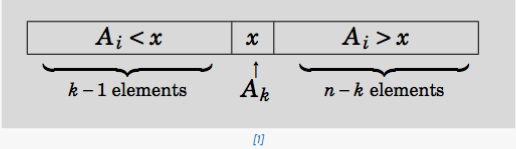
> Use a **divide and conquer** strategy to efficiently compute **the i^th smallest number** in an unsorted list of size n.

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).

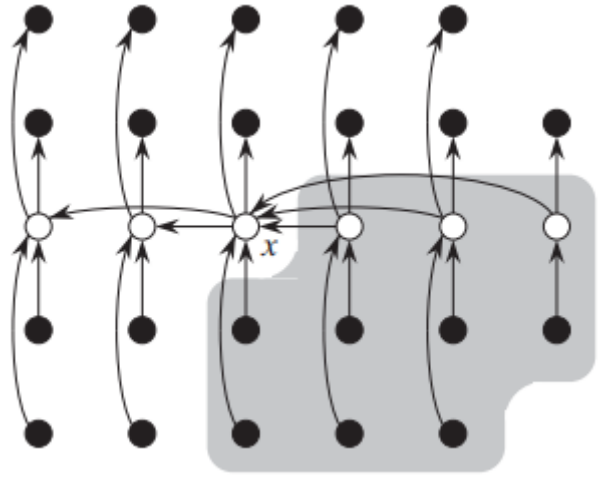
1. Divide the list into sublists each of length five (if there are fewer than five elements available for the last list, that is fine).
2. Sort each sublist and determine the median. Sorting very small lists takes linear time since these sublists have five elements, and this takes $O(n)$ time. In the 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)

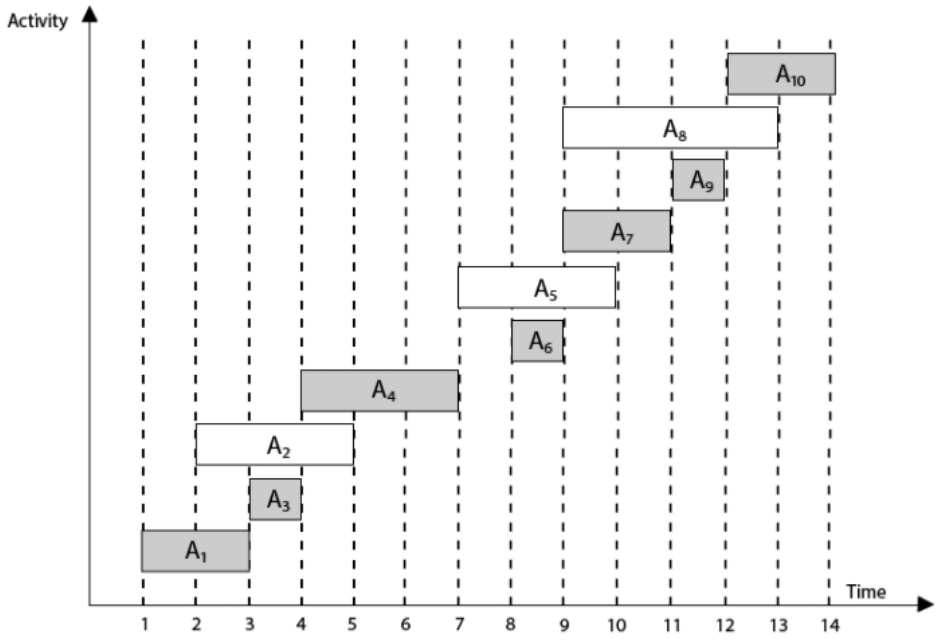
16 Greedy Algorithms

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.

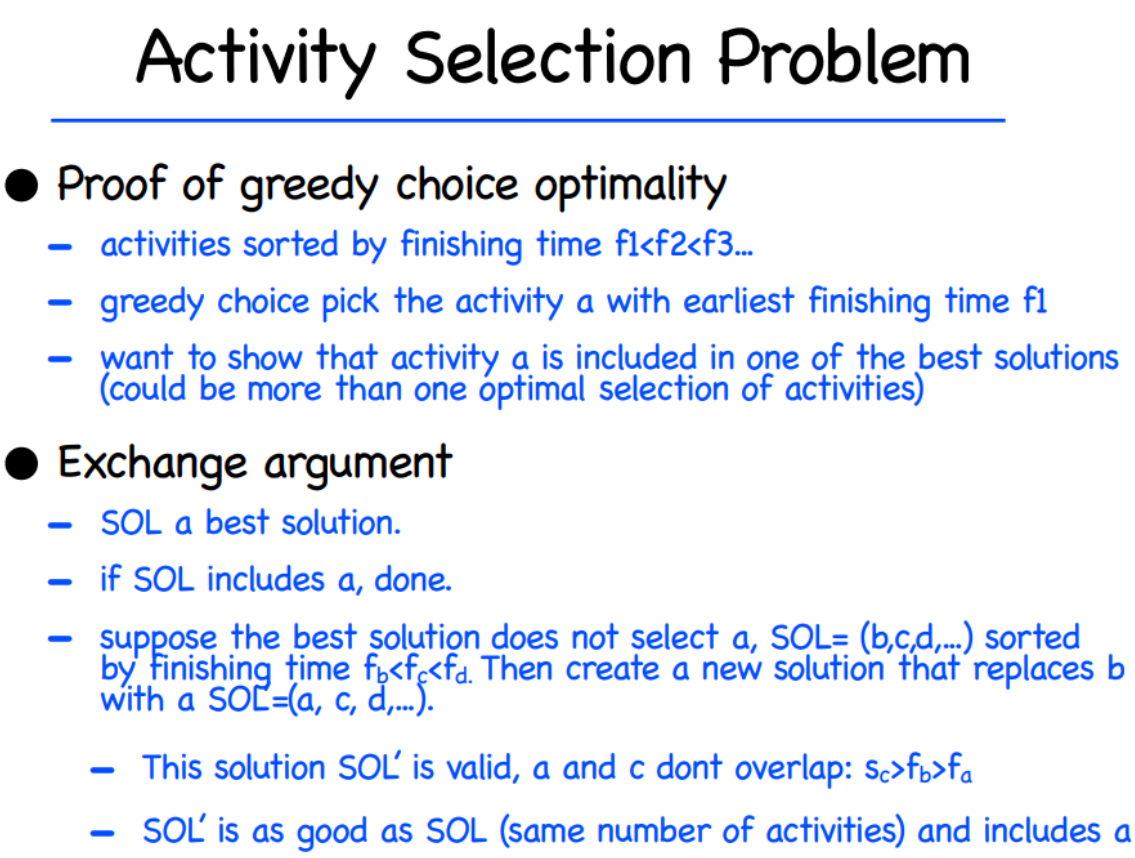
总结

- 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

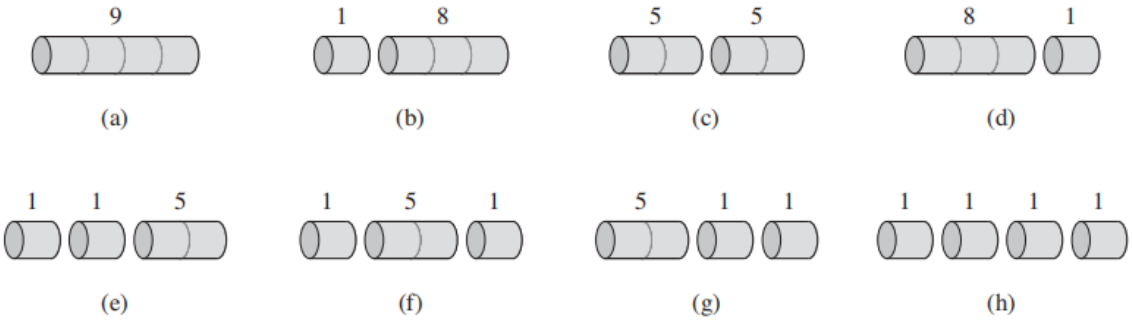
切绳子问题定义如下：

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
```

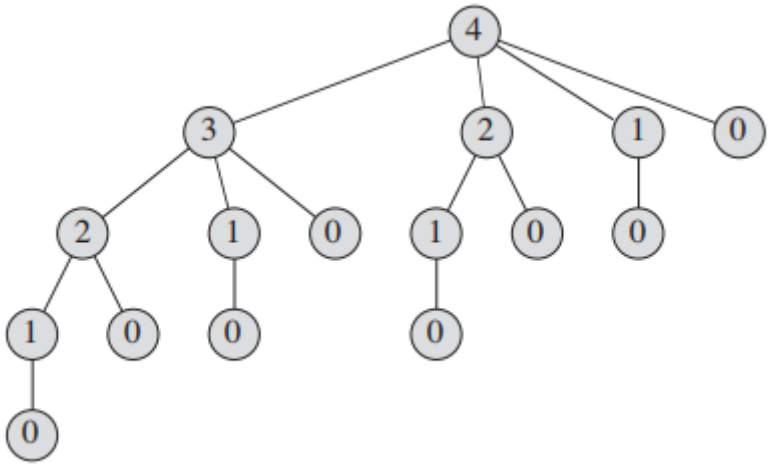
```
@param:
**p** : array p[1..n] of price

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:

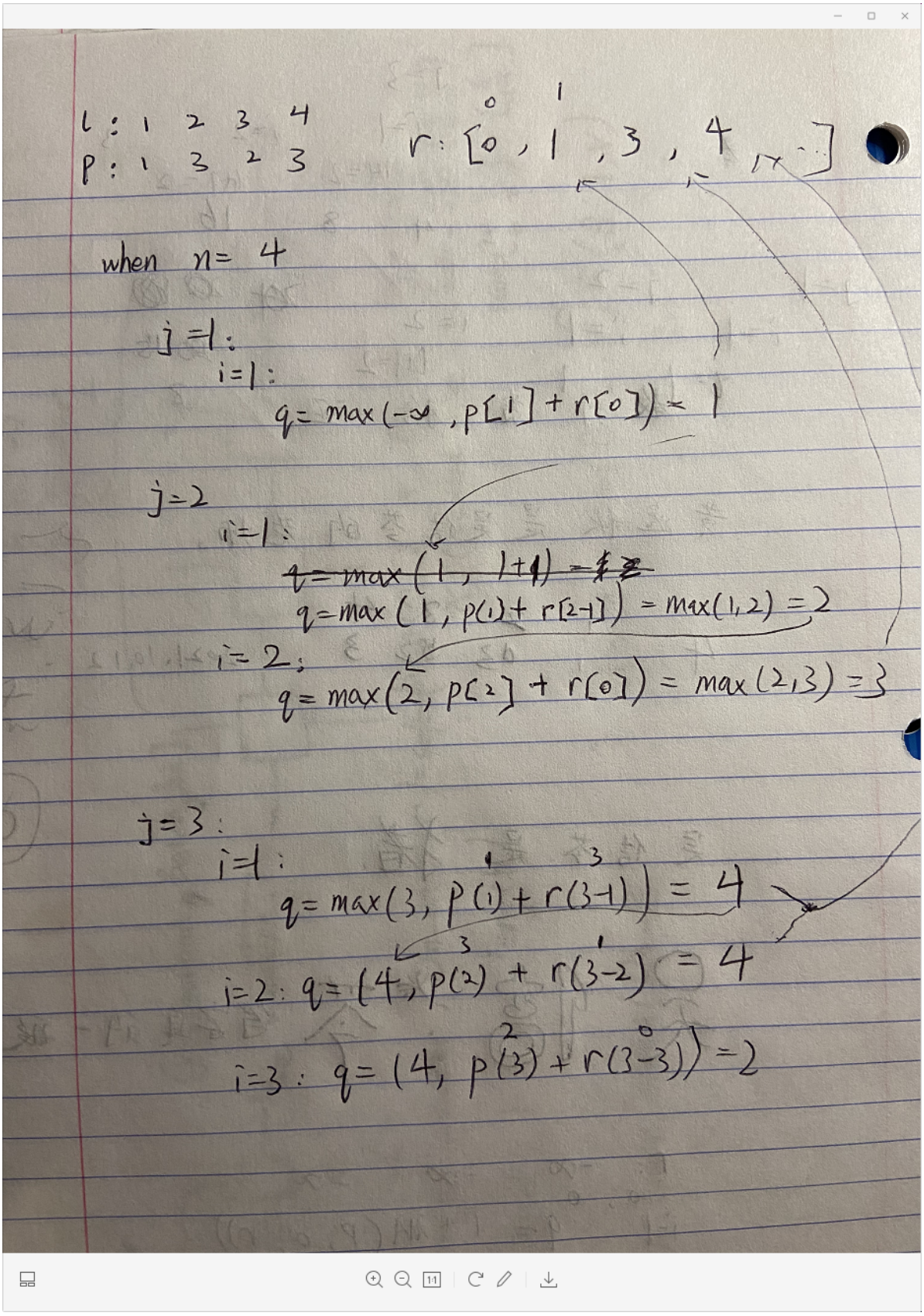
```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

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 $p_{i-1} \cdot p_i$.

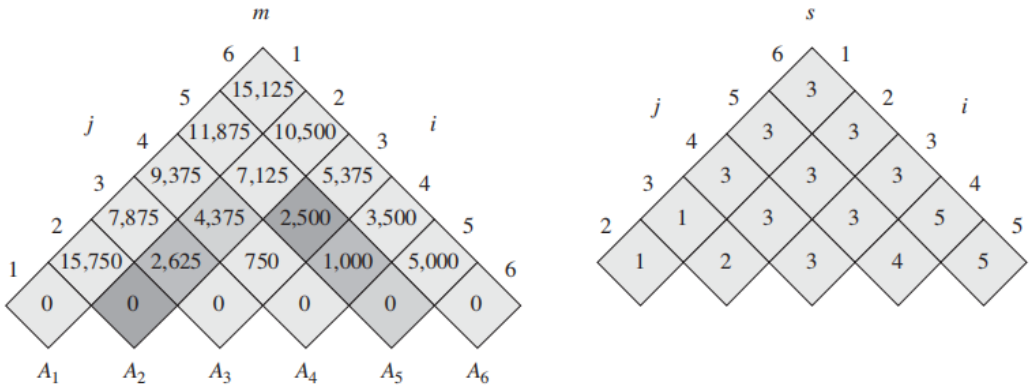
bottom-up approach：

- 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.

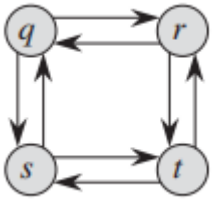
有些问题没有Optimal substructure性质，比如给定一个没有权值的**有向图**(directed graph)：

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

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**.）

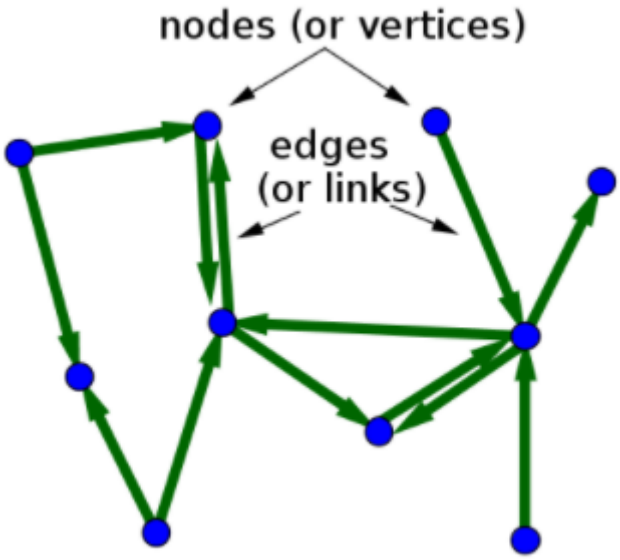


2. 无权值的最长路径问题(**Unweighted longest simple path:**): 找到某顶点u到某顶点v的 最多边的数量。

   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

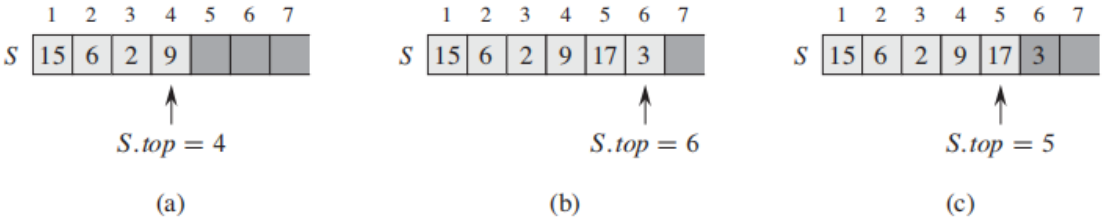


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is 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

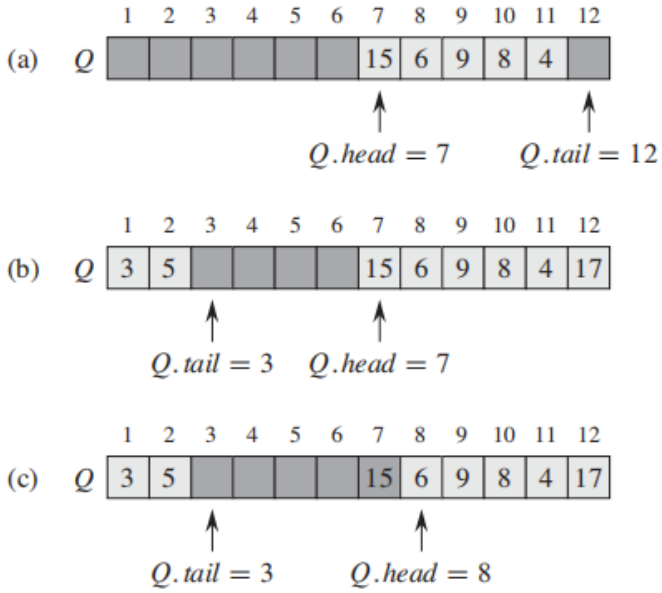


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The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

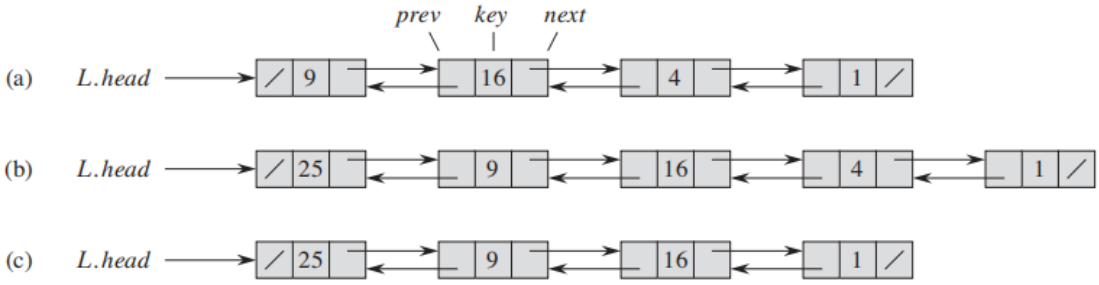


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, 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:

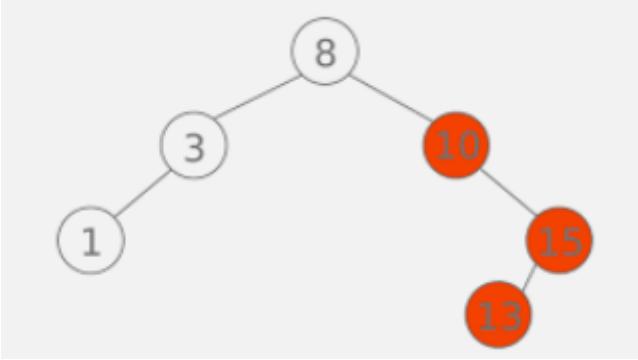
- 如果x.left $\neq$ NIL, 那么Predecessor就是x.left中的最大值;
- 否则，Predecessor不存在；

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.

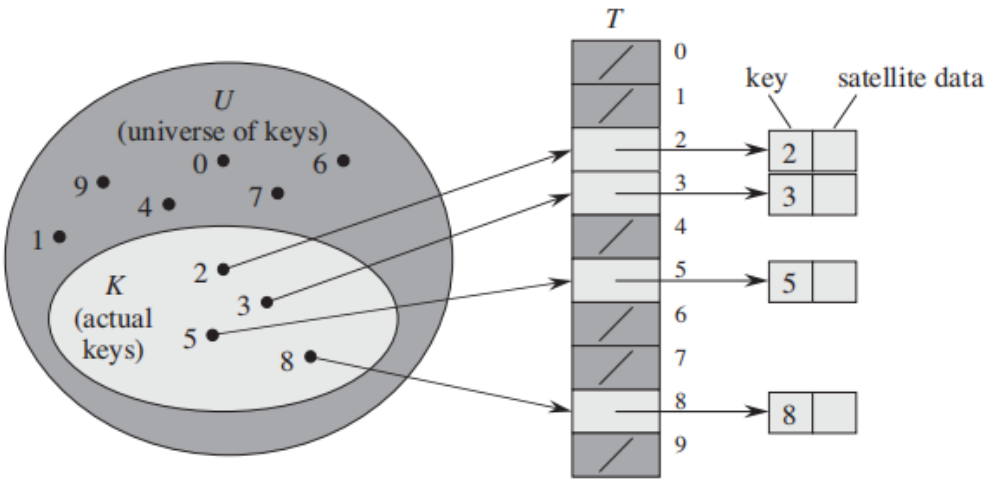
直接寻址法 适用于key比较少的时候。

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

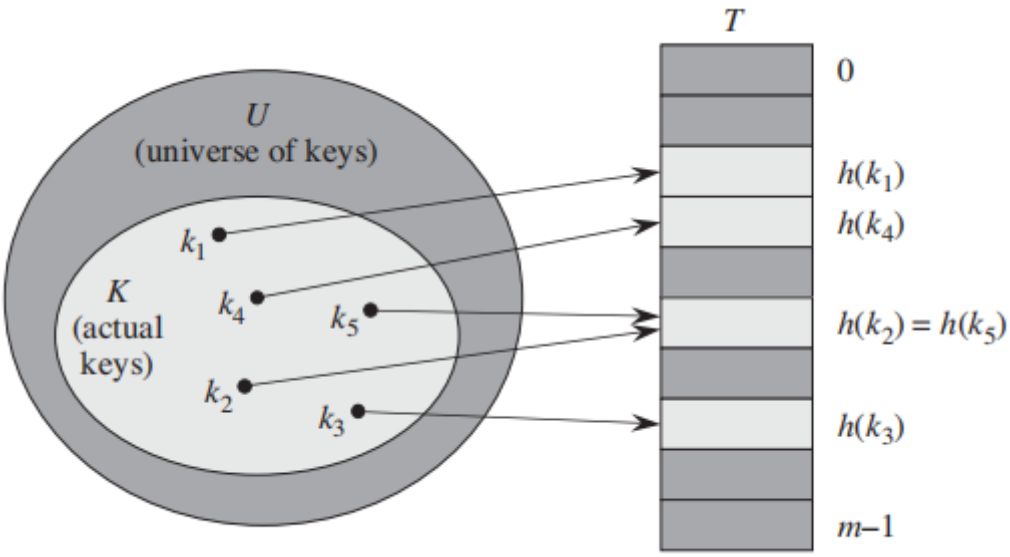
> With direct addressing, an element with key k is stored in slot k.
>
> With hashing, this element is stored in slot **h(k)**; that is, we use a **hash function** h to compute the
>
> slot from the key k.
>
> Here, h maps the universe U of keys into the slots of a **hash**
>
> **table** T[0 .. m-1]:

> 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；

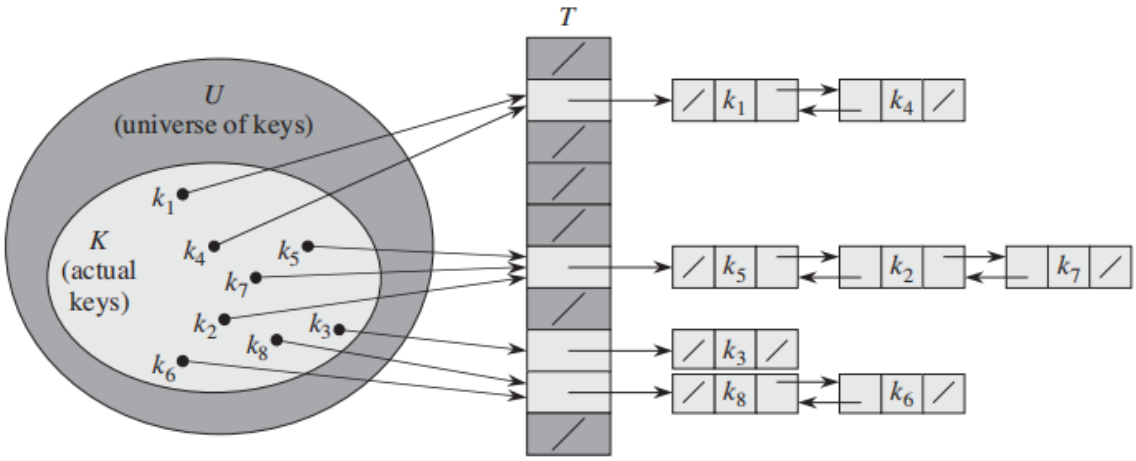
发生冲突并不可怕，有一些有效的方法能化解冲突带来的结果；（冲突本身是不可能解决的）

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。

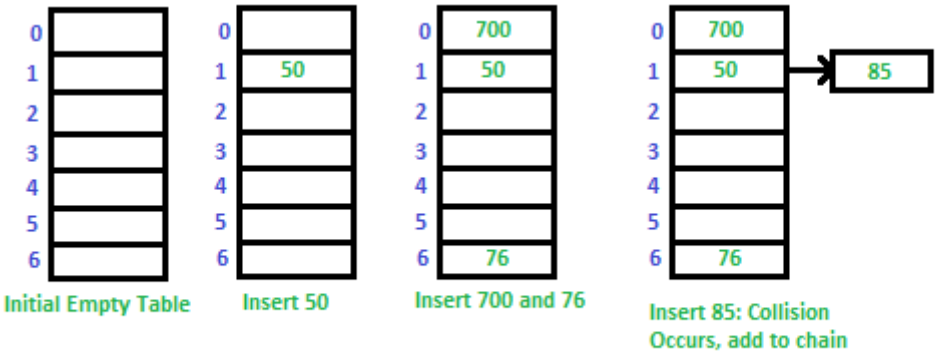
下面介绍原书中提供的最简单的冲突解决方法；

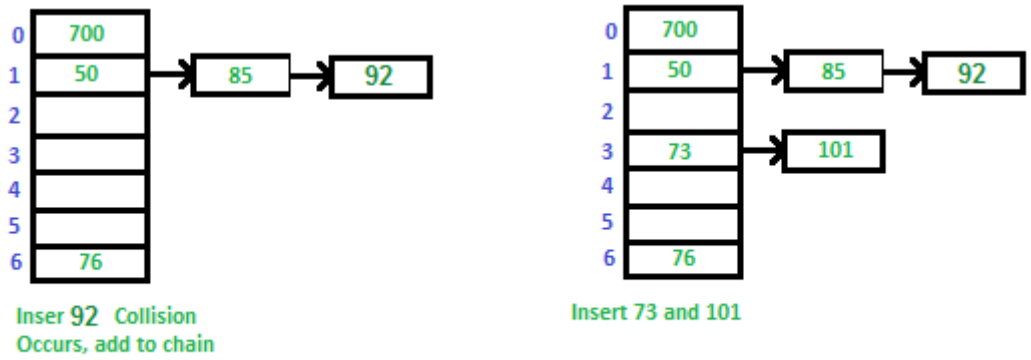
1. Chaining:



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

python 位运算：

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：

下表中变量 a 为 60，b 为 13，二进制格式如下：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

实施hash_djb2:

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

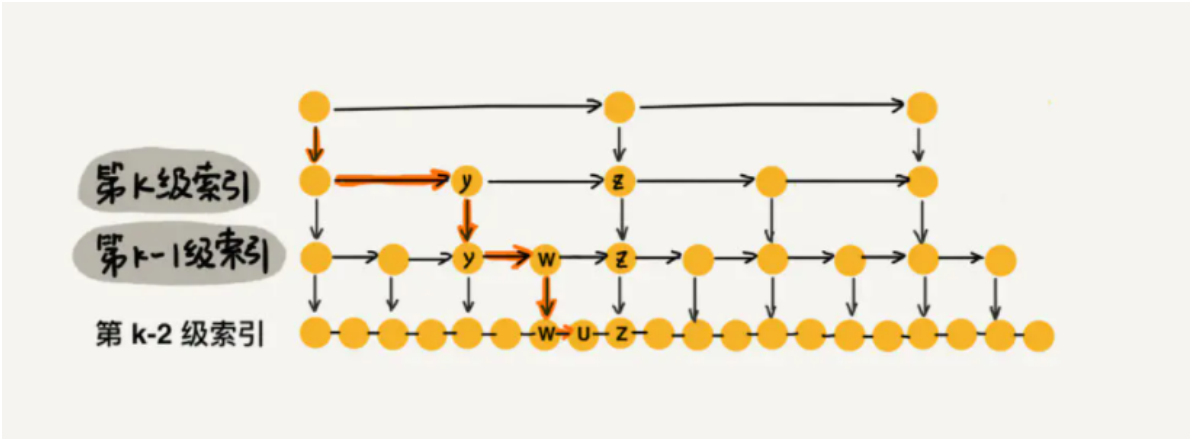
这个函数的magic在于他的两个魔法数字：33和5381，这里用的是5381；

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
```

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。

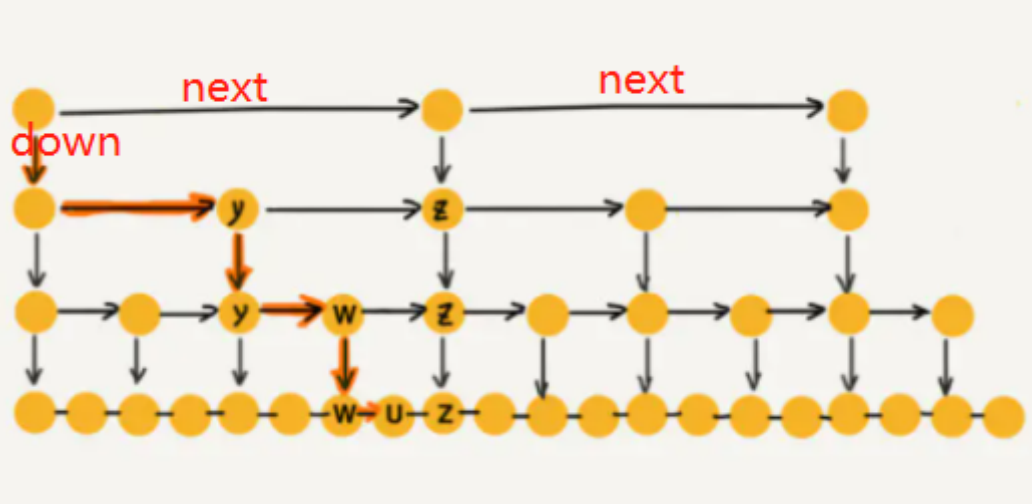
https://leetcode.com/problems/design-skiplist/discuss/?currentPage=1&orderBy=most_votes&query=

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：

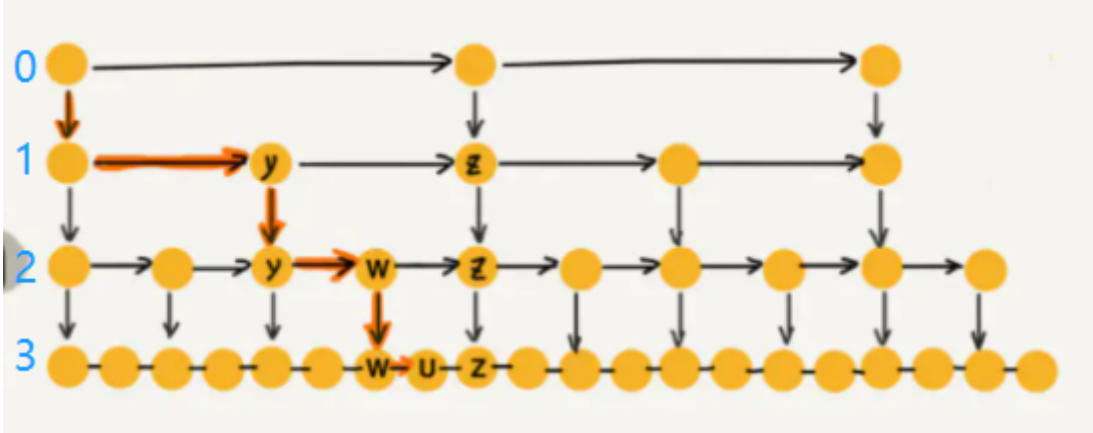
```
class Skiplist:
    def __init__(self):
        self.levels = []
        self.max_level = 4
        prev = None
        for i in range(self.max_level):
            node = Node(-math.inf)
            self.levels.append(node)
            if prev:
                prev.down = node
            prev =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

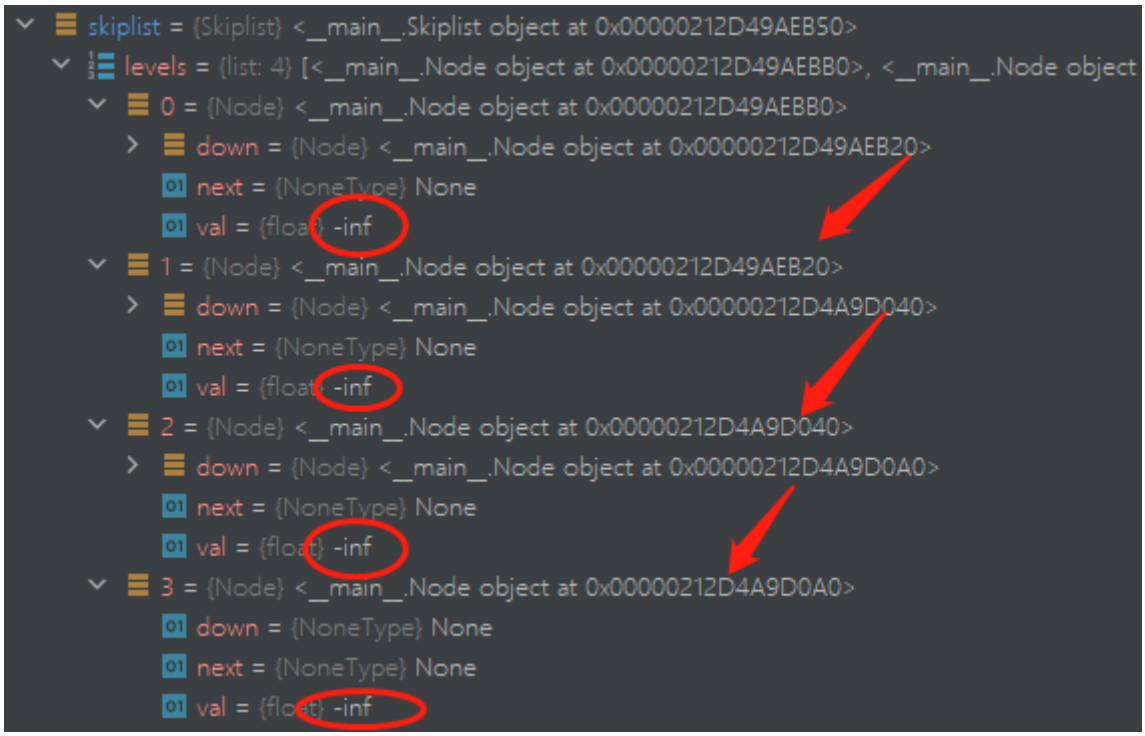
> 后面统一一下口径，**最高级**的链表指 **index = 0**, 元素最少的那一条链表；



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`.

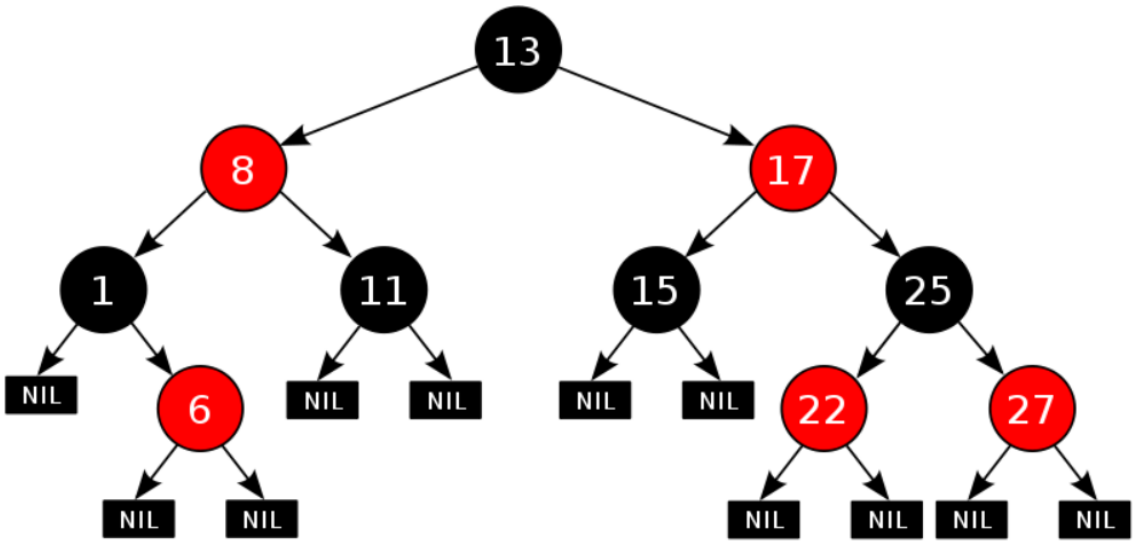
A red-black tree is a binary tree that satisfies the following ***red-black properties***:

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。
```

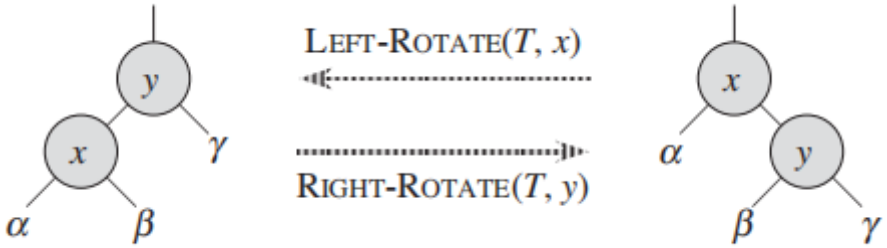


figure 来自原文**13.2** *p313*

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**

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）

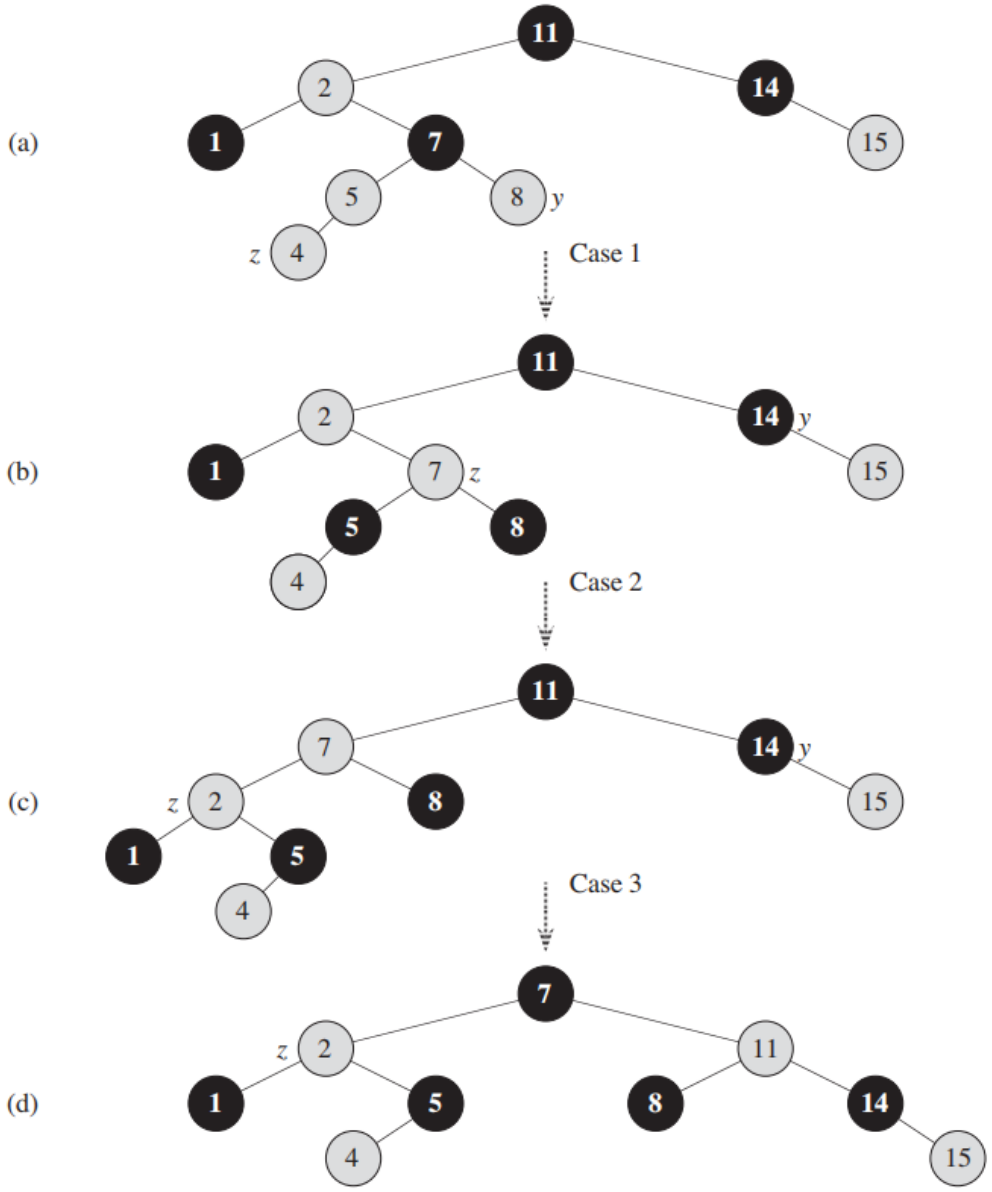


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. 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)

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