《算法导论》参考答案

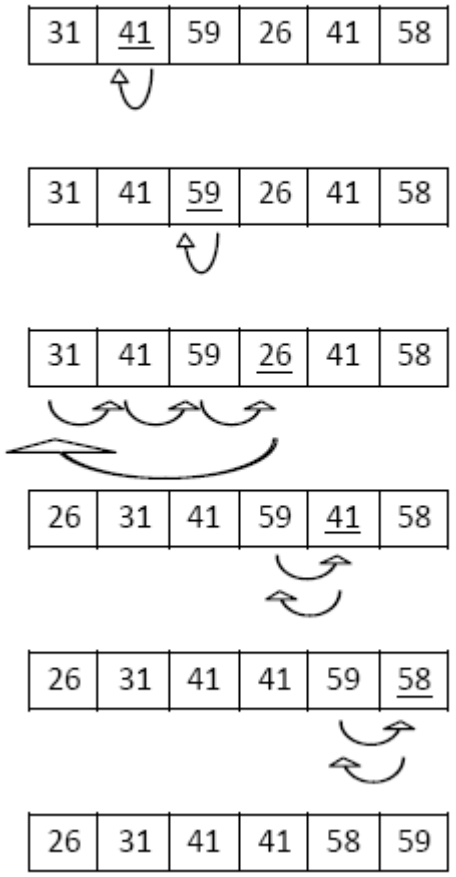
第 2 章

2.1-1

代码:

```
INSERTION-SORT(A)
1    for j ← 2 to length[A]
2         do key ← A[j]
3              ΔInsert A[j] into the sorted sequence A[1..j-1]
4              i ← j – 1
5              While I > 0 and A[i] > key
6                   do A[i+1] ← A[i]
7                        i ← j – 1
8              A[i+1] ← key
```

图解:



2.1-2

In line 5 of INSERTION-SORT alter A[i] > key to A[i] < key in order to sort the elements in nonincreasing order.

```
INSERTION-SORT(A)
for j←2 to length[A]
  do key←A[j]
     //Insert A[j] into the sorted sequence A[1..j-1]
     i←j-1
     while i>0 and A[i]<key
        do A[i+1] ← A[i]
           i←i-1
     A[i+1] ← key
```

2.1-3

Algorithm 1 LINEAR-SEARCH(A, v)

Input: $A = \langle a_1, a_2, \ldots a_n \rangle$ and a value v.
Output: An index i such that $v = A[i]$ or **nil** if $v \notin A$
for $i \leftarrow 1$ **to** n **do**
  if $A[i] = v$ **then**
    return i
  end if
end for
return nil

As a loop invariant we say that none of the elements at index $A[1, \ldots, i-1]$ are equal to v. Clearly, all properties are fullfilled by this loop invariant.

2.1-4

A、B各存放了一个二进制n位整数的各位数值，现在通过二进制的加法对这两个数进行计算，结果以二进制形式把各位上的数值存放在数组C中

key存储临时计算结果，flag为进位标志符，按位相加，8、9行不能少

```
BINARY-ADD(A,B,C)
1 flag← 0
2 for j←1 to n
3 do
4  key←A[j]+B[j]+flag
5  C[j] ← key mod 2
6     if key >1
7        flag←1
8 if flag=1
9   C[n+1] ← 1
```

2.2-1

$$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3).$$

2.2-2

Assume that FIND-MIN(A, r, s) returns the index of the smallest element in A between indices r and s. Clearly, this can be implemented in $O(s - r)$ time if $r \geqslant s$.

Algorithm 2 SELECTION-SORT(A)

Input: $A = \langle a_1, a_2, \ldots a_n \rangle$
Output: sorted A.
for $i \leftarrow 1$ **to** $n - 1$ **do**
  $j \leftarrow$ FIND-MIN(A, i, n)
  $A[j] \leftrightarrow A[i]$
end for

As a loop invariant we choose that $A[1, \ldots, i - 1]$ are sorted and all other elements are greater than these. We only need to iterate to $n - 1$ since according to the invariant the nth element will then the largest.

The n calls of FIND-MIN gives the following bound on the time complexity:

$$\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$$

This holds for both the best- and worst-case running time.

2.2-3

Given that each element is equally likely to be the one searched for and the element searched for is present in the array, a linear search will on the average have to search through half the elements. This is because half the time the wanted element will be in the first half and half the time it will be in the second half. Both the worst-case and average-case of LINEAR-SEARCH is $\Theta(n)$.

2.2-4

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

2.3-1

如下所示：

3, 9, 26, 38, 41, 49, 52, 57

3, 26, 41, 52　　9, 38, 49, 57

3, 41　　26, 52　　38, 57　　9, 49

3　41　　52　26　　38　57　9　49

2.3-2

```
void Merge(int *A,int p,int q,int r)
{
 //构建左半部分和右半部分的辅助数组
 int n1=q-p+1;
 int n2=r-q;
 int *L=new int[n1];
 int *R=new int[n2];
 for (int i=0;i<n1;i++)
 {
  L[i]=A[p+i-1];
```

```
 }
 for(int j=0;j<n2;j++)
 {
  R[j]=A[q+j];
 }
 int i=0;
 int j=0;
 int k=p-1;
 while((i<=n1-1)&&(j<=n2-1))
 {
  if(L[i]<=R[j])
  {
   A[k]=L[i];
   i++;
  }
  else
  {
   A[k]=R[j];
   j++;
  }
  k++;
 }
 while(i<=n1-1)
 {
  A[k]=L[i];
  i++;
  k++;
 }
 while(j<=n2-1)
 {
  A[k]=R[j];
  j++;
  k++;
 }
delete[]L;
delete []R;
}
```

2.3-3

由题意得：

根据数学归纳法：

给出基本条件（base case）：

当 n=2 时，T（2）= 2 lg2 = 2。符合条件。

给出假设：

当 $n=2^t$ 时，$T(2^t) = 2^t \lg 2^t$ 成立。

那么，

当 $n=2^{t+1}$ 时，$T(2^{t+1}) = 2\,T(2^{t+1}/2) + 2^{t+1}$

$= 2T(2^t) + 2^{t+1}$

$= 2(2^t \lg 2t) + 2^{t+1}$

$= 2^{t+1} \lg 2^{t+1}$

所以,得证！

2.3-4

Since it takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$, we get the recurrence

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

The solution to this recurrence is $T(n) = \Theta(n^2)$.

2.3-5

给出一个递归算法. 显然最坏情况的运行时间为(lg n).

```
BINARY-SEARCH(A, v, p, r)
if p ≥ r and v ≠ A[p] then
    return nil
end if[n/2]
j←A[[ (r - p)/2]]
if v = A[j] then
    return j
else
    if v < A[j] then
        return BINARY-SEARCH(A; v; p; j)
    else
        return BINARY-SEARCH(A; v; j; r)
    endif
end if
```

2.3-6

将插入排序的顺序查找改为二分查找的算法：

```
INSERTION-SORT(A)
9    for j ← 2 to length[a]
10       do key ← a[j]
11          △insert a[j] into the sorted sequence a[1..j-1]
12          high ← j - 1
13          low ← 1
14          while low < high
15              mid = ( low + high ) / 2
16              If key == A[mid] then
17                  break
18              if key < A[mid] then
19                  high ← mid - 1
20              if key > A[mid] then
21                  Low ← mid + 1
22          for i ← mid to j – 1
23              A[i+1] ← a[i]
24          A[mid] ← key
```

在最坏情况下，二分查找的时间复杂度是 nlgn，但插入时数组移动的时间复杂度仍是 n^2。故总体运行时间不能改善为 $\Theta(n \lg n)$。（但若排序中采用链表的数据结构，则可改善。）

2.3-7

Give a $\Theta(n \lg n)$ time algorithm for determining if there exist two elements in an set S whose sum is exactly some value x.

Algorithm 4 CHECKSUMS(A, x)

Input: An array A and a value x.
Output: A boolean value indicating if there is two elements in A whose sum is x.

```
A ← SORT(A)
n ← length[A]
for i ← to n do
  if A[i] ⩾ 0 and BINARY-SEARCH(A, A[i] − x, 1, n) then
    return true
  end if
end for
return false
```

Clearly, this algorithm does the job. (It is assumed that **nil** cannot be true in the **if**-statement.)

第 3 章

3.1-1

因为 f(n)和 g(n)都是渐近非负的函数，所以根据定义有：存在 N_f, N_g,使得：当 $n>N_f$ 时,f(n)≥0,同时，当 $n> N_g$ 时，g(n) ≥0.所以，我们取 $N_0=\max\{N_f, N_g\}$,此时，当 $n>N_0$ 时,同时有 f(n)≥0，g(n) ≥0.下面我们取 $C_1=1/2$，$C_2=1$，根据 f(n),g(n)的非负性保证，当 $n>N_0$ 时，有:

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

.所以，得证！

3.1-2

To show that $(n+a)^b = \Theta(n^b)$, we want to find constants $c_1, c_2, n_0 > 0$ such that $0 \le c_1 n^b \le (n+a)^b \le c_2 n^b$ for all $n \ge n_0$.

Note that

$$\begin{aligned} n+a &\le n+|a| \\ &\le 2n \qquad \text{when } |a| \le n, \end{aligned}$$

and

$$\begin{aligned} n+a &\ge n-|a| \\ &\ge \frac{1}{2}n \qquad \text{when } |a| \le \tfrac{1}{2}n. \end{aligned}$$

Thus, when $n \ge 2|a|$,

$$0 \le \frac{1}{2}n \le n+a \le 2n.$$

Since $b > 0$, the inequality still holds when all parts are raised to the power b:

$$0 \le \left(\frac{1}{2}n\right)^b \le (n+a)^b \le (2n)^b,$$

$$0 \le \left(\frac{1}{2}\right)^b n^b \le (n+a)^b \le 2^b n^b.$$

Thus, $c_1 = (1/2)^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ satisfy the definition.

3.1-3

Let the running time be $T(n)$. $T(n) \ge O(n^2)$ means that $T(n) \ge f(n)$ for some function $f(n)$ in the set $O(n^2)$. This statement holds for any running time $T(n)$, since the function $g(n) = 0$ for all n is in $O(n^2)$, and running times are always nonnegative. Thus, the statement tells us nothing about the running time.

3.1-4

$2^{n+1} = O(2^n)$ 因为 $2^{n+1} = 2\times 2^n \le 2\times 2^n$, 所以成立

$2^{2n} = O(2^n)$ 不成立。用反证法：如果有 $2^{2n} = O(2^n)$成立，根据定义需要有：$2^{2n} \le c2^n$，由此需要：n≤lgc.矛盾！对于任意大的 n 是不可能成立的，因为 c 是一个常数。

3.1-5

首先证明充分性：即由 f(n)= Θ(g(n))推出 f(n)= O(g(n))且 f(n)= Ω (g(n))

根据 Θ 定义：有存在 N_0,C_0, , 使得：当 n> N_0 时有：$C_0 g(n) \le f(n) \le C_1 g(n)$,在根据 O，和 Ω 定义有 f(n)= O(g(n))且 f(n)= Ω (g(n)).

在证明必要性：即由 f(n)= O(g(n))且 f(n)= Ω (g(n))推出 f(n)= Θ(g(n))

根据 O，和 Ω 定义有：存在 N_1, C_1 使得：当 n> N_1 时有$f(n) \le C_1 g(n)$，同样地，存在 N_2, C_0 使得：当 n> N_2 时有$C_0 g(n) \le f(n)$，此时，我取 N_0=max(N_1, N_2),则有如下事实成立：当 n> N_0 时有：$C_0 g(n) \le f(n) \le C_1 g(n)$，即 f(n)= Θ(g(n)).所以，得证！

3.1-6

由定义可易证，Θ记号渐近地给出一个函数的上界和下界，O 记号给出一个函数的渐近上界，而Ω给出渐近下界。

3.1-7

Solution: Let f∈o (g(n))∩ ω(g(n)). That means f= o (g(n))= ω(g(n)). By the definition of o, we have

$$\lim_{n\to\infty}(f(n)/g(n)) = 0$$

But the definition of ω, we have

$$\lim_{n\to\infty}(f(n)/g(n)) = \infty$$

Because of $0 \neq \infty$, we have a obvious contradiction.

3.1-8

$$\Omega(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le cg(n,m) \le f(n,m) \text{ for all } n \ge n_0 \text{ and } m \ge m_0\}.$$

$$\Theta(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that } 0 \le c_1 g(n,m) \le f(n,m) \le c_2 g(n,m) \text{ for all } n \ge n_0 \text{ and } m \ge m_0\}.$$

3.2-1

依题意可知，若$n_1 \le n_2$，

则有$f(n_1) \le f(n_2)$, $g(n_1) \le g(n_2)$ f(n1)≤f(n2), g(n1)≤g(n2)。

$$[f(n_1)+g(n_1)]-[f(n_2)+g(n_2)]$$
$$=f(n_1)-f(n_2)+g(n_1)-g(n_2) \le 0$$

故 f(n)+g(n)是单调递增的，后两个类似可得证。

3.2-2

$\log_b a \bullet \log_b n = \log_b n \bullet \log_b a$ commutativity of $\bullet$

$\log_b a^{\log_b n} = \log_b n^{\log_b a}$ $(\log x)y = \log x^y$ (both sides)

$a^{\log_b n} = n^{\log_b a}$ $x^{\log_x y} = y$ (both sides)

3.2-3

$\forall c > 0, \exists n_0 > 0$,使得对所有的$n \geq n_0$,有$0 \leq c2^n < n!$,故$n! = \omega(2^n)$

$\forall c > 0, \exists n_0 > 0$,使得对所有的$n \geq n_0$,有$0 \leq n! \leq cn^n$，故$n! = o(n^n)$

3.2-4

设$\lceil \lg n \rceil = m$,

则有$m! = \sqrt{2\pi m}\left(\frac{m}{e}\right)^m e^{2m} > \left(\frac{m}{e}\right)^m e^{2m}$

$= (me)^m = e^{m(\ln m+1)} > n^{\ln m+1} > n^{\ln \ln n}$

So $\lceil \lg n \rceil !$ *is* not polynomially bounded.

$\lceil \lg \lg n \rceil = m$, $m! < m^m < (2^m)^m = 2^{m^2} < 2^{2^{m-1}}$

and $\lg \lg n \geq m-1$, $n \geq 2^{2^{m-1}}$

$\lceil \lg \lg n \rceil ! < 2^{2^{m-1}} \leq n$。So $\lceil \lg \lg n \rceil !$ *is* polynomially bounded.

3.2-5

后者大

3.2-6

数学归纳法易证

3.2-7

用数学归纳法证明

第 4 章

4.1-1

因为包含上取整函数，又要找一个上界，所以我们要用一些技巧：

假设：$T(n) \le c \lg(n - b)$,对$\lceil n/2 \rceil$成立，进而，我们有：

$$T(n) \le c \lg(\lceil n/2 \rceil - b) + 1$$

$$\le c \lg(n/2 - b + 1) + 1$$

$$= c \lg(\frac{n - 2b + 2}{2}) + 1$$

$$= c \lg(n - 2b + 2) - c \lg 2 + 1$$

$$\le c \lg(n - b)$$

最后一个不等号成立需要，$b \ge 2, c \ge 1$

所以，得证！

4.1-2

当 $n = 1$ 时，$T(n) = 1$, $T(n) \ge cn \lg n = c \times 1 \times \lg 1 = 0$

假设 $T(n) \ge cn \lg n$ 对 $n/2$ 成立

$$T(n) \ge 2(\lceil cn/2 \rceil \lg(\lceil n/2 \rceil)) + n \ge cn \lg(n/2) + n$$

$$= cn \lg n - cn \lg 2 + n = cn \lg n - cn \lg 2 + n$$

$$= cn \lg n - cn + n$$

$$\ge cn \lg n \text{ (当 } c < 1 \text{ 时)}$$

综上所述，能取到一个小于 1 的 c，使得 $T(n) \ge cn \lg n$，则 $T(n) = \Omega(n \lg n)$

又 $T(n) = O(n \lg n)$,所以 $T(n) = \Theta(n \lg n)$

4.1-3

T(n)=cnlgn+n

4.1-4

先证$O(n\lg n)$，即要证$T(n)\le cn\lg n$

设$T(n/2)\le c(n/2)\lg(n/2)$成立，

则$T(n)=T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n)$

$$\le c(\lceil n/2\rceil)\lg(\lceil n/2\rceil)+c(\lfloor n/2\rfloor)\lg(\lfloor n/2\rfloor)+\Theta(n)$$

$$\le c\frac{n+1}{2}\lg(\frac{n+1}{2})+c\frac{n}{2}\lg(\frac{n}{2})+\Theta(n)$$

$$c\frac{n+1}{2}\lg(\frac{n+1}{2})+c\frac{n}{2}\lg(\frac{n}{2})+\Theta(n)\le cn\lg n$$

$$\Leftrightarrow c\frac{n+1}{2}\lg(\frac{n+1}{2})+c\frac{n}{2}\lg(\frac{n}{2})+\Theta(n)-cn\lg n\le 0$$

$$\Leftrightarrow c\frac{n}{2}\lg(1+\frac{1}{n})+\frac{c}{2}\lg(n+1)-\frac{2n+1}{2}c+\Theta(n)\le 0$$

$\lg(1+\frac{1}{n})$是递减函数，取$n\ge 1$，有$\lg(1+\frac{1}{n})\le 1$

$$\therefore c\frac{n}{2}\lg(1+\frac{1}{n})+\frac{c}{2}\lg(n+1)-\frac{2n+1}{2}c+\Theta(n)\le\frac{c}{2}\lg(n+1)+\Theta(n)-\frac{n+1}{2}c$$

$$=\frac{c}{4}(2\lg(n+1)-n-2)+\frac{n}{4}(4\Theta(1)-c)$$

取$n\ge 3$，有$2\lg(n+1)-n-2\le 0$，取$c\ge 4\Theta(1)$，有$4\Theta(1)-c\le 0$

$$\therefore c\frac{n+1}{2}\lg(\frac{n+1}{2})+c\frac{n}{2}\lg(\frac{n}{2})+\Theta(n)\le cn\lg n$$

即$T(n)\le cn\lg n$

再证$\Omega(n\lg n)$，即要证$T(n)\geq cn\lg n$

设$T(n/2)\geq c(n/2)\lg(n/2)$成立，

则$T(n)=T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n)$

$$\geq c(\lceil n/2\rceil)\lg(\lceil n/2\rceil)+c(\lfloor n/2\rfloor)\lg(\lfloor n/2\rfloor)+\Theta(n)$$

$$\geq c\frac{n}{2}\lg(\frac{n}{2})+c\frac{n-1}{2}\lg(\frac{n-1}{2})+\Theta(n)$$

$$c\frac{n}{2}\lg(\frac{n}{2})+c\frac{n-1}{2}\lg(\frac{n-1}{2})+\Theta(n)\geq cn\lg n$$

$$\Leftrightarrow c\frac{n}{2}\lg(\frac{n}{2})+c\frac{n-1}{2}\lg(\frac{n-1}{2})+\Theta(n)-cn\lg n\geq 0$$

$$\Leftrightarrow c\frac{n}{2}\lg(1-\frac{1}{n})-\frac{c}{2}\lg(n-1)-\frac{2n-1}{2}c+\Theta(n)\geq 0$$

$\lg(1-\frac{1}{n})$是递增函数，取$n\geq 2$, 有$\lg(1-\frac{1}{n})\geq -1$

$$\therefore c\frac{n}{2}\lg(1-\frac{1}{n})-\frac{c}{2}\lg(n-1)-\frac{2n-1}{2}c+\Theta(n)\geq\Theta(n)-\frac{c}{2}\lg(n-1)-\frac{3n-1}{2}c$$

$$=\frac{c}{2}(n+1-\lg(n-1))+(\Theta(1)-2c)n$$

取$n\geq 3$, 有$n+1-\lg(n-1)\geq 0$，取$c\leq\frac{1}{2}\Theta(1)$, 有$\Theta(1)-2c\geq 0$

$$\therefore c\frac{n}{2}\lg(\frac{n}{2})+c\frac{n-1}{2}\lg(\frac{n-1}{2})+\Theta(n)\geq cn\lg n$$

即$T(n)\geq cn\lg n$

4.1-5

往证：存在常数N_0, C_0,使得：当$n>N_0$时 $T(n)\leq C_0 n\lg n$.

假设：$T(\lfloor n/2\rfloor+17)\leq C_0(\lfloor n/2\rfloor+17)\lg(\lfloor n/2\rfloor+17)$

我们有：

$T(n) \le 2C_0(\lfloor n/2 \rfloor + 17)\lg(\lfloor n/2 \rfloor + 17) + n$

$\le 2C_0(n/2 + 17)\lg(n/2 + 17) + n$

$= C_0(n+34)[\lg(n+34) - 1] + n$

$= C_0 n\lg(n+34) - C_0 n + 34C_0\lg(n+34) - 34C_0 + n$ ……(*1)

我们取$C_0 \ge 2$

则有(*1) $\le C_0 n\lg(n+34) + 34C_0\lg(n+34) - n - 34C_0$

$< C_0 n\lg(n+34) + 34C_0\lg(n+34) - n$

$= C_0 n\lg n + C_0 n\lg(n+34) - C_0 n\lg n + 34C_0\lg(n+34) - n$ ……(*2)

下面我们只需证明存在常数N_0，$C_0 \ge 2$ 使得当$n > N_0$时，有

$C_0 n\lg(n+34) - C_0 n\lg n + 34C_0\lg(n+34) - n \le 0$

此时即有：(*2)<=$C_0 n\lg n$ （即我们往证的结论）

$C_0 n\lg(n+34) - C_0 n\lg n + 34C_0\lg(n+34) - n$

$= C_0 n(\lg(n+34) - \lg n) + 34C_0\lg(n+34) - n$

$= C_0 n(34\ln 2/\xi) + 34C_0\lg(n+34) - n \qquad (n \le \xi \le 34 + n)$

$\le 34C_0 n\ln 2/(34+n) + 34C_0\lg(n+34) - n$

此时只需取$C_0 = 3, N_0 = 100$,即可使当$n > N_0$时，有

$C_0 n\lg(n+34) - C_0 n\lg n + 34C_0\lg(n+34) - n \le 0$

所以,得证！

4.1-6

$T(n) = 2T(\sqrt{n}) + 1$

令 $m = \lg n$,则有 $n = 2^m$,$\sqrt{n} = 2^{m/2}$，则原式可以化为 $T(2^m) = 2T(2^{m/2}) + 1$,再令 $S(m) = T(2^m)$,则有 $S(m) = 2S(m/2) + 1$。利用递归树或者主方法可得：$S(m) = O(m)$,再利用代换可得：$T(n) = T(2^m) = S(m) = O(m) = O(\lg n)$

4.2-1

Determine an upper bound on $T(n) = 3T(\lfloor n/2 \rfloor) + n$ using a recursion tree. We have that each node of depth i is bounded by $n/2^i$ and therefore the contribution of each level is at most $(3/2)^i n$. The last level of depth $\lg n$ contributes $\Theta(3^{\lg n}) = \Theta(n^{\lg 3})$. Summing up we obtain:

$$
\begin{aligned}
T(n) &= 3T(\lfloor n/2 \rfloor) + n \\
&\leqslant n + (3/2)n + (3/2)^2 n + \cdots + (3/2)^{\lg n - 1} n + \Theta(n^{\lg 3}) \\
&= n \sum_{i=0}^{\lg n - 1} (3/2)^i + \Theta(n^{\lg 3}) \\
&= n \cdot \frac{(3/2)^{\lg n} - 1}{(3/2) - 1} + \Theta(n^{\lg 3}) \\
&= 2(n(3/2)^{\lg n} - n) + \Theta(n^{\lg 3}) \\
&= 2n\frac{3^{\lg n}}{2^{\lg n}} - 2n + \Theta(n^{\lg 3}) \\
&= 2 \cdot 3^{\lg n} - 2n + \Theta(n^{\lg 3}) \\
&= 2n^{\lg 3} - 2n + \Theta(n^{\lg 3}) \\
&= \Theta(n^{\lg 3})
\end{aligned}
$$

We can prove this by substitution by assumming that $T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor^{\lg 3} - c\lfloor n/2 \rfloor$. We obtain:

$$\begin{aligned} T(n) &= 3T(\lfloor n/2 \rfloor) + n \\ &\leqslant 3c\lfloor n/2 \rfloor^{\lg 3} - c\lfloor n/2 \rfloor + n \\ &\leqslant \frac{3cn^{\lg 3}}{2^{\lg 3}} - \frac{cn}{2} + n \\ &\leqslant cn^{\lg 3} - \frac{cn}{2} + n \\ &\leqslant cn^{\lg 3} \end{aligned}$$

Where the last inequality holds for $c \geqslant 2$.

4.2-2

The shortest path from the root to a leaf in the recursion tree is $n \to (1/3)n \to (1/3)^2 n \to \cdots \to 1$. Since $(1/3)^k n = 1$ when $k = \log_3 n$, the height of the part of the tree in which every node has two children is $\log_3 n$. Since the values at each of these levels of the tree add up to n, the solution to the recurrence is at least $n \log_3 n = \Omega(n \lg n)$.

4.2-3

Draw the recursion tree of $T(n) = 4T(\lfloor n/2 \rfloor) + cn$. The height of the tree is $\lg n$, the out degree of each node will be 4 and the contribution of the ith level will be $4^i \lfloor cn/2^i \rfloor$. The last level contributes $4^{\lg n}\Theta(1) = \Theta(n^2)$. Hence we have a bound on the sum given by:

$$\begin{aligned} T(n) &= 4T(\lfloor n/2 \rfloor) + cn \\ &= \sum_{i=0}^{\lg n - 1} 4^i \cdot \lfloor cn/2^i \rfloor + \Theta(n^2) \\ &\leqslant \sum_{i=0}^{\lg n - 1} 4^i \cdot cn/2^i + \Theta(n^2) \\ &= cn \sum_{i=0}^{\lg n - 1} 2^i + \Theta(n^2) + \Theta(n^2) \\ &= cn \cdot \frac{2^{\lg n} - 1}{2 - 1} + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Using the substitution method we can verify this bound. Assume the following clever induction hypothesis. Let $T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor^2 - c\lfloor n/2 \rfloor$. We have:

$$\begin{aligned} T(n) &= 4T(\lfloor n/2 \rfloor) + cn \\ &\leqslant 4(c\lfloor n/2 \rfloor^2 - c\lfloor n/2 \rfloor) + cn \\ &< 4c(n/2)^2 - 4cn/2 + cn \\ &= cn^2 - 2cn + cn \\ &= cn^2 - cn \end{aligned}$$

4.2-4

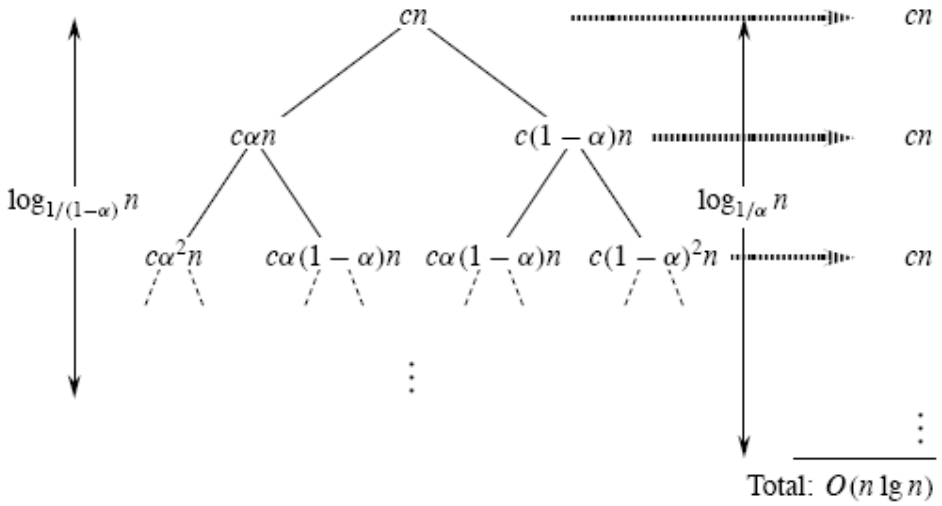
$\Theta(n^2)$

4.2-5

$T(n) = T(\alpha n) + T((1-\alpha)n) + n$

We saw the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$ in the text. This recurrence can be similarly solved.

Without loss of generality, let $\alpha \geq 1-\alpha$, so that $0 < 1-\alpha \leq 1/2$ and $1/2 \leq \alpha < 1$.



The recursion tree is full for $\log_{1/(1-\alpha)} n$ levels, each contributing cn, so we guess $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$. It has $\log_{1/\alpha} n$ levels, each contributing $\leq cn$, so we guess $O(n \log_{1/\alpha} n) = O(n \lg n)$.

Now we show that $T(n) = \Theta(n \lg n)$ by substitution. To prove the upper bound, we need to show that $T(n) \leq dn \lg n$ for a suitable constant $d > 0$.

$$\begin{aligned}
T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\
&\leq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\
&= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\
&\leq dn \lg n \ ,
\end{aligned}$$

if $dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \leq 0$. This condition is equivalent to

$$d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \leq -c \ .$$

Since $1/2 \leq \alpha < 1$ and $0 < 1-\alpha \leq 1/2$, we have that $\lg \alpha < 0$ and $\lg(1-\alpha) < 0$. Thus, $\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha) < 0$, so that when we multiply both sides of the inequality by this factor, we need to reverse the inequality:

$$d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$$

or

$$d \geq \frac{c}{-\alpha \lg \alpha + -(1-\alpha)\lg(1-\alpha)} \ .$$

The fraction on the right-hand side is a positive constant, and so it suffices to pick any value of d that is greater than or equal to this fraction.

To prove the lower bound, we need to show that $T(n) \geq dn \lg n$ for a suitable constant $d > 0$. We can use the same proof as for the upper bound, substituting $\geq$ for $\leq$, and we get the requirement that

$$0 < d \leq \frac{c}{-\alpha \lg \alpha - (1-\alpha)\lg(1-\alpha)} .$$

Therefore, $T(n) = \Theta(n \lg n)$.

4.3-1

a. $T(n) = 4T(n/2) + n$. 因为 $n = O(n^{2-\varepsilon})$，第一种情况，所以有 $T(n) = \Theta(n^2)$.

b. $T(n) = 4T(n/2) + n^2$. 因为 $n^2 = \Theta(n^2)$，所以有 $T(n) = \Theta(n^2 \lg n)$.

c. $T(n) = 4T(n/2) + n^3$. 因为 $n^3 = \Omega(n^{2+\varepsilon})$ 第三种情况，所以有 $T(n) = \Theta(n^3)$.

4.3-2

解：（在此题中，具体套用主定理的过程省略）应用主定理我们知道算法 A 的时间复杂度为 $T(n) = \Theta(n^{\lg 7})$，我们分类讨论 a 的取值：

a<16 时， $T'(n) = \Theta(n^2)$ 此时，A'比算法 A 更快

a=16 时，$T'(n) = \Theta(n^2 \lg n)$ 此时，A'比算法 A 更快

a>16 时，$T'(n) = \Theta(n^{\lg\sqrt{a}})$ 此时，若要 A'比算法 A 更快，则需要$\lg\sqrt{a}<7$，由此，a<49,所以，a 的最大整数解为 48

4.3-3

解：$n^{\log_b a} = 1$. 与 $\Theta(1)$同阶，所以是 $\Theta(\lg n)$. (直接套公式)

4.3-4

不能运用主方法

4.3-5

$$T(n) = T(n/2) + n(\sin(n - \pi/2) + 2)$$

We are in Case 3 of the Master Theorem, and the regularity condition is:

$(n/2)\ (\sin(n/2 - \pi/2) + 2) \leq c\ n(\sin(n - \pi/2) + 2)$

$$c \geq (1/2)\frac{\sin(n/2 - \pi/2) + 2}{\sin(n - \pi/2) + 2}$$

To see it is impossible to satisfy this for all large n, choose:

$n = 2\pi k$

where k is odd. \|/ relies on k being odd

Then $\sin(n/2 - \pi/2) = \sin(\pi k - \pi/2) = \sin(\pi/2) = 1$

$\sin(n - \pi/2) = \sin(2\pi k - \pi/2) = \sin(-\pi/2) = -1$

And we get

$$c \geq (1/2)\frac{1+2}{-1+2}$$

$c \geq 3/2$

Thus, we can't choose $c < 1$ to satisfy the condition.

第 5 章

5.1-1

因为本身就是一个排序过程

5.2-1

第一种想法：由于已经假设应聘者以随机的顺序出现，所以前 i 个也是以随机的顺序出现。这些前 i 个应聘者中的任何一个都等可能地是目前最有资格。应聘者 i 比应聘者 1 到 i-1 都没资格的概率是 i-1/i，因此也以 1/i 的概率被雇用。

所以正好雇一个的概率为：1*1/2*2/3*3/4*……*n-1/n=1/n.

而正好雇用 n 个的概率为：1*1/2*1/3*……*1/n=1/n!.

另一想法：n 个人随即出现，所以对他们进行全排，有 n！种排法，而我们需要只雇佣一次，所以把最好那个排第一，其余 n-1 个人进行全排，共有(n-1)!种排法，所以只雇一次的概率为(n-1)!/n! 得 1/n。

而雇用 n 次需要各个应聘者严格地按照水平递增的顺序排列，这种情况只占所有 n!种情况的 1 种，所以是 1/n!.

5.2-2

We make three observations:

1. Candidate 1 is always hired.
2. The best candidate, i.e., the one whose rank is n, is always hired.
3. If the best candidate is candidate 1, then that is the only candidate hired.

Therefore, in order for HIRE-ASSISTANT to hire exactly twice, candidate 1 must have rank $i \leq n-1$ and all candidates whose ranks are $i+1, i+2, \ldots, n-1$ must be interviewed after the candidate whose rank is n. (When $i = n-1$, this second condition vacuously holds.)

Let E_i be the event in which candidate 1 has rank i; clearly, $\Pr\{E_i\} = 1/n$ for any given value of i.

Letting j denote the position in the interview order of the best candidate, let F be the event in which candidates $2, 3, \ldots, j-1$ have ranks strictly less than the rank of candidate 1. Given that event E_i has occurred, event F occurs when the best candidate is the first one interviewed out of the $n-i$ candidates whose ranks are $i+1, i+2, \ldots, n$. Thus, $\Pr\{F \mid E_i\} = 1/(n-i)$.

Our final event is A, which occurs when HIRE-ASSISTANT hires exactly twice. Noting that the events $E_1, E_2, \ldots, E_n$ are disjoint, we have

$$\begin{aligned} A &= F \cap (E_1 \cup E_2 \cup \cdots \cup E_{n-1}) \\ &= (F \cap E_1) \cup (F \cap E_2) \cup \cdots \cup (F \cap E_{n-1}) \ . \end{aligned}$$

and

$$\Pr\{A\} = \sum_{i=1}^{n-1} \Pr\{F \cap E_i\} \ .$$

By equation (C.14),

$$\begin{aligned} \Pr\{F \cap E_i\} &= \Pr\{F \mid E_i\} \Pr\{E_i\} \\ &= \frac{1}{n-i} \cdot \frac{1}{n} \ , \end{aligned}$$

and so

$$\begin{aligned} \Pr\{A\} &= \sum_{i=1}^{n-1} \frac{1}{n-i} \cdot \frac{1}{n} \\ &= \frac{1}{n} \sum_{i=1}^{n-1} \frac{1}{n-i} \\ &= \frac{1}{n} \left(\frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{1} \right) \\ &= \frac{1}{n} \cdot H_{n-1} \ , \end{aligned}$$

where H_{n-1} is the nth harmonic number.

5.2-3

每一次投骰子：$E(x_i) = 1/6 * (1 + 2 + 3 + 4 + 5 + 6) = 7/2$

则投 n 次骰子总和的期望值为:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = n \times 7/2 = 7n/2$$

5.2-4

Let X_i=I{The event that people get the proper hat}

Then I{A} is the indicator random variable.

Make $X=X_1+X_2+\ldots+X_n$

$$E[X] = E\left[\sum X_i\right] = \sum E[X_i];\ i = 1,2,\ldots n$$

$E[X_i]=1/n$

Thus $E[X]=n\times1/n=1$,the average number is 1.

5.2-5

Let X_{ij} be an indicator random variable for the event where the pair $A[i]$, $A[j]$ for $i < j$ is inverted, i.e., $A[i] > A[j]$. More precisely, we define $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ for $1 \le i < j \le n$. We have $\Pr\{X_{ij} = 1\} = 1/2$, because given two distinct random numbers, the probability that the first is bigger than the second is $1/2$. By Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/2$.

Let X be the the random variable denoting the total number of inverted pairs in the array, so that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

We want the expected number of inverted pairs, so we take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \ .$$

We use linearity of expectation to get

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1/2 \\
&= \binom{n}{2}\frac{1}{2} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{2} \\
&= \frac{n(n-1)}{4} \ .
\end{aligned}$$

Thus the expected number of inverted pairs is $n(n-1)/4$.

5.3-1

RANDOMIZE-IN-PLACE(A)

$n \leftarrow length[A]$
swap $A[1] \leftrightarrow A[\text{RANDOM}(1, n)]$
for $i \leftarrow 2$ **to** n
    do swap $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

The loop invariant becomes

> **Loop invariant:** Just prior to the iteration of the **for** loop for each value of $i = 2, \ldots, n$, for each possible $(i-1)$-permutation, the subarray $A[1 \ldots i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

The maintenance and termination parts remain the same. The initialization part is for the subarray $A[1 \ldots 1]$, which contains any 1-permutation with probability $(n-1)!/n! = 1/n$.

5.3-2

Although PERMUTE-WITHOUT-IDENTITY will not produce the identity permutation, there are other permutations that it fails to produce. For example, consider its operation when $n = 3$, when it should be able to produce the $n! - 1 = 5$ non-identity permutations. The **for** loop iterates for $i = 1$ and $i = 2$. When $i = 1$, the call to RANDOM returns one of two possible values (either 2 or 3), and when $i = 2$, the call to RANDOM returns just one value (3). Thus, there are only $2 \cdot 1 = 2$ possible permutations that PERMUTE-WITHOUT-IDENTITY can produce, rather than the 5 that are required.

5.3-3

The PERMUTE-WITH-ALL procedure does not produce a uniform random permutation. Consider the permutations it produces when $n = 3$. There are 3 calls to RANDOM, each of which returns one of 3 values, and so there are 27 possible outcomes of calling PERMUTE-WITH-ALL. Since there are $3! = 6$ permutations, if PERMUTE-WITH-ALL did produce a uniform random permutation, then each permutation would occur $1/6$ of the time. That would mean that each permutation would have to occur an integer number m times, where $m/27 = 1/6$. No integer m satisfies this condition.

In fact, if we were to work out the possible permutations of $\langle 1, 2, 3 \rangle$ and how often they occur with PERMUTE-WITH-ALL, we would get the following probabilities:

permutation	probability
$\langle 1, 2, 3 \rangle$	4/27
$\langle 1, 3, 2 \rangle$	5/27
$\langle 2, 1, 3 \rangle$	5/27
$\langle 2, 3, 1 \rangle$	5/27
$\langle 3, 1, 2 \rangle$	4/27
$\langle 3, 2, 1 \rangle$	4/27

5.3-4

PERMUTE-BY-CYCLIC chooses *offset* as a random integer in the range $1 \le offset \le n$, and then it performs a cyclic rotation of the array. That is, $B[((i + offset - 1) \bmod n) + 1] \leftarrow A[i]$ for $i = 1, 2, \ldots, n$. (The subtraction and addition of 1 in the index calculation is due to the 1-origin indexing. If we had used 0-origin indexing instead, the index calculation would have simplied to $B[(i + offset) \bmod n] \leftarrow A[i]$ for $i = 0, 1, \ldots, n - 1$.)

Thus, once *offset* is determined, so is the entire permutation. Since each value of *offset* occurs with probability $1/n$, each element $A[i]$ has a probability of ending up in position $B[j]$ with probability $1/n$.

This procedure does not produce a uniform random permutation, however, since it can produce only n different permutations. Thus, n permutations occur with probability $1/n$, and the remaining $n! - n$ permutations occur with probability 0.

5.3-5

令 $m = n^3$，生成的全排列个数有 m^n，

其中不重复的有 $P(m,n) = m*(m-1)*\cdots*(m-n+1)$，

所以，所有元素唯一概率为 $P(m,n)/m^n$

要证 $P(m,n)/m^n \ge 1-1/n$

由于 $P(m,n)/m^n \ge [(m-n)/m]^n$

故需证：$[(m-n)/m]^n \ge 1-1/n$ 易证

令 $m = n^3$，生成的全排列个数有 m^n，

其中不重复的有 $P(m,n) = m*(m-1)*\cdots*(m-n+1)$，

所以，所有元素唯一概率为 $P(m,n)/m^n$

要证 $P(m,n)/m^n \ge 1-1/n$

由于 $P(m,n)/m^n \ge [(m-n)/m]^n$

故需证：$[(m-n)/m]^n \ge 1-1/n$ 易证

5.3-6

第 6 章

6.1-1

There is a most $2^{h+1}-1$ vertices in a complete binary tree of height h. Since the lower level need not be filled we may only have 2^h vertices.

6.1-2

Since the height of an n-element heap must satisfy that $2^h \leqslant n \leqslant 2^{h+1}-1 < 2^{h+1}$. we have $h \leqslant \lg n < h+1$. h is an integer so $h = \lfloor \lg n \rfloor$.

6.1-3

The max-heap property insures that the largest element in a subtree of a heap is at the root of the subtree.

6.1-4

The smallest element in a max-heap is always at a leaf of the tree assuming that all elements are distinct.

6.1-5

不一定是最小堆

6.1-6

No, the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ is not a max-heap.

6.1-7

根据完全二叉树的性质可以得到

6.2-1

见图 6-2

6.2-2

参考 MAX-HEAPIFY 可得

6.2-3

不变

6.2-4

无需调整

6.2-5

对以 i 为根结点的子树上每个点用循环语句实现

6.2-6

Setting the root to 0 and all other nodes to 1, will cause the 0 to propagate to bottom of the tree using at least $\lg n$ operations each costing $O(1)$. Hence we have a $\Omega(\lg n)$ lower bound for MAX-HEAPIFY.

6.3-1

见图 6-3

6.3-2

为了保证在调用 MAX-HEAPIFY(A, i)时，以 LEFT(i)和 RIGHT(i)为根的两棵二叉树都是最大堆。

6.3-3

Let H be the height of the heap.

Two subtleties to beware of:

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H - 1$ can have either height 0 or height 1.

For a complete binary tree, it's easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h. But the proof for an incomplete tree is tricky and is not derived from the proof for a complete tree.

Proof By induction on h.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$). In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H - 1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H - 1$ that are not parents of depth-H nodes.

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n - x$ is odd, because the $n - x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even). Here are two ways to do this: The first requires more cleverness, and the second requires more algebraic manipulation.

1. First method of proving the base case:
   - If n is odd, then x is even, so all nodes have siblings—i.e., all internal nodes have 2 children. Thus (see Exercise B.5-3), # of internal nodes $=$ # of leaves $- 1$.

So, n = # of nodes = # of leaves + # of internal nodes = $2 \cdot$ # of leaves $- 1$. Thus, # of leaves $= (n+1)/2 = \lceil n/2 \rceil$. (The latter equality holds because n is odd.)

- If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n+1$ nodes, where $n+1$ is odd, so the case we analyzed above would apply. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case above, # of leaves $+ 1 = \lceil (n+1)/2 \rceil = \lceil n/2 \rceil + 1$. (The latter equality holds because n is even.)

In either case, # of leaves $= \lceil n/2 \rceil$.

2. Second method of proving the base case:

   Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

   - If x is even, there are $x/2$ nodes at depth $H-1$ that are parents of depth H nodes, hence $2^{H-1} - x/2$ nodes at depth $H-1$ that are not parents of depth-H nodes. Thus,

     $$\begin{aligned} \text{total \# of height-0 nodes} &= x + 2^{H-1} - x/2 \\ &= 2^{H-1} + x/2 \\ &= (2^H + x)/2 \\ &= \lceil (2^H + x - 1)/2 \rceil \qquad \text{(because } x \text{ is even)} \\ &= \lceil n/2 \rceil \ . \end{aligned}$$

     ($n = 2^H + x - 1$ because the complete tree down to depth $H-1$ has $2^H - 1$ nodes and depth H has x nodes.)

   - If x is odd, by an argument similar to the even case, we see that

     $$\begin{aligned} \text{\# of height-0 nodes} &= x + 2^{H-1} - (x+1)/2 \\ &= 2^{H-1} + (x-1)/2 \\ &= (2^H + x - 1)/2 \\ &= n/2 \\ &= \lceil n/2 \rceil \qquad \text{(because } x \text{ odd} \Rightarrow n \text{ even)} \ . \end{aligned}$$

Inductive step: Show that if it's true for height $h-1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - n_0 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

Note that the nodes at height h in T would be at height $h-1$ if the leaves of the tree were removed—that is, they are at height $h-1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h-1$ in T', we have

$$n_h = n'_{h-1} .$$

By induction, we can bound n'_{h-1}:

$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil . \qquad \blacksquare$$

6.4-1

见图 6-4

6.4-2

HEAPSORT 仍然正确，因为每次循环的过程中还是会运行 MAX-HEAP 的过程。

6.4-3

按升序排列的数组 A 堆排序只须 build-heap 构造堆，以线性时间运行时间为 O(n)；按降序排列的数组运行时间为 O(n)+(n-1)lgn.

6.4-4

Show that the worst-case running time of heapsort is $\Omega(n \lg n)$. This is clear since sorting has a lower bound of $\Omega(n \lg n)$

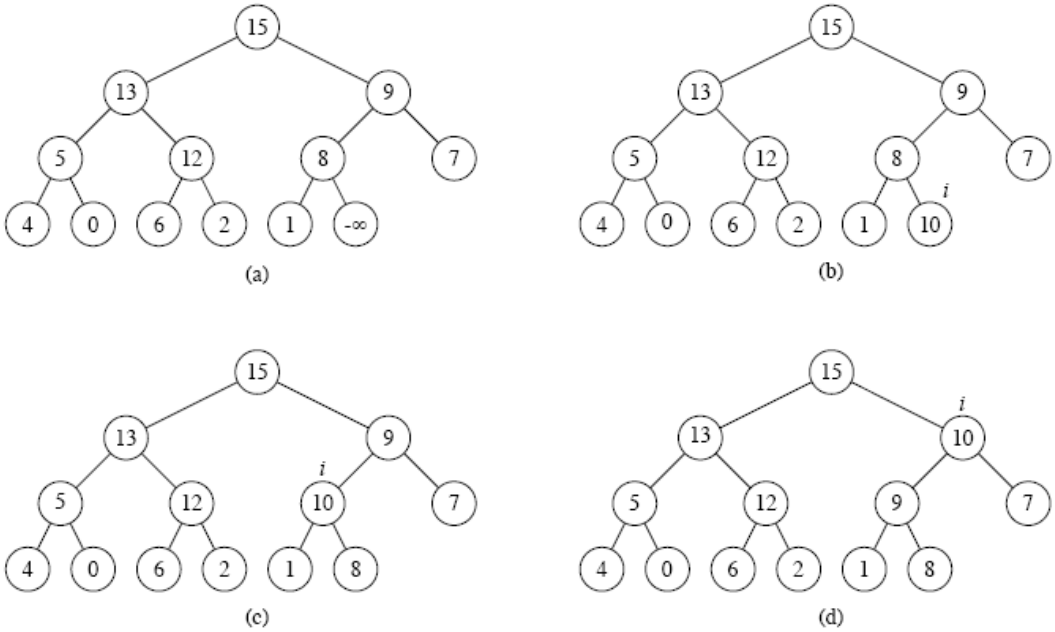
6.4-5

堆排序算法在最坏的情况下为 O(nlgn)，那在最佳情况就是 Ω(nlgn)。

6.5-1

据图 6-5

6.5-2



6.5-3

To support operations for a min-heap simply swap all comparisons between keys or elements of the heap in the max-heap implementation.

6.5-4

Since the heap data structure is represented by an array and deletions are implemented by reducing the size of the array there may be undefined values in the array past the end of the heap. Therefore it is essential that the MAX-HEAP-INSERT sets the key of the inserted node to $-\infty$ such that HEAP-INCREASE-KEY does not fail.

6.5-5

By the following loop invariant we can prove the correctness of HEAP-INCREASE-KEY:

At the start of each iteration of the **while** loop of lines $4-6$, the array $A[1 \ldots \textit{heap-size}[A]]$ satifies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

Initialization: Before the first iteration of the while the only change of the max-heap is that $A[i]$ is increased an may therefore violate the max-heap property.

Maintenance: Immediately before the iteration i and the child that violated the max-heap property has been exchanged thus restoring the max-heap property between these. This can only destroy the max-heap property between i and the parent of i.

Termination: The termination condition of the **while** states that at the end of the iteration the max-heap property between i and its parent is restored or the i is the root of the heap.

We see by the loop invariant that the heap property is restored at end of the iteration.

6.5-6

根据最先进先出队列以及堆栈的性质，运用优先级队列中所支持的操作，可易实现。

6.5-7

Algorithm 5 HEAP-DELETE(A, i)

Input: A max-heap A and integers i.
Output: The heap A with the element a position i deleted.

```
A[i] ↔ A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
key ← A[i]
if key ⩽ A[PARENT(i)] then
   MAX-HEAPIFY(A, i)
else
   while i > 1 and A[PARENT(i)] < key do
      A[i] ↔ A[PARENT(i)]
      i ← PARENT(i)
   end while
end if
```

6.5-8

Given k sorted lists with a total of n elements show how to merge them in $O(n \lg k)$ time. Insert all k elements a position 1 from each list into a heap. Use EXTRACT-MAX to obtain the first element of the merged list. Insert element at position 2 from the list where the largest element originally came from into the heap. Continuing in this fashion yields the desired algorithm. Clearly the running time is $O(n \lg k)$.

第七章

7.1-1

见图 7-1

7.1-2

When all the elements in A are the same, notice that the comparison in line 4 of PARTITION is always satified and i therefore is incremented in each iteration. Since initially $i \leftarrow p-1$ and $i+1$ is returned the returned value is $r-1$.

To make PARTITION return $(p+r)/2$ when all elements are the same, simply modify the algorithm to check for this case explicitly.

7.1-3

The running time of PARTITION is $\Theta(n)$ since each iteration of the **for** loop involves a constant number of operations and there is $\Theta(n)$ iterations in total.

7.1-4

To make QUICKSORT sort in nonincreasing order replace the $\leqslant$ comparison in PARTITION line 4 with $\geqslant$.

7.2-1

7.2-2

If the elements in A are the same, then by exercise $7.1-2$ the returned element from each call to PARTITION(A, p, r) is $r-1$ thus yielding the worst-case partitioning. The total running time is easily seen to be $\Theta(n^2)$.

7.2-3

If the elements in A are distinct and sorted in decreasing order then, as in the previous exercise, we have worst-case partitioning. The running time is again $\Theta(n^2)$.

7.2-4

因为本来的输入几乎已排好序，那么在插入排序中需移动的元素就非常少；而在快速排序中，就很可能出现最坏情况，即划分的两个区域不称。

7.2-5

The minimum depth follows a path that always takes the smaller part of the partition—i.e., that multiplies the number of elements by α. One iteration reduces the number of elements from n to αn, and i iterations reduces the number of elements to $\alpha^i n$. At a leaf, there is just one remaining element, and so at a minimum-depth leaf of depth m, we have $\alpha^m n = 1$. Thus, $\alpha^m = 1/n$. Taking logs, we get $m \lg \alpha = -\lg n$, or $m = -\lg n / \lg \alpha$.

Similarly, maximum depth corresponds to always taking the larger part of the partition, i.e., keeping a fraction $1-\alpha$ of the elements each time. The maximum depth M is reached when there is one element left, that is, when $(1-\alpha)^M n = 1$. Thus, $M = -\lg n / \lg(1-\alpha)$.

All these equations are approximate because we are ignoring floors and ceilings.

7.2-6

7.3-1

We may be interested in the worst-case performance, but in that case, the randomization is irrelevant: it won't improve the worst case. What randomization can do is make the chance of encountering a worst-case scenario small.

7.3-2

(1) 由 Quick Sort 算法最坏情况分析得知：n 个元素每次都划 n-1 和 1 个，因为是 p<r 的时候才调用，所以为Θ（n）

(2) 最好情况是每次都在最中间的位置分，所以递推式是：

N（n）= 1+ 2*N（n/2）

不难得到：N(n) =Θ（n）

7.4-1

7.4-2

T(n)=2*T(n/2)+ Θ（n）

可以得到 T(n) =Θ（n *lg*n）

可得最佳运行时间Ω(n lgn)

7.4-3

纯数学问题，对式子求导即可得。

7.4-4

见 RANDOMIZED-QUICKSORT.ppt

7.4-5

见快速排序改进算法(7.4-5).pdf

7.4-6

第八章

8.1-1

决策树是一颗二叉树，每个节点表示元素之间一组可能的排序，它予以进行的比较相一致，比较的结果是树的边。先来说明一些二叉树的性质，令 T 是深度为 d 的二叉树，则 T 最多有 2^片树叶。具有 L 片树叶的二叉树的深度至少是 logL。所以，对 n 个元素排序的决策树必然有 n!片树叶（因为 n 个数有 n!种不同的大小关系），所以决策树的深度至少是 log(n!)

8.1-2

8.1-3

If the sort runs in linear time for m input permutations, then the height h of the portion of the decision tree consisting of the m corresponding leaves and their ancestors is linear.

Use the same argument as in the proof of Theorem 8.1 to show that this is impossible for $m = n!/2$, $n!/n$, or $n!/2^n$.

We have $2^h \geq m$, which gives us $h \geq \lg m$. For all the possible m's given here, $\lg m = \Omega(n \lg n)$, hence $h = \Omega(n \lg n)$.

In particular,

$$\lg \frac{n!}{2} = \lg n! - 1 \geq n \lg n - n \lg e - 1$$

$$\lg \frac{n!}{n} = \lg n! - \lg n \geq n \lg n - n \lg e - \lg n$$

$$\lg \frac{n!}{2^n} = \lg n! - n \geq n \lg n - n \lg e - n$$

8.1-4

Let S be a sequence of n elements divided into n/k subsequences each of length k where all of the elements in any subsequence are larger than all of the elements of a preceding subsequence and smaller than all of the elements of a succeeding subsequence.

Claim

Any comparison-based sorting algorithm to sort s must take $\Omega(n \lg k)$ time in the worst case.

Proof First notice that, as pointed out in the hint, we cannot prove the lower bound by multiplying together the lower bounds for sorting each subsequence. That would only prove that there is no faster algorithm *that sorts the subsequences independently*. This was not what we are asked to prove; we cannot introduce *any* extra assumptions.

Now, consider the decision tree of height h for any comparison sort for S. Since the elements of each subsequence can be in any order, any of the $k!$ permutations correspond to the final sorted order of a subsequence. And, since there are n/k such subsequences, each of which can be in any order, there are $(k!)^{n/k}$ permutations of S that could correspond to the sorting of some input order. Thus, any decision tree for sorting S must have at least $(k!)^{n/k}$ leaves. Since a binary tree of height h has no more than 2^h leaves, we must have $2^h \geq (k!)^{n/k}$ or $h \geq \lg((k!)^{n/k})$. We therefore obtain

$$\begin{aligned} h &\geq \lg((k!)^{n/k}) \\ &= (n/k)\lg(k!) \\ &\geq (n/k)\lg((k/2)^{k/2}) \\ &= (n/2)\lg(k/2)\ . \end{aligned}$$

The third line comes from $k!$ having its $k/2$ largest terms being at least $k/2$ each. (We implicitly assume here that k is even. We could adjust with floors and ceilings if k were odd.)

Since there exists at least one path in any decision tree for sorting S that has length at least $(n/2)\lg(k/2)$, the worst-case running time of any comparison-based sorting algorithm for S is $\Omega(n \lg k)$. ■

8.2-1

见图 8-2

8.2-2 和 8.2-3

Notice that the correctness argument in the text does not depend on the order in which A is processed. The algorithm is correct no matter what order is used!

But the modified algorithm is not stable. As before, in the final **for** loop an element equal to one taken from A earlier is placed before the earlier one (i.e., at a lower index position) in the output arrray B. The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.

In particular, the algorithm still places the elements with value k in positions $C[k-1]+1$ through $C[k]$, but in the reverse order of their appearance in A.

8.2-4

Given n integers from 1 to k show how to count the number of elements from a to b in O(1) time with O(n + k) preprocessing time. As shown in COUNTING-SORT we can produce an array C such that C[i] contains the number of elements less than or equal to i. Clearly, C[b] − C[a] gives the desired answer.

8.3-1

见图 8-3

8.3-2

Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1 \ldots j-1]$, we do it the following way: compare $A[j]$ to $A[i]$, starting with $i = j-1$ and going down to $i = 1$. Continue at long as $A[j] < A[i]$.

Merge sort as defined is stable, because when two elements compared are equal, the tie is broken by taking the element from array L which keeps them in the original order.

Heapsort and quicksort are not stable.

One scheme that makes a sorting algorithm stable is to store the index of each element (the element's place in the original ordering) with the element. When comparing two elements, compare them by their values and break ties by their indices.

Additional space requirements: For n elements, their indices are $1 \ldots n$. Each can be written in $\lg n$ bits, so together they take $O(n \lg n)$ additional space.

Additional time requirements: The worst case is when all elements are equal. The asymptotic time does not change because we add a constant amount of work to each comparison.

8.3-3

Basis: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that radix sort works for $d - 1$ digits, we'll show that it works for d digits.

Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d - 1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d - 1$ digits.

The sort on digit d will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d - 1$ digits when their dth digits are equal, and the elements are already sorted by their low-order $d - 1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose dth digits were equal—elements that *were* in the right order after the sort on their lower-order digits.

8.3-4

Treat the numbers as 2-digit numbers in radix n. Each digit ranges from 0 to $n-1$. Sort these 2-digit numbers with radix sort.

There are 2 calls to counting sort, each taking $\Theta(n+n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

8.3-5(*)

8.4-1

见图 8-4

8.4-2

The worst-case running time for the bucket-sort algorithm occurs when the assumption of uniformly distributed input does not hold. If, for example, all the input ends up in the first bucket, then in the insertion sort phase it needs to sort all the input, which takes $O(n^2)$ time.

A simple change that will preserve the linear expected running time and make the worst-case running time $O(n \lg n)$ is to use a worst-case $O(n \lg n)$-time algorithm like merge sort instead of insertion sort when sorting the buckets.

8.4-3

3/2，1/2

8.4-4(*)

8.4-5(*)

第九章

9.1-1

Show how to find the the second smallest element of n elements using $n+\lceil \lg n \rceil - 2$ comparisons. To find the smallest element construct a tournament as follows: Compare all the numbers in pairs. Only the smallest number of each pair is potentially the smallest of all so the problem is reduced to size $\lceil n/2 \rceil$. Continuing in this fashion until there is only one left clearly solves the problem.

Exactly $n-1$ comparisons are needed since the tournament can be drawn as an n-leaf binary tree which has $n-1$ internal nodes (show by induction on n). Each of these nodes correspond to a comparison.

We can use this binary tree to also locate the second smallest number. The path from the root to the smallest element (of height $\lceil \lg n \rceil$) must contain the second smallest element. Conducting a tournament among these uses $\lceil \lg n \rceil - 1$ comparisons.

The total number of comparisons are: $n-1+\lceil \lg n \rceil - 1 = n + \lceil \lg n \rceil - 2$.

9.1-2

在同时找到最大值和最小值时，每个元素都参与了与最大值和最小值的比较，比较了两次。将数组成对处理。两两互相比较，将大的与最大值比较，小的与最小值比较。每两个元素需要的比较次数是，两两比较一次，大者与最大值比较一次，小者与最小值比较一次，共三次。

9.2-1

长度为 0 的数组，RANDOMIZED-SELECT 直接返回，当然不会递归调用。

9.3-1

Consider the analysis of the algorithm for groups of k. The number of elements less than (or greater than) the median of the medians x will be at least $\lceil \frac{k}{2} \rceil \left(\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \rceil - 2 \right) \geqslant \frac{n}{4} - k$. Hence, in the worst-case SELECT will be called recursively on at most $n - \left(\frac{n}{4} - k \right) = \frac{3n}{4} + k$ elements. The recurrence is

$$T(n) \leqslant T(\lceil n/k \rceil) + T(3n/4 + k) + O(n)$$

Solving by substitution we obtain a bound for which k the algorithm will be linear. Assume $T(n) \leqslant cn$ for all smaller n. We have:

$$\begin{aligned} T(n) &\leqslant c \left\lceil \frac{n}{k} \right\rceil + c \left(\frac{3n}{4} + k \right) + O(n) \\ &\leqslant c(\frac{n}{k} + 1) + \frac{3cn}{k} + ck + O(n) \\ &\leqslant \frac{cn}{k} + \frac{3cn}{k} + c(k+1) + O(n) \\ &= cn \left(\frac{1}{k} + \frac{3}{4} \right) + c(k+1) + O(n) \\ &\leqslant cn \end{aligned}$$

Where the last equation only holds for $k > 4$. Thus, we have shown that the algorithm will compute in linear time for any group size of 4 or more. In fact, the algorithm is $\Omega(n \lg n)$ for $k = 3$. This can be shown by example.

9.3-2

9.3-3

Quicksort can be made to run in $O(n \lg n)$ time worst-case by noticing that we can perform "perfect partitioning": Simply use the linear time select to find the median and perform the partitioning around it. This clearly achieves the bound.

9.3-4

只用比较的过程来确定第 i 小的元素，当然可以确定 n 个元素中每个元素与第 i 小的元素之间的大小关系。

9.3-5

We assume that are given a procedure MEDIAN that takes as parameters an array A and subarray indices p and r, and returns the value of the median element of $A[p..r]$ in $O(n)$ time in the worst case.

Given MEDIAN, here is a linear-time algorithm SELECT′ for finding the ith smallest element in $A[p..r]$. This algorithm uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

```
SELECT′(A, p, r, i)
if p = r
   then return A[p]
x ← MEDIAN(A, p, r)
q ← PARTITION(x)
k ← q − p + 1
if i = k
   then return A[q]
elseif i < k
   then return SELECT′(A, p, q − 1, i)
else return SELECT′(A, q + 1, r, i − k)
```

Because x is the median of $A[p..r]$, each of the subarrays $A[p..q-1]$ and $A[q+1..r]$ has at most half the number of elements of $A[p..r]$. The recurrence for the worst-case running time of SELECT′ is $T(n) \le T(n/2) + O(n) = O(n)$.

9.3-6

这个用分治就可以了：
将 A[p～q]等分成 t 个大小相等的集合（每个小集合含 k 个元素）
{
找到第 k*(t/2)大的元素 M;
根据 M 将集合分成两部分 A[p～r]和 A[r+1～q]，使得前面部分的元素 <=M，后面的元素>M;
将 A[p～r]等分成 t/2 个小集合;
将 A[r+1～q]等分成 t-(t/2)个小集合;
}

9.3-7

1.找S的中位数x，将S中的每个数与x相减，再求绝对值。//每个数的下标与其绝对值建立对应关系

2.在n个绝对值中找第k小y;

3.依次检查每个绝对值，如果它小于等于y，那么对应的数是与x最近的k个数之一。

- $T(n)=O(n)$

9.3-8

Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m, and let's suppose that it's in $X[k]$. Then k elements of X are less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k)=k$ elements of Y that are greater than or equal to m.

Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n-k] \le X[k] \le Y[n-k+1]$. A boundary case occurs for $k=n$. Then $n-k=0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \le Y[1]$.

Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then $X[k]$ is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$.

Thus, we can use a binary search to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n-k] \le X[k] \le Y[n-k+1]$ or $k=n$ and $X[k] \le Y[n-k+1]$; if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n-k] \le Y[k] \le X[n-k+1]$ or $k=n$ and $Y[k] \le X[n-k+1]$; such a $Y[k]$ is the median. Since each binary search takes $O(\lg n)$ time, we spend a total of $O(\lg n)$ time.

Here's how we write the algorithm in pseudocode:

```
TWO-ARRAY-MEDIAN(X, Y)
n ← length[X]                    ▷ n also equals length[Y]
median ← FIND-MEDIAN(X, Y, n, 1, n)
if median = NOT-FOUND
  then median ← FIND-MEDIAN(Y, X, n, 1, n)
return median

FIND-MEDIAN(A, B, n, low, high)
if low > high
  then return NOT-FOUND
  else k ← ⌊(low + high)/2⌋
       if k = n and A[n] ≤ B[1]
         then return A[n]
       elseif k < n and B[n − k] ≤ A[k] ≤ B[n − k + 1]
         then return A[k]
       elseif A[k] > B[n − k + 1]
         then return FIND-MEDIAN(A, B, n, low, k − 1)
       else return FIND-MEDIAN(A, B, n, k + 1, high)
```

9.3-9

In order to find the optimal placement for Professor Olay's pipeline, we need only find the median(s) of the y-coordinates of his oil wells, as the following proof explains.

Claim

The optimal y-coordinate for Professor Olay's east-west oil pipeline is as follows:

- If n is even, then on either the oil well whose y-coordinate is the lower median or the one whose y-coordinate is the upper median, or anywhere between them.
- If n is odd, then on the oil well whose y-coordinate is the median.

Proof We examine various cases. In each case, we will start out with the pipeline at a particular y-coordinate and see what happens when we move it. We'll denote by s the sum of the north-south spurs with the pipeline at the starting location, and s' will denote the sum after moving the pipeline.

We start with the case in which n is even. Let us start with the pipeline somewhere on or between the two oil wells whose y-coordinates are the lower and upper medians. If we move the pipeline by a vertical distance d without crossing either of the median wells, then $n/2$ of the wells become d farther from the pipeline and $n/2$ become d closer, and so $s' = s + dn/2 - dn/2 = s$; thus, all locations on or between the two medians are equally good.

Now suppose that the pipeline goes through the oil well whose y-coordinate is the upper median. What happens when we increase the y-coordinate of the pipeline by $d > 0$ units, so that it moves above the oil well that achieves the upper median? All oil wells whose y-coordinates are at or below the upper median become d units farther from the pipeline, and there are at least $n/2 + 1$ such oil wells (the upper median, and every well at or below the lower median). There are at most $n/2 - 1$ oil wells whose y-coordinates are above the upper median, and each of these oil wells becomes at most d units closer to the pipeline when it moves up. Thus, we have a lower bound on s' of $s' \geq s + d(n/2 + 1) - d(n/2 - 1) = s + 2d > s$. We conclude that moving the pipeline up from the oil well at the upper median increases the total spur length. A symmetric argument shows that if we start with the pipeline going through the oil well whose y-coordinate is the lower median and move it down, then the total spur length increases.

We see, therefore, that when n is even, an optimal placement of the pipeline is anywhere on or between the two medians.

Now we consider the case when n is odd. We start with the pipeline going through the oil well whose y-coordinate is the median, and we consider what happens when we move it up by $d > 0$ units. All oil wells at or below the median become d units farther from the pipeline, and there are at least $(n + 1)/2$ such wells (the one at the median and the $(n - 1)/2$ at or below the median. There are at most $(n - 1)/2$ oil wells above the median, and each of these becomes at most d units closer to the pipeline. We get a lower bound on s' of $s' \geq s + d(n + 1)/2 - d(n - 1)/2 = s + d > s$, and we conclude that moving the pipeline up from the oil well at the median increases the total spur length. A symmetric argument shows that moving the pipeline down from the median also increases the total spur length, and so the optimal placement of the pipeline is on the median. ■ (claim)

Since we know we are looking for the median, we can use the linear-time median-finding algorithm.

第 15 章

15.1-1

```
PRINT-STATIONS(l, j, n)
    i ← l*
   if j=0
      then    return
      else
      if(j>1)   i ← li[j]
            PRINT-STATION(l, j-1, n)

 if   j = n
     then i ← l*
                print "line" i ", station" j
     else i ← li[j+1]
                print "line" i ", station" j
```

15.1-2

1．j = n 时,由(15.8), $r_i(j) = 1 = 2^{n-n}$ 成立

2．假设当 j = k 时, $r_i(k) = 2^{n-k}$

当 j = k – 1 时

$r_i(k-1) = r_1(k) + r_2(k) = 2^{n-k} + 2^{n-k} = 2^{n-(k-1)}$ 成立

3．综上，得证

15.1-3

$$f_i[j] = \sum_{i=1}^{2}\sum_{j=1}^{n} ri(j) = 2(\sum_{j=1}^{n} 2^{n-j}) = 2(2^n - 1) = 2^{n+1} - 1$$

15.1-4

原来 $f_i[j]$含有 2n 个表项 ，$l_i[j]$含有 2n-2 个表项，$l_i[j]$保持不变，而将 $f_i[j]$改为四个表项，两个记录 $f_i[j-1]$，两个记录 $f_i[j]$。因为每次求 $f_i[j]$只需知道上一步的 f_i 值，没有必要将所有 i 的 f_i 值都记录。这样，空间需求便缩减到了 2n+2，而仍能输出通过工厂的最快路线上的所有装配站。

15.1-5

证明：如果 $l_1[j]=2$,则有 $f_2(j-1)+t_{2,j-1}+a_{1,j}<f_1(j-1)+a_{1,j}$

同样地，如果 $l_2[j]=1$,则有 $f_1(j-1)+t_{2,j-1}+a_{2,j}< f_2(j-1)+a_{2,j}$

两式相加：$t_{2,j-1}+t_{2,j-1}<0$.题目中说，$t_{i,j}$ 都非负，所以矛盾，得证！

15.2-1

Solve the matrix chain order for a specific problem. This can be done by computing MATRIX-CHAIN-ORDER(p) where $p = \langle 5, 10, 3, 12, 5, 50, 6 \rangle$ or simply using the equation:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

The resulting table is the following:

i\j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

The table is computed simply by the fact that $m[i,i] = 0$ for all i. This information is used to compute $m[i,i+1]$ for $i = 1, \ldots n-1$ an so on.

最终答案：$((A_1A_2)((A_3A_4)(A_5A_6)))$

15.2-2

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
if (i = j)
        return A[i]
if (j =i+1)
        return MATRIX-MULTIPLY (A[i], A[j])
else
        B1 =MATRIX-CHAIN-MULTIPLY (A, s, i, S[i,j]);
        B2= MATRIX-CHAIN-MULTIPLY (A, s, S[i,j]+1, j);
        return MATRIX-MULTIPLY (B1, B2);
```

15.2-3

当 n=1 时，有 P（n）=1 成立

现在假设当任意的 k<n 都成立，即 P（k）≥$c2^k$, 则有

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \ge \sum_{k=1}^{n-1} c\,2^k c2^{n-k} = c^2(n-1)2^n$$

由此，对于任意给定的 c 总存在一个 $N_0 = \frac{1}{c} + 1$，使得当 $n > N_0$ 时，有 $P(n) \ge c2^n$

所以，得证

15.2-4

Each time the l-loop executes, the i-loop executes $n - l + 1$ times. Each time the i-loop executes, the k-loop executes $j - i = l - 1$ times, each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n}(n - l + 1)(l - 1)2$. Thus,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) &= \sum_{l=2}^{n}(n - l + 1)(l - 1)2 \\
&= 2\sum_{l=1}^{n-1}(n - l)l \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}$$

15.2-5

用递归

15.3-1

Running RECURSIVE-MATRIX-CHAIN is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

Consider the treatment of subproblems by the two approaches.

- For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left- and right-half subproblem results is thus the product of the number of ways to do the left half and the number of ways to do the right half.
- For each possible place to split the matrix chain, RECURSIVE-MATRIX-CHAIN finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left- and right-half subproblem results is $O(1)$.

Section 15.2 argued that the running time for enumeration is $\Omega(4^n/n^{3/2})$. We will show that the running time for RECURSIVE-MATRIX-CHAIN is $O(n3^{n-1})$.

To get an upper bound on the running time of RECURSIVE-MATRIX-CHAIN, we'll use the same approach used in Section 15.2 to get a lower bound: Derive a recurrence of the form $T(n) \le \ldots$ and solve it by substitution. For the lower-bound recurrence, the book assumed that the execution of lines 1–2 and 6–7 each take at least unit time. For the upper-bound recurrence, we'll assume those pairs of lines each take at most constant time c. Thus, we have the recurrence

$$T(n) \le \begin{cases} c & \text{if } n = 1 , \\ c + \displaystyle\sum_{k=1}^{n-1} (T(k) + T(n-k) + c) & \text{if } n \ge 2 . \end{cases}$$

This is just like the book's $\ge$ recurrence except that it has c instead of 1, and so we can be rewrite it as

$$T(n) \le 2 \sum_{i=1}^{n-1} T(i) + cn .$$

We shall prove that $T(n) = O(n3^{n-1})$ using the substitution method. (Note: Any upper bound on $T(n)$ that is $o(4^n/n^{3/2})$ will suffice. You might prefer to prove one that is easier to think up, such as $T(n) = O(3.5^n)$.) Specifically, we shall show that $T(n) \le cn3^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) \le c = c \cdot 1 \cdot 3^{1-1}$.

Inductively, for $n \ge 2$ we have

$$\begin{aligned}
T(n) &\le 2 \sum_{i=1}^{n-1} T(i) + cn \\
&\le 2 \sum_{i=1}^{n-1} ci3^{i-1} + cn \\
&\le c \cdot \left(2 \sum_{i=1}^{n-1} i3^{i-1} + n \right) \\
&= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2} \right) + n \right) \qquad \text{(see below)} \\
&= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n \right) \\
&= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\
&\le cn3^{n-1} \text{ for all } c > 0, n \ge 1 .
\end{aligned}$$

Running RECURSIVE-MATRIX-CHAIN takes $O(n3^{n-1})$ time, and enumerating all parenthesizations takes $\Omega(4^n/n^{3/2})$ time, and so RECURSIVE-MATRIX-CHAIN is more efficient than enumeration.

Note: The above substitution uses the fact that

$$\sum_{i=1}^{n-1} i x^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

This equation can be derived from equation (A.5) by taking the derivative. Let

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 .$$

Then

$$\sum_{i=1}^{n-1} i x^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

15.3-2

Draw a nice recursion tree. The MERGESORT algorithm performs at most a single call to any pair of indices of the array that is being sorted. In other words, the subproblems do not overlap and therefore memoization will not improve the running time.

15.3-3

这个问题同样具有最优子结构。下面进行分析：
采用动态规划法的第一步就是寻找最优子结构，然后利用这一子结构，就可以根据子问题的最优解结构造出原问题的一个最优解。
用记号 $A_{i..j}$ 表示对乘积 $A_iA_{i+1}\cdots A_j$ 的求积结果，其中 i<j.
这个问题的最优子结构如下：假设 $A_iA_{i+1}\cdots A_j$ 的一个最优加全部括号的乘法把乘积 A_k 与 A_{k+1} 之间分开，则对 $A_iA_{i+1}\cdots A_j$ 最优加全部括号的“前缀”子链 $A_iA_{i+1}\cdots A_k$ 的加全部括号必须是 $A_iA_{i+1}\cdots A_K$ 的一个最优加全部括号，因为如果对 $A_iA_{i+1}\cdots A_k$ 有一个代价更大的加全部括号，那么把它替换到 $A_iA_{i+1}\cdots A_j$ 的最优加全部括号中就会产生 $A_iA_{i+1}\cdots A_j$ 的另一种加全部括号，而它会具有更大的代价，产生矛盾。所以判断这个问题同样具有最优子结构。

15.3-4

我们要求解到达 S1, j 最快路线，必须先知道到达 S1, j-1 和 S2, j-1 的最快路线，以此类推，要使到达某一个装配站的路线最快，必须使到达它前一个装配站的路线最快，所以在求解到达每个装配站的最快路线时，必须重复求解到达某些装配站的最快路线，这就是重复子问题。
如果我们在求解装配线调度问题时采用开销比较大的递归算法，就会在递归树中看到，某些值被求解了多次，也正是这个原因使得此种算法拥有指数级的时间复杂度。
所以装配线调度问题比较适合用动态规划求解，即将重复子问题的解存于表中以便后面调用，避免重复计算。

15.3-5

此处求解时用了贪心策略，就是每次选取矩阵 A_k 分裂时，总使得 $p_{i-1}p_kp_j$ 最小。实例略。

15.4-1

The LCS is <1， 0， 0， 1， 1， 0>

	Yi	0	1	0	1	1	0	1	1	0
Xi	0	0	0	0	0	0	0	0	0	0
1	0	↑ 0	1	← 1	1	1	← 1	1	1	← 1
0	0	1	↑ 1	2	← 2	← 2	2	← 2	← 2	2
0	0	1	↑ 1	2	↑ 2	↑ 2	3	← 3	← 3	3
1	0	↑ 1	2	↑ 2	3	3	↑ 3	4	4	← 4
0	0	1	↑ 2	3	↑ 3	↑ 3	4	↑ 4	↑ 4	5
1	0	↑ 1	2	↑ 3	4	4	↑ 4	5	5	↑ 5
0	0	1	2	3	↑ 4	↑ 4	5	↑ 5	↑ 5	6
1	0	↑ 1	2	↑ 3	4	5	↑ 6	6	6	↑ 6

15.4-2

```
PRINT_LCS(c,x,y,i,j)
  if x[i]=y[j]
     PRINT_LCS(c,x,y,i-1,j-1)
     print x[i]
     else if c[i –1]>=c[i,j-1]
            PRINT_LCS(c,x,y,i-1,j)
            else PRINT_LCS(c,x,y,i,j-1)
```

15.4-3

Give an efficient memoized implementation of LCS-LENGTH. This can done directly by using:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Algorithm 8 LCS-LENGTH(X, Y)

Input: The two strings X and Y.
Output: The longest common substring of X and Y.

```
m ← length[X]
n ← length[Y]
for i ← 1 to m do
   for j ← 1 to n do
      c[i, j] ← −1
   end for
end for
return LOOKUP-LENGTH(X, Y, m, n)
```

Algorithm 9 LOOKUP-LENGTH(X, Y, i, j)

```
if c[i, j] > −1 then
  return c[i, j]
end if
if i = 0 or j = 0 then
  c[i, j] ← 0
else
  if x_i = y_i then
    c[i, j] ← LOOKUP-LENGTH(X, Y, i − 1, j − 1) + 1
  else
    c[i, j] ← max{LOOKUP-LENGTH(X, Y, i, j − 1), LOOKUP-LENGTH(X, Y, i − 1, j)}
  end if
end if
return c[i, j]
```

15.4-4

When computing a particular row of the c table, no rows before the previous row are needed. Thus only two rows—$2 \cdot length[Y]$ entries—need to be kept in memory at a time. (Note: Each row of c actually has $length[Y] + 1$ entries, but we don't need to store the column of 0's—instead we can make the program "know" that those entries are 0.) With this idea, we need only $2 \cdot \min(m, n)$ entries if we always call LCS-LENGTH with the shorter sequence as the Y argument.

We can thus do away with the c table as follows:

- Use two arrays of length $\min(m, n)$, *previous-row* and *current-row*, to hold the appropriate rows of c.
- Initialize *previous-row* to all 0 and compute *current-row* from left to right.
- When *current-row* is filled, if there are still more rows to compute, copy *current-row* into *previous-row* and compute the new *current-row*.

Actually only a little more than one row's worth of c entries—$\min(m, n) + 1$ entries—are needed during the computation. The only entries needed in the table when it is time to compute $c[i, j]$ are $c[i, k]$ for $k \leq j - 1$ (i.e., earlier entries in the current row, which will be needed to compute the next row); and $c[i - 1, k]$ for $k \geq j - 1$ (i.e., entries in the previous row that are still needed to compute the rest of the current row). This is one entry for each k from 1 to $\min(m, n)$ except that there are two entries with $k = j - 1$, hence the additional entry needed besides the one row's worth of entries.

We can thus do away with the c table as follows:

- Use an array a of length $\min(m, n) + 1$ to hold the appropriate entries of c. At the time $c[i, j]$ is to be computed, a will hold the following entries:
  - $a[k] = c[i, k]$ for $1 \leq k < j - 1$ (i.e., earlier entries in the current "row"),
  - $a[k] = c[i - 1, k]$ for $k \geq j - 1$ (i.e., entries in the previous "row"),
  - $a[0] = c[i, j - 1]$ (i.e., the previous entry computed, which couldn't be put into the "right" place in a without erasing the still-needed $c[i - 1, j - 1]$).

- Initialize a to all 0 and compute the entries from left to right.
  - Note that the 3 values needed to compute $c[i, j]$ for $j > 1$ are in $a[0] = c[i, j-1]$, $a[j-1] = c[i-1, j-1]$, and $a[j] = c[i-1, j]$.
  - When $c[i, j]$ has been computed, move $a[0]$ ($c[i, j-1]$) to its "correct" place, $a[j-1]$, and put $c[i, j]$ in $a[0]$.

15.4-5

Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ we wish to find the longest monotonically increasing subsequence. First sort the elements of X and create another sequence X′. Finding the longest common subsequence of X and X′ yields the longest monotonically increasing subsequence of X. The running time is $O(n^2)$ since sorting can be done in $O(n \lg n)$ and the call to LCS-LENGTH is $O(n^2)$.

15.4-6

```
#include <iostream>
using namespace std;
int find(int *a,int len,int n)//修改后的二分查找，若返回值为 x，则 a[x]>=n
{
    int left=0,right=len,mid=(left+right)/2;
    while(left<=right)
    {
        if(n>a[mid]) left=mid+1;
        else if(n<a[mid]) right=mid-1;
        else return mid;
        mid=(left+right)/2;
    }
    return left;
}
int main()
{
    int n,a[100],c[100],i,j,len;//新开一变量 len,用来储存每次循环结束后 c 中已经求出值的元素的最大下标
    while(cin>>n)
    {
        for(i=0;i<n;i++)
            cin>>a[i];
        b[0]=1;
        c[0]=-1;
        c[1]=a[0];
        len=1;//此时只有 c[1]求出来，最长递增子序列的长度为 1.
        for(i=1;i<n;i++)
        {
            j=find(c,len,a[i]);
            c[j]=a[i];
            if(j>len)//要更新 len,另外补充一点：由二分查找可知 j 只可能比 len 大 1
                len=j;//更新 len
```

```
            }
            cout<<len<<endl;
        }
        return 0;
}
```

15.5-1

可给出如下算法：

```
CONSTRUCT-OPTIAML-BST(root, i, j)
if(i==1&&j==n)
        print    root[i, j] is the root
if (i<j)
        print root[i, root[i,j]-1] is the left child of K_root[i,j]
        CONSTRUCT-OPTIAML-BST(root, i, root[i, j]-1)
        print root[root[i,j]+1, j] is the right child of K_root[i,j]
        CONSTRUCT-OPTIAML-BST(root, root[i, j], j)
if(i==j)

        print d_{i-1} is the left child of K_i
        print d_i is the right child of K_i
if(i>j)
        print d_j is the right child of K_j
```

15.5-2

可参考书中的例子解答

15.5-3

$O(n^3)$

式（15.17）花费的时间是 O(n)。注意算法 OPTIMAL-BST，虽然将 w[i,j]用 15.17 式计算，但是这个结果的 w[i,j]可以用于第 10 行中的计算，因此总的来说，时间复杂度仍然为 $O(n)*O(n)*(O(n)+O(n))=O(n^3)$.

15.5-4

第九行替换为：

```
if   i=j
     r<-j
     else for    r<-root[i,j-1] to root[i+1,j]
```

第 16 章

16.1-1

动态规划时间复杂度为$O(n^3)$,贪心算法时间复杂度为O(n)

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ max_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

```
DYNAMICACTIVITYSELECTOR(S)
 1   initialize c[i, j] = 0
 2   for i ← 1 to n
 3       do for j ← 2 to n
 4              do if i ≥ j
 5                    then c[i, j] ← ∅
 6                    else  for k ← i + 1 to j − 1
 7                              do if c[i, j] < c[i, k] + c[k, j] + 1
 8                                    then c[i, j] ← c[i, k] + c[k, j] + 1
 9                                         s[i, j] ← k
```

16.1-2

证明：对于任意非空子问题 Sij,设 am 是 Sij 中具有最迟开始时间的活动

sm=max{Sk， ak∈Sij}

那么，1）活动 am 在 Sij 的某最大兼容子活动中被使用

2）子问题 Smj 为空，所以选择 am 将使子问题 Sim 为唯一可能非空的问题

首先，证明 2)。假设 Smj 非空，则有活动 ak 将满足

$S_m<f_m<=S_k<f_k<=S_j$

ak 同时也在 Sij 中，且具有比 am 更晚的开始时间，这与 am 的选择相矛盾，所以推出 Smj 为空。

然后，证明 1)。设 Aij 为 Sij 的最大兼容活动子集，且将 Aij 中的活动按开始时间单调递增排序，又设 ak 为 Aij 的最后一个活动，如果 ak=am，则得到结论。如果 ak≠am，则构造子集 A' ij=Aij−{ak}∪{am}, 因为在 Aij 中 ak 是最后一个活动，而 sm>=sk, 所以 A' ij 中的活动是不相交的，此时 Aij 与 A' ij 的活动数目相同，因此 A' ij 是包含 am 的 Sij 的最大兼容活动的集合。

16.1-3

Find the smallest number of lecture halls to schedule a set of activities S in. To do this efficiently move through the activities according to starting and finishing times. Maintain two lists of lecture halls: Halls that are busy at time t and halls that are free at time t. When t is the starting time for some activity schedule this activity to a free lecture hall and move the hall to the busy list. Similarly, move the hall to the free list when the activity stops. Initially start with zero halls. If there are no halls in the free list create a new hall.

The above algorithm uses the fewest number of halls possible: Assume the algorithm used m halls. Consider some activity a that was the first scheduled activity in lecture hall m. i was put in the mth hall because all of the $m-1$ halls were busy, that is, at the time a is scheduled there are m activities occuring simultaneously. Any algorithm must therefore use at least m halls, and the algorithm is thus optimal.

The algorithm can be implemented by sorting the activities. At each start or finish time we can schedule the activities and move the halls between the lists in constant time. The total time is thus dominated by sorting and is therefore $\Theta(n \lg n)$.

16.1-4

Show that selecting the activity with the least duration or with minimun overlap or earliest starting time does not yield an optimal solution for the activity-selection problem. Consider figure 4:

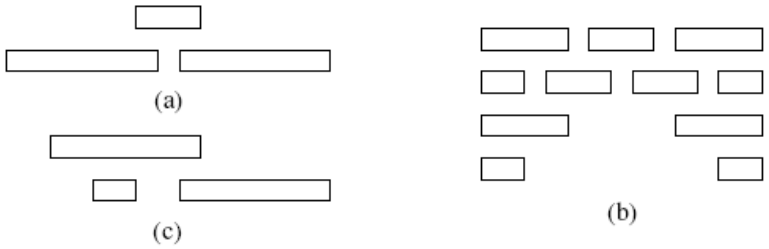


Figure 4: Three examples with other greedy strategies that go wrong

Selecting the activity with the least duration from example a will result in selecting the topmost activity and none other. Clearly, this is worse than the optimal solution obtained by selecting the two activities in the second row.

The activity with the minimun overlap in example b is the middle activity in the top row. However, selecting this activity eliminates the possibility of selecting the optimal solution depicted in the second row.

Selecting the activity with the earliest starting time in example c will yield only the one activity in the top row.

16.2-1

背包问题：

选择物品 i 装入背包时，可以选择物品 i 的一部分，而不一定要全部装入背包，1≤i≤n。此问题可以描述为：给定 C>0, wi>0,vi>0, 1≤i≤n,要求找出一个 n 元 0-1 向量(x1,x2,…xn),xi 0≤xi ≤1， 1≤i≤n,使得∑xi ≤C，而且∑vi xi 达到最大。

贪心选择性质：

若 vi / wi ≥vi+1 / wi+1 ，1≤i≤n-1，则存在背包问题的一个最优解(x1,x2,…xn)，使得 x1 =min {W/ w1 ,1}。

分三种情况证明背包问题的这个贪心选择性质

（1）∑wi ≤W ，此时唯一的最优解为（x1,x2,…xn)。

（2）∑wi >W ，且 vi / wi =v1 / w1 ，2≤i≤n ，则它的每一个解跟最优解相同

（3）∑wi >W ，S=（x1,x2,…xn)，S 包含于{,x2,…xn}使得 x1∈S。

事实上，设 S 包含于{,x2,…xn } 是背包问题的一个最优解，且 x1 不属于 S。对任意 i∈S，取 Si =S∪{ x1}-{ xi }，则 Si 满足贪心选择性质的最优解。

16.2-2

The 0/1 knapsack problem exibits the optimal substructure given in the book: Let i be the highest numbered item among $1, \ldots, n$ items in an optimal solution S for W with value $v(S)$. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ with value $v(S') = v(S) - v_i$.

We can express this in the following recursion. Let $c[i, w]$ denote the value of the solution for items $1, \ldots, i$ and maximum weight w.

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geqslant w_i \end{cases}$$

Notice that the last case determines whether or not the ith element should be included in an optimal solution. We can use this recursion to create a straight forward dynamic programming algorithm:

Algorithm 10 DYNAMIC-0-1-KNAPSACK (v, w, n, W)

Input: Two sequences $v = \langle v_1, \ldots, v_n \rangle$ and $w = \langle w_1, \ldots, w_n \rangle$ the number of items n and the maximum weight W.
Output: The optimal value of the knapsack.

```
for w ← 0 to W do
  c[0, w] ← 0
end for
for i ← 1 to n do
  c[i, 0] ← 0
  for w ← 1 to W do
    if w_i ⩽ w then
      if v_i + c[i − 1, w − w_i] > c[i − 1, w] then
        c[i, w] ← v_i + c[i − 1, w − w_i]
      else
        c[i, w] ← c[i − 1, w]
      end if
    else
      c[i, w] ← c[i − 1, w]
    end if
  end for
end for
return c[n, W]
```

For the analysis notice that there are $(n + 1) \cdot (W + 1) = \Theta(nW)$ entries in the table c each taking $\Theta(1)$ to fill out. The total running time is thus $\Theta(nW)$.

16.2-3

此问题的形式化描述是，给定 $C>0$, $w_i>0$, $v_i>0$, $1 \leqslant i \leqslant n$, 且分别对 w_i, v_i 进行排序后有 $w_i \leqslant w_{i+1}$, $v_i \geqslant v_{i+1}$, $1 \leqslant i \leqslant n-1$, 要求找一个 n 元向量 $(x_1, x_2, \cdots, x_n)$, $x_i \in \{0, 1\}$, $1 \leqslant i \leqslant n$, 使得 $\sum_{i=1}^{n} w_i x_i \leqslant C$, 且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。我们用贪心算法求解这个问题。

此问题采用以重量最轻者先装的贪心选择策略，具体算法如下：

```
int Partion( int* w, int l, int r)
{   int  i, j;
    i = l; j = r;
    int temp = w[ i] ;
    do{
       while( ( w[ j]  > = temp) &&( i < j) )    j--;
       if( i < j)    w[ i++ ]  = w[ j ] ;
       while( ( w[ i]  < = temp) &&( i < j) )    i++;
       if( i < j)    w[ j - - ]  = w[ i] ;
    } while( i!   = j) ;
    w[ i]  = temp;
    return i;
}
void QuickSort( int* w, int l, int r)
{   int i;
    if( l < r) {
       i = Partion( w, l, r) ;
       QuickSort( w, l, i - 1) ;
       QuickSort( w, i + 1, r) ;
       }
}
```

```
void Loading ( int* x, int* w, int c, int n)
{   int* t = new int [ n + 1 ] ;
   QuickSort( w, t, n) ;
   for( int i = 1; i < = n; i ++ )
      x[ i]  = 0;
   for( int i = 1; i < = n&&w[ t[ i] ]  < = c; i++)
{   x[ t[ i] ]  = 1;


       c -  = w[ t[ i] ] ;
   }
}
```

算法的时间复杂度为 $O(n\log_2 n)$ 。

16.2-4

The optimal strategy is the obvious greedy one. Starting will a full tank of gas, Professor Midas should go to the farthest gas station he can get to within n miles of Newark. Fill up there. Then go to the farthest gas station he can get to within n miles of where he filled up, and fill up there, and so on.

Looked at another way, at each gas station, Professor Midas should check whether he can make it to the next gas station without stopping at this one. If he can, skip this one. If he cannot, then fill up. Professor Midas doesn't need to know how much gas he has or how far the next station is to implement this approach, since at each fillup, he can determine which is the next station at which he'll need to stop.

This problem has optimal substructure. Suppose there are m possible gas stations. Consider an optimal solution with s stations and whose first stop is at the kth gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m - k$ stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $s - 1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

This problem also has the greedy-choice property. Suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the kth station as its first stop. No station beyond the kth works as a first stop, since Professor Midas runs out of gas first. If a solution chooses a station $j < k$ as its first stop, then Professor Midas could choose the kth station instead, having at least as much gas when he leaves the kth station as if he'd chosen the jth station. Therefore, he would get at least as far without filling up again if he had chosen the kth station.

If there are m gas stations on the map, Midas needs to inspect each one just once. The running time is $O(m)$.

16.2-5

Describe an algorithm to find the smallest unit-length set, that contains all of the points $\{x_1, \ldots x_n\}$ on the real line. Consider the following very simple algorithm: Sort the points obtaining a new array $\{y_1, \ldots y_n\}$. The first interval is given by $[y_1, y_1 + 1]$. If y_i is the leftmost point not contained in any existing interval the next interval is $[y_i, y_i + 1]$ and so on.

This greedy algorithm does the job since the rightmost element of the set must be contained in an interval and we can do no better than the interval $[y_1, y_1 + 1]$. Additionally, any subproblem to the optimal solution must be optimal. This is easily seen by considering the problem for the points greater than $y_1 + 1$ and arguing inductively.

16.2-6

Use a linear-time median algorithm to calculate the median m of the v_i/w_i ratios. Next, partition the items into three sets: $G = \{i : v_i/w_i > m\}$, $E = \{i : v_i/w_i = m\}$, and $L = \{i : v_i/w_i < m\}$; this step takes linear time. Compute $W_G = \sum_{i \in G} w_i$ and $W_E = \sum_{i \in E} w_i$, the total weight of the items in sets G and E, respectively.

- If $W_G > W$, then do not yet take any items in set G, and instead recurse on the set of items G and knapsack capacity W.
- Otherwise ($W_G \leq W$), take all items in set G, and take as much of the items in set E as will fit in the remaining capacity $W - W_G$.
- If $W_G + W_E \geq W$ (i.e., there is no capacity left after taking all the items in set G and all the items in set E that fit in the remaining capacity $W - W_G$), then we are done.
- Otherwise ($W_G + W_E < W$), then after taking all the items in sets G and E, recurse on the set of items L and knapsack capacity $W - W_G - W_E$.

To analyze this algorithm, note that each recursive call takes linear time, exclusive of the time for a recursive call that it may make. When there is a recursive call, there is just one, and it's for a problem of at most half the size. Thus, the running time is given by the recurrence $T(n) \leq T(n/2) + \Theta(n)$, whose solution is $T(n) = O(n)$.

16.2-7

Sort A and B into monotonically decreasing order.

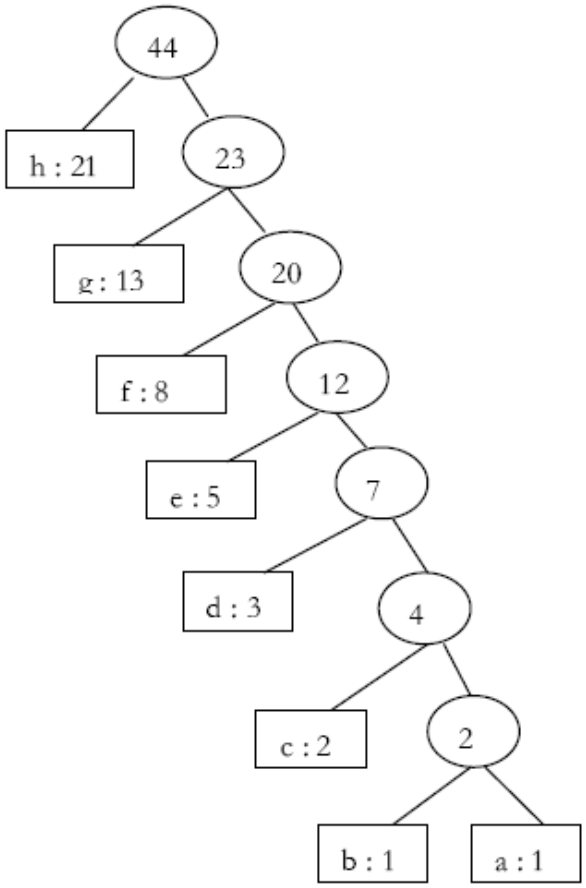
Here's a proof that this method yields an optimal solution. Consider any indices i and j such that $i < j$, and consider the terms ${a_i}^{b_i}$ and ${a_j}^{b_j}$. We want to show that it is no worse to include these terms in the payoff than to include ${a_i}^{b_j}$ and ${a_j}^{b_i}$, i.e., that ${a_i}^{b_i}{a_j}^{b_j} \geq {a_i}^{b_j}{a_j}^{b_i}$. Since A and B are sorted into monotonically decreasing order and $i < j$, we have $a_i \geq a_j$ and $b_i \geq b_j$. Since a_i and a_j are positive and $b_i - b_j$ is nonnegative, we have ${a_i}^{b_i - b_j} \geq {a_j}^{b_i - b_j}$. Multiplying both sides by ${a_i}^{b_j}{a_j}^{b_j}$ yields ${a_i}^{b_i}{a_j}^{b_j} \geq {a_i}^{b_j}{a_j}^{b_i}$.

Since the order of multiplication doesn't matter, sorting A and B into monotonically increasing order works as well.

16.3-1

要达到了最优前缀编码，必须考虑到前缀的每一种情况，故要是满二叉树。

16.3-2



a:1111111
b:1111110
c:111110
d:11110
e:1110
f:110
g:10
h:0

下面可以对其可推广性进行证明：
我们只需要证明：对任意的 n≥1,总有下列式子成立：

(1) $f(n+1) \leq \sum_{i=1}^{n} f(i) \leq f(n+2)$

(2) $\sum_{i=1}^{n} f(i) = f(n+2) - 1$

我在这里只证明第一个不等式，第二个同理可以证明。

证明：用数学归纳法证明。对 n=1 时，显然 f(2)≤f(1) ≤f(3)

假设，当 n=k(k>1)时，(1)式成立，即

$$f(k+1) \le \sum_{i=1}^{k} f(i) \le f(k+2)$$ 成立，那么当 n = k + 1 时

$$f(k+2) = f(k) + f(k+1) \le f(k+1) + f(k+1)$$

$$\le \sum_{i=1}^{k} f(i) + f(k+1) \le f(k+2) + f(k+1) = f(k+3)$$

由此，便证明了

$$f(k+2) \le \sum_{i=1}^{k+1} f(i) \le f(k+3)$$

所以，得证！

16.3-3

16.3-4

频度排列为 f(1)>=f(2)>=...f(i)>=...>=f(j)>=...>=f(n)，

假设我们设计出了最优编码方案，各字母的编码长度分别为 t(1)，t(2),...,t(n)，

此时的全文编码长度为 T=f(1)*t(1)+...+f(i)*t(i)+...+f(j)*t(j)+...+f(n)*t(n)

假设对于某个 i <j，此时有 t(i)>t(j)，那么我们将第 i 个字母和第 j 个字母的编码方式互换之后，由于 f(i)*t(j)+f(j)*t(i) <f(i)*t(i)+f(j)*t(j) ，所以互换之后的整体编码长度比原来更短，这与原来的方案“最佳”相矛盾。

所以上述假设不可能成立，命题得证

16.3-5

用2n-1位表示树的结构，内部结点用1表示，叶子结点用0表示。用nlog(n)位表示字母序列，每个字母的二进制编码长度为log(n)，总共需要nlog(n)位。

16.3-6

那就推广到树的结点有三个孩子结点，证明过程同引理 16.3 的证明。

16.3-7

此时生成的 Huffman 树是一颗满二叉树，跟固定长度编码一致。

16.3-8

Show that we cannot expect to compress a file of randomly chosen bits. Notice that the number of possible source files S using n bits and compressed files E using n bits is $2^{n+1} - 1$. Since any compression algorithm must assign each element $s \in S$ to a distinct element $e \in E$ the algorithm cannot hope to actually compress the source file.

第 24 章

24.1-1

同源顶点 s 的运行过程，见图 24-4

24.1-2

由于 Bellman-Ford 算法是运用松弛技术 ，对每个顶点$v \in V$，逐步减小从源 s 到 v 的最短路径的权的估计值 d[v]直至其达到实际最短路径的权，所以对每一个顶点$v \in V$，若从 s 到 v 存在一条通路，当且仅当该算法终止时，有$d[v] < \infty$

24.1-3

The proof of the convergence property shows that for every vertex v, the shortest-path estimate $d[v]$ has attained its final value after *length* (any shortest-weight path to v) iterations of BELLMAN-FORD. Thus after m passes, BELLMAN-FORD can terminate. We don't know m in advance, so we can't make the algorithm loop exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m+1$ iterations (i.e., after one iteration that makes no changes).

```
BELLMAN-FORD-(M+1)(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
changes ← TRUE
while changes = TRUE
    do changes ← FALSE
       for each edge (u, v) ∈ E[G]
           do RELAX-M(u, v, w)

RELAX-M(u, v, w)
if d[v] > d[u] + w(u, v)
  then d[v] ← d[u] + w(u, v)
       π[v] ← u
       changes ← TRUE
```

The test for a negative-weight cycle (based on there being a d that would change if another relaxation step was done) has been removed above, because this version of the algorithm will never get out of the **while** loop unless all d's stop changing.

24.1-4

24.1-5*

24.1-6

修改 Bellman-Ford 算法，先找到负环上的一个节点，再依次找到负环上的一个节点，再依次找到负环上的其他节点。

24.2-1

见图 24-5

24.2-2

最后一次不影响结果

24.2-3

We'll give two ways to transform a PERT chart $G = (V, E)$ with weights on vertices to a PERT chart $G' = (V', E')$ with weights on edges. In each way, we'll have that $|V'| \leq 2|V|$ and $|E'| \leq |V| + |E|$. We can then run on G' the same algorithm to find a longest path through a dag as is given in Section 24.2 of the text.

In the first way, we transform each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enter v will enter v' in E', and all edges in E that leave v will leave v'' in E'. In other words, if $(u, v) \in E$, then $(u'', v') \in E'$. All such edges have weight 0. We also put edges (v', v'') into E' for all vertices $v \in V$, and these edges are given the weight of the corresponding vertex v in G. Thus, $|V'| = 2|V|$, $|E'| = |V| + |E|$, and the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

In the second way, we leave vertices in V alone, but we add one new source vertex s to V', so that $V' = V \cup \{s\}$. All edges of E are in E', and E' also includes an edge (s, v) for every vertex $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new source s. The weight of edge $(u, v) \in E'$ is the weight of vertex v in G. In other words, the weight of each entering edge in G' is the weight of the vertex it enters in G. In effect, we have "pushed back" the weight of each vertex onto the edges that enter it. Here, $|V'| = |V| + 1$, $|E'| \leq |V| + |E|$ (since no more than $|V|$ vertices have in-degree 0 in G), and again the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

24.2-4

We count the number of directed paths in a directed acyclic graph $G = (V, E)$ as follows. First perform a topological sort of the input. Then for all $v \in V$ compute, $B(v)$ defined as follows.

$$B(v) = \begin{cases} 1 & v \text{ is last in the order} \\ 1 + \sum_{(v,w)\in E} B(w) & \text{otherwise} \end{cases}$$

$B(v)$ computes the number of directed paths beginning at v since if v is last in the order the only path starting at v is the empty one. Otherwise for each node w, $(v, w) \in E$, (v, w) concatenated with the paths from w and the empty path are the paths starting from v. We then compute the number of directed paths after v in the topological order. We denote this by $D(v)$ and we obtain the following.

$$D(v) = B(v) + \sum_{(v,w)\in E} D(w)$$

Since the nodes of G are ordered topologically $B(v)$ and $D(v)$ can be computed in linear. Thus the total running time is $O(E + V)$.

24.3-1

见图 24-6

24.3-2

Dijkstra 算法基本原理是：每次新扩展一个距离最短的点，更新与其相邻的点的距离。当所有边权都为正时，由于不会存在一个距离更短的没扩展过的点，所以这个点的距离永远不会再被改变，因而保证了算法的正确性。不过根据这个原理，用 Dijkstra 求最短路的图不能有负权边，因为扩展到负权边的时候会产生更短的距离，有可能就破坏了已经更新的点距离不会改变的性质。

24.3-3

Consider stopping Dijkstra's algorithm just before extracting the last vertex v from the priority-queue. The shortest path estimate of this vertex must the shortest path since all edges going into v must have been relaxed. Additionally, v was to be extracted last so it will have the largest shortest path of all vertices and any relaxation from v will therefore not alter shortest path estimates. Therefore the modified algorithm is correct.

24.3-4

Consider to problem of computing the most reliable channel between two vertices. Observe that this is equivalent to a shortest path problem on the graph with $w(e) = \lg(r(e))$ for all $e \in E$ which can be solved using Dijkstra's algorithm. The reliability of the most reliable path to any node v can then be found as $2^{d[v]}$.

24.3-5

24.3-6

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \to \{1, \ldots W-1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortests path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimun shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

24.3-7

Consider the problem from the above exercise. Notice that every time a node v is extracted by EXTRACT-MIN the relaxations performed on the neighbour of v gives shortests path estimates in the range $\{d[v], \ldots d[v] + W - 1\}$. Hence after every EXTRACT-MIN operation only W distinct shortest path estimates are in the priority queue at any time.

Converting the array implementation to a binary heap of the previous exercise must give a running time of $O(V + E) \lg W)$ since both the EXTRACT-MIN operation and the DECREASE-KEY operation take $O(\lg W)$ time. If we use a fibonnacci heap the running time can be further improved to $O(V \lg W + E)$.

24.3-8

这种情况下不会破坏已经更新的点的距离。

24.4****

24.5****

第 25 章

25.1-1

见图 25-1

25.1-2

为了保证递归定义式 25.2 的正确性

25.1-3

The matrix $L^{(0)}$ corresponds to the identity matrix

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

of regular matrix multiplication. Substitute 0 (the identity for $+$) for ∞ (the identity for min), and 1 (the identity for $\cdot$) for 0 (the identity for $+$).

25.1-4

25.1-5

The all-pairs shortest-paths algorithm in Section 25.1 computes

$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1}$

where $l_{ij}^{(n-1)} = \delta(i, j)$ and $L^{(0)}$ is the identity matrix. That is, the entry in the ith row and jth column of the matrix "product" is the shortest-path distance from vertex i to vertex j, and row i of the product is the solution to the single-source shortest-paths problem for vertex i.

Notice that in a matrix "product" $C = A \cdot B$, the ith row of C is the ith row of A "multiplied" by B. Since all we want is the ith row of C, we never need more than the ith row of A.

Thus the solution to the single-source shortest-paths from vertex i is $L_i^{(0)} \cdot W^{n-1}$, where $L_i^{(0)}$ is the ith row of $L^{(0)}$—a vector whose ith entry is 0 and whose other entries are ∞.

Doing the above "multiplications" starting from the left is essentially the same as the BELLMAN-FORD algorithm. The vector corresponds to the d values in BELLMAN-FORD—the shortest-path estimates from the source to each vertex.

- The vector is initially 0 for the source and ∞ for all other vertices, the same as the values set up for d by INITIALIZE-SINGLE-SOURCE.

25.1-6

25.1-7

25.1-8

By overriding previous matrices we can reduce the space used by FASTER-ALL-PAIRS-SHORTEST-PATH to $\Theta(n^2)$.

25.1-9

The presence of a negative-weight cycle can be determined by looking at the diagonal of the matrix $L^{(n-1)}$ computed by an all-pairs shortest-path algorithm. If the diagonal contains any negative number there must be a negative-weight cycle.

25.1-10

As in the previous exercise when can determine the presence of a negative-weight cycle by looking for a negative number in the diagonal. If $L^{(m)}$ is the first time for which this occurs then clearly the negative-weight cycle has length m. We can either use SLOW-ALL-PAIRS-SHORTEST-PATH in the straightforward manner or perform a binary search for m using FASTER-ALL-PAIRS-SHORTEST-PATH.

25.2-1

见图 25-4

25.2-2

25.2-3

25.2-4

With the superscripts, the computation is $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$. If, having dropped the superscripts, we were to compute and store d_k or d_{kj} before

using these values to compute d_{ij}, we might be computing one of the following:

$$d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right)$$
$$d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right)$$
$$d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right)$$

In any of these scenarios, we're computing the weight of a shortest path from i to j with all intermediate vertices in $\{1, 2, \ldots, k\}$. If we use $d_{ik}^{(k)}$, rather than $d_{ik}^{(k-1)}$, in the computation, then we're using a subpath from i to k with all intermediate vertices in $\{1, 2, \ldots, k\}$. But k cannot be an *intermediate* vertex on a shortest path from i to k, since otherwise there would be a cycle on this shortest path. Thus, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. A similar argument applies to show that $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Hence, we can drop the superscripts in the computation.

25.2-5

25.2-6

Here are two ways to detect negative-weight cycles:

1. Check the main-diagonal entries of the result matrix for a negative value. There is a negative weight cycle if and only if $d_{ii}^{(n)} < 0$ for some vertex i:

   - $d_{ii}^{(n)}$ is a path weight from i to itself; so if it is negative, there is a path from i to itself (i.e., a cycle), with negative weight.
   - If there is a negative-weight cycle, consider the one with the fewest vertices.
     - If it has just one vertex, then some $w_{ii} < 0$, so d_{ii} starts out negative, and since d values are never increased, it is also negative when the algorithm terminates.
     - If it has at least two vertices, let k be the highest-numbered vertex in the cycle, and let i be some other vertex in the cycle. $d_{ik}^{(k-1)}$ and $d_{ki}^{(k-1)}$ have correct shortest-path weights, because they are not based on negative-weight cycles. (Neither $d_{ik}^{(k-1)}$ nor $d_{ki}^{(k-1)}$ can include k as an intermediate vertex, and i and k are on the negative-weight cycle with the fewest vertices.) Since $i \rightsquigarrow k \rightsquigarrow i$ is a negative-weight cycle, the sum of those two weights is negative, so $d_{ii}^{(k)}$ will be set to a negative value. Since d values are never increased, it is also negative when the algorithm terminates.

   In fact, it suffices to check whether $d_{ii}^{(n-1)} < 0$ for some vertex i. Here's why. A negative-weight cycle containing vertex i either contains vertex n or it does not. If it does not, then clearly $d_{ii}^{(n-1)} < 0$. If the negative-weight cycle contains vertex n, then consider $d_{nn}^{(n-1)}$. This value must be negative, since the cycle, starting and ending at vertex n, does not include vertex n as an intermediate vertex.
2. Alternatively, one could just run the normal FLOYD-WARSHALL algorithm one extra iteration to see if any of the d values change. If there are negative cycles, then some shortest-path cost will be cheaper. If there are no such cycles, then no d values will change because the algorithm gives the correct shortest paths.

25.2-7

25.2-8

We wish to compute the transitive closure of a directed graph $G = (V, E)$. Construct a new graph $G^* = (V, E^*)$ where E^* is initially empty. For each vertex v traverse the graph G adding edges for every node encountered in E^*. This takes $O(VE)$ time.

25.2-9

25.3-1

25.3-2

25.3-3

h(v)=0, h(u)=0, w=w+h(u)-h(v)=w

25.3-4

25.3-5

$\because$ 有0圈， $\hat{\omega}(u,v) \leftarrow \omega(u,v)+h(u)-h(v)$

$\therefore \hat{\omega}(u,v_1)+\hat{\omega}(v_1,v_2)+\cdots+\hat{\omega}(v_{n-1},v_n)+\hat{\omega}(v_n,v)+\hat{\omega}(v,u)=0$

又$\hat{\omega} \geq 0$，则对每一条边有：$\hat{\omega}(u,v)=0$

25.3-6