Solution of CLRS 3^{rd}

January 11, 2014

1 The Role of Algorithms in Computing

Exercise 1.1-1

Use insertion sort when you play card.

Exercise 1.1-2

energy conversion rate

Exercise 1.1-3

Array, fast appending element but deletion is slow

Exercise 1.1-4

The goals of both these problems are to find a shortest path from one point to another point. There are lots of difference. The most important item is, the former is P and the latter is NP-Hard.

Exercise 1.1-5

Give the best way to find some element from a sorted array. (Binary search) Give the square root of a real number. (The binary system can not describe a irrational number precisely)

Exercise 1.2-1

Find the guy who runs fastest from 100 candidates in the Olympic Game. Tournament Sort.

Exercise 1.2-2

When merge sort beats insertion sort, we have $64n \lg n > 8n^2, n \in \mathbb{N}, n > 46$

Exercise 1.2-3

$n = 15$ when $n \in \mathbb{N}$

Problem 1-1

skip the parts of day/month/year/century and the part of $n \lg n$

	1 second	1 minute	1 hour
$\lg n$	2^{10^6}	60^{10^6}	3600^{10^6}
$\sqrt{n}$	10^{12}	$3600 \cdot 10^{12}$	$(3600 \cdot 10^6)^2$
n	10^6	$60 \cdot 10^6$	$3600 \cdot 10^6$
n^2	1000	7746	60000
n^3	100	390	1532
2^n	19	24	29
$n!$	9	11	14

2 Getting Started

Exercise 2.1-1

31 <u>41</u> 59 26 41 58
31 41 <u>59</u> 26 41 58
26 31 41 <u>59</u> 41 58
26 31 41 41 <u>59</u> 58
26 31 41 41 58 <u>59</u>

Exercise 2.1-2

NON-INCREASING-SORT(A)

```
1  for j = 2 to A.length
2     key = A[j]
3     i = j − 1
4     while i > 0 and A[i] < key
5        A[i + 1] = A[i]
6        i = i − 1
7     A[i + 1] = key
```

Exercise 2.1-3

LINEAR-SEARCH(A, v)

```
1  for i = 1 to A.length
2     if A[i] = v
3        return i
4  return Nil
```

loop invariant : $A[1..i-1]$ do not contain v before ith iteration starts.
Initialization:
Trivial for the first iteration
Maintenance:
If the $i-1$th iteration found v, index $i-1$ will be returned. So as the current loop.
Termination:
It ends when $i = A.length$, one of the position in A will be returned, or it will return Nil.

Exercise 2.1-4

Input: Two binary arrays, A and B, $A.length = n$ and $B.length = n$, $f(X) = X_1X_2...X_n$, where $n = X.length$ and X is array
Output: An array C, where $f(C) = f(A) + f(B)$

SUM(A, B)

```
 1  init array C with 0
 2  Carry = 0
 3  for i = 1 to A.length
 4     n = A[i] + B[i] + Carry
 5     if n > 1
 6        C[i] = n − 1
 7        Carry = 1
 8     else
 9        C[i] = n
10        Carry = 0
11  C[n + 1] = Carry
12  return C
```

Exercise 2.2-1

$\Theta(n^3)$

Exercise 2.2-2

SELECTION-SORT(A)

```
1  for i = 1 to A.length − 1
2     Find minimum element in A[i...A.length − 1], denote it as A[j]
3     Swap A[i] and A[j]
4  return A
```

loop invariant: $A[1...i-1]$ are the first $(i-1)th$ elements in $A[n]$ before iteration i begins. Since before nth iterations, $A[1...n-1]$ are the first $(n-1)th$ elements, so $A[n]$ must be the largest element, the array has been in sorted order. Both the best-case and worst-cast are $\Theta(n^2)$

Exercise 2.2-3

average case: $n/2$, $\Theta(n)$
worst case: n, $\Theta(n)$

Exercise 2.2-4

We compute all the possible input to get the optimal answer offline. Return the pre-computed result, it will cost $\Theta(1) for online part$

Exercise 2.3-1

$3\ 41 \to 3, 41 \qquad 52\ 26 \to 26, 52 \qquad 38\ 57 \to 38, 57 \qquad 9\ 49 \to 9, 49$
$3, 41\ 26, 52 \to 3, 26, 41, 52 \qquad 38, 57\ 9, 49 \to 9, 38, 49, 57$
$3, 26, 41, 52\ 9, 38, 49, 57 \to 3, 9, 26, 38, 41, 49, 52, 57$

Exercise 2.3-2

MERGEV2(A, p, q, r)

```
 1  same as line 1-7 in MERGE
 2  i = 1
 3  j = 1
 4  for k = p to r
 5     same as line 13-17 in MERGE
 6     if i = n_1
 7        copy R[j...n] to A[k + 1...r]
 8        break
 9     if j = n_2
10        copy L[i...n] to A[k + 1...r]
11        break
```

Exercise 2.3-3

Basis: For $k = 2$, $T(4) = 2T(2) + 4 = 8$ and $4 \lg 4 = 8$, $k = 2$ holds
Inductive step: If $T(2^{k-1})$ holds, $T(2^{k-1}) = 2^{k-1} \lg(2^{k-1}) = (k-1) \cdot 2^{k-1}$. By recurrence, we have $T(2^k) = 2T(2^k/2) + 2^k = 2 \cdot 2^{k-1}(k-1) + 2^k = k \cdot 2^k$, $T(2^k)$ also holds.

Exercise 2.3-4

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

Exercise 2.3-5

BinarySearch(A, i, j, v)

```
1  if i < j
2     return Nil
3  mid = (i + j)/2
4  if A[mid] = v
5     return mid
6  else if A[mid] < v
7     return BinarySearch(A, i, mid − 1, v)
8  else
9     return BinarySearch(A, mid + 1, j, v)
```

the worst case occurs when the function return Nil, the range of current search is always half of the last one, so the worst case is $\Theta(\lg n)$

Exercise 2.3-6

No, we can't. Even we use $\Theta(\lg n)$ time to find the position to insert, we still need $\Theta(n)$ time to move the elements after it to the correct positions.

Exercise 2.3-7

CheckSumInArray(A, sum)

```
1  MergeSort(A)
2  for i = 1 to A.length
3     j = BinarySearch(A, i + 1, A.length, sum − A[i])
4     if j ≠ Nil
5        return True
6  return False
```

Another solution:

CheckSumInArray(A, sum)

```
1  S = MergeSort(A)
2  S' = MergeSort(sum − A)
3  if sum%2 = 0
4     Check if there exists two elements which value is sum/2
5  return S ∩ S' (This step can be done by Merge)
```

Problem 2-1

a. To sort an array with length k costs ck^2 time, so sort n/k arrays will cost $n/k \cdot ck^2 = \Theta(nk)$ time.
b. To merge n/k lists, the first merge action needs $n/k-1$ comparisons. And it will create a tree with height $\lg(n/k)$. So the rest merge needs $\lg(n/k)-1$ comparison each time. The total cost is $n/k-1+(n-1)(\lg(n/k)-1) = \Theta(n\lg(n/k))$, the solution is based on loser tree. Another solution is using heap.
c. $k = \Theta(\lg n)$
d. Let a be the constant of insertion-sort, b be the constant of merge sort, we should choose k to make the inequality $ank + b\lg(n/k) \leq bn\lg n$ holds.

Problem 2-2

a. We need to prove the set $\{x \mid x \in A'\} = \{x \mid x \in A\}$, which means the elements after sorting are the same as before.
b. *loop invariant*: The minimum element in array $A[j-1...A.length]$ is in the position $j-1$
(1)Initialization:
$A[A.length]$ is the min element of $A[A.length...A.length]$ before the first comparison.
(2)Maintenance:
Before the $j = k+1$ iteration begins, the min element of $A[k...A.length]$ is $A[k]$, after line 3 and 4, the min element will be switch to the position $k-1$.
(3)Termination:
After the final iteration ends, $A[i]$ is the min element of $A[i...A.length]$
c. *loop invariant*: The array $A[1...i]$ contains the first ith elements in the original array.
(1)Initialization:
It's trivially that $A[1]$ is the minimum element in the array.
(2)Maintenance:
Before ith iteration starts, $A[1...i-1]$ contains the first $i-1$th elements, and the ith iteration will switch ith element to its correct position.
(3)Termination:
After the final iteration, $A[1...A.length]$ is in sorted order.
d. worst case running time is $n(n-1)/2 = \Theta(n)$. Same as insertion sort.

Problem 2-3

a. $\Theta(n)$
b.

PolynomialSum(A, x)

```
1  sum = 0
2  for k = 0 to n
3     s = 1
4     for i = 1 to k
5        s = s · x
6     sum = sum + s
7  return sum
```

$\Theta(n^2)$, much more slower than $Horner's$
c.
(1)Initialization: Before the first iteration $i = n$ start, $y = 0$, it's trivial.
(2)Maintenance: The equality holds before $i - 1$th iteration, such that we have $y = \sum_{k=0}^{n-i} a_{k+i}x^k$. And after $i - 1$th iteration, we compute $y = \sum_{k=0}^{n-i} a_{k+i}x^k \cdot x + a_i = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$
(3)Termination: Before the iteration $i = -1$, which means the algorithm terminates, the equality holds.
d. Followed by **c.**

Problem 2-4

a. 2,1 3,1 8,6 8,1 6,1
b. $n, n-1...1$, $n(n-1)/2$ pairs.
c. Two variables are equal. Since the insertion sort needs to switch each inversion pairs.
d. In Merge process, Add $invertPairs = 0$ to the first line. And add $invertPairs = invertPairs + n_1 - i + 1$ between line 16-17.

3 Growth of Functions

Exercise 3.1-1

The following inequality always holds, $\frac{f(n)+g(n)}{2} \leq \max(f(n), g(n)) \leq f(n) + g(n)$, which proves $\max(f(n), g(n)) = \Theta(f(n) + g(n))$

Exercise 3.1-2

Since $(n+a)^b = \sum_{k=0}^{b} C_b^k n^{b-k} a^k = n^b + \sum_{k=1}^{b} C_b^k n^{b-k} a^k$, and $n^b \leq n^b + \sum_{k=1}^{b} C_b^k n^{b-k} a^k \leq (1+b)n^b$, a and b are constants. If we set $c_1 = 1$ and $c_2 = (1+b)$, we have $(n+a)^b = \Omega(n^b)$ and $(n+a)^b = O(n^b)$.

Exercise 3.1-3

At least is meaningless.

Exercise 3.1-4

Yes. No.

Exercise 3.1-5

Three inequality. (1)$c_1 g(n) \leq f(n) \leq c_2 g(n)$ and (2)$c_1 g(n) \leq f(n)$ and (3)$f(n) \leq c_2 g(n)$, (1)holds can prove(2) and (3). (2) and (3) hold at the same time can prove(1).

Exercise 3.1-6

Let $f(n)$ be the running time. worst-case running time equal to $O(g(n))$ means $f(n) \leq c_1 g(n)$, best-case running time equal to $\Omega(g(n))$ means $c_2 g(n) \leq f(n)$, then followed by 3.1-5.

Exercise 3.1-7

$f(n) = o(g(n))$ means for any constant c, $f(n) < cg(n)$, $f(n) = \omega(g(n))$ means for any constant c, $f(n) > cg(n)$. Obviously, there is no $f(n)$ such that satisfies two conditions at the same time.

Exercise 3.1-8

$f(n,m) = \Theta(g(n,m))$, there exist positive c_1, c_2, n_0, m_0 such that $c_1 g(n,m) \leq f(n,m) \leq c_2 g(n,m)$ for all $n \geq n_0$ and $m \geq m_0$.$f(n,m) = \Omega(g(n,m))$, there exist positive c, n_0, m_0 such that $cg(n,m) \leq f(n,m)$ for all $n \geq n_0$ and $m \geq m_0$.

Exercise 3.2-1

If $n_1 < n_2$, then $f(n_1) < f(n_2)$ and $g(n_1) < g(n_2)$ hold. Obviously, $f(n_1) + g(n_1) < f(n_2) + g(n_2)$ and $f(g(n_1)) < f(g(n_2))$ hold. If $f(n) \geq 0$ and $g(n) \geq 0$, then $f(n_1) \cdot g(n_1) < f(n_2) \cdot g(n_2)$.

Exercise 3.2-2

$a^{\log_b c} = a^{\log_a c / \log_a b} = (a^{\log_a c})^{1/\log_a b} = c^{1/\log_a b} = c^{\log_b a}$

Exercise 3.2-3

$\lg(n!) = \lg 1 + \lg 2 + ... + \lg n \leq cn \lg n = O(n \lg n)$. If set $c < 1$, then $(1 - c) \lg n \geq \lg e$ holds for some positive n_0 when $n \geq n_0$. So $n \lg(n/e) \geq cnlgn$, and $\lg \sqrt{2\pi n} + \lg(n/e)^n + \lg(1+\Theta(1/n)) \geq cn \lg n$, by Stirling's approximation, $\lg(n!) = \Omega(n \lg n)$, we prove that $\lg(n!) = \Theta(n \lg n)$. Also, by Stirling's approximation, we can prove $n! = \omega(2^n)$ and $n! = o(n^n)$.

Exercise 3.2-4

The former is but the latter is not. By Stirling's approximation, we have $\sqrt{2\pi f(n)}(f(n)/e)^{f(n)}(1 + \Theta(1/\lg n))$, we only need to prove $f(n) = \lg n^{\lg n}$ is polynomial bound and $g(n) = \lg \lg n^{\lg \lg n}$ is not. $\lim_{n\to\infty} \frac{\lg \lg n}{k} = \infty \Leftrightarrow \lim_{n\to\infty} \frac{\lg f(n)}{k \lg n} = \infty \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{n^k} = \infty$. Which implies $f(n)$ is not polynomial bound. And using a similar process can prove $g(n)$ is polynomial bound.

Exercise 3.2-5

$\lg^* \lg m$ is larger. Suppose we have $m = 2^{2^{2\cdots}}$, where the number of 2 is k. $\lg^* \lg m = \lg k$ and $\lg \lg^* m = k - 1$, we can also prove the integer with number of 2 between k and $k+1$ are true. which mean $\lg^* \lg m = o(lgk)$ and $\lg \lg^* m = \omega(k)$. So the former expression is larger.

Exercise 3.2-6

$\phi^2 = ((1 - \sqrt{5})/2)^2 = (1 - 2\sqrt{5} + 5)/4 = (1 - \sqrt{5})/2 + 1$, its conjugate is the same.

Exercise 3.2-7

$F_1 = 1$ is correct for $i = 1$.By strong induction, the equality of F_i holds for any $i \leq j-1$ and $F_j = F_{j-1}+F_{j-2} = (\phi^{j-1}-\phi'^{j-1})/\sqrt{5}+(\phi^{j-2}-\phi'^{j-2})/\sqrt{5} = (\phi^{j-2}(\phi+1) - \phi'^{j-2}(\phi'+1))/\sqrt{5}$, by the equality in Exercise 3.2-6, we have $F_j = (\phi^j + \phi'^j)/\sqrt{5}$

Exercise 3.2-8

Suppose $k = \Theta(n/lnn)$, then $(1)c_1 n/lnn \leq k \leq c_2 n/lnn$ and $(2)ln(c_1 n/lnn) \leq lnk \leq ln(c_2 n/lnn)$, $(1)\cdot(2)$ will prove that there exist c_1 and c_2 such that $c_1 n \leq klnk \leq c_2 n$.

Problem 3-1

a. Let $c = d + 1$, we will get $p(n) \leq cn^k$
b. Let $c = a_d$, we will get $p(n) \geq cn^k$
c. Let $c_1 = a_d$ and $c_2 = d + 1$, we will get $c_1 n^k \leq p(n) \leq c_2 n^k$
d. and **e.** Same as **a.** and **b.**

Problem 3-2

No	Yes	No	No	No
No	Yes	No	No	No
No	No	No	No	No
No	No	No	Yes	No
Yes	No	Yes	No	Yes
Yes	No	Yes	No	Yes

Problem 3-3

a. $2^{2^{n+1}} = \Theta(2^{2^n}) = \Omega((n+1)!) = \Omega(n!) = \Omega(n2^n) = \Omega(2^n) = \Omega(e^n) = \Omega((3/2)^n) = \Omega(n^{\lg\lg n}) = \Omega((\lg n)!) = \Omega(n^3) = \Omega(4^{\lg n}) = \Theta(n^2) = \Omega(n \lg n) = \Theta(\lg n!) = \Omega(n) = \Theta(2^{\lg n}) = \Omega(\sqrt{2}^{\lg n} = \Omega(2^{\sqrt{2\lg n}}) = \Omega(\lg^2 n) = \Omega(\ln n) = \Omega(\sqrt{\lg n}) = \Omega(\ln\ln n) = \Omega(2^{\lg^* n}) = \Omega(\lg^* n) = \Omega(\lg^* \lg n) = \Omega(\lg \lg^* n) = \Omega(n^{1/\lg n}) = \Theta(1)$

b. $f(n) = \begin{cases} g_1(n) & \text{if } n\%30 = 1 \\ g_2(n) & \text{if } n\%30 = 2 \\ & \quad ... \\ g_{30}(n) & \text{if } n\%30 = 0 \end{cases}$

Problem 3-4

a. $g(n) = n^{\sin n}$,$f(n) = n$ is a counter-example.
b. $f(n) = n$,$g(n) = n^2$ is a counter-example.
c. $f(n) = O(g(n)) \implies f(n) \leq cg(n) \implies lg(f(n)) \leq lg(cg(n)) = lgc + lg(g(n)) \implies lg(f(n)) = O(lg(g(n)))$
d. $f(n) = n$ and $g(n) = 2n$ is a counter-example.
e. $f(n) = \frac{1}{n}$ is a counter-example.
f. $f(n) = O(g(n)) \implies f(n) \leq cg(n) \implies g(n) \geq (1/c)f(n) \implies g(n) = \Omega(f(n))$
g. $f(n) = a^n$, where a is the constant
h. let $f(n) = n$, $n + n^2 \neq \Theta(n)$

Problem 3-5

a. We need to prove if $f(n) = O(g(n))$ doesn't hold, we must have $f(n) = \overset{\infty}{\Omega}(g(n))$. If $f(n) \neq O(g(n))$, there is not any constant $c > 0$, which satisfies $f(n) \leq cg(n)$ for all $n > n_0$. So for any constant $c > 0$ and $n_0 > 0$, we can always find an $n > n_0$ and make $f(n) \geq cg(n) \geq 0$(both $f(n)$ and $g(n)$ are non-negative). For example, we find $n_1 > n_0$ which let the inequality founds, then we can find $n_2 > n_1$ and make this process infinitely. Which means $f(n) = \overset{\infty}{\Omega}(g(n))$ holds. An counter-example of using Ω, $f(n) = n^{\sin n}$ and $g(n) = n^{\cos n}$
b. The advantage is giving us a bound easily. The disadvantage is, the bound is not strict
c. Nothing happens.
d. $\widetilde{\Omega}(g(n)) = \{f(n)$:there exist positive constants c, k and n_0 such that $cg(n)lg^k(n) \leq f(n)$ for all $n \geq n_0\}$ $\widetilde{\Theta}(g(n)) = \{f(n)$:there exist positive constants c_1, c_2, k and n_0 such that $0 \leq c_1g(n)lg^k(n) \leq f(n) \leq c_2g(n)lg^k(n)$ for all $n \geq n_0\}$.

Problem 3-6

a. n
b. $\lg^* n$
c. $\lceil \lg n \rceil$
d. $\lfloor \lg n \rfloor$
e. $\lg \lg n$
f. ∞
g. $\frac{\lg \lg n}{\lg 3}$
h. 2

4 Probabilistic Analysis and Randomized Algorithms

Exercise 5.1-1

If a candidate a is better than b, than any candidate x better than a is also better than b. And any candidate worse than b is worse than a. Such than any pair of the candidate set can be compared, which implies the rank of the candidates are in total order.

Exercise 5.1-2

Use the binomial representation x of the number $b - a + 1$, suppose the bit length of x is L_x, we repeat $L_x = \lg(b - a + 1)$ iterations of RANDOM$(0, 1)$. And if we get a binomial number n, which is larger than $b - a + 1$, we throw the result and do the iterations again. The expected running time is $\frac{1}{Pr\{n \leq b-a+1\}} \cdot O(\lg(b - a + 1))$. The $Pr\{n \leq b - a + 1\}$ is at most $1 - (1/2)^{\lg(b-a+1)} = (b - a)/(b - a + 1)$, such that the expected running time is $O((b - a + 1)/(b - a) \cdot \lg(b - a + 1))$

Exercise 5.1-3

We run BIASED-RANDOM twice. And set $I = \{$output sequence is 0 and 1, output sequence is 1 and 0$\}$, the appearance probability of these two events are both $1/2$ in the total set I. If we get the output sequence which is not in I, we rerun the procedure until we get the event in I. Assume the random

variable X_i indicates the ith round result of BIASED-RANDOM. Then the expected running time is $1/Pr\{X_1X_2 \in I\} = \frac{1}{2p(1-p)}$

Exercise 5.2-1

If the best candidate is the first, we will hire exact one time. Its probability is $1/n$. If the candidates appear from worst to best, we will hire exactly n time, the probability is $1/n!$

Exercise 5.2-2

Since candidate 1 will be always hired. We denote X_i as the indicator that only the candidate 1 and candidate i are hired(where $i > 1$). Which means, candidate 1 is the best candidate in $[1, i-1]$ and candidate i is the best in $[1, n]$. So $Pr\{X_i\} = \frac{1}{i-1} \cdot \frac{1}{n}$. Since X_i is independent for different i, we have the total probability $Pr\{x\} = \sum_{i=2}^{n} Pr\{X_i\} = \sum_{i=2}^{n} \frac{1}{i-1} \cdot \frac{1}{n} = \frac{1}{n} \cdot \sum_{i=2}^{n} \frac{1}{i-1} = \frac{1}{n} \cdot H_{n-1}$

Exercise 5.2-3

Denote X_{ij} as the dice i has sum j. We have the expected total sum $E[X] = \sum_{i=1}^{n} \sum_{j=1}^{6} E[X_{ij}] = \sum_{i=1}^{n} \sum_{j=1}^{6} (j/6) = 21n/6$

Exercise 5.2-4

Denote X_i as the event that people i get his/her hat. $E[X_i] = 1/n$, and X_i is independent for different i, we have $\sum_{i=1}^{n} E[X_i] = 1$

Exercise 5.2-5

Denote X_{ij} as the event that $i < j$ and $A[i] > A[j]$, since A form a uniform random permutation, $E[X_{ij}] = Pr\{X_{ij}\} = 1/2$, then we have the expected number of inversions $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1/2) = n(n-1)/4$

Exercise 5.3-1

RANDOMIZED-IN-PLACE'(A)

```
1  n = A.length
2  swap A[1] with A[RANDOM(i, n)]
3  for i = 2 to n
4     swap A[2] with A[RANDOM(i, n)]
```

In **Initialization**, before the first loop iteration, $A[1]$ contains 1-permutation with probability $1/n$.

Exercise 5.3-2

No. For input sequence $< 1, 2, 3 >$, PERMUTE-WITHOUT-IDENTITY can not produce the sequence $< 1, 3, 2 >$

Exercise 5.3-3

No. For input sequence $< 1, 2, 3 >$, if the code produce a uniform random permutation, we have the probability $1/3! = 1/6$ to get some specified output. But for PERMUTE-WITH-ALL, it can only produce any permutation with some multiple of probability $\frac{1}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} = \frac{1}{27}$. And 1/6 can't be divided by 1/27, such that this code does not produce uniform random permutation.

Exercise 5.3-4

For input $< 1, 2, 3 >$,PERMUTE-BY-CYCLIC can't produce permutation $< 2, 1, 3 >$

Exercise 5.3-5

There are $C^n_{n^3+n-1}$ ways to put n numbers to n^3 slots.. And there are $C^n_{n^3}$ ways to put n numbers to n distinct slots in n^3 candidates. Such that the probability is $\frac{C^n_{n^3}}{C^n_{n^3+n-1}}$

Exercise 5.3-6

PERMUTE-BY-SORTING once again.

Exercise 5.3-7

loop invariant: All the element in $S[1...n-m+i]$ will be chosen to i subset in round i in RANDOM-SAMPLE with probability $\frac{i}{n-m+i}$.
Initialization: In round 1, any element in $S[1...n-m+1]$ can be chosen, each with probability $\frac{1}{n-m+1}$
Maintenance: Before ith iteration, $S[1...n-m+i-1]$ will be chosen to $i-1$ subset with probability $\frac{i-1}{n-m+i-1}$. And In ith iteration, the probability of choosing $S[n-m+i]$ is $\frac{1}{n-m+i}+\frac{i-1}{n-m+i}=\frac{i}{n-m+i}$, each of the $S[1...n-m+i-1]$ could be chosen with probability
$\frac{i-1}{n-m+i-1}+(1-\frac{i-1}{n-m+i-1})\cdot\frac{1}{n-m+i}=\frac{i}{n-m+i}$, such that all in $S[1...n-m+i]$ has probability $\frac{i}{n-m+i}$.
Termination: It ends after mth iteration, so all the elements in $S[1...n]$ will be chosen to m subset with probability $\frac{n}{m}$.

Exercise 5.4-1

Suppose there are k people in the room except you. Let X_i denote the event that the birthday of ith people is the same as yours. $Pr[x]=\sum_{i=1}^{k}Pr[X_i]=k\cdot\frac{1}{365}=\frac{k}{365}\geq\frac{1}{2}$, $k\geq\frac{365}{2}$. Let Y denote the event that there is only one person which has birthday of July 4. Let Z denote the event that all people are not July 4 birthday. $1-Pr[Y]-Pr[Z]=1-\frac{k}{365}\cdot(\frac{364}{365})^k-(\frac{364}{365})^k\geq\frac{1}{2}$, $k>611$

Exercise 5.4-2

Let X_i denote the event that when we toss ith balls, we get some bin which contains two balls. $Pr[X_i]=(\prod_{k=0}^{i-1}\frac{b-k}{b})\cdot\frac{i-1}{b}\leq e^{\frac{-i(i-1)}{2b}}\cdot\frac{i-1}{b}$. By the pigeon hole principle, we know if we toss $b+1$ balls, we necessarily get some bin with two balls. So the expected number of ball tossed is $\sum_{i=2}^{b}i\cdot Pr[X_i]=\sum_{i=2}^{b}i\cdot(\prod_{k=0}^{i-1}\frac{b-k}{b})\cdot\frac{i-1}{b}$

Exercise 5.4-3

Pairwise independence is sufficient, for equality $E[X] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}$, we needn't consider the independence for different i, j. And for any specified i, j, their birthday is pairwise independent, such that $E_{ij} = \frac{1}{n}$.

Exercise 5.4-4

Let X_{ijk} denote the event that person i, j and k have the same birthday. And $E[X_{ijk}] = \frac{1}{365^2}$, for n people, $E[X] = \sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=j+1}^{n} E[X_{ijk}] = \frac{1}{365^2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=j+1}^{n} 1 = \frac{1}{2\cdot 365^2}\cdot\sum_{i=1}^{n-2} i(i+1) \leq \frac{1}{2\cdot 365^2}\cdot(\frac{(n-2)^3}{3} - \frac{1}{3} + \frac{(n-1)(n-2)}{2})$. To make $E[X] \geq 1$, $n \geq 95$

Exercise 5.4-5

The probability of forming a k-permutation is $1 \cdot \frac{n-1}{n} \cdot ... \cdot \frac{n-k+1}{n} = \prod_{i=0}^{k-1} \frac{n-i}{n} \leq e^{-k(k-1)/2n}$. Same as the probability that k people which has the same birthday in n people.

Exercise 5.4-6

Let random indicator variable X_i denote after n tossed, the ith bin is empty. Such that $E[X_i] = Pr[X_i] = (1 - 1/n)^n \leq \frac{1}{e}$, $E[X] = \sum_{i=1}^{n} E[X_i] \leq \frac{n}{e}$. Let random indicator variable Y_i denote after n tossed, the ith bin has exactly one ball. And $E[Y_i] = Pr[Y_i] = C_n^1 \cdot (1-\frac{1}{n})^{n-1} \cdot \frac{1}{n} = (1-\frac{1}{n})^{n-1} \leq \frac{1}{e} \cdot (1-\frac{1}{n})^{-1}$, $E[Y] = \sum_{i=1}^{n} E[Y_i] \leq \frac{n}{e} \cdot (1-\frac{1}{n})^{-1} = \frac{n^2}{e(n-1)}$

5 Heaps

Exercise 6.1-1

minimum: 2^h
maximum: $2^{h+1} - 1$

Exercise 6.1-2

Assume the height is x, the inequality $2^x \leq 2^{x+1} - 1 \implies \lg(n+1) - 1 \leq x \leq \lg n$, we have $x = \lfloor \lg n \rfloor$

Exercise 6.1-3

Just because $i \geq max(i.left, i.right)$ hold.

Exercise 6.1-4

It could be any leaf node.

Exercise 6.1-5

No. An example:$1, 3, 2$

Exercise 6.1-6

No.

Exercise 6.1-7

If a node i is a leaf node, then we have $2i + 1 > n$, such that $i > (n-1)/2$, which means the minimum index of leaf node is $\lfloor n/2 \rfloor + 1$

Exercise 6.2-1

Swap 3 and 10. Swap 3 and 9.

Exercise 6.2-2

MAX-HEAPIFY(A, i)

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

Running time is the same.

Exercise 6.2-3

It returns directly.

Exercise 6.2-4

The current node is leaf such that it returns.

Exercise 6.2-5

MAX-HEAPIFY-ITERATIVE(A)

```
1  i = 1
2  l = LEFT(i)
3  r = RIGHT(i)
4  while True:
5     if l ≤ A.heap − size
6        if A[l] < A[i] min = l
7        else min = i
8     else if l > A.heap − size
9        break
10    if r ≤ A.heap − size
11       if A[r] < A[min] min = r
12    if min ≠ i
13       exchange A[i] with A[min]
14       i = min
15    else
16       break
```

Exercise 6.2-6

The n-element heap has height $\lfloor \lg n \rfloor$, if max heapify do the swap for all the elements on the path, it will cost $O(\lfloor \lg n \rfloor)$

Exercise 6.3-1

Swap 10 and 22. Swap 17 and 19. Swap 3 and 84. Swap 5 and 84, then swap 5 and 22, finally swap 5 and 10.

Exercise 6.3-2

A counter example, $5, 3, 17, 10, 84$. After we do the max heapify from 1 to $\lfloor A.length/2 \rfloor$, the array is $17, 84, 5, 10, 3$, which is not a max heap.

Exercise 6.3-3

Note that the leaf node has height 0. As we know, the leaf node of a binary tree will be less than $\lceil n/2 \rceil$. And the max node number of each level will be at most half of the next level, such that $\lceil n/2^{h+1} \rceil$ holds.

Exercise 6.4-1

After building max heap, the array is $25, 13, 20, 8, 7, 17, 2, 5, 4$ 25 will be the top, swap 4 and 25, and do the max heapify, the array is $20, 13, 17, 8, 7, 4, 2, 5, 25$. (Skip the rest.)

Exercise 6.4-2

Initialization: Before the first iteration, $A[1..n]$ is a max heap and $A[n + 1..n]$ which has no elements.
Maintenance: Before the iteration $i = k$, $A[1..k]$ is a max-heap. And $A[k + 1..n]$ contains the $n - k$ largest elements. After this iteration, $A[k]$ is the max element in the subarray $A[1..k]$, such that $A[k..n]$ contains the $n-k+1$ largest elements. And $A[1..k]$ keeps the max-heap property because we do the max heapify.
Termination: Before $i = 1$, $A[2..n]$ contains the $n - 1$ largest elements, which means $A[1..n]$ has been sorted.

Exercise 6.4-3

Increasing order: Consider the $n/2$ leaf nodes which are in increasing order. After the first $n/2$ iteration of build-max-heap procedure, there are $n/4$ elements got promotion, which means their height will be decrease one. And in these $n/4$ elements, at least half of them will keep their position until the build-max-heap finished, because the node is in increasing order. Such that we have $n/8$ elements have height $\lfloor \lg n \rfloor - 1$ or $\lfloor \lg n \rfloor - 2$. And in the max-heapify procedure, in order to let these node to the top level, it will cost at least $(n/8) \cdot (\lfloor \lg n \rfloor - 2)$ to move them, such that the running time is $O(nlgn)$
Decreasing order: After the build-max-heap procedure, the array is in the original order. And each time we exchange $A[1]$ with $A[i]$, because $A[i]$ is the

smallest element, so it must be down to the leaf of the current heap, such that the running time is $O(\lg i)$. Such that the total running time is $O(nlgn)$

Exercise 6.4-4

The worst-running time of building max heap is $\Omega(n)$. And if the worst running time of max-heapify is $\Omega(lgn)$, after $n-1$ iterations, the worst running time is $\Omega(nlgn)$

Exercise 6.4-5

See the paper of Sedgwick.

Exercise 6.5-1

extract 15. Move 1 to top. Swap 1 and 13, swap 1 and 12, swap 1 and 6. This is the first iteration.(skip the rest)

Exercise 6.5-2

Insert 10 into the right child of 8, swap 10 and 8, swap 10 and 9.

Exercise 6.5-3

HEAP-MINIMUM(A)

```
1  return A[1]
```

HEAP-EXTRACT-MIN(A)

```
1  if A.heap − size < 1
2     return "heap underflow"
3  min = A[1]
4  A[1] = A[A.heap − size]
5  MIN-HEAPIFY(A, 1)
6  return min
```

HEAP-DECREASE-KEY(A, i, key)

```
1  if key > A[i]
2     error "new key is larger than current key"
3  A[i] = key
4  while i > 1 and A[Parent(i)] > A[i]
5     exchange A[i] with A[Parent[i]]
6     i = Parent(i)
```

MIN-HEAP-INSERT(A, key)

```
1  A.heap − size = A.heap − size + 1
2  A[A.heap − size] = −∞
3  HEAP-DECREASE-KEY(A, A.heap − size, key)
```

Exercise 6.5-4

Because if not, HEAP-INCREASE-KEY will return error.

Exercise 6.5-5

Initialization: Before the first loop, $A[1...A.heap-size]$ is a max heap.Since we only increase $A[i]$ and all the other elements are original value, which means except the case $A[i]$ maybe larger than its parent. All the other elements are keep in max-heap order.
Maintenance: Before each iteration, the subtree rooted at $A[i]$ is a max heap. And after we change the order of $A[i]$ and its parent, the subtree rooted at $A[parent(i)]$ still be a max heap. Because we only swap i and its parent, and the other nodes do not changed. Such that the entire array are in max heap order.
Termination: When the loop ended, A is a max heap.

Exercise 6.5-6

HEAP-INCREASE-KEY(A, i, key)

```
1  if key < A[i]
2     error "new key is smaller than current key"
3  key = A[i]
4  while i > 1 and A[Parent(i)] < key
5     A[i] = A[Parent[i]]
6     i = Parent(i)
7  A[i] = key
```

Exercise 6.5-7

Use a min heap. And we assign the first element we want to push it to the heap with priority 1, and each time we insert new element to the heap. We increase its priority with one. Since the priority of older element is always smaller than new element, it will be extracted early, we get a FIFO queue. If each time we insert new element to the heap, we decrease its priority with one, then the priority of older element is always larger than the new, such that it will be extracted later, we get a stack.

Exercise 6.5-8

HEAP-DELETE(A, i)

```
1  exchange A[i] with A[A.heap − size]
2  A[A.heap − size] = A[A.heap − size] − 1
3  MAX-HEAPIFY(A, i)
```

Exercise 6.5-9

We build a min heap with the first elements of k sorted array. The node contains two fields, its value and the array index(from 1 to k). This step takes $O(k)$. And each time we extract the top element, we assign the top node with the element which has the same array index as last extracted node. Then we do the procedure min heapify. And this loop takes $O((n-k)\lg k)$, we get the first $n-k$ smallest element. In the end, we use the procedure HEAP-SORT(we don't use the BUILD-MAX-HEAP) to get the last k largest element, it will take $O(k \lg k)$. The total procedure costs $O(n \lg k)$

Problem 6-1

a. No,they didn't. Consider the case $1, 2, 3, 4, 5$, BUILD-MAX-HEAP will create a max heap $5, 4, 3, 1, 2$, and BUILD-MAX-HEAP' will create a max heap $5, 4, 2, 1, 3$
b. The worst case occurs we swap all the elements from the path of leaf to root when we do the MAX-HEAP-INSERT. The entire procedure will cost $\sum_{i=2}^{n} \lg i = \lg n! = \Theta(n \lg n)$

Problem 6-2

a. Same as binary heap. In BFS order, from left to right.
b. $\lfloor \log_d(nd - n) \rfloor$
c.

CHILDREN(A, i, d)

```
1  k = d(i − 1) + 2
2  return A[k...min(A.length, k + d − 1)]
```

PARENT(A, i, d)

```
1  return ⌊(i − 2)/d⌋ + 1
```

MAX-HEAPIFY(A, d)

```
1  i = 1
2  while True:
3     max = c where c is the index of max element in CHILDREN(A, i, d)
4     if CHILDREN(i, d) = Nil break
5     else if A[max] > A[i]
6        swap A[max] and A[i]
7        i = max
8     else break
```

EXTRACT-MAX(A, d)

```
1  if A.heap − size < 1
2     return "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap − size]
5  MAX-HEAPIFY(A, 1)
6  return max
```

MAX-HEAPIFY takes $O(d \cdot \lfloor \log_d(nd - n) \rfloor)$. So the EXTRACT-MAX is the same running time.

e.

INCREASE-KEY(A, i, k)

```
1  if k < A[i]
2     error "new key is smaller than current key"
3  k = A[i]
4  while i > 1 and PARENT(A, i, d) < k
5     A[i] = A[Parent[i]]
6     i = Parent(i)
7  A[i] = k
```

Running time is the height of heap at most, which is $O(\lfloor \log_d(nd - n) \rfloor)$.

d.

INSERT-KEY(A, k)

```
1  A.heapSize = A.heapSize + 1
2  A[A.heapSize] = -∞
3  INCREASE-KEY(A, A.heapSize, k)
```

Running time is the same as INCREASE-KEY

Problem 6-3

a.

2	4	9	∞
3	8	16	∞
5	14	∞	∞
12	∞	∞	∞

b. $Y[1, 1]$ is the smallest element in the first line, such that $Y[1, 2...n]$ are all ∞. At the same time, they are the smallest elements in the corresponding rows, such that $Y[1...m, 1...n]$ are all ∞. The table is empty. We can do the same argue for the case of $Y[m, n] \neq \infty$ to prove its a full table.

c.

EXTRACT-MIN-AUX(Y, i, j)

```
 1  minX = i
 2  minY = j
 3  if i + 1 ≤ m and Y[i + 1, j] < Y[i, j]
 4     minX = i + 1
 5     minY = j
 6  if j + 1 ≤ n and Y[i, j + 1] < Y[minX, minY]
 7     minX = i
 8     minY = j + 1
 9  if minX ≠ i and minY ≠ j
10     Swap Y[i, j] and Y[minX, minY]
11     EXTRACT-MIN-AUX(Y, minX, minY)
```

EXTRACT-MIN(Y)

```
1  min = Y[1, 1]
2  Y[1, 1] = ∞
3  EXTRACT-MIN-AUX(Y, 1, 1)
4  return min
```

$T(p) = T(p-1) + O(1), T(1) = O(1) \implies T(p) = O(p) = O(m+n)$

d.

POP-AUX(Y, i, j)

```
 1  maxX = i
 2  maxY = j
 3  if i − 1 ≥ 1 and Y[i − 1, j] < Y[i, j]
 4     maxX = i − 1
 5     minY = j
 6  if j − 1 ≥ 1 and Y[i, j − 1] < Y[minX, minY]
 7     minX = i
 8     minY = j − 1
 9  if maxX ≠ i and maxY ≠ j
10     Swap Y[i, j] and Y[maxX, maxY]
11     POP-AUX(Y, maxX, maxY)
```

INSERT(Y, k)

```
1  i = 1
2  j = 1
3  while Y[i, j] ≠ ∞ and i < m
4     i = i + 1
5  while Y[i, j] ≠ ∞ and j < n
6     j = j + 1
7  Y[i, j] = k
8  POP-AUX(Y, i, j)
```

e. Use EXTRACT-MIN in a n^2 size table will take $O(n)$, to extract all n^2 elements, we need at most $O(n^3)$.
f. Search from the left-bottom position.

SEARCH(Y, k)

```
1  Let (i, j) be the coordinate of element in the left-bottom position.
2  if k > Y[i, j] and j + 1 ≤ n Search Y[i, j + 1] recursively
3  else if k < Y[i, j] and i − 1 ≥ 1 Search Y[i − 1, j] recursively
4  else return the current index.
```

6 QuickSort

Exercise 7.1-1

After the PARTITION, the array $A = 9, 5, 8, 7, 4, 2, 6, 11, 21, 13, 19, 12$

Exercise 7.1-2

```
PARTITION(A, p, r)
 1  x = A[r]
 2  i = p − 1
 3  changed = False
 4  for j = p to r − 1
 5     if A[j] ≤ x
 6        i = i + 1
 7        exchange A[i] with A[j]
 8        if A[j] ≠ x
 9           changed = True
10  exchange A[i + 1] with A[r]
11  if changed is True
12     return i + 1
13  else
14     return ⌊(p + r)/2⌋
```

Exercise 7.1-3

Line 3 iterates $n - 1$ times, and each exchange cost $\Theta(1)$, which causes a $\Theta(n)$ running time.

Exercise 7.1-4

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

Exercise 7.2-1

a. Prove $T(n) = O(n^2)$. We assume $T(n) \leq c_1 n^2$. Then $T(n) = T(n-1) + \Theta(n) \leq c_1(n-1)^2 + c_2 n = c_1 n^2 + (c_2 - 2c_1)n + 1$, if we set $c_2 < 2c_1 - 1$, we can let the inequality holds. **b.** Prove $T(n) = \Omega(n^2)$. If we set $c_2 \geq 2c_1$, we can make $c_1(n-1)^2 + c_2 n = c_1 n^2 + (c_2 - 2c_1)n + 1 \geq c_1 n^2$ holds. Combine a. and b., $T(n) = \Theta(n^2)$

Exercise 7.2-2

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

Exercise 7.2-3

We pick either the largest or the smallest by sequence for each round, we have $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Exercise 7.2-4

In the almost sorted input, QUICKSORT will usually produce a split of 1 and $n-1$, which will make the total time close to $\Theta(n^2)$. But INSERTION-SORT tend to make a $\Theta(n)$ running time because we needn't move a lot of elements during each scanning round.

Exercise 7.2-5

To the best case, which corresponds to the minimum level leafs in the node, we always choose the part α. Suppose its level is k. Such that we have $\alpha^k n = 1$, where $k = -\lg n / \lg \alpha$. To the worst case, we always choose the part $1 - \alpha$ such that we have $(1-\alpha)^k n = 1$, where $k = -\lg n / \lg(1-\alpha)$

Exercise 7.2-6

In order to get a split more balanced than $1 - \alpha$ to α, we should choose the value between $[\alpha, 1/2 - \alpha + 1/2]$. Since we choose the value randomly, it is uniform distributed in the range $[0, 1]$, such that we have the probability of $1/2 - \alpha + 1/2 - \alpha = 1 - 2\alpha$ to get a better balance.

Exercise 7.3-1

Because the worst-case doesn't occur from time to time.

Exercise 7.3-2

The number of calling RANDOM is equal to the non-leaf node number of recursion tree. So the best case is $n(\lfloor \lg n \rfloor - 1) = \Theta(n \lg n)$. The worst case is $n - 1 = \Theta(n)$

Exercise 7.4-1

$\max_{0 \le q \le n-1}(T(q) + T(n-q-1)) + \Theta(n) \ge T(0) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = c(\sum_{i=1}^{n} i) = \Omega(n^2)$

Exercise 7.4-2

The minimum height of the recursion tree is $\lg n$, which corresponds to the best case running time $\Omega(n \lg n)$

Exercise 7.4-3

$f(q) = q^2 + (n-q-1)^2 \implies f'(q) = 4q - 2n + 2$ when $q \in [0, (n-1)/2]$, its first derivative is less than 0, and when $q \in [(n-1)/2, n-1]$, it is larger than 0, which means the value of $f(q)$ achieves the max when $q = 0$ or $q = n - 1$

Exercise 7.4-4

$E[X] = \sigma_{i=1}^{n-1} \sigma_{k=1}^{n-i} 2/(k+1) > \sigma_{i=1}^{n-1} \sigma_{k=1}^{n-i} 1/(k+1) \implies$ expected running time is $\Omega(n \lg n)$

Exercise 7.4-5

To sort n/k array with length k by insertion sort, we need $(n/k) \cdot O(k^2) = O(nk)$. And in the recursion tree of quick sort, we just need to pick the first $O(\lg(n/k))$ level to split the array into n/k sub arrays. And this procedure costs $O(n \lg n/k)$, such that the total running time is $O(nk + n \lg(n/k))$. In theory, if we want to make $c_1(nk + n \lg(n/k)) \le c_2 n \lg n \implies (k - \lg k) \le$

$(c_2 - c_1) \lg n/c_1$. And we can pick $k < (c_2 - c_1) \lg n/c_1$ to get a nearly optimal solution. In practice, we just need to get the average running time from statistics, and we can know the bound of k.

Exercise 7.4-6

To get a better approximation than α to $1 - \alpha$, we need to pick the partition which from the range $[\alpha, 1 - \alpha]$, and we have a chance of $1 - 2\alpha$ to get such value. Since we must guarantee that this value is in the mid in three rounds, we can pick the value from $[1 - \alpha, 1]$ and $[0, \alpha]$ (Since it is an approximation so we only need to pick the lower bound), such that we have a chance at least $6(1 - 2\alpha)\alpha^2$. And at worst, we have the probability of $1 - 6(1 - 2\alpha)\alpha^2$ to get an α to $1 - \alpha$ partition.

Problem 7-1

a.
1. 6,19,9,5,12,8,7,4,11,2,13,21 $i = 1$,$j = 11$
2. 6,2,9,5,12,8,7,4,11,19,13,21 $i = 2$,$j = 10$
3. 6,2,9,5,12,8,7,4,11,19,13,21 $i = 10$,$j = 9$

b. After the first round, $i = 1$ and $j = k$, where $k \geq 1$. If $k = 1$, the procedure will return directly and it couldn't access any element outside of the range $[p..r]$. If $k > 1$, i will arrive at k and j will arrive at 1 definitely if the procedure didn't return before it. And in this specify round, $i > j$ hold and the procedure will also return. Such that we also couldn't access any element not in the range.

c. We have prove that j is in the range $[p..r]$ by question b, we only need to prove that $j \neq r$. Suppose we can get $j = r$. After the first round, $i = 1$ and $j = r$ must hold. But in the second round, we do $j = j - 1$ at first and this will let $j < r$, which cause a contradiction.

d. Suppose the HOARE-PARTITION loops n rounds. Before the nth round beginning(which means $i < j$ holds), we can promise that the elements $A[p..i]$ are less than $A[j, r]$ definitely, because all the elements are swapped at most once. Denote the i and j with i' and j'. And in the nth round, j will never move to the position which is smaller than i'. And i will never move to the position which is larger than j'. Consider the pointer j first. It will arrive at some position $j = k$, where the elements in $A[k, j']$ are larger than any of the elements in $A[p, i]$. Combine with the previous conclusion, the elements

in $A[k, r]$ are larger than the elements in $A[p, i']$. Now we move the pointer i, it will stop at some position $i = k'$, where $k' \geq k$ hold(Remember that the procedure ends in this round). If $k' = k$, we have prove the answer. If $k' > k$, the $A[k, k']$ must satisfy the condition $A[i] \geq x$ and $A[i] \leq x$ at the same time, which means $A[i] = x$ must hold. And in this case, elements in $A[k + 1, r]$ are larger than $A[p, k]$ when $k = j$
e. Replace the PARTITION with HOARE-PARTITION will be ok.

Problem 7-2

a. RANDOMRIZED-QUICKSORT will form the recursion $T(n) = T(0) + T(n - 1) + \Theta(n)$, such that the running time is $\Theta(n^2)$
b.

PARTITION'(A, p, r)

```
 1  q = p − 1
 2  t = p − 1
 3  pivot = A[r]
 4  i = p
 5  while i < r
 6     if A[i] < A[r]
 7        t = t + 1
 8        Swap A[t] and A[i]
 9        q = q + 1
10        Swap A[i] and A[q]
11     if A[i] = A[r]
12        t = t + 1
13        Swap A[t] and A[i]
14  Swap A[t] and A[r]
15  return q + 1, t
```

c.

QUICKSORT'(A, p, r)

```
1  if p ≥ r
2     return
3  q, t = PARTITION'(A, p, r)
4  QUICKSORT'(A, p, q − 1)
5  QUICKSORT'(A, t + 1, r)
```

d. No, we needn't adjust anything.

Problem 7-3

a. Because we pick the element randomly and the result is uniform distributed. Such that $E[X_i] = 1/n$
b. If $X_q = 1$, we pick the element in position q and the array will be split into two subarrays with length $q-1$ and $n-q$. The partition procedure will take $\Theta(n)$, such that the equality holds.
c. $E[\Sigma_{q=1}^{n} X_q(T(q-1)+T(n-q)+\Theta(n))] = (1/n)\cdot\Sigma_{q=1}^{n}(T(q-1)+T(n-q))+(1/n)\cdot\Sigma_{q=1}^{n}\Theta(n) = (1/n)\cdot\Sigma(T(0)+T(n-1)+T(1)+T(n-2)+...+T(n-1)+T(0))+(1/n)\cdot n\cdot\Theta(n) = (2/n)\cdot\Sigma(T(0)+T(1)+...+T(n-1))+\Theta(n) = (2/n)\cdot\Sigma(T(2)+T(3)+...+T(n-1))+\Theta(n)$(note that $T(0)$ and $T(1)$ are the lower order of $\Theta(n)$), $(2/n)\cdot\Sigma(T(2)+T(3)+...+T(n-1))+\Theta(n) = (2/n)\cdot\Sigma_{q=2}^{n-1}E[T(q)]+\Theta(n)$
d. $\sum_{k=2}^{n-1} k\lg k = \sum_{k=2}^{\lceil n/2\rceil-1} k\lg k + \sum_{k=\lceil n/2\rceil}^{n-1} k\lg k \le \sum_{k=2}^{\lceil n/2\rceil-1} k\lg(n/2) + \sum_{k=\lceil n/2\rceil}^{n-1} n\lg k = (-1/8)(n^2-2n-8)(1-\lg n)+n[\sum_{k=1}^{n-1}\lg k - \sum_{k=1}^{n/2}\lg k] = (-1/8)(n^2-2n-8)(1-\lg n)+n[n\lg n-(n/2)\lg(n/2)+n/2] \le (-1/8)n^2+(1/2)n^2\lg n$
e. $E[T(n)] = (2/n)\sum_{q=2}^{n-1}E[T(q)]+\Theta(n) \le (2/n)[(-1/8)n^2+(1/2)n^2\lg n]+\Theta(n) = n\lg n-(1/4)n+\Theta(n) \le an\lg n$, when we pick $a > 1$.

Problem 7-4

a. line 5 set $p = q+1$, which means the next iteration of **while** is equal to TAIL-RECURSIVE-QUICKSORT$(A, q+1, r)$
b. When we partition the subarray (p, r), we always get the $q = p+1$, it will cause the stack depth to be $\Theta(n)$
c.

TAIL-RECURSIVE-QUICKSORT(A, p, r)

```
1  while p < r
2  q = PARTITION(A, p, r)
3  if q − p > r − q
4     TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
5     r = q − 1
6  else
7     TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
8     p = q + 1
```

Problem 7-5

a. $p_i = \frac{i-1}{n} \cdot \frac{1}{n} \cdot \frac{n-i}{n} \cdot P_3 = \frac{6}{n^3}(ni - n - i^2 + i)$

b. The ordinary implementation chooses x with probability $\frac{1}{n}$. And $p_{(n+1)/2} = \frac{6}{n^3}\frac{(n-1)^2}{4} = \frac{3}{2}\frac{(n-1)^2}{n^3}$. $\lim_{n\to\infty} \frac{\frac{3}{2}\frac{(n-1)^2}{n^3}}{\frac{1}{n}} = \frac{2}{3}$

c. The ordinary implementation will get a "good" split with probability $\sum_{n/3\le i\le 2n/3} 1/n \approx \frac{1}{3}$. And $\sum_{n/3\le i\le 2n/3} p_i = \frac{6}{n^3} \cdot \sum_{n/3\le i\le 2n/3} (n-i)(i-1) \approx \frac{6}{n^3} \cdot \frac{13n^3}{162} = \frac{39}{81}$. We increase the probability $\frac{4}{27}$

d. For a $\Omega(n \lg n)$ quicksort, the median-of-3 implementation will also give a recursion of $T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$, which means it only affects the constant factor.

Problem 7-6

a.

FUZZYSORT(a, b, p, r)

```
1  if p ≥ r
2     return
3  q = PARTITION(a, p, r)
4  let {x} be the pairs which has a_q < a_i < b_q
5  let {y} be the pairs which has a_q < b_i < b_q
6  FUZZYSORT(a − {x} − {y}, b, p, q − 1)
7  FUZZYSORT(a − {x} − {y}, b, q + 1, r)
```

b. FUZZYSORT has the similar time as QUICKSORT except that it get rid of some pairs with specified attribute, so its running time is $\Theta(n \lg n)$. And

if the condition in **b.** holds, line 4 and 5 will remove all the pairs from the original array in the first call of FuzzySort, which will make its running time be $\Theta(n)$

7 Sorting in Linear Time

Exercise 8.1-1

$n - 1$

Exercise 8.1-2

$$\sum_{k=1}^{n} \lg k \geq \int_1^n \lg k \, dk = [k \lg k]_1^n - \int_1^n k \, d(\lg k) = n \lg n - n = \Omega(n \lg n)$$

Exercise 8.1-3

For a binary tree with $n!/2$ leaf, its height should be larger than $\lg n!/2$, which is not $O(n)$, the same as $n!/n$ or $n!/2^n$ leaf.

Exercise 8.1-4

For k specified groups, we have $[(n/k)!]^k$ different leaves. And the height of the tree is $\lg [(n/k)!]^k = k \cdot \frac{n}{k} \lg [(n/k)!] = \Omega(n \lg \frac{n}{k})$

Exercise 8.2-1

$C =< 2, 2, 2, 2, 1, 0, 2 > C =< 2, 4, 6, 8, 9, 9, 11 > B =< 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6 >$

Exercise 8.2-2

For any pair $A[i] = A[j]$ and $i < j$, line12 ensures that the process will fill in the B with $A[j]$ earlier than $A[i]$

Exercise 8.2-3

The modification do the same thing as the original one, but it will change the order of two identical items, such that it's not stable.

Exercise 8.2-4

Use line 1-9 in COUNTING-SORT, now $C[i]$ contains the number of elements less than or equal to i. If we want to compute the elements in range $[a, b]$, we just return $C[b] - C[a]$

Exercise 8.3-1

Round1: SEA,TEA,MOB,TAB,DOG,RUG,DIG,BIG,
BAR,EAR,TAR,COW,ROW,NOW,BOX,FOX
Round2: TAB,BAR,EAR,TAR,SEA,TEA,DIG,BIG,
MOB,DOG,COW,ROW,NOW,BOX,FOX,RUG
Round3: BAR,BIG,BOX,COW,DIG,DOG,EAR,FOX,
MOB,NOW,ROW,RUG,SEA,TAB,TAR,TEA

Exercise 8.3-2

Define the structure number, pos, if number is equal, sorted by position. Additional time is $O(n \lg n)$, space is $O(n)$

Exercise 8.3-3

For the elements with $digits = 1$, Radix sort equals to the common comparison sort, such that it works. Assume for the elements with digits less than k, the radix sort works. Now we use radix sort to deal with the elements with digit length equal to k. Since all the elements have been sorted with digit k and the elements which have same digit k will be kept in order by their lower $k - 1$ digits(the previous result is stable), such that all elements are in their correct position.

Exercise 8.3-4

Denote n_k as the value at position k of integer n, we split the n into $d_{n^3}...d_{n^2+1}$, $d_{n^2}...d_{n+1}$,$d_n...d_1$, sort each range with counting sort. The counting sort will cost $O(n)$ time because the upper bound of range is n, such that the whole procedure is $O(n)$

Exercise 8.3-5

The piles of card will create a tree and each non leaf node will have 10(0-9) children. Such that the total number is $9^{\sum_{i=1}^{d} i} = 3^{d(d+1)}$

Exercise 8.4-1

Bucket0:
Bucket1:.13,.16
Bucket2:.20
Bucket3:.39
Bucket4:.42
Bucket5:.53
Bucket6:.64
Bucket7:.79,.71
Bucket8:.89
Bucket9:

Exercise 8.4-2

When all elements fall into the same bucket, the running time is $\Theta(n^2)$. We can use quick sort instead of insertion sort when we sort the elements in each bucket.

Exercise 8.4-3

$E[X^2] = Pr\{X^2 = 0\} \cdot 0 + Pr\{X^2 = 1\} \cdot 1 + Pr\{X^2 = 4\} \cdot 4 =$
$Pr\{X^2 = 1\} \cdot 1 + Pr\{X^2 = 4\} \cdot 4 = 2 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 4 = \frac{3}{2}$
$E^2[X] = [Pr\{X = 0\} \cdot 0 + Pr\{X = 1\} \cdot 1 + Pr\{X = 2\} \cdot 2]^2 = [2 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 +$
$\frac{1}{2} \cdot \frac{1}{2} \cdot 2]^2 = 1$

Exercise 8.4-4

Assign bucket i with range $(\sqrt{\frac{i-1}{n}}, \sqrt{\frac{i}{n}}]$, $i \in [1, n]$. Such that the distance of n points from the origin are uniformly distributed in these ranges. Compute the d_i for each i and put it into proper bucket, then concatenate each bucket.

Exercise 8.4-5

We can split the range $[0, 1]$ into n buckets. Because $P(x) \in [0, 1]$ and if $x_i < x_j$, $P(x_i) < P(x_j)$. We compute $P(i)$ for each number i and put it into bucket k with range $[\frac{k-1}{n}, \frac{k}{n})$ and $\frac{k-1}{n} \leq P(i) < \frac{k}{n}$.

Problem 8-1

a. An input with size n has $n!$ probable permutations,which correspond to $n!$ leaves. And each permutation are equally likely. Such that each leaf has a chance of $\frac{1}{n!}$ to arrive.
b. The recursion equality is clearly correct.
c. The value of $D(T)$ can only be generated by $D(LT)$ and $D(RT)$ and k. We must choose $d(RT)$ and $d(LT)$ to generate $d(T)$, if not, cut and paste rule will give an smaller answer. Such that the equality of $d(k)$ holds.
d. $f(i) = i \lg i + (k-i) \lg(k-i)$, $f'(i) = \lg i + \frac{1}{\ln 2} + \frac{k}{(i-k) \ln 2} - [\lg(k-i) + \frac{1}{(i-k) \ln 2}]$, when $i = k/2$, $f'(i) = 0$, which minimized $f(i)$
e. $D(T_A) \geq d(T_A)$ and T_A has $n!$ available leaves, such that $d(T_A) = d(n!) = \Omega(n! \lg(n!))$, T_A has $n!$ leaves, which means it has $n!$ different routes to leaves at least. Such that the average length of a route is at least $\frac{\Omega(n! \lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n)$. We can conclude that the average running time to sort n elements is $\Omega(n \lg n)$
f. do not understand the question.

Problem 8-2

a. Count Sort
b. Quick Sort, we only need to partite 0 and 1
c. Bubble Sort
d. Count Sort is a good candidate. Its running time is $\Theta(n)$, such that the total running time is $\Theta(bn)$ **e.** Use $O(k)$ to count the number of the elements less than 1, less than 2, ... less than k. Denote them as $n_1, n_2, ...n_k$. Now we split the array into k ranges, $[0, n_1 - 1], [n_1, n_1 + n_2 - 1] ... [n_{k-1}, n_{k-1} + n_k - 1]$. Then we can swap the current element to specified range.

COUNTING-SORT-MODIFIED(A, k)

```
1  SubArrayStart = The array C in COUNTING-SORT(A, k)
2  delete last element from SubArrayStart
3  add 0 to the header of SubArrayStart
4  for i from 0 to A.size − 1
5     while (SubArrayStart[A[i]] ≤ i)
6        Swap A[i] and SubArrayStart[A[i]]
7        SubArrayStart[A[i]] = SubArrayStart[A[i]] + 1
```

Problem 8-3

a. First, use the counting sort by the number length. It will take $O(n)$, since the number which has more digits must be larger than the one which has less digits. Then use the radix sort to deal with the numbers with same length. The total running time is $O(n)$

b.

STRING-SORT-AUX$(A, begin, end, curCharPos)$

COUNTING-SORT from $begin$ to end by $curCharPos$, in this step, $[begin, end]$ has been split into k sub ranges their index is $[begin, i_1], [i_1 + 1, i_2], [i_2 + 1, i_3]...[i_{k-1} + 1, end]$ and each sub range has the same header.
COUNTING-SORT each range by the $x_i = length(A_i) - cuCharPos$
we only need to partite each sub range into two parts, one are the strings which has $x_i > 0$ (current character is the last character in the string), another are the $x_i = 0$ (current char is not the last), computing x_i is $O(1)$.Assume the begin and end of the first part in sub range i is y_i, z_i. We have k pairs $y_1, z_1, y_2, z_2...y_k, z_k$.
For each pair, call STRING-SORT-AUX$(A, y_i, z_i, curCharPos + 1)$

STRING-SORT(A)

```
1  STRING-SORT-AUX(A, 0, A.length − 1, 0)
```

Each letter will be used twice in two COUNTING-SORT. And we know the sum of letter is n, such that the running time is $O(n)$

Problem 8-4

a. For each red jug, compare it with each blue jug, if its amount is larger than k blue jug, put it at the position k. Do the same procedure for each

blue jug, and then make the pair of the jugs with same position. The total running time is $\Theta(n)$

b. Let $(< blue_1, red_1 >, < blue_2, red_2 >, ..., < blue_n, red_n >)$ denote a output pair sequence of some specified algorithm, the total number of such sequence is $(n!) \cdot (n!) = (n!)^2$. By the decision tree model, the height of tree is $\Omega(\lg(n!)^2) = \Omega(n \lg n)$, which prove the lower bound.

c. Assume the sum of current blue jugs and red jugs is n. We randomly choose a blue jug, compare it with each red jugs, split the red jugs into three parts, the red jugs with amount less than that blue jugs, the only red jug with same amount, the red jug with larger amount. And we do the same procedure to the blue jugs by using the only red jug with same amount. The step cost $O(n)$. And we recursively call the algorithm for the corresponding parts. The entire algorithm is the same as QUICK-SORT, such that its average running time is $O(n \lg n)$, the worst case is $O(n^2)$

Problem 8-5

a. The array which is in ascending/descending order is 1-sorted.

b. $1, 2, 3, 4, 5, 6, 7, 8, 10, 9$

c. If array is k-sorted, $\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \iff \frac{A[i]}{k} + \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i+k]}{k} \iff A[i] \leq A[i+k]$ for each i.

d. Sort k sub array $< A_0, A_k, A_{2k}... >, < A_1, A_{k+1}, A_{2k+1}... > ... < A_k - 1, A_{2k-1}, A_{3k-1}... >$ independently. Each sub array has at most n/k elements, such that using MERGE-SORT will get a running time $O(n/k \lg(n/k))$. And the total running time is $O(n \lg(n/k))$

e. We can use a heap to merge k sorted sub array. The leaf of heap is k, such that the height of heap is at most $\lg k$. The total running time is $O(n \lg k)$

g. Since condition d. holds, we must sort k groups elements, and the length of each group is n/k. The sum of permutation number is $(\frac{n}{k}!)^k$, so if we use the comparison sort, the running time is $\Omega((\lg \frac{n}{k}!)^k) = \Omega(k \cdot \frac{n}{k} \cdot \lg \frac{n}{k}) = \Omega(n \lg \frac{n}{k})$. k is a constant, the lower bound is $\Omega(n \lg n)$

Problem 8-6

a. The possible way number is C_{2n}^n

b. By decision tree model, the height of tree is at least $\lg(C_{2n}^n) = \lg(2n)! - 2\lg(n!)$. Use the Stirling's approximation, we have a $2n - o(n)$ lower bound.

c. Assume there are two sorted array, A and B. A_i and B_j are consecutive and $A_i < B_j$, then the following inequality holds for $i > 1$ and $j > 1$(the situation of $i = 0$ or $j = 0$ is trivial), $B_{j-1} \leq A_i \leq B_j \leq A_{i+1}$. In the process of merge, B_{j-1} must be popped earlier than B_j. Because B_{j-1} is less than A_i, then B_{j-1} will be popped earlier than A_i, such that the headers of A and B are A_i and B_j. At this time, these two elements must be compared.
d. The worst case occurred when the number of consecutive pair maximized in two sorted array. And we can always create the following situation, assume two arrays are $A_1, A_2...A_n$ and $B_1, B_2, ...B_n$, $A_1 < B_1 < A_2 < B_2 < ... < A_n < B_n$, such that the sum of consecutive pairs is $n + n - 1 = 2n - 1$, the lower bound has been proved.

Problem 8-7

a. Assume $A[q] \leq A[p]$. We know $A[q]$ has been put into a wrong position, so it should be put before $A[p]$, such that there must be an element put into wrong position, at the same time it's smaller than $A[p]$. It's contradict to the condition that $A[p]$ is the smallest value in A being put into the wrong position.
b. Because $A[q] > A[p]$, by the rule of compare exchange algorithm, $q < p$ holds. But $A[q] = 1$ and $A[p] = 0$, which means the algorithms failed to sort the 0-1 array correctly.
c.

8 Elementary Data Structures

Exercise 10.1-1

4
4 1
4 1 3
4 1
4 1 8
4 1

Exercise 10.1-2

Put two top pointers at the A[1] and A[n]. And the pointer at A[1] increase with each push, the pointer at A[n] decrease.

Exercise 10.1-3

4
4 1
4 1 3
1 3
1 3 8
3 8

Exercise 10.1-4

ENQUEUE(Q, x)

```
1  if Q.tail == (Q.head + Q.length) mod Q.length
2     return overflow
3  Q[Q.tail] = x
4  if Q.tail == Q.length
5     Q.tail = 1
6  else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
1  if Q.head == (Q.tail + Q.length) mod Q.length
2     return underflow
3  x = Q[Q.head]
4  if Q.head == Q.length
5     Q.head = 1
6  else Q.head = Q.head + 1
7  return x
```

Exercise 10.1-5

PUSHFRONT(D, x)

```
1  D[D.head] = x
2  D.head = D.head − 1
```

PopFront(D)

```
1  x = D[D.head]
2  D.head = D.head + 1
3  return x
```

PushBack(D, x)

```
1  D[D.tail] = x
2  D.tail = D.tail + 1
```

PopBack(D)

```
1  x = D[D.tail]
2  D.tail = D.tail − 1
3  return x
```

Exercise 10.1-6

EnQueue(A, B, x)

```
1  Push(A, x)
```

DeQueue(A, B)

```
1  while A is not empty
2     x = Pop(A)
3     Push(B, x)
4  return Pop(B)
```

Without amortized analysis, the running time of EnQueue is $O(1)$, the running time of DeQueue is $O(n)$

Exercise 10.1-7

Pop(A, B)

```
1  x = DeQueue(A)
2  return x
```

PUSH(A, B, x)

```
1  while A is not empty
2     t = DEQUEUE(A)
3     ENQUEUE(B, t)
4  PUSH(A, x)
5  while B is not empty
6     t = DEQUEUE(B)
7     ENQUEUE(A, t)
```

Without amortized analysis, the running time of POP is $O(1)$, the running time of PUSH is $O(n)$

Exercise 10.2-1

Yes, we can. The procedure is the same as double-linked list. But we can't implement the DELETE in $O(1)$, because we have to retrieve all the elements in the linked list to find the target and then delete it. It will cost $O(n)$

Exercise 10.2-2

PUSH(A, x)

```
1  append element at the header of A
```

POP(A)

```
1  x = A.head
2  delete A.head
3  return x
```

Exercise 10.2-3

ENQUEUE(A, x)

```
1  append element at the tail of A
```

DEQUEUE(A)

```
1  x = A.head
2  delete A.head
3  return x
```

Exercise 10.2-4

LIST-SEARCH'(L, k)

```
1  L.nil.key = k
2  x = L.nil.next
3  while x.key ≠ k
4     x = x.next
5  if x ≠ L.nil
6     return x
```

Exercise 10.2-5

SEARCH and DELETE will take $O(n)$. INSERT will take $O(1)$.

Exercise 10.2-6

Use two single linked list. Connect the tail of the first list and the head of the second list. It will cost $O(1)$.

Exercise 10.2-7

RESERVE(L)

```
1  p = new node
2  newHead = p
3  for i from 1 to n
4     t = new node
5     p.next = t
6     p = p.next
7  p.next = L.nil.next
8  L.nil.next = newHead
9  return L.nil
```

Exercise 10.2-8

SEARCH(L, k)

1 $x = L.nil.np$ XOR $L.tail$
2 $prev = L.nil$
3 **While** $x \neq L.nil$ and $x.key \neq k$
4 $\quad temp = x$
5 $\quad x = x.np$ XOR $prev$
6 $\quad prev = temp$
7 **return** x

INSERT(L, x)

1 $x.np = L.nil$ XOR $(L.nil.np$ XOR $L.tail)$
2 $t = L.nil.np$ XOR 0
3 $tNext = t.np$ XOR $L.nil$
4 $tNewPrev = x$
5 $t.np = tNext$ XOR $tNewPrev$
6 $LNext = x$
7 $LPrev = 0$
8 $L.nil.np = LNext$ XOR $LPrev$

DELETE(L, x)

1 $t = L.nil.np$ XOR $L.tail$
2 $prev = L.nil$
3 **While** $x \neq L.nil$ and $t \neq k$
4 $\quad temp = t$
5 $\quad t = t.np$ XOR $prev$
6 $\quad prev = temp$
7 $xNext = x.np$ XOR $prev$
8 $prev.np = prev.np$ XOR x XOR $xNext$
9 $xNext.np = prev$ XOR $(xNext.np$ XOR $x)$

REVERSE(L)

1 Swap $L.nil$ and $L.tail$

Exercise 10.3-1

index	1	2	3	4	5	6
prev	6	1	2	3	4	5
key	13	4	8	19	5	11
next	2	3	4	5	6	1

index	1	2	3	4	5	6
key|*prev*|*next*	13|4|16	4|7|1	8|10|4	19|13|7	5|16|10	11|1|13

Exercise 10.3-2

ALLOCATE-OBJECT(L)

```
1  if free == −1
2     error "out of space"
3  else
4     x = free
5     free = L[x + 1]
6     return x
```

FREE-OBJECT(L, x)

```
1  L[x + 1] = free
2  free = x
```

Exercise 10.3-3

Because we only need to modify the header of the free list.

Exercise 10.3-4

ALLOCATE-OBJECT($key, next, prev, top$)

```
1  prev[top] = top − 1
2  next[top] = top + 1
3  top = top + 1
4  return key[top − 1]
```

FREE-OBJECT$(key, next, prev, top, x)$

1 $next[prev[x]] = next[x]$
2 $prev[next[x]] = prev[x]$
3 $key[x] = key[top - 1]$
4 $next[next[top - 1]] = x$
5 $next[x] = next[top - 1]$
6 $prev[x] = prev[top - 1]$
7 $top = top - 1$

Exercise 10.3-5

COMPACTIFY-LIST(L, F)

```
 1  cur = F
 2  while cur ≠ Nil
 3     key[cur] = −∞
 4     cur = next[cur]
 5  cur = L
 6  LTail = 1
 7  while cur ≠ Nil
 8     if key[LTail] == −∞
 9        key[LTail] = key[cur]
10     else
11        Swap key[cur] and key[LTail]
12        next[cur] = next[LTail]
13        prev[cur] = prev[LTail]
14        next[prev[LTail]] = cur
15        prev[next[LTail]] = cur
16     cur = next[cur]
17     LTail = LTail + 1
18  cur = 1
19  while cur ≤ m
20     prev[cur] = cur − 1
21     next[cur] = cur + 1
22  prev[1] = Nil
23  next[LTail] = Nil
24  prev[LTail + 1] = Nil
25  next[m] = Nil
26  F = LTail + 1
27  return L, F
```

Exercise 10.4-1

L1:18;L2:12,10;L3:7,4,2,21;L4:5

Exercise 10.4-2

```
TRAVERSE-AUX(x)
1  print x.value
2  if x.left − child ≠ Nil
3     TRAVERSE-AUX(x.leftChild)
4  if x.right − sibling ≠ Nil
5     TRAVERSE-AUX(x.rightSibling)

TRAVERSE(T)
1  TRAVERSE-AUX(T.root)
```

Exercise 10.4-3

```
TRAVERSE(T)
1  create stack S
2  PUSH(S, T.root)
3  while not S.empty
4     n = POP(S)
5     visit n
6     if n.left ≠ Nil
7        PUSH(S, n.left)
8     if n.right ≠ Nil
9        PUSH(S, n.right)
```

Exercise 10.4-4

```
TRAVERSE(T)
 1  create array visited[n]
 2  for i from 1 to n
 3     if visited[i]
 4        continue
 5     create stack S
 6     PUSH(S, T[i])
 7     while not S.empty
 8        x = POP(S)
 9        visit x
10        visited[x] = true
11        if x.leftChild ≠ Nil
12           PUSH(S, x.leftChild)
13        if x.nextSibling ≠ Nil
14           PUSH(S, x.RightSibling)
```

Exercise 10.4-5

```
InOrderWalkWithoutStack(root)
 1  n = root
 2  while true
 3    while n.left exists
 4      n = n.left
 5    visit n
 6    if n.right exists
 7      n = n.right
 8    else if n.left not exists and n.right not exists
 9      while n.parent exists and n.parent.left ≠ n
10        n = n.parent
11      if n.parent not exists
12        break
13      visit n.parent
14      while n.parent exists and n.parent.right not exists or n.parent.right = n
15        n = n.parent
16      if n.parent not exists
17        break
18      n = n.parent.right
```

Exercise 10.4-6

Two pointers: left-Child, SiblingOrParent. One boolean: IsSibling. If IsSibling is true, SiblingOrParent points to the sibling. And IsSibling is false if and only if current node is the last sibling node. Such that if we want to access the parent of current node, we need follow the pointer of nextsibling until reach the last sibling node, then go to the parent node. This action will take the time linear in the number of children.

Problems 10-1

	unsorted,singly	sorted,singly	unsorted,doubly	sorted,doubly
SEARCH(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
INSERT(L, k)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
DELETE(L, k)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
SUCCESSOR(L, k)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
PREDECESSOR(L, k)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
MINIMUM(L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
MAXIMUM(L)	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Problems 10-2

a.

MAKE-HEAP()

1 create an empty list L
2 **return** L

$O(1)$

INSERT(L, x)

1 search the target position linearly, insert x
2 **return** L

$O(n)$

MINIMUM(L)

1 **return** $L.head$

$O(1)$

EXTRACT-MIN(L)

1 $x = L.head$
2 delete the head of L
3 **return** x

$O(1)$

UNION(L_1, L_2)

1 merge L_1 and L_2

2 **return** the new header of merged list

$O(|L_1| + |L_2|)$

Problems 10-3

a. Since the sequence of integer returned by RANDOM$(1, n)$ are the same, which means in the first $t-1$ iterations, both of these two procedures do not find the element. And in the tth iteration, RANDOM has either reached the position of element k or the position which is the previous of element k. Such that the number of **for** loop is at least t, and both the for and $while$ is at least t.

b. **for** take t iterations, and each iteration takes $O(1)$, such that the total is $O(t)$. **while** will execute $E[X_t]$ rounds to let the pointer move to the position of element k, so the entire procedure will take $O(t + E[X_t])$

c. X_t takes on values from the natural numbers, by **C.25**, $E[X_t] = \sum_{r=1}^{\infty} Pr\{X_t \geq i\} = \sum_{r=1}^{n} Pr\{X_t \geq r\} = \sum_{r=1}^{n}(1 - Pr\{X \leq r\})^t = \sum_{r=1}^{n}(1 - r/n)^t$

d. $\sum_{r=0}^{n-1} r^t \leq [\frac{r^{t+1}}{t+1}]_0^n \leq n^{t+1}/(t+1)$

e. $\sum_{r=1}^{n}(1 - r/n)^t = (1/n^t)\sum_{r=1}^{n}(n-r)^t = (1/n^t)\sum_{r=0}^{n-1}(r)^t \leq (1/n^t)n^{t+1}/(t+1) = n/(t+1)$

f. by **b.** and **e.**, we will get the result.

g. $f(t) = t + n/t$ arrives max when we set $t = \sqrt{n}$, such that it runs in $O(\sqrt{n})$

h.

9 Hash Tables

Exercise 11.1-1

FIND-MAXIMUM-ELEMENT(T,m)

```
1  max = −∞
2  for i from 1 to m
3     if T[i] > max
4        max = T[i]
5  return max
```

Exercise 11.1-2

First, we need a function to convert original key to a non-negative number. And this function should be a one-to-one mapping. Now we simply assume that the key is a non-negative number. We set the corresponding bit to one if we want to insert a new element into the collection, set the corresponding bit to zero if we want to erase it. Return the bit if we want to search it.

Exercise 11.1-3

Each table entry has a pointer to the header of doubly linked list which store the satellite data. Since the keys are not distinct, SEARCH return the header of corresponding linked list which contain all the satellite data with the same keys. And INSERT, DELETE both do the operations on the linked list(remember that we have the pointer to the satellite data when we delete object, we just need to remove it from doubly linked list). Trivially, all of the operations is $O(1)$

Exercise 11.1-4

Define the structure $S\{intkey; varobj\}$, the obj here is the satellite data, such that it can be any type. And make T be a stack with type S. A is the huge array.

INSERT(T,A,key,obj)

1 $T.size = T.size + 1$
2 $A[key] = T.size$
3 $T[T.size].key = key$
4 $T[T.size].obj = obj$

Now we have a double confirm, the case of the random value in the huge array which is larger than the stack size or it's not equal to the value in the stack can be handled.

SEARCH(T,A,key)

1 $p = A[key]$
2 **if** $p \notin [0, T.size]$ or $T[p].key \neq key$
3 **return** key not exists.
4 **return** $T[p].obj$

DELETE(T,A,key)

1 $A[T[T.size].key] = key$
2 Swap $T[A[key]]$ and $T[T.size]$
3 release $T[T.size]$
4 $T.size = T.size - 1$
5 $A[key] = -\infty$

10 Binary Search Tree

Exercise 12.1-1

2. 10,4,17,1,5,16,21
3. 17,10,21,4,16,1,5
4. 21,17,10,4,16,1,5
5. 21,17,16,10,4,1,5
6. 21,17,16,10,5,4,1

Exercise 12.1-2

The former property is $p.left \leq p \leq p.right$ but the latter is $p \leq p.left$ and $p \leq p.right$. No, it can not. Since we can not make sure the the relationship of the elements in the left subtree and the elements in the right.

Exercise 12.1-3

```
InOrderWalkWithStack(root)
 1  NodeStack.push(root)
 2  pop = false
 3  while sizeof(NodeStack) > 0
 4    n = NodeStack.top
 5    if n.left exists and pop = false
 6      NodeStack.push(n.left)
 7    else
 8      visit n
 9      NodeStack.pop
10      pop = true
11      if n.right exists
12        NodeStack.push(n.right)
13        pop = false
```

```
InOrderWalkWithoutStack(root)
 1  n = root
 2  while true
 3    while n.left exists
 4      n = n.left
 5    visit n
 6    if n.right exists
 7      n = n.right
 8    else if n.left not exists and n.right not exists
 9      while n.parent exists and n.parent.left ≠ n
10        n = n.parent
11      if n.parent not exists
12        break
13      visit n.parent
14      while n.parent exists and n.parent.right not exists or n.parent.right = n
15        n = n.parent
16      if n.parent not exists
17        break
18      n = n.parent.right
```

Exercise 12.1-4

preorder: Visit current node and visit left subtree and right subtree recursively.
postorder: Visit left subtree and right subtree recursively, then visit current node.

Exercise 12.1-5

Building a BST implies we sort the n elements.

Exercise 12.2-1

c and e are incorrect.

Exercise 12.2-2

TreeMinimum($root$)

```
1  if root.left exists
2     return TreeMinimum(root.left)
3  else
4     return root
```

TreeMaximum($root$)

```
1  if root.right exists
2     return TreeMinimum(root.right)
3  else
4     return root
```

Exercise 12.2-3

Tree-Predecessor(p)

```
1  while p.parent.right ≠ p
2     p = p.parent
3  p = p.parent.left
4  while p.right ≠ Nil
5     p = p.right
6  return p
```

Exercise 12.2-4

Level1: 10. Level2: 2, Level3: 4, Level4: 3, 5. Search path: 10, 2, 4, 3. 5 is to the right of path, but it is smaller than 10.

Exercise 12.2-5

For the case of successor. It should be the left-most node in right sub tree. A left-most node cannot have left child. So does the case of predecessor.

Exercise 12.2-6

Analyze the procedure of finding successor, if the node x doesn't have right child, its successor should be the lowest parent which left child is the ancestor of x.

Exercise 12.2-7

Tree-Minimum costs $O(lgn)$ time, and $n-1$calls to Tree-Successor will visit $2(n-1)$ nodes at most, which is corresponding to the $2(n-1)$ pop/push operations in InOrderWalk with stack, such that the algorithms runs within $O(lgn)+O(n)=O(n)$ time.

Exercise 12.2-8

We can decompose the k-successor visiting procedure into two sub parts. The first part is that we go back to the parent of the current node if it is the left child of its parent. Obviously, there are at most h such nodes. And the second part is we traverse the subtree of the current node. Assume the number of these kinds of nodes are k', so we have $k' < k$, to traverse these nodes we need at most $O(2k')$ time cost(one for push and one for pop). Combine these parts, we need at most $O(h+k)$ time.

Exercise 12.2-9

Assume x is the left child of y such that we have $x < y$. Suppose y' satisfies $x < y' < y$, then y' must be to the left side of y, and to the right side of x. So it must be in the right subtree of x. But x is a leaf node. It's a contradict. So does the case of x is the right child of y.

Exercise 12.3-1

```
TREE-INSERT(n, v)
 1  if v ≥ n.value
 2     if n.left = Nil
 3        n.left = v
 4     else
 5        TREE-INSERT(n.left, v)
 6  else
 7     if n.right = Nil
 8        n.right = v
 9     else
10        TREE-INSERT(n.right, v)
```

Exercise 12.3-2

Since the place we first insert the node is the child of the node which we examine to find the correct place.

Exercise 12.3-3

worst-case: $O(n)$
best-case: $O(nlgn)$

Exercise 12.3-4

It's commutative. The deletion of node n will only impact the subtree which root is n. Such that if x is not the ancestor of y and vice versa, the deletion sequence of x and y can be changed. Now assume y is the ancestor of x. Consider the case that x is in the left subtree of y. the deletion of x and y will move the left-most nodes in their right subtree. And they couldn't be the same. So this case doesn't matter. Then we consider the case of x is in the right subtree of y. The only subcase we need to take consideration is x is the left-most node in the right subtree of y. We can prove that the sequence doesn't impact by the graph(Omit the graph now).

Exercise 12.3-5

We needn't modify the procedure of TREE-SEARCH.

```
TREE-INSERT(n, v)
 1  if v ≥ n.value
 2    if n.left = Nil
 3      n.left = v
 4    else
 5      TREE-INSERT(n.left, v)
 6  else
 7    if n.right = Nil
 8      n.right = v
 9    else
10      TREE-INSERT(n.right, v)
```

Exercise 12.3-6

Add a random procedure to choose pred/succ with 50% chance.

Exercise 12.4-*

Problem 12.3

Problem 12.4

To do

Problem 12-1

a. $O(n^2)$
b. $O(n^2)$
c. $O(n)$
d. worst-case performance is $O(n^2)$

Problem 12-2

Create a radix tree, it will cost $O(n)$. And traverse it by depth-first search, and output the string when meet the terminal point. This procedure will

cost $O(n)$

11 Dynamic Programming

Exercise 15.1-1

Use mathematic induction. $T(0) = 2^0 = 1$ is correct. Assume $T(n-1) = 2^{n-1}$ is correct. We have $T(n-1) = 1 + \sum_{j=0}^{n-2} T(j)$ and $T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-2} T(j) + T(n-1) = 2T(n-1) = 2 \cdot 2^{n-1} = 2^n$.

Exercise 15.1-2

Greedy Algorithm will give the incorrect answer with following assignment.

$lengthi$	2	3	4	5
$pricep_i$	10	8	9	10

Exercise 15.1-3

Modify line 7 in MEMOIZED-CUT-ROD-AUX to $q = max(q, p[i]+MCRAUX(p, n-i, r) - c[i])$

Exercise 15.1-4

log $r[i]$ with the best solution in last step.

Exercise 15.1-5

FIBONACCI-AUX(n, r)

```
1  if n = 0
2     return 0
3  if r[n] > 0
4     return r[n]
5  return FIBONACCI-AUX(n − 1, r)+FIBONACCI-AUX(n − 2, r)
```

FIBONACCI(n)

```
1  set r[1...n] to 0
2  return FIBONACCI-AUX(n, r)
```

For the problem with size n, the subproblem has $n+1$ vertices and $2(n+1-2) = 2n-2$ edges.(Each node has two out edges except the node 0 and 1)

Exercise 15.2-1

$(A_1A_2)((A_3A_4)(A_5A_6))$

Exercise 15.2-2

MATRIX-CHAIN-MULTIPLY(A, s, i, j)

```
1  k = s[i][j]
2  if k = 0
3     return 0
4  else
5     return MATRIX-CHAIN-MULTIPLY(A, s, i, k) +
6     MATRIX-CHAIN-MULTIPLY(A, s, k + 1, j) +
7     A[i] · A[k] · A[j]
```

Exercise 15.2-3

$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} c^2 \cdot 2^n \geq c \cdot 2^n$ holds when $c \geq 1$

Exercise 15.2-4

vertices number is $n(n-1)/2$. And the edges number is $\sum_{i=1}^{n-1}(n-i)\sum_{j=2}^{i-1} j$, which means, we have $n-i$ subproblems with size i, and for each subproblem, we need to solve the sub of subproblem with size 2 to $i-1$ totally.

Exercise 15.2-5

$m[i, j]$ is needed by computing the set $\{m[a, j] \| a \in [1, i-1]\}$ and $\{m[i, b] \| b \in [j+1, n]\}$, so $R(i, j) = n + i - j - 1$

Exercise 15.2-6

Use strong mathematic induction. For the case $n = 2$, there is only one way to do full parenthesization, it's correct. Now assume for any $k \leq n-1$, the statement is true. Then we prove the case n, let $f(k)$ be the number

of parentheses pairs, we have $f(k) = k - 1$ and $f(n-k) = n - k - 1$ for $2 \leq k \leq n - 1$. So for the case n, the number of parentheses pairs is $f(k) + f(n-k) + 1 = k - 1 + n - k - 1 + 1 = n - 1$. The statement is true for n.

Exercise 15.3-1

RECURSIVE-MATRIX-CHAIN is better, the number of enumeration is 2^{n-1} and the recursion method use only $O(n^3)$ time.

Exercise 15.3-2

Recursion Tree:
Level1:(1,16) Level2:(1,8),(9,16) Level3:(1,4),(5,8),(9,12),(13,16)
Level4:(1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(13,14),(15,16)
The difference is that MERGE-SORT doesn't have overlapping subproblems.

Exercise 15.3-3

Yes, it does. Just replace line 5 in MEMOIZED-MATRIX-CHAIN(p) with $m[i, j] = -\infty$, and line 7 in LOOKUP-CHAIN(m, p, i, j) with if $q > m[i, j]$

Exercise 15.3-4

Given the sequence $(1, 4, 2, 5, 3)$, use the greedy approach, we'll have the result 44. But if we deal with example from left to right, we'll get the answer 33.

Exercise 15.3-5

Assume we have the following table

$lengthi$	1	2
$pricep_i$	10	1
$limitl_i$	1	1

And we need to cut a rod with length 3. We need to resolve the subproblem with length 1 and length 2. For these subproblems, both of the best choice should be length 1 rod. But we can choose the rod with length 1 only once. So the subproblems with length 2 can not use the optimal solution.

Exercise 15.3-6

Let $R_iR_{i+1}...Ri+k$ denotes the trade sequence start with fixed $R_i = a$ and end with fixed $R_{i+k} = b$. We need to solve the subproblem of R_iR_{i+1} and $R_{i+2}...R_{i+k}$ the latter should be the optimal solution, and then we integrate them into one to get the optimal solution of the sequence $R_i...R_{i+k}$. Obviously, if it's not optimal, we can use the cut-and-paste to get a better answer. Suppose we have $r_{1n} = 1$ and $r_{ij} = 10$ for $\{(i,j)|i \in [1,n] \& j \in [1,n] \& (i,j) \neq (1,n)\}$. And we have $c_1 = 0, c_i = 10000000$ for any $i \in [2,n]$, the best solution is to change the currency 1 with currency n directly. But it doesn't contain any optimal substructure.

Exercise 15.4-1

$< 1, 0, 1, 1, 0, 1 >$

Exercise 15.4-2

RECONSTRUCT-LCS(c, x, y, i, j)

```
1  if i = −1 or j = −1
2     return
3  if c[i][j] = c[i − 1][j]
4     return RECONSTRUCT-LCS(c, x, y, i − 1, j)
5  else if c[i][j] = c[i][j − 1]
6     return RECONSTRUCT-LCS(c, x, y, i, j − 1)
7  else
8     return RECONSTRUCT-LCS(c, x, y, i − 1, j − 1) + x[i]
```

Exercise 15.4-3

```
LCS-LENGTH(c, x, y, i, j)
 1  if i = −1 or j = −1
 2     return 0
 3  if c[i][j] ≠ −1
 4     return c[i][j]
 5  if x_i = y_j
 6     k = LCS-LENGTH(c, x, y, i − 1, j − 1) + 1
 7     c[i][j] = k
 8     return k
 9  else
10     k_1 = LCS-LENGTH(c, x, y, i − 1, j)
11     k_2 = LCS-LENGTH(c, x, y, i, j − 1)
12     k = max(k_1, k_2)
13     c[i][j] = k
14     return k
```

Exercise 15.4-4

Assume $min(m, n) = m$, we create two arrays, $current[m]$ and $last[m]$, we initialize all the elements in last array with value 0. And in each iteration, we use $last[i-1], last[i], current[i-1]$ to compute $current[i]$. We'll get the answer after n iterations. This method use $2 \cdot min(m, n) entries$.
We can use one array $current[m]$ and two additional variable $last_{i-1}$ and $last_i$ to compute LCS. Suppose we are computing $current[i]$, we save the $last_{i-1}$ and $last_i$ in advance. And each time we get the result of $current[i]$, before we replace the current[i] with new value, we save it to $last_i$ and we save the previous $last_i$ to the variable $last_{i-1}$ Which means we can use it to compute the next item in $current[m]$. This method use $min(m, n) + O(1)$ spaces.

Exercise 15.4-5

MONO-INCREASING-SUBSEQUENCE-AUX(c, x, i)

```
1  if i = 1
2     return 1
3  if c[i] > −1
4     return c[i]
5  for j from 1 to i
6     k=MONO-INCREASING-SUBSEQUENCE-AUX(x, j)
7     if x_i > x_j and k + 1 > c[i]
8        c[i] = k + 1
9  return c[i]
```

MONO-INCREASING-SUBSEQUENCE(x)

```
1  let c be the array which length is same as array x
2  Init c with −1
3  MONO-INCREASING-SUBSEQUENCE-AUX(c, x, x.length)
4  return Max Element in c
```

Exercise 15.4-6

Create an AVL Search tree by the order of inserting. The tree node contains three extra fields(except the link to the), value, max subsequent length and a link to its previous. And each time after we insert a new element to the tree, we find its previous and set the max subsequent length value of the current node be the value of its previous plus one. And we link the current node to its previous. After the insertion ended, we traverse the tree, and get the node with max subsequent value. And by the link, we can find the longest monotonically increasing sub-sequence. Since we use an AVL tree, the insertion costs $O(nlgn)$, and the traverse will cost $O(n)$.

```
CONSTRUCT-OPTIMAL-BST-AUX(root, i, j)
1  if i ≥ j
2     return
3  k = root[i][j]
4  if i ≤ k − 1
5     print root[i][k − 1] is the left child of k
6     CONSTRUCT-OPTIMAL-BST-AUX(root, i, k − 1)
7  if k + 1 ≤ j
8     print root[k + 1][j] is the right child of k
9     CONSTRUCT-OPTIMAL-BST-AUX(root, k + 1, j)

CONSTRUCT-OPTIMAL-BST(root)
1  print root[1][root.length] is the root
2  CONSTRUCT-OPTIMAL-BST-AUX(root, 1, root.length)
```

Exercise 15.5-2

It's just a computing process, leave it to be solved later.

Exercise 15.5-3

It will not affect the asymptotic running time. Computing (15.12) takes $O(n)$, so the running time of the entire algorithm is still $O(n^3)$

Exercise 15.5-4

From this fact, each time we compute $root[i, j]$, in line 10 we needn't retrieve r from i to j, we could scan it from $root[i - 1, j]$ to $root[i, j + 1]$. (Need to prove the correctness)

Problem 15-1

LONGEST-SIMPLE-PATH-AUX(g, c, i, t)

```
1  if c[i][t] ≠ −∞
2     return graph[i][t]
3  if g[i][t] ≠ ∞
4     c[i][t] = g[i][t]
5     return graph[i][t]
6  for j in {j|g[i][j] ≠ ∞}
7     c[i][t] = max(c[i][t], g[i][j]+LONGEST-SIMPLE-PATH-AUX(g, c, j, t))
8     return c[i][t]
```

LONGEST-SIMPLE-PATH(g, s, t)

```
1  Init array c[g.width][g.height] with −∞
2  return LONGEST-SIMPLE-PATH-AUX(g, c, s, t)
```

It looks like a topological sort graph. Since we access the internal nodes and edges from s to v only once, the efficiency is $O(|V| + |E|)$

Problem 15-2

LONGEST-PALINDROME-SUBSEQUENCE(s)

```
1  maxLcs = 0
2  mid = 0
3  for i from 2 to s.length
4     t = LCS(S[1...I-1],REVERSE(S[I+1...S.LENGTH]))
5     if t.length > maxLcs
6        maxT = t
7        maxLcs = t.length
8        mid = i
9  return maxT + s[mid] + Reverse(maxT)
```

It takes $O(n^3)$

LONGEST-PALINDROME-SUBSEQUENCE(s)

```
1  LCS(s,Reverse(s))
```

It takes $O(nlgn)$

Problem 15-3

BITONIC-TOUR-AUX(g, c, i)

```
1  if c[i] < ∞
2     return c[i]
3  for all k in {k|g[k] > g[i]}
4     c[i] = min(c[i], ∑_{s>i}^{s<k} g[i][s]+BITONIC-TOUR-AUX(g, s))
5  return c[i]
```

BITONIC-TOUR(g)

```
1  sort graph node, order from left to right, denote the sorted array with g
2  Init array c with ∞
3  BITONIC-TOUR-AUX(g, c, 1)
```

Problem 15-4

SPLIT-LINE-AUX(a, c, i, j)

```
1  if c[i][j] ≠ −1
2     return c[i][j]
3  if ∑_{k=i}^{j} a[k] < M
4     c[i][j] = ∑_{k=i}^{j} a[k]
5     return c[i][j]
6  for k from i + 1 to j − 1
7     c[i][j] = min(c[i][j], SPLIT-LINE-AUX(a, c, i, k) + SPLIT-LINE-AUX(a, c, k, j))
8  return c[i][j]
```

SPLIT-LINE(a)

```
1  init array c with −1
2  return SPLIT-LINE-AUX(a, c, 1, a.length)
```

Problem 15-5

a.

```
EDIT-DISTANCE-AUX(x, y, c, i, j)
 1  if i = 0 or j = 0
 2     return 0
 3  if c[i][j] < ∞
 4     return c[i][j]
 5  minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i − 1, j − 1) + cost(replace)
 6  minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i − 1, j) + cost(delete)
 7  minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i, j − 1) + cost(insert)
 8  if x[i] = y[j]
 9     minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i − 1, j − 1) + cost(copy)
10  if i − 1 > 0 and j − 1 > 0 and x[i] = y[j − 1] and x[i − 1] = y[j]
11     minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i − 2, j − 2) + cost(twiddle)
12  c[i][j] = minDist
13  return c[i][j]
```

```
EDIT-DISTANCE(x, y)
1  minDist = ∞
2  for i from 1 to x.length
3     init array c with ∞
4     minDist = min(minDist, EDIT-DISTANCE-AUX(x, y, c, i − 1, j) + cost(kill(i, x.length)))
5  return minDist
```

The running time is $O(n^3)$ and space requirement is $\Theta(n^2)$

b.

We assign the element in operation set with different values. Assume current character is c, $cost(copy) = 1$ and only if c is not space. $cost(replace) = -2$, if c is space. $cost(replace) = -2$ if c is not space and we replace c with space. $cost(replace) = -1$ if we replace c without space. $cost(delete) = 0$. And the same method for the other operations.

Problem 15-6

```
BEST-PARTY-AUX(c, root)
 1  if root = Nil
 2     return 0
 3  if c[root] < ∞
 4     return c[root]
 5  x = root.leftChild
 6  root_0 = 0
 7  init set y
 8  while x ≠ Nil
 9     root_0.add(BEST-PARTY-AUX(c, x))
10     add all childs of x into y
11     x = x.NextSibling
12  root_1 = 0
13  for all x ∈ y
14     root_1.add(BEST-PARTY-AUX(c, x))
15  c[root] = max(root_0, root_1)
16  return c[root]
```

```
BEST-PARTY(root)
1  init array c with ∞
2  return BEST-PARTY-AUX(c, root)
```

The running time is $\Theta(n)$, because the in-edges of each node in sub status graph are at most two. And we have $\Theta(n)$ nodes.

Problem 15-7

a. Use BFS Traverse to extend the current node. Assume we are extending the node v_i, we only extend the nodes which are the neighbours of v_i and with value $v_i v_j = \phi_i$. The running time is $O(n)$.Note that we only need to return an arbitrary path so we will not access the visited node again.
b. When using BFS, we extend the child with the sorted order, then if a path is returned, it must be the most probable path.

Problem 15-8

a. We have $T(i) = 3T(i-1)$ and $T(1) = n$, such that $T(m) = \Omega((3n)^m)$
b.

```
LOWEST-DISRUPTION-AUX(c, A, i, j)
 1  if i = m + 1
 2     return 0
 3  if c[i][j] < ∞
 4     return c[i][j]
 5  minDis = ∞
 6  if j − 1 > 0
 7     minDis = min(minDis, LOWEST-DISRUPTION-AUX(c, A, i + 1, j − 1))
 8  if j + 1 ≤ n
 9     minDis = min(minDis, LOWEST-DISRUPTION-AUX(c, A, i + 1, j + 1))
10  minDis = min(minDis, LOWEST-DISRUPTION-AUX(c, A, i + 1, j))
11  c[i][j] = minDis
12  return c[i][j]
```

```
LOWEST-DISRUPTION(A)
1  minDis = ∞
2  for i from 1 to n
3     init array c with ∞
4     minDis = min(minDis, LOWEST-DISRUPTION-AUX(c, A, i, 1))
5  return minDis
```

Running time is $O(mn)$

Problem 15-9

LEAST-COST-CUT-AUX(c, L, i, j)

```
1  if L(i, j) contains no other element
2     return 0
3  if c[i][j] ≠ ∞
4     return c[i][j]
5  minCost = ∞
6  for k ∈ L(i, j)
7     minCost = min(minCost, LEAST-COST-CUT-AUX(c, L, i, k)+
8     LEAST-COST-CUT-AUX(c, L, k + 1, j) + j − i)
9  c[i][j] = minCost
10 return minCost
```

LEAST-COST-CUT(L, S)

```
1  init array c with ∞
2  LEAST-COST-CUT-AUX(c, L, 1, S.length)
```

Problem 15-10

a. Assume there exists an optimal solutions, which has more than one choice for the investment in some year k. The choices are $d_{i_1} \cdot r_{i_1 k}, d_{i_2} \cdot r_{i_2 k} ... d_{i_t} \cdot r_{i_t k}$, where $d_{i_1} + d_{i_2} + ... + d_{i_t} = d$. Let $r_{i_m k}$ is the highest rate in $(r_{i_1 k}, r_{i_2 k} ... r_{i_t k})$, if we let $d_{i_m} = d$ and set the other parameters to zero, we can get a higher reward than the previous decision, which cause a contradiction.
b. If we want to know the best reward of choosing ith investment in the jth year, we should get the best reward of each investment of $j - 1$th year, then we can use the table r_{ij} for all $i \in n$ to compute it.
c.

```
OPTIMAL-INVESTMENT-AUX(c, r, i, j)
1  if c[i][j] ≠ ∞
2     return c[i][j]
3  maxInvest = 0
4  for all s ∈ n
5     for all t ∈ n, t ≠ s
6        maxInvest = max(maxInvest, OPTIMAL-INVESTMENT-AUX(c, r, s, j − 1) · r_tj − f_2)
7     maxInvest = max(maxInvest, OPTIMAL-INVESTMENT-AUX(c, r, s, j − 1) · r_sj − f_1)
8  c[i][j] = maxInvest
9  return c[i][j]

OPTIMAL-INVESTMENT(r)
1  init array c with ∞
2  init c[i][1] for all i ∈ n with r_i1
3  maxInvest = OPTIMAL-INVESTMENT-AUX(c, r, 1, 10)
4  return 10000 · maxInvest
```

d. Assume we have two investments for choice, and $r_{11} = 1.2, r_{12} = 1.4, r_{21} = 1.4, r_{22} = 1.2, f_1 = 1, f_2 = 2$, and in the second year, we can only choose $r_{11} = 1.2$ and $r22 = 1.2$, and this choice is not in any optimal sub-structure.

Problem 15-11

```
MIN-COST-AUX(c, h, d, i, j)
 1  if c[i][j] ≠ ∞
 2     return c[i][j]
 3  minCost = ∞
 4  for k ∈ [1, D]
 5     if d_i ≥ k
 6        extra = (d_i − k − m) · c
 7        if extra < 0
 8           extra = 0
 9        minCost = min(minCost, MIN-COST-AUX(c, h, d, i − 1, k) + extra + h[k])
10  c[i][j] = minCost
11  return c[i][j]
```

MIN-COST(h, d)

```
1  init array c with ∞
2  init c[1][1...m] with 0
3  init c[1][j|j ∈ (m, D]] with (j − m) · c
4  return MIN-COST-AUX(c, h, d, 1, n)
```

Problem 15-12

MAX-VORP-AUX$(c, vorp, price, i, j)$

```
 1  if i = 0
 2     return 0
 3  if c[i][j] ≠ ∞
 4     return c[i][j]
 5  maxVorp = 0
 6  for all people k which is in position i
 7     if j − price[k] > 0
 8        maxVorp = max(maxVorp,
 9        MAX-VORP-AUX(c, vorp, price, i − 1, j − price[k]) + vorp[k])
10     maxVorp = max(maxVorp, MAX-VORP-AUX(c, vorp, price, i − 1, j))
11  c[i][j] = maxVorp
12  return c[i][j]
```

MAX-VORP$(vorp, price, N, X)$

```
1  init array c with ∞
2  return MAX-VORP-AUX(c, vorp, price, N, X)
```

12 B-Trees

Exercise 18.1-1

If $t = 1$, the node containing 0 key appears, and it is meaningless.

Exercise 18.1-2

$t = 2$ or $t = 3$

Exercise 18.1-3

Level 1:{2} Level 2:{1}{3,4,5}
Level 1:{3} Level 2:{1,2}{4,5}
Level 1:{4} Level 2:{1,2,3}{5}

Exercise 18.1-4

$1 + 2 \cdot (2t)^{h-1}$

Exercise 18.1-5

wait until Chapter 13 finished.