字符串

串的存储

顺序存储

数组表示。或用第 0 个元素存储串长，或在最后一个字符后添加‘¥0’。

链式存储

模式匹配

朴素模式匹配

时间复杂度：$O(m)$， $O(n+m)$， $O(n*m)$

```
index(S, T, pos)
    i=pos
    j=1
    while(i <= S[0] && j <= T[0])
        if(S[i] == T[j])
            ++i
            ++j
        else
            i = i - j + 2 // next position from the first position of last match
            j = 1
    if(j > T[0])
        return i - T[0]
    else
        return 0
```

KMP 模式匹配

时间复杂度：$O(m)$，$O(n+m)$，$O(n+m)$

KMP 适用于模式与主串之间存在许多“部分匹配”。
假设模式第 k 个字符开始不与主串第 i 个字符匹配，则前 k－1 个字符的子串满足以下式子，且不存在 满足该式子。

$t_1t_2...t_{k-1}=s_{i-k+1}s_{i-k+2}...s_{i-1}$

同时，我们有

$t_{j-k+1}t_{j-k+2}...t_{j-1}=s_{i-k+1}s_{i-k+2}...s_{i-1}$

于是，$t_{j-k+1}t_{j-k+2}...t_{j-1}=t_1t_2...t_{k-1}$ 我们得知模式中下一个进行匹配的字符位是 k。

$$next[j]=\begin{cases}0 & j=1\\ \max\{k\mid 1<k<j\wedge t_1...t_{k-1}=t_{j-k+1}...t_{j-1}\} & k\text{存在}\\ 1 & \text{其他情况}\end{cases}$$

求解 next

设 $next[j]=k$，我们有 $t_1t_2...t_{k-1}=t_{j-k+1}t_{j-k+2}...t_{j-1}$。

如果 $t_k=t_j$，则有 $t_1t_2...t_k=t_{j-k+1}t_{j-k+2}...t_j$，$next[j+1]=next[j]+1$

如果 $t_k\neq t_j$，将模式作为主串和模式。向右滑动至以模式中第 $next[k]$ 位和主串第 j 位比较。

再按 $t_{next[k]}$ 与 t_j 是否相等，进行匹配。如果持续不等，

$t_{next[...next[k]]}\neq t_j,\text{until } next[...next[k]]=0$

```
getNext(T, next)
    i=1
    next[1]=0
    j=0
    while(i < T[0])
        if(j == 0 || T[i]==T[j])
            ++i
            ++j
            next[i]=j
        else
            j=next[j]

getNextVal(T, next)
    i=1
    next[1]=0
    j=0
    while(i < T[0])
        if(j == 0 || T[i]==T[j])
            ++i
            ++j
            if(T[i]!=T[j])
                next[i]=j
            else
                // if t[i]!=s[k], and, t[i]==t[j], it means t[j]!=s[k], thus, we
            go back to next[j]
                // consider S='aaabaaaab', T='aaaab'
                next[i]=next[j]
        else
            j=next[j]

indexKMP(S, T, pos)
    i = pos
    j = 1
    while(i <= S[0] && j <= T[0])
        if(j ==0 || S[i] == T[j])
            ++i
            ++j
        else
            j = next[j]
    if(j > T[0])
        return i – T[0]
    else
        return 0
```

链表

栈

FILO
假设 S 有个属性 top[S]，指向最近插入的元素。S[1]是栈底；S[top]是栈顶。
相关操作：

```
// top[S] 是 S 中 top 的 index
empty(S)
    return top[S]==0

full(S)
    return top[S]==length[S]

push(S,x)
    if full(S)
        return
    top[S]=top[S]+1
    S[top[S]]=x

pop(S)
    if empty(S)
        return
    top[S]=top[S]-1
    return S[top[S]+1]
```

栈的链式结构

不存在溢出，也不存在空间浪费。但是不是顺序存取。

```
class LinkedStackNode<T>:
    T data
    LinkedStackNode<T> next

class LinkedStack<T> :
    LinkedStackNode<T> top
    int count

    empty()
        return top==null

    push(T e)
        LinkedStackNode node = new LinkedStackNode(e)
        node.next = top
        top=node
        count++

    pop()
        if empty()
            return
        LinkedStackNode node=top
        top=top.next
        count--
        return node
```

栈的应用

递归

斐波那契数列

$$F(n\}=\begin{cases} n & \mathrm{n}\le 1 \\ F(n-1)+F(n-2) & \mathrm{n}>1 \end{cases}$$

四则运算

后缀表示法（逆波兰式）计算

```
calculate(S1,E)
    for i=1 to length[E]
        if number E[i]
            push(S1, E[i])
        else
            first=pop(S1)
            second=pop(S1)
            push(S1, operate(first, second,E[i]))
    return pop(S1)
```

中缀表示法转化后缀表示法

队列

FIFO

Q[1,...,n]来实现最多容纳 **n-1** 个元素的队列。head[Q]指向队列的头；tail[Q]指向新元素将会被插入的地方。队列元素位置为 head[Q],...,tail[Q]-1。

相关操作：

```
empty(Q)
    return head[Q]==tail[Q]

full(Q)
    return head[Q]==tail[Q]+1

size(Q)
    return (tail[Q] - head[Q] + n) mod n

enQueue(Q,x)
    Q[tail[Q]]=x
    if tail[Q]==length[Q]
        tail[Q]=1
    else
        tail[Q]=tail[Q]+1

deQueue(Q)
    x=Q[head[Q]]
```

```
    if head[Q]==length[Q]
        head[Q]=1
    else
        head[Q]=head[Q]+1
    return x

full1(Q)
    return head[Q]==(tail[Q]+1)  mod n

enQueue1(Q,x)
    if full1(Q)
        return
    Q[tail[Q]]=x
    tail[Q]=(tail[Q]+1) % n

deQueue1(Q)
    if empty(Q)
        return
    x=Q[head[Q]]
    head[Q]=(head[Q]+1)  % n
    return x
```

队列链式结构

```
class QNode<T>:
    T data
    QNode<T> next
class Queue<T>:
    QNode<T> front
    QNode<T> rear
enQueue(Queue<T> Q, T e)
    QNode<T> node = new QNode<T>
    node.data=e
    node.next=null
    Q.rear.next=node
    Q.rear=node

deQueue(Queue<T> Q)
    if Q.front==Q.rear
        return
    p=Q.front.next
    e=p.data
    Q.front.next=p.next
```

```
    if Q.rear==p
        Q.rear=Q.front
    return e
```

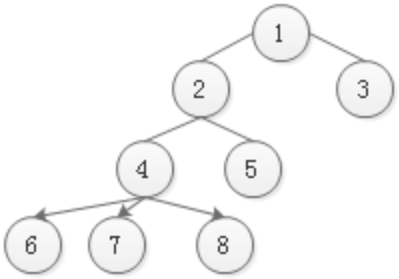
优先级队列

优先级队列是二叉堆。用来维护由一组元素构成的集合 S。每个元素都有一个关键词 key。
应用场景：作业调度；事件模拟；
最大优先级队列支持以下操作：

```
insert(S, x)
    heapSize[S]++
    S[heapSize[S]]= −∞
    increaseKey(S,heapSize[S],x)
maximum(S)
    return S[1]
extractMax(S)
    max=S[1]
    S[1]=S[heapSize[S]]
    heapSize[S]--
    maxHeapify(S,1)
    return max
increaseKey(S, x, key)
    if key<S[x]
        return
    A[x]=k
    i=x
    while i>1 &&A[parent(i)]<A[i]
        swap(A,i,parent(i))
        i=parent(i)
```

树

树是 n 个结点的集合。n=0，称为空树。任意一颗非空树：1）有且只有一个根；2）n>1 时，其余结点可分为 m 个互不相交的有限集 $T_1...T_m$，其中每个集合本身也是树，称为根的子树。



度：结点的子树数量。树的度为结点度的最大值。

层次：根为第一层；根的孩子为第二层；以此类推。
深度：树中结点的最大层次。
高度：同深度。
注意：算法导论中，高度和深度是从 0 开始。

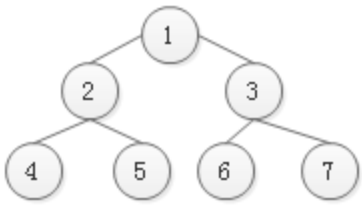
二叉树

二叉树（Binary Tree）：一个 n 个结点的集合。或为空；或为由一个根结点和两颗互不相交的分别称为根结点的左子树和右子树的二叉树组成。该树特性如下：

1. 最多只有两颗子树；
2. 子树有左右之分，不能混淆；

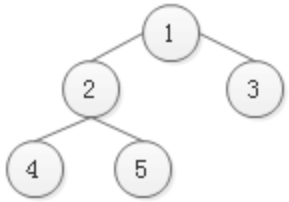
斜树：只有左子树或只有右子树的二叉树，分别称为左（右）斜树。
满二叉树：所有分支结点都有左子树和右子树，且叶子结点在同一层。

1. 叶子只可能在最后一层；
2. 非叶子结点度=2；
3. 相同深度或者高度的二叉树中，满二叉树结点数最多；叶子数最多；



完全二叉树：

一颗 n 个结点的二叉树，按层次从左到右分别编号为 1...n。如果编号为 i 的结点与同样深度的满二叉树中编号为 i 的结点位置相同，则这颗树为完全二叉树。



1．叶子结点只出现在最下两层；
2．同样结点数的二叉树，完全二叉树深度最小；

二叉树的性质

设二叉树的高度为$h \geq 0$，元素个数为 n。

1．第 i 层，至多有2^i个结点。

2．$2^h \leq n \leq 2^{h+1}-1$

3．一颗完全二叉树中，$h=\lfloor \lg^n \rfloor$

4．叶子结点数n_0，度为 2 的结点数为n_2, $n_0 = n_2 + 1$

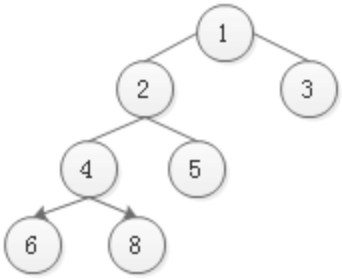
5．叶子结点的下标是$\lfloor n/2 \rfloor$+1,...,n。下标从 1 开始。

6．对于完全二叉树，结点 i，$1 \leq i \leq n$，如果$i > 1$，双亲结点为$\lfloor i/2 \rfloor$；如果 2i>n,则结点 i 没有孩子；如果 2i+ 1>n,则结点 i 没有右孩子；

二叉树的存储

1. 数组顺序存储

按完全二叉树的编号顺序存于数组中，不存在的序号以 NULL 表示。



1	2	3	4	5	N	N	6	8	N	N	N	N	N	N

上图中，我们需要数组的长度为 $2^{h+1}-1=2^{3+1}-1=15$，而该二叉树只有 7 个结点。

对于不是完全二叉树而已，顺序存储存在浪费空间的可能。

2. 二叉链表存储

left	data	right

二叉树的遍历

假设二叉树为 BTree:

```
class BTree<T>{
    BTree left;
    BTree right;
    T data;
}
```

前序遍历

```
preOrder(t)
    if t == null
        return
    traverse t.data
    preOrder(t.left)
    preOrder(t.right)
```

中序遍历

```
inOrder(t)
    if t == null
        return
    inOrder(t.left)
    traverse t.data
```

```
    inOrder(t.right)
```

后序遍历

```
postOrder(t)
    if t == null
    return
    postOrder(t.left)
    postOrder(t.right)
    traverse t.data
```

层次遍历

```
deepOrder(t)
    inQueue(queue, t)
    while notEmpty(queue)
        temp=outQueue(queue)
        traverse temp.data
        if temp.left != null
            inQueue(queue, temp.left)
        if temp.right != null
            inQueue(queue, temp.right)
```

已知前序和中序，可以唯一确定一颗二叉树；
已知后序和中序，可以唯一确定一颗二叉树；

图

算法分析

正确性分析:循环不变式

在循环中，成立的一个性质，保持算法的正确性。
初始化：第一次循环之前是正确的。
保持：如果在某次循环之前是正确的，那么在下一次循环也应该是正确的。
终止：循环结束时，是正确的。

例：插入排序算法中，始终保持 A[1...i-1]是有序的。

复杂度分析

算法复杂度一般与输入规模同步增长。可以分为最好、最坏、平均等三种情况分析。

渐进记号

Θ记号

$f(n)=\Theta(g(n))$ ：给定函数 g(n), $\Theta(g(n))$ 表示函数集合：

$\Theta(g(n))=\{f(n):0\le c_1 g(n)\le f(n)\le c_2 g(n),\exists c_1\ge 1,c_2\ge 1,n_0\ge 1,\forall n\ge n_0\}$

O记号

$f(n)=\mathrm{O}(g(n))$ ：给定函数 g(n), $\mathrm{O}(g(n))$ 表示函数集合：

$$O(g(n)) = \{ f(n) : 0 \le f(n) \le cg(n), \exists c \ge 1, n_0 \ge 1, \forall n \ge n_0 \}$$

Ω记号

$f(n) = \Omega(g(n))$ ： 给 定 函 数 g(n), $\Omega(g(n))$ 表 示 函 数 集 合 ：

$$\Omega(g(n)) = \{ f(n) : cg(n) \le f(n), \exists c \ge 1, n_0 \ge 1, \forall n \ge n_0 \}$$

定理 1：对任意函数 $f(n), g(n)$， $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$

定理 2： $\Theta(\lg^n) \subset \Theta(n) \subset \Theta(n \lg^n) \subset \Theta(n^2)$

传递性：

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

自返性：

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

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

对称性：

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

递归式

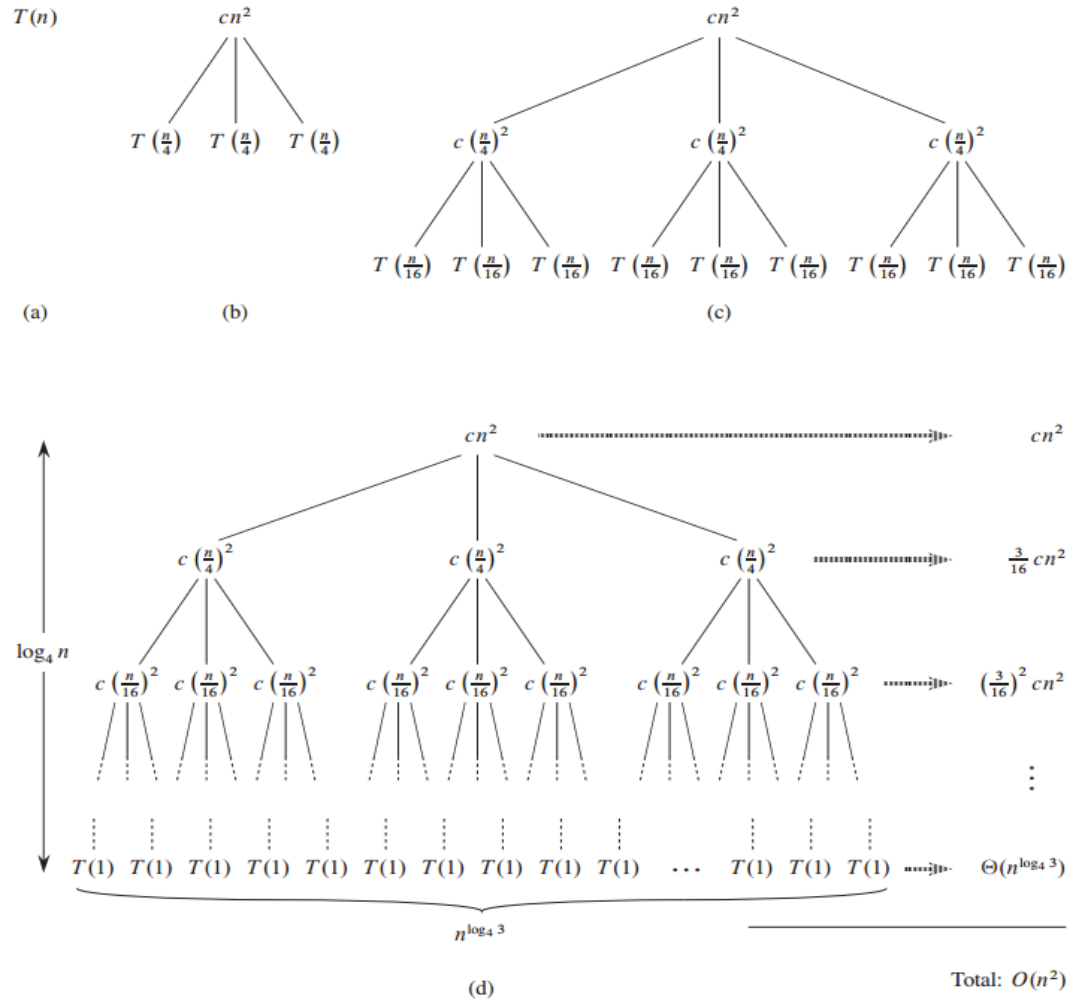
代换法

1. 猜解的形式：经验猜测；先猜宽界，再缩小。
2. 数学归纳法证明存在有效常数。

递归树

每个结点表示递归函数调用集合中一个子问题代价。

$T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$



主方法

设 $a\geq 1, b>1$，$T(n)$ 为递归式

$$T(n)=aT(n/b)+f(n)$$

我们有以下结论：

1) $\exists \varepsilon>0, f(n)=O(n^{\log_b^a-\varepsilon}) \Rightarrow T(n)=\Theta(n^{\log_b^a})$

2) $f(n)=\Theta(n^{\log_b^a})\Rightarrow T(n)=\Theta(n^{\log_b^a}\lg^n)$

3) $\exists\varepsilon>0, f(n)=\Omega(n^{\log_b^{a+\varepsilon}})$ 且$\exists c<1,$对足够大的$n, af(n/b)\le cf(n)\Rightarrow T(n)=\Theta(f(n))$

例子：

1. 计算 $T(n)=9T(n/3)+n$

a=9, b=3, $n^{\log_3^9}=n^2$; 取 $\varepsilon=1$, $f(n)=O(n^{\log_b^{a-\varepsilon}})=O(n)$, 应用规则 1），$T(n)=O(n^{\log_b^a})=O(n^2)$

2. 计算 $T(n)=T(2n/3)+1$

a=1, b=3/2, $n^{\log_{3/2}^1}=1$; 应用规则 2），$T(n)=\Theta(n^{\log_b^a}\lg^n)=\Theta(\lg^n)$

3. 计算 $T(n)=3T(n/4)+n\lg^n$

a=3, b=4, $n^{\log_4^3}=O(n^{0.793})$; 取 $\varepsilon=0.2$, $f(n)=\Omega(n^{\log_b^{a+\varepsilon}})$,

$3f(n/4)=3(n/4)\lg^{n/4}\le(3/4)n\lg^n=cf(n), c=3/4$

应用规则 3），$T(n)=\Theta(f(n))=\Theta(n\lg^n)$

随机算法

```
permuteBySorting(A)
    n=length(A)
    for i=1 to n
        P[i]=random(1, n^3)
    sort A using P as sort keys
    return A
```

```
permuteInPlace(A)
    n=length(A)
    for i=1 to n
        swap(A[i], A[random(i,n)])
```

排序

选择排序

原地排序：√
稳定排序：√

时间复杂度：$\Theta(n^2), \Theta(n^2), \Theta(n^2)$

空间复杂度：$\Theta(1)$

步骤：

```
for i=1 to length[A]-1
    minIdx=i
    for j=i+1 to length[A]
        if A[j] < A[minIdx]
            minIdx=j
    if minIdx != i
        swap(A,i,minIdx)
```

冒泡排序

原地排序：√
稳定排序：√

时间复杂度：$\Theta(n)$ if A 基本有序, $\Theta(n^2), \Theta(n^2)$

空间复杂度：$\Theta(1)$

步骤：

```
for i = 1 to length[A] - 1
```

```
flag = false
for i = 1 to length[A]
    flag=true // it means no element is swapped
    for j = length[A] to i + 1
        if A[j] < A[j-1]
            swap(A, j-1, j)
            flag=false
    if flag
        return
```

插入排序

原地排序：√
稳定排序：√

时间复杂度：$\Theta(n)$ if A 基本有序， $\Theta(n^2), \Theta(n^2)$

空间复杂度：$\Theta(1)$

步骤：

```
for i = 2 to length[A]
    key = A[i]
    j = i - 1
    while j > 0 && A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key
```

并归排序

原地排序：×
稳定排序：√

时间复杂度：$\Theta(n\lg^n), \Theta(n\lg^n), \Theta(n\lg^n)$

空间复杂度：$\Theta(n)$

步骤：

```
mergeSort(A, p, r):
    if p < r
        q = p + (r - p) / 2
        mergeSort(A, p, q)
        mergeSort(A, q + 1, r)
        merge(A, p, q, r)

merge(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q
    l = array(1...n1 + 1)
    r = array(1...n2 + 1)
    for i=1 to n1
        l[i] = A[p + i - 1]
    for i = 1 to n2
        r[i] = A[q + i]
    // sentinel element
    l[n1 + 1] = ∞
    r[n2 + 1] = ∞
    i=1
    j=1
    for k = p to r
        if l[i] <= r[j]
            A[k]=l[i]
            i=i+1
        else
            A[k]=r[j]
            j=j+1
```

堆排序

（二叉）堆是一颗完全二叉树。分最大堆和最小堆。

最大堆: A[parent(i)]>=A[i], i>1

最小堆：A[parent(i)]<=A[i], i>1

原地排序：√

稳定排序：X

时间复杂度：$\Theta(n\lg^n), \Theta(n\lg^n), \Theta(n\lg^n)$

空间复杂度：$\Theta(1)$

步骤：

```
parent(i)
    return ⌊n/2⌋

left(i)
    return 2i

right(i)
    return 2i+1

// O(lg^n)
maxHeapify(A,i)
    l=left(i)
    r=right(i)
    if l<=heapSize[A] && A[l]>A[i]
        largest=l
    else
        largest=i
    if r<=heapSize[A] && A[r>A[largest]
        largest=r
    if largest != i
        swap(A, i, largest)
        maxHeapify(A, largest)
// O(n)
buildMaxHeap(A)
    heapSize[A]=length[A]
    for i = ⌊length[A]/2⌋ to 1
        maxHeapify(A,i)

heapSort(A)
    buildMaxHeap(A)
    for i=length[A] to 2
        swap(A,1,i)
        heapSize[A]=heapSize[A]-1
        maxHeapify(A,1)

maxHeapify(A,i,m)
```

```
    largest=A[i]
    for j=2*i to m:j *= 2
        if j < m && A[j] < A[j+1]
            ++j
        if largest >= A[j]
            break
        A[i]=A[j]
        i=j
    A[i]=largest
heapSort(A)
    for i = ⌊length[A]/2⌋ to 1
        maxHeapify(A,i,length[A])

    for i = length[A] to 2
        swap(A,1,i)
        maxHeapify(A,1,i-1)
```

快速排序

快速排序适用于数据量很大的情况。

原地排序：√

稳定排序：X

时间复杂度：$\Theta(n\lg^n), \Theta(n\lg^n), \Theta(n^2)$ if 基本有序

空间复杂度：$\Theta(\lg^n) \sim \Theta(n)$

步骤：

```
quickSort(A, p, r)
    if p<r
        q=partition(A,p,r)
        quickSort(A,p,q-1)
        quickSort(A,q+1,r)
partition(A,p,r)
    x=A[r]
    i=p-1
    // ensure A[p]...A[i] <= x
```

```
    // A[i+1]...A[j-1]>x
    for j=p to r-1
        if A[j]<=x
            i++
            swap(A,i,j)
    swap(A,i+1,r)
    return i+1

randomizedPartition(A,p,r)
    i=random(p,r)
    swap(A,r,i)
    return partition(A,p,r)

randomizedQuickSort(A, p, r)
    if p<r
        q=partition(A,p,r)
        randomizedPartition(A,p,q-1)
        randomizedPartition(A,q+1,r)

partition(A,p,r)
    pivot=A[p]
    while p<r
        while p<r && A[r]>=pivot
            r--
        // make sure the value <= pivot will be in lower position
        swap(A,p,r)
        while p<r && A[p]<=pivot
            p++
        // make sure the value >= pivot will be in lower position
        swap(A,p,r)
    return p

// 三数取中法
pivotPosition(A,p,r)
    m=p+(r-p)/2;
    if A[p]>A[r]
        swap(A,p,r)
    if A[m]>A[r]
        swap(A,m,r)
    if A[m]>A[p]
        swap(A,p,m)
    return p

partition1(A,p,r)
```

```
    pivot=A[p]
    while p<r
        while p<r && A[r]>=pivot
            r--
        A[p]=A[r]
        while p<r && A[p]<=pivot
            p++
        A[r]=A[p]
    A[p]=pivot
    return p

quickSort1(A,low,high)
    if high-low  > INSERTION_SORT_THRESHOLD
        while low<high
            pivot=partition1(A,low,high)
            quickSort1(A,low,pivot-1)
            low=pivot+1
    else
        insertSort(A,low,high)
```

计数排序

假设 $0 \le A[i] \le k, \text{if } 0 \le i \le length[A]$。基本思路：**对每一个输入元素 x，确定小于 x 的元素个数。**

原地排序：X

稳定排序：√

时间复杂度：$O(n+k), O(n+k), O(n+k)$; if $k = O(n)$, T(n)= $O(n)$

空间复杂度：$O(n+k)$

步骤：

```
for i=0 to k
    C[i]=0
for i=1 to length[A]
    C[A[i]]=C[A[i]]+1
//record the count of numbers which are less or equal to i
for i=1 to k
    C[i]=C[i]+C[i-1]
// make sure it's stable
for i=length[A] to 1
```

B[C[A[i]]]=A[i]
C[A[i]]=C[A[i]]-1

基数排序

按位排序必须是稳定排序。否则，最终的排序结果将不正确。
原地排序：
稳定排序：
时间复杂度：
空间复杂度：
步骤：

```
// 每个数有 d 位数字，第一位是最低位
for i = 1 to d
    stable sort A on digit i
```

定理：给定 n 个 d 位数，每一个数位可以取 k 种可能值。基数排序算法时间复杂度 $T(n)=\Theta(d(n+k))$

定理：给定 n 个 d 位数和任何正整数 $r \le b$，基数排序时间复杂度 $T(n)=\Theta((b/r)(n+2^r))$

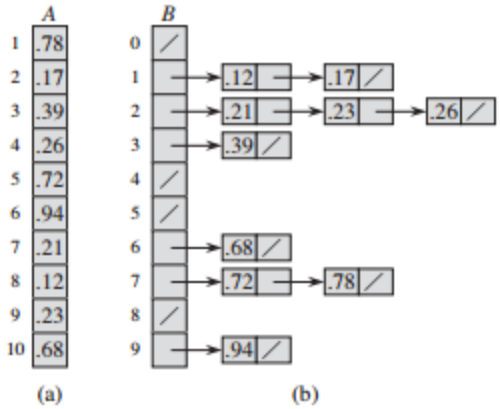
桶排序

原地排序：
稳定排序：
时间复杂度：

输入均匀分布时，$T(n)=\Theta(n)$

空间复杂度：
步骤：

```
// 0<=A[i]<1
n=length[A]
for i = 1 to n
    insert A[i] to list B[nA[i]]
for i = 0 to n-1
    sort list B[i] with insertion sort
concatenate B[0],B[1],…,B[n-1]  in order
```



总结

最好情况下，也就是说基本有序时，冒泡、插入排序更胜一筹；
最坏情况下，堆排序、归并排序强过快速排序以及其他简单排序；
如果在乎内存使用量，不适合归并排序和快速排序；

基于比较的排序算法，最好时间复杂度不会好于$\Theta(n\lg^n)$；

查找

二分查找

线性查找

同时找出最大最小值

$T(n)=\Theta(3\lfloor n/2 \rfloor)$

步骤：

```
findMaxAndMin(A)
    i=2
    if length[A] % 2 == 0
        i=3
        if A[0]>A[1]
            max=A[0]
            min=A[1]
```

```
        else
            max = A[1]
            min = A[0]
    else
        max=min=A[0]
    for j=i to length[A]-1:j  += 2
        if A[j]>A[j+1]
            if A[j]>max
                max=A[j]
            if A[j+1]<min
                min=A[j+1]
        else
            if A[j+1]>max
                max=A[j+1]
            if A[j]<min
                min=A[j]
```

以期望线性时间选择

平均 $T(n) = \mathrm{O}(n)$

最坏 $T(n) = \mathrm{O}(n^2)$

```
randomizedSelect(A,p,r,i)
    if p==r
        return A[p]
    q=randomizedPartition(A,p,r)
    k=q-p+1
    if i==k
        return A[q]
    if i<k
        randomizedSelect(A,p,q-1,i)
    else
        randomizedSelect(A,q+1,r,i-k)
```

最坏情况线性时间选择

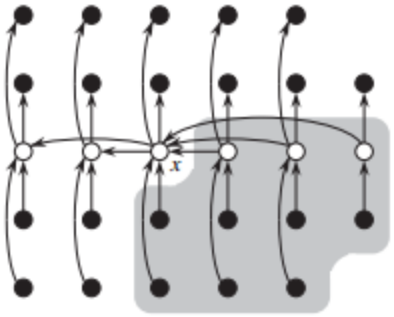


Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

The SELECT algorithm determines the ith smallest of an input array of n > 1 distinct elements by executing the following steps. (If n = 1, then SELECT merely returns its only input value as the ith smallest.)

1. Divide the array into $\lfloor n/5 \rfloor$ groups of 5 elements each

and at most one group made up of the remaining n mod 5 elements.

2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each

group (of which there are at most 5) and then picking the median
from the sorted list of group elements.

3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in

step 2. (If there are an even number of medians, then by our convention, x is the lower median.)

4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n-k$ elements on the high side of the partition.

5. If $i=k$, then return x. Otherwise, use SELECT recursively to find the ith

smallest element on the low side if $i<k$, or the $(i-k)$th smallest element on

the high side if $i>k$.

二叉查找树

AVL 树

红黑树

分治法

将原问题化解为规模较小的子问题，并不断调用自身解决子问题。分为三步：
分解（Divide）：原问题分解为一系列小问题。
解决（Conquer）：递归解各个子问题。如果子问题小到一定规模，可以直接求解。
合并（Combine）：子问题结果合并成原问题的解。

时间复杂度：$T(n)=aT(n/b)+f(n), a\geq 1, b\geq 1, f(n)$是分解子问题的时间复杂度。如归并排序时间复杂度计算式：$T(n)=\begin{cases}\Theta(1) & n=1\\ 2T(n/2)+\Theta(n) & if\ n>1\end{cases}$

空间复杂度：

```
getNext(T, next)
i=1
next[1]=0
j=0
while(i < T[0])
    if(j == 0 || T[i]==T[j])
        ++i
        ++j
        next[i]=j
    else
        j=next[j]
```

动态规划

贪婪法

Math

常用函数

下取整和上取整

$$x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1, x\in R$$

$$\lfloor n/2\rfloor+\lceil n/2\rceil=n, n\in Z$$

$$\left\lceil \lceil n/a\rceil /b\right\rceil=\lceil n/ab\rceil, n\in R, n\ge 0, a\in Z, b\in Z, a>0, b>0$$

$$\left\lfloor \lfloor n/a\rfloor /b\right\rfloor=\lfloor n/ab\rfloor, n\in R, n\ge 0, a\in Z, b\in Z, a>0, b>0$$

$$\lceil a/b\rceil\le (a+(b-1))/b, a\in Z, b\in Z, a>0, b>0$$

$$\lfloor a/b\rfloor\ge (a-(b-1))/b, a\in Z, b\in Z, a>0, b>0$$

取模运算

$$a \bmod n = a - \lfloor a/n\rfloor n$$

$$a\equiv b(\bmod n) \text{ if } (a \bmod n)=(b \bmod n)$$

$$(b-a) \bmod n = 0 \text{ if } a\equiv b (\bmod n)$$

$$a+c\equiv b+d (\bmod n) \text{ if } a\equiv b(\bmod n)\wedge c\equiv d(\bmod n)$$

$$ac\equiv bd (\bmod n) \text{ if } a\equiv b(\bmod n)\wedge c\equiv d(\bmod n)$$

指数式

根据 $\lim_{n\to\infty}\frac{n^b}{a^n}=0 \text{ if } a\in R, b\in R, a>1$，任何底大于 1 的指数函数比任何多项式函数增长更快。

对数

$$\log_c^{ab}=\log_c^a+\log_c^b$$

$$_c\log_c^a+\log_c^b=_c\log_c^a\,_c\log_c^b=ab$$

$$_c^{\log_c^{ab}}=ab$$

$$\log_b^{a^n}=n\log_b^a$$

$$\log_b^a=\frac{\log_c^a}{\log_c^b}$$

$$\log_b^{1/a}=-\log_b^a$$

$$\log_b^a=\frac{1}{\log_a^b}$$

根据 $\lim_{n\to\infty}\frac{(\lg^n)^b}{(2^a)^{\lg^n}}=\lim_{n\to\infty}\frac{(\lg^n)^b}{n^a}=0 \text{ if } a\in R, b\in R, a>0$，任何正多项式函数都比多项对数函数增长更快。

阶乘

$$\lg^{n!}=\Theta(n\lg^n)$$

其他常用技巧

双向指针

位运算

算法题

串

数组

链表

Math

Next permutation

Q:

Assume we have a sequence $a_1 ... a_n$ where n >= 1, we want to find the next permutation.

A:

Example	6	8	7	4	3	2
Step 1	6	8	7	4	3	2
Step 2	6	8	7	4	3	2
Step 3	7	8	6	4	3	2
Step 4	7	8	6	4	3	2
	7	2	3	4	6	8

1. From right to left, find the first digit (PartitionNumber)which violate the increase trend, in this example, 6 will be selected since 8,7,4,3,2 already in a increase trend.
2. From right to left, find the first digit which large than PartitionNumber, call it changeNumber. Here the 7 will be selected.
3. Swap the PartitionNumber and ChangeNumber.
4. Reverse all the digit on the right of partition index.