1.Divide and Conquer

输入:数组的长度n 以及一个数组a

输出:极小元素的下标index

观察:只需要用循环扫一遍数组中的所有元素，进行比较返回最小值的下标index就行了，最小值一定为极小值

算法：

index <- n

for i <- 0 to n-1 do

if a[i] < a[index] then

index <- i

复杂度:

时间 O(n) 为循环所需复杂度

空间 O(n) 为数组所占空间

算法正确性:很明显如果一个元素A是最小的，在a[i] 与a[index]比较的时候一定会被换入index，用循环检测了每个元素中一定包含最小元素。而最小值的左右两侧一定大于最小值。

（这是一种极其不严谨的做法）

输入:数组的长度n 以及一个数组a

输出:极小元素的下标index

观察:通过三分来查找极小值

算法:

left <- -1

right <- n

while left + 2 < right do

mid1 <- (right - left) / 3 + left

mid2 <- (right - left) / 3 \* 2 + left

if a[mid1] < a[mid2] then

right <- mid2 - 1

else

left <- mid1 + 1

for i <- left to right do

if a[i] < a[i-1] and a[i] < a[i+1] then

return i

复杂度:

时间 O(logn) 为三分所需时间

空间 O(n) 为数组所占空间

算法正确性:当我们三分的时候，left, mid1, mid2, right, 通过比较a[mid1] 和 a[mid2] 一定能确定极小值确定于[left, mid2]或者[mid1, right]中，这样每次缩小三分之一的区间，最终能求到极小值。

2.Dynamic Programming

输入:数组长度n 以及一个数组cost

输出:到最高台阶的最小花费ans

观察:这是一个简单的走台阶dp题，只要进行决策选择就行了

Optimal substructure: dp[i] 表示从0级走到i级的最小花费

DP equation: dp[i] = min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2])

算法:

dp[0] <- dp[1] <- 0

for i <- 2 to n do

dp[i] <- min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2])

ans <- dp[n]

复杂度:

时间 O(n) 为循环所需复杂度

空间 O(n) 为数组所占空间 cost,dp均占O(n)

算法正确性:由于dp问题的子结构性质，我们只要保证每一个子结构最优以及转移过程中，考虑了所有情况即可说明算法最优。题目可行决策为走一步或者两步，对应的转移方程为dp[i]从dp[i-1]和dp[i-2]转移而来，恰好一一对应。所以该算法的正确性得以证明。

3.Greedy

输入:数组长度n 以及一个数组p

输出:最小平均等待时间ans

观察:这是一个贪心做最小时间的任务就能得到最优的情况，因为等待时间不会随着次序交换而发生变化。

算法:

ans <- 0

wait\_time <- 0

p.sort()

for i <- 1 to n do

ans <- ans + wait\_time

wait\_time <- wait\_time + p[I]

ans <- ans / n

复杂度:

时间 O(nlogn) 为排序的时间复杂度

空间 O(n) 为数组所占空间

算法正确性:

我们首先猜测任务一定是从小到大做的，那么如果有两个任务i,j(i<j)不是从小到大排的，我们分别计算他们的代价

1,2,3 (a) i (b) j (c) n

1,2,3 (a) j (b) i (c) n

对于（a）(c)两处的任务，其等待时间是不变的

对于（b）处任务，改动前变动到改动后增加了j-i的等待时间，所以时间小的任务优先做一定能降低总等待时间。

4.Linear Programming Formulation

min sigma(aijdij) + sigma(cjfj)

s.t. sigma(aij) <= fj, 1 <= j <= n (1)

sigma(aij) = 1, 1 <= i <= m (2)

aij 属于 {0,1} (3)

fj 属于 {0,1} (4)

约束(1)为每个设施只能服务一个客户，没选中的设施不能服务顾客

约束(2)为每个客户有且仅有被一个设施服务

约束(3)(4)为决策变量的01约束

5.Palindromic String

(1) find the longest palindromic subsequence

输入：一个字符串s

输出：最长回文子序列长度L

观察：考虑用一个动态规划来解决最长回文子序列的问题

Optimal substructure: dp[i][j]表示从i到j的最长回文序列长度

DP equation:

dp[i][j] = 0 (i > j)

dp[i][j] = 1 (i = j)

dp[i][j] = dp[i+1][j-1] + 2 (i < j 且 s[i] == s[j])

dp[i][j] = max(dp[i][j-1], dp[i+1][j]) (i < j 且 s[i] != s[j])

算法:

s <- '$' + s

n <- s.length()

all element in array dp fill 0

for i <- n downto 1 do

dp[i][i] <- 1

for j <- i+1 to n do

if s[j] == s[i] then

dp[i][j] <- dp[i+1][j-1] + 2

else

dp[i][j] <- max(dp[i+1][j], dp[i][j-1])

L <- dp[1][n]

复杂度：

时间复杂度O(n^2) 由两重循环嵌套而成，每次循环中状态转移的时间为O(1)

空间复杂度O(n^2) 为dp数组所占的空间

算法正确性：我们需要证明最长的回文子序列可以由这样的dp方法拓展而成，假设回文子序列为(..a..b..c..d..)，对应在原字符串的位置为(..pa..pb..pc..pd),那么我们需要证明dp[pa][pd]可以由dp[pb][pc]转移而来，而事实就是如此。我们观察转移方程，当两侧元素不相等时，dp[i][j] 会从dp[i][j-1]和dp[i+1][j]中选择较大的转移而来，那么我们不同的重复这个过程,dp[pa]一定能从dp[pb]扩展而来，而dp[pd]也一定能从dp[pc]扩展而来。

(2) find the longest palindromic substring

输入：一个字符串s

输出：最长回文子串长度L和最长回文子串sub\_s

观察：考虑通过枚举方法来解决最长回文子串的问题

算法：

s <- s.fill('#') (e.x.‘aba’ -> '#a#b#a#')

n <- s.length()

L <- 1

position <- 1

for i <- 0 to n - 1 do

now <- 1

while i-now>=0 and i+now<n and s[i-now]==s[i+now] do

now <- now + 1

if L < now then

L <- now

position <- i

sub\_s <- s[i-L, i+L]

sub\_s.remove('#')

L <- L - 1

复杂度:

时间复杂度O(n^2) 由两重循环嵌套而成

空间负责度O(n) 为字符串所占空间

算法正确性：每个回文子串都可以由这种做法扩展得到。如果最长子串为奇数长度，那么最后扩展完成为"#.#.#.#"的形式；如果为偶数长度，那么最后扩展完成为"#.#.#"的形式。最后只要将长度和子串做一些逆变换即可。

6.Integer Partition

(1)prove and provide an algorithm

Prove:已知(1+K)\*K >= n,那么我们设left =(1+K)\*K-n。如果left为0，显然1到K就是n的一个方案；如果left不为0，我们也可以从1到K中找出一些数字之和等于left。这个找的策略如下，首先如果left>K我们就先取K,令left = left - K，K=K-1;直到left <= K,我们直接把left从1到K中拿出来。

输入：一个整数K

输出：一个方案数组a,代表一种可行解

观察：按照证明的策略实现即可

算法:

n <- 1

while n\*(n+1)<K do

n <- n + 1

assign t[];

for i <- 1 to n do

t[i] <- 1

left <- n \* (n + 1) - K

if left != 0 do

t[left] <- 0

now <- 0

for i <- 1 to n do

if t[i] == 1 then

now <- now + 1

a[now] <- i

复杂度:

时间复杂度O(n),为循环复杂度

空间复杂度O(n),为数组所占空间

(2)construct a set with the minimum number of positive integers

Set: {1,2,4,...,2^t|2^t-1<n 且 2^(t+1)-1>=n}

Prove:每一个数字都可以转化为二进制数，如果当前位为1，就拿集合中对应位置的幂。

(3)count the number of special integer partitions of n

输入：一个待分割整数n

输出：可分割的方案数ans

观察：可以用类似递推的方法来计数

Optimal substructure: dp[i][j]将i分割，其中最大的数字为j的方案数

DP equation:

dp[i][j] = 0 (i < j)

dp[i][j] = 1 (i = j)

dp[i][j] = sigma(dp[i-j][k]) 0 <= k < j (i > j)

算法:

dp[0][0] <- 1

for i <- 1 to n do

for j <- 1 to i do

dp[i][j] <- 0

for k <- 0 to j-1 do

dp[i][j] <- dp[i][j] + dp[i-j][k]

ans <- 0

for j <- 1 to i do

ans <- ans + dp[n][j]

复杂度：

时间复杂度O(n^3),为循环嵌套的复杂度

空间复杂度O(n^2),为数组占用空间

算法正确性：我们考虑dp[i][j]中的所有方案，一定都有数字j，所以去掉j以后的所有方案为dp[i-j][k],由于去掉j之后不知道所有方案中的当前最大数为多少，所以k也需要枚举，把小于j的数字所对应的方案全部加起来。