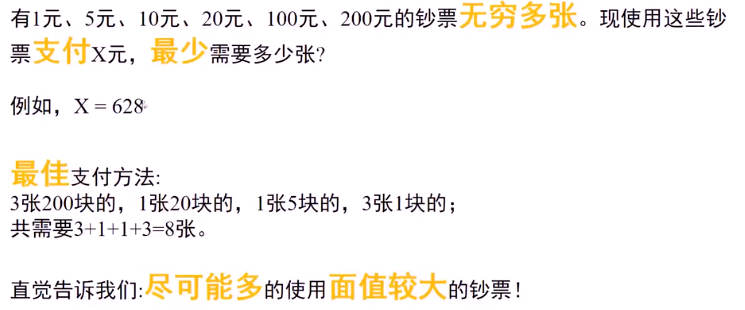
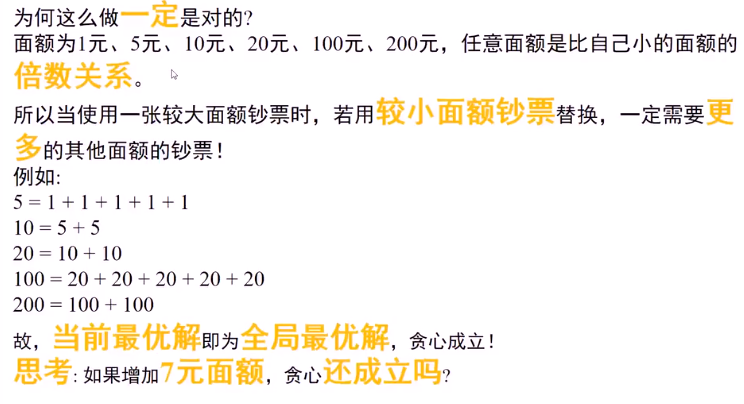
# 概述





贪心算法（greedy algorithm，贪婪算法）是分阶段执行的，每一个阶段都根据当前的情况来判断，而不考虑后续的发展。

一般来说，这种算法选出的解是**局部最佳（local best）解**。该算法预设了这样一个前提，就是认为全局最优解可以由局部最优解所推出。

贪心法是遵循某种规律，不断贪心的选取当前最优策略的算法设计方法。即，贪心算法不追求最优解，只找到满意解。

# 条件

有待处理的问题必须满足下列两项条件，才能用贪婪算法求出最优的解：

1. 具备贪心选择性质（greedy choice property）
2. 具备最优子结构（optimal substructure）

贪心选择性质：该性质意味着全局最优解可以由局部最优解（也就是在贪心策略下所选出的解）所推出。贪婪算法在针对当前这一步做决定时，可以参考前面几步的决定，但是不会依赖后续的步骤。它总是会选出局部最优的解，并将原问题约简为更小的问题，然后在更小的问题上继续寻找其局部最优解，并继续约简。

最优子结构：如果某个问题的最优解可以由其各个子问题的最优解所构成，那么该问题就具备最优子结构。这意味着把子问题的解法拼合起来可以解决最初所要求解的那个大问题。

# 特点

优点：直观、易懂，实现简单。算法一旦做出决定，就不用回过头来去重新检查前面计算过的那些值。

缺点：并非所有问题都能那么解决，对于很多问题，在某个小范围内所做的最优决策，未必是整个问题的最优决策。

# 适用场合

排序：选择排序、拓扑排序

优先级队列：堆排序

霍夫曼编码压缩算法

Prim算法与Kruskal算法

加权图中的最短路径算法（Dijkstra算法）

用零钱换整钱的问题

分数背包（fractional knapsack）问题

按照大小及权重（或者说级别）来合并不相交的集合（disjoint set）

找零钱问题

作业调度算法

把贪婪算法当成一种近似算法，来解决某些复杂的问题

赫夫曼编码

# 其它案例

## 找回零钱问题

## 装箱问题

## 最大连续子数组和

题目要求：在一个整数数组中，求和最大的子数组的值。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

最大连续子数组和

\*/

int MaxSubarray(vector<int>& vec)

{

int sum = vec[0];

int curmax= vec[0];

int i;

for(i=1;i<vec.size();i++)

{

curmax += vec[i];

if(curmax < 0)

curmax = 0;

if(curmax > sum)

sum = curmax;

}

if(sum<0)

{

for(i=0;i<vec.size();i++)

if(sum < vec[i])

sum = vec[i];

}

return sum;

}

int main()

{

int array[]={-1,1,-3,4,-1,2,1,-5,4};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<MaxSubarray(vec)<<endl;

return 0;

}

## 分糖果

题目要求：现在为已经站成一排的小朋友分糖果，保证每个小朋友至少有一个糖果，同时保证各自比相邻小朋友高的所分的糖果要比他的邻居多，按照这样的分食方法，最少需要多少糖果？

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

进行两次扫描，一次从左向右，一次从右向左

第一次扫描的时候维护对于每一个小孩左边所需要最少的糖果数量

存入数组对应元素中，第二次扫描的时候维护右边所需的最少糖果数量

并且比较将左边和右边大的糖果数量存入结果数组对应元素中

\*/

int candy(vector<int> &ratings)

{

vector<int> candy(ratings.size(),1);

int sum,i;

for(i=1;i<ratings.size();i++)

{

if(ratings[i] > ratings[i-1])

candy[i] = candy[i-1]+1;

}

sum = candy[ratings.size()-1];

for(i=ratings.size()-2;i>=0;i--)

{

int cur =1;

if(ratings[i] > ratings[i+1])

cur = candy[i+1]+1;

sum += max(cur,candy[i]);

candy[i] = cur;

}

return sum;

}

/\*

更清晰的思路

\*/

int candy2(vector<int> &ratings)

{

vector<int> candy(ratings.size(),1);

int sum,i;

for(i=1;i<ratings.size();i++)

{

if(ratings[i] > ratings[i-1])

candy[i] = candy[i-1]+1;

}

sum = candy[ratings.size()-1];

for(i=ratings.size()-2;i>=0;i--)

{

int cur =1;

if(ratings[i] > ratings[i+1] && candy[i] <= candy[i+1])

candy[i] = candy[i+1]+1;

sum += candy[i];

}

return sum;

}

int main()

{

int array[]={4,2,6,8,5};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<candy(vec)<<endl;

return 0;

}

## 跳远游戏

题目要求：给定一个整数数组，数组中的元素代表在当前位置能够向当前跳的最远距离，判断给定的这个跳远策略能否跳到最后的位置。

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

贪心思想，时刻计算当前位置和当前位置能跳的最远长度，并始终和界限比较

若在任意位置出现最大跳步为0，那么就无法继续跳下去

在任意位置出现最大跳步+当前位置 >界限，那么说明可以跳出去

\*/

bool canJump(vector<int>& vec)

{

if(vec.size() <=0)

return true;

int maxstep = vec[0];

for(int i=1;i<vec.size();i++)

{

if(maxstep == 0)

return false;

maxstep--;

if(maxstep < vec[i])

maxstep = vec[i];

if(maxstep+i >= vec.size()-1)

return true;

}

}

int main()

{

int array[]={2,3,1,1,4};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<canJump(vec)<<endl;

return 0;

}