算法复习

概念

算法效率计算

算法的概念

算法是求解问题的一系列计算步骤，用来将输入数据转换成输出结果。

算法的目标

1. 正确性
2. 可使用性
3. 可读性
4. 健壮性
5. 高效率与低存储量需求

分治法

分治法（英语：Divide and conquer）是建基于多项分支递归的一种很重要的算法范型。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

循环递归[编辑]

在每一层递归上都有三个步骤：

1. 分解：将原问题分解为若干个规模较小，相对独立，与原问题形式相同的子问题。
2. 解决：若子问题规模较小且易于解决时，则直接解。否则，递归地解决各子问题。
3. 合并：将各子问题的解合并为原问题的解。

显堆栈

```
void merge_sort(int array[], unsigned int first,
unsigned int last)
 {
    int mid = 0;
    if(first<last)
    {
        mid = (first+last)/2;
        merge_sort(array, first, mid);
        merge_sort(array, mid+1,last);
        merge(array,first,mid,last);
    }
 }
```

回溯法

书上：是一个类似琼剧的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时就“回溯”，尝试其他路径。

维基百科：对于某些计算问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，尤其适用于约束满足问题（在解决约束满足问题时，我们逐步构造更多的候选解，并且在确定某一部分候选解不可能补全成正确解之后放弃继续搜索这个部分候选解本身及其可以拓展出的子候选解，转而测试其他的部分候选解）。

分支限界法

分支限界法是一种在问题的解空间树上搜索问题解的算法，求解目标是找出满足约束条件的一个解，或是在满足约束条件的解中找出是某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

动态规划

动态规划通常基于一个递推公式级一个或多个初始状态，当前子问题的解将由上一次子问题的解推出。

区别：背包问题，01背包问题

背包问题：一个背包总容量为V，现在有N个物品，第i个 物品体积为weight[i]，价值为value[i]，每个物品都有无限多件，现在往背包里面装东西，怎么装能使背包的内物品价值最大?

01背包问题：一个背包总容量为V，现在有N个物品，第i个 物品体积为weight[i]，价值为value[i]，现在往背包里面装东西，怎么装能使背包的内物品价值最大?

完全背包问题与01背包类似，但不同之处在于完全背包问题中，物品有无限多件，每个物品可以选择多次放入背包。也就是说，每个物品的数量是无限的，目标仍然是最大化装载的物品总价值。往背包里面添加物品时，只要当前背包没装满，可以一直添加。

因此他们的状态转移方程也不同：

01背包问题的状态转移方程：`dp[i][j] = max(dp[i - 1][j], values[i - 1] + dp[i - 1][j - weights[i - 1]])`
完全背包问题的状态转移方程：`dp[j] = max(dp[j], values[i - 1] + dp[j - weights[i - 1]])`

代码填空

递归

P46

```cpp
#include<iostream>
using namespace std;
int temp=0;
move(int n,char a,char b,char c){

    if(n==0){

        return 0;//函数出口
    }
    move(n-1,a,c,b);//第一步，将n-1个盘子从A柱移动至B柱（C柱为工具柱）
    temp++;//输出移动过程前计数
    cout<<a<<"--->"<<c<<endl;//输出移动过程
    move(n-1,b,a,c);//第二步将B柱的n-1个盘子移动至C柱（A柱为工具柱）
//  cout<<b<<"--->"<<c<<endl;
}


int main(){
    int n;
    char a='A',b='B',c='C';
    cin>>n;
    move(n,a,b,c);
    cout<<temp;
    return 0;
}
```

蛮力法

P122

最小三角形路径

运用动态规划

LCR 100. 三角形最小路径和

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle)
    {
        int row = triangle.size();

        // 我们不开辟新空间，直接遍历并修改二维数组，所以不必设置初始值
        for (int i = 1; i < row; ++i)
        {
            for (int j = 0; j <= i; ++j)
            {
                // 当前位置的最小路径和，应该是上一层的位置的最小路径和+当前位置的值。
                // 求出来我们是直接放到原始数组中的，不要混淆。
                // 由于上面的位置已经被遍历修改过，已经是该位置的最小路径和，所以直接加等。
                if (j == 0)
                    triangle[i][j] += triangle[i-1][j];
                else if (j == i)
                    triangle[i][j] += triangle[i-1][j-1];
                else
                    triangle[i][j] += min(triangle[i-1][j-1], triangle[i-1][j]);
            }
        }

        // 求返回值：最底层的数组中的最小值
        int minSum = triangle[row - 1][0];
        for (auto e : triangle[row - 1])
        {
            if (minSum > e) minSum = e;
        }
        return minSum;
    }
};
```

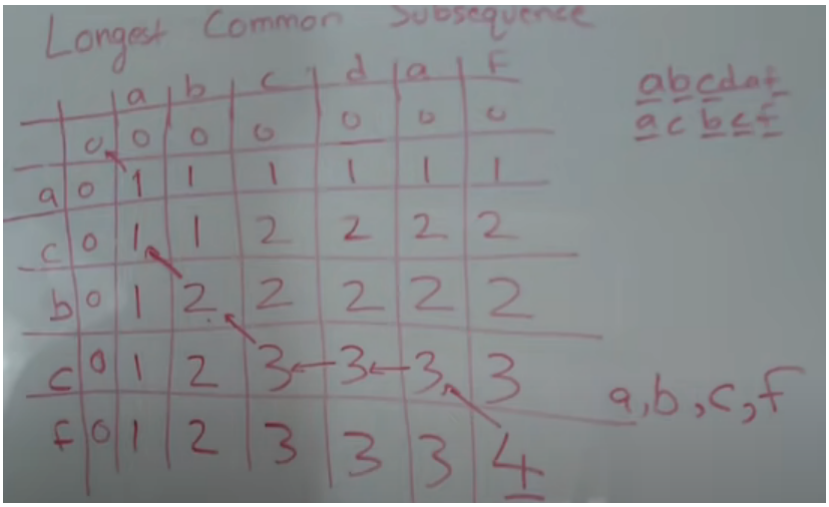
```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int row = triangle.size();
        //自下而上遍历
        for (int i = row - 2; i >= 0; --i) // 最后一行作初始值 不参与遍历
        {
            for (int j = 0; j <= i; ++j)
            {
                triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1]);
            }
        }

        return triangle[0][0];
    }
};
```

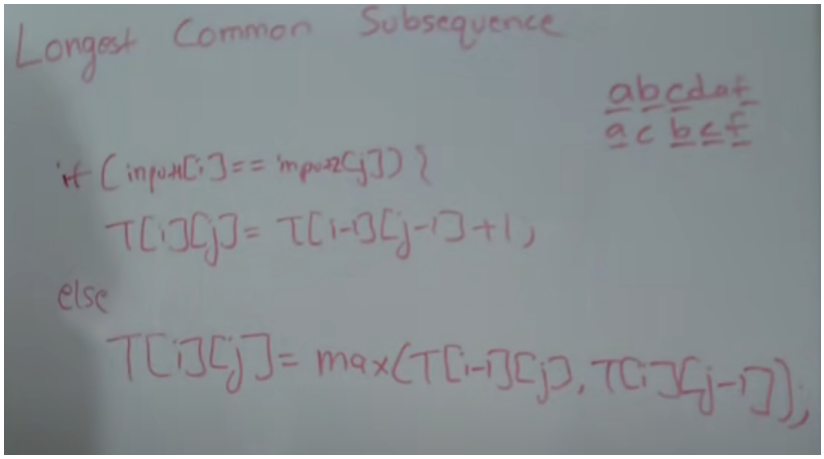
书上做法back输出就是从后面倒退，如果上左有相同则移过去，如果不相同，则往左上角，然后翻转

最长公共子序列

```
int longestCommonSubsequence(char * text1, char * text2)
{
    int a[1010][1010];
    for(int i=1;i<=strlen(text1);i++)
    {
        for(int j=1;j<=strlen(text2);j++)
        {
            if(text1[i-1]==text2[j-1])
            {
```

```
                a[i][j]=a[i-1][j-1]+1;
            }
            else
            {
                a[i][j]=fmax(a[i][j-1],a[i-1][j]);
            }
        }
    }
    return a[strlen(text1)][strlen(text2)];
}
```





视频：Longest Common Subsequence - YouTube

算法设计学习

斐波那契

斐波那契数列的定义是 `f(n+1)=f(n)+f(n-1)`，生成第 `n` 项的做法有以下几种：

暴力搜索：

原理：把 `f(n)` 问题的计算拆分成 `f(n-1)` 和 `f(n-2)` 两个子问题的计算，并递归，以 `f(0)` 和 `f(1)` 为终止条件。
缺点：大量重复的递归计算，例如 `f(n)` 和 `f(n - 1)` 两者向下递归需要 各自计算 `f(n - 2)` 的值。

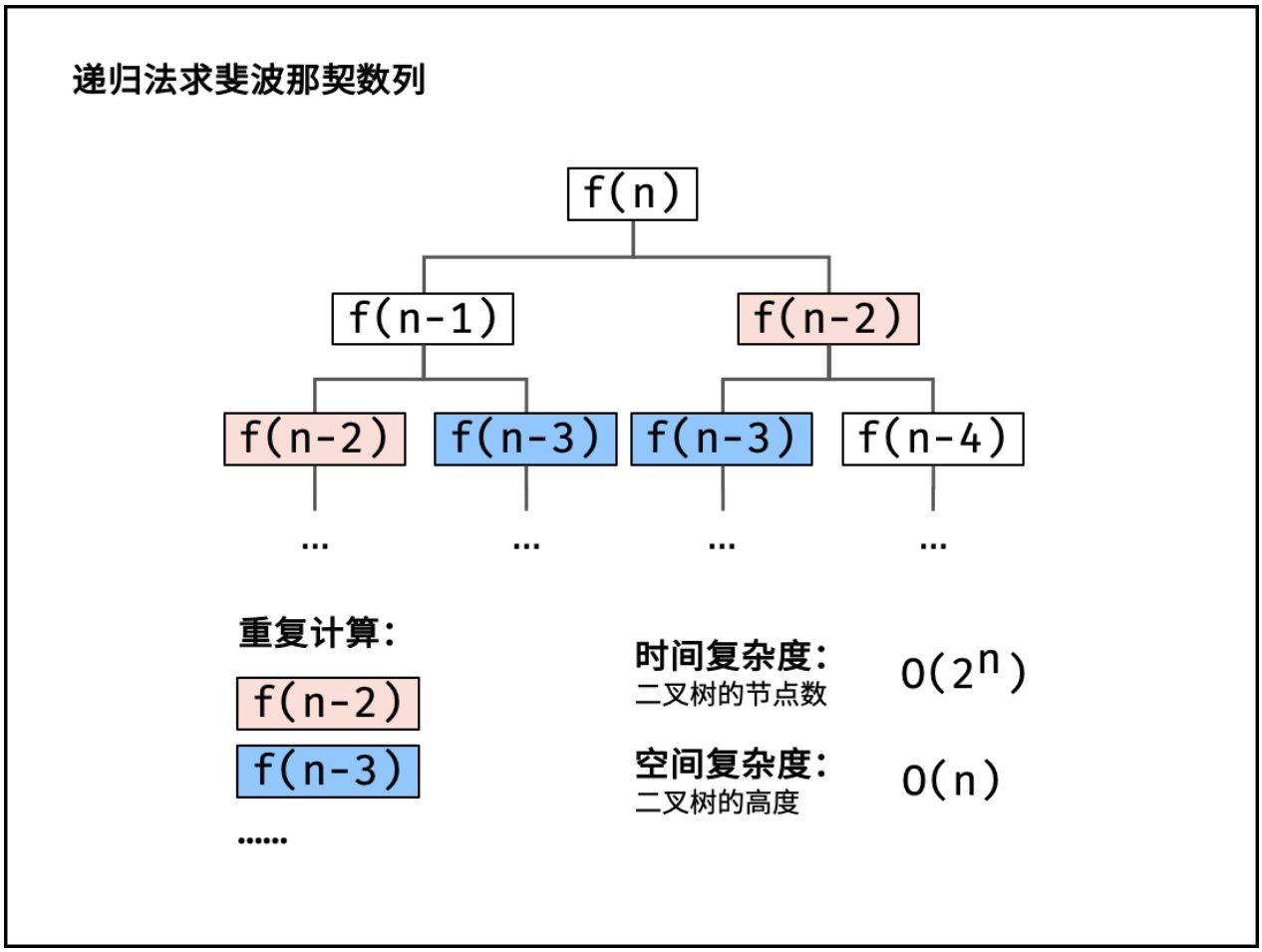
```
#include<bits/stdc++.h>
using namespace std;
int Fib(int n);
int fib(int n);
int main()
{

    int r=Fib(7);
    cout<<r<<endl;
    return 0;
}
int Fib(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}
```

记忆化递归：

原理：在递归法的基础上，新建一个长度为 `n` 的数组，用于在递归时存储 `f(0)` 至 `f(n)` 的数字值，重复遇到某数字则直接从数组取用，避免了重复的递归计算。
缺点：记忆化存储需要使用 O(N)的额外空间。

动态规划：

原理：以斐波那契数列性质 `f(n+1)=f(n)+f(n-1)` 为转移方程。
从计算效率、空间复杂度上看，动态规划是本题的最佳解法。



动态规划解析:

状态定义: 设 `dp` 为一维数组，其中 `dp[i]` 的值代表 斐波那契数列第 iii 个数字。

转移方程: `dp[i+1]=dp[i]+dp[i-1]` ，即对应数列定义 `f(n+1)=f(n)+f(n-1)` 。

初始状态: `dp[0]=0, dp[1]=1` ，即初始化前两个数字。

返回值: `dp[n]` ，即斐波那契数列的第 `n` 个数字。

状态压缩:

若新建长度为 `n` 的 `dp` 列表，则空间复杂度为 `O(N)` 。

由于 `dp` 列表第 `i` 项只与第 `i-1` 和第 `i-2` 项有关，因此只需要初始化三个整形变量 `sum, a, b` ，利用辅助变量 `sum` 使 `a,b` 两数字交替前进即可（具体实现见代码）。

节省了 `dp` 列表空间，因此空间复杂度降至 `O(1)` 。

```
class Solution {
public:
    int fib(int n) {
        int a = 0, b = 1, sum;
        for(int i = 0; i < n; i++){
            sum = a + b;
            a = b;
            b = sum;
        }
        return a;
    }
};
```

作者：Krahets
链接：https://leetcode.cn/problems/fibonacci-number/
来源：力扣（LeetCode）

水仙花数

“水仙花数”是指一个**三位数**，其各位数字立方和等于该数本身。

用循环判断就行。

```
#include<stdio.h>
#include "math.h"
main()
{ int i,iG,iS,iB;
  for(i=100;i<1000;i++)
   { iG=i%10;  /*计算个位数*/
     iS=i/10%10;  /*计算十位数*/
     iB=i/100;   /*计算百位数*/
     if(i==pow(iG,3)+pow(iS,3)+pow(iB,3))
        printf("%d  ",i);
    }
 }
```

找到最大和次大的数

遍历

直接m1为最大值, m2为次大值，每个数字遍历一遍。

```
#include <iostream>
#include <vector>

using namespace std;

int test1(vector<int> v)
{
    // 这里设m1为最大值， m2为次大值， idx为最大值对应的数组下标
    // 由于题目中nums中元素为自然数， 所以只需取-1即可
    int m1 = -1, m2 = -1, idx = 0;
    // 对数组中元素进行遍历
    for (int i = 0; i < v.size(); ++i)
    {
        // 如果当前数组元素大于m1， 执行交换
        if (v[i] > m1)
        {
            m2 = m1;
            m1 = v[i];
            idx = i;
        }
        // 如果当前数组元素在m1和m2之间， 记录m2的值
        else if (v[i] > m2) {
            m2 = v[i];
        }
    }
    return m1>=m2*2?idx:-1;
}


int main(int argc, char const *argv[])
{
    vector<int> v = {3,2,1,6,7};
    // vector<int> v = {3,2,1,7};
    cout<<test1(v)<<endl;
    return 0;
```

```
}
```

排序

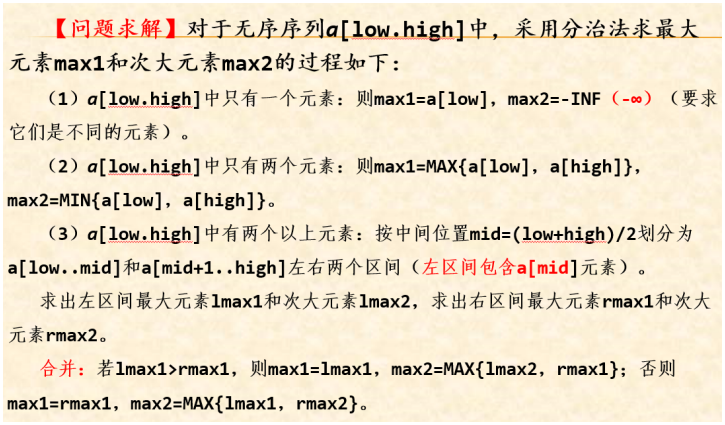
简单粗暴直接sort

非排序优化：分治与递归

分治法是将问题分成多个子问题，然后最后将问题合并起来，从而求得其解，减治法是将问题分解，但是没有将解合并，解就在子问题的解中。折半查找属于减治法。

可以拆成子问题，求解一个数组最大和次大的数，就先把它子数组的最大和次大的数求出来，把他分成几个小的子数组，然后把这些子数组的最大和次大再进行比较得出这个数组的最大和次大的数。

然后分的终止条件就是这个数组只有一个数或者两个数，只有一个数那么就最大值为它，次大值为-INF(不存在)，只有两个数就直接比较就行。


【问题求解】对于无序序列a[low.high]中，采用分治法求最大元素max1和次大元素max2的过程如下：
（1）a[low.high]中只有一个元素：则max1=a[low]，max2=-INF（-∞）（要求它们是不同的元素）。
（2）a[low.high]中只有两个元素：则max1=MAX{a[low]，a[high]}，max2=MIN{a[low]，a[high]}。
（3）a[low.high]中有两个以上元素：按中间位置mid=(low+high)/2划分为a[low..mid]和a[mid+1..high]左右两个区间（左区间包含a[mid]元素）。
求出左区间最大元素lmax1和次大元素lmax2，求出右区间最大元素rmax1和次大元素rmax2。
合并：若lmax1>rmax1，则max1=lmax1，max2=MAX{lmax2，rmax1}；否则max1=rmax1，max2=MAX{lmax1，rmax2}。

```
void solve(int a[],int low,int high,int &max1,int &max2)
{   if (low==high)       //区间只有一个元素
    {   max1=a[low];     max2=-INF;  }
    else if (low==high-1)    //区间只有两个元素
    {   max1=max(a[low],a[high]);
max2=min(a[low],a[high]); }
    else              //区间有两个以上元素
    {   int mid=(low+high)/2;
    int lmax1,lmax2;
    solve(a,low,mid,lmax1,lmax2);          //左区间求lmax1和lmax2
    int rmax1,rmax2;
    solve(a,mid+1,high,rmax1,rmax2);   //右区间求lmax1和lmax2
    if (lmax1>rmax1)
    {   max1=lmax1;
```

```
        max2=max(lmax2,rmax1);   //lmax2,rmax1中求次大元素
    }
    else
    {   max1=rmax1;
        max2=max(lmax1,rmax2);   //lmax1,rmax2中求次大元素
    }
    }
}
```

二分查找

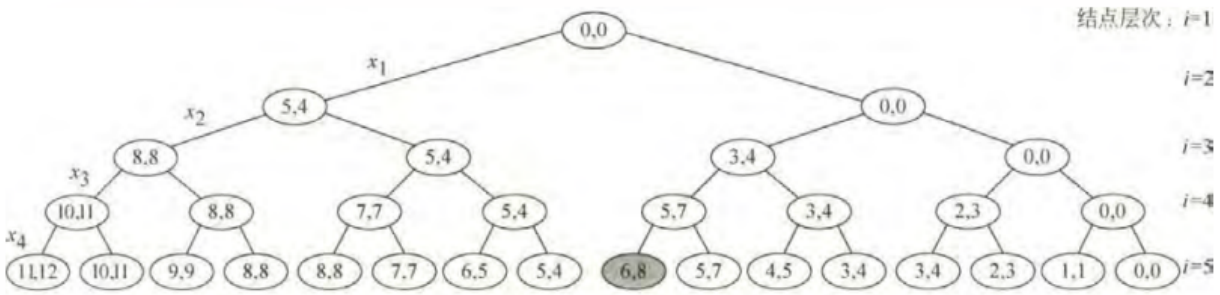
```
int BinSearch(int a[], int low, int high, int k)
//拆半查找算法
{  int mid;
   if (low<=high)          //当前区间存在元素时
   {    mid=(low+high)/2;    //求查找区间的中间位置
       if (a[mid]==k)        //找到后返回其物理下标mid
          return mid;
       if (a[mid]>k)         //当a[mid]>k时
          return BinSearch(a, low, mid-1, k);
       else                 //当a[mid]<k时
          return BinSearch(a, mid+1, high, k);
   }
  else return -1;          //若当前查找区间没有元素时返回-1
}
```

0/1背包问题解空间树

表 4.2　4 个物品的信息

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

对于表 4.2 所示的 4 个物品，在限制背包总重量 $W=6$ 时描述问题求解过程的解空间树如图 5.10 所示，每个结点中的两个数值为(tw,tv)。



采用优先队列式分支限界法求解

一个 0/1 背包问题是 $n=3$，重量为 $w=(16,15,15)$，价值为 $v=(45,25,25)$，背包限重为 $W=30$，求放入背包总重量小于等于 W 并且价值最大的解，设解向量为 $x=(x_1,x_2,x_3)$，其解空间树如图 6.5 所示。本节通过队列式和优先队列式两种分枝限界法求解该问题。

```
struct NodeType                                  //队列中的结点类型
{   int no;                                      //结点编号
    int i;                                       //当前结点在搜索空间中的层次
    int w;                                       //当前结点的总重量
    int v;                                       //当前结点的总价值
    int x[MAXN];                                 //当前结点包含的解向量
    double ub;                                   //上界
    bool operator<(const NodeType &s) const      //重载<关系函数
    {
        return ub<s.ub;                          //ub 越大越优先出队
    }
};
```

46/16=2.8 25/15=1.6 ub=45+(30-16)*1.6=67.4

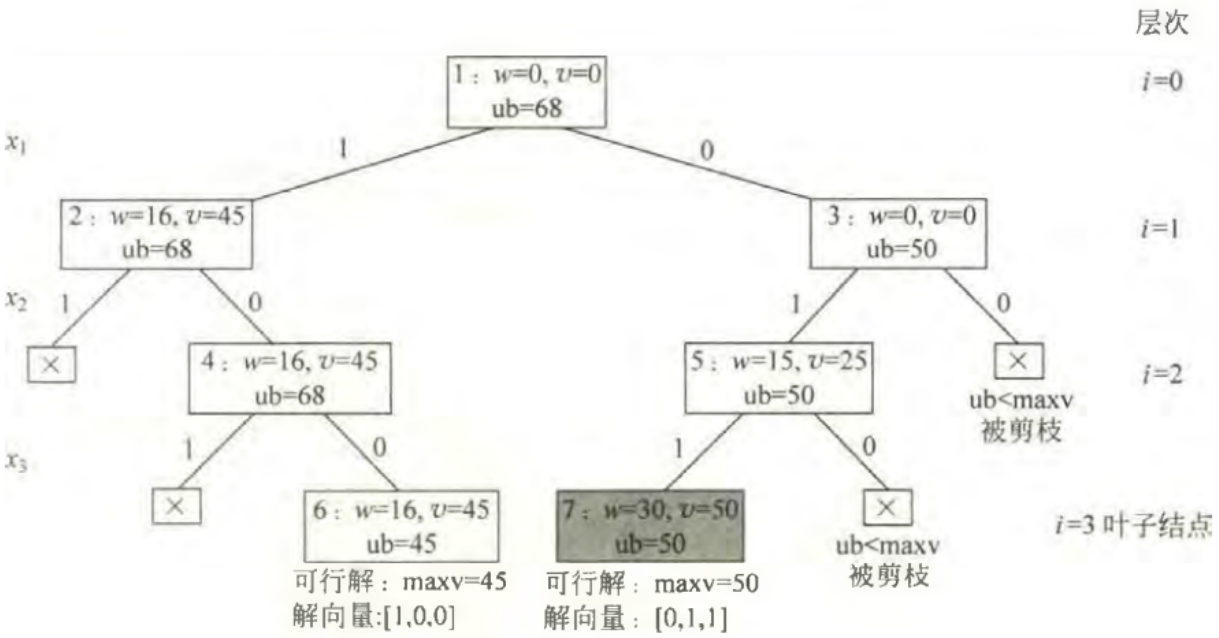


图 6.8　采用优先队列式分枝限界法求解 0/1 背包问题的过程

动态规划解决最短路径

在提供的C++代码中，`cost[]` 数组用于存储从源顶点（顶点0）到图中每个顶点的最短路径的累积成本。以下是它的工作原理的详细说明：

1. `cost[]` 数组被初始化为一个较大的值（`MAX`），对于除源顶点（顶点0）之外的所有顶点都设置为该值。这是因为从源顶点到它自身的路径成本始终为零。

```
for(i = 0; i < n; i++)
{
    cost[i] = MAX;   // 将成本初始化为较大的值
    path[i] = -1;    // 将路径初始化为-1
}

cost[0] = 0;         // 到达源顶点的成本为0
```

1. 代码然后遍历图中的顶点，根据最小成本路径更新 `cost[]` 数组。内部循环比较通过顶点 `i` 到达顶点 `j` 的成本与当前到达顶点 `j` 的成本。如果通过顶点 `i` 的新路径更短，则更新 `cost[j]`，并将顶点 `j` 的前驱顶点设置为 `i`（存储在 `path[]` 数组中）。

```
for(j = 1; j < n; j++)
{
    for(i = j - 1; i >= 0; i--)
    {
        if(arc[i][j] + cost[i] < cost[j])
        {
            cost[j] = arc[i][j] + cost[i];
            path[j] = i;  // 更新顶点j的前驱顶点
        }
    }
}
```

1 最后，函数返回 `cost[n-1]` ，它表示从源顶点到图中最后一个顶点的最短路径的累积成本。

```
return cost[n-1];
```

总之， `cost[]` 数组跟踪从源顶点到图中每个顶点的最小成本，考虑到路径上的累积成本。

源代码：

```
#include<iostream.h>
const int N = 20;
const int MAX = 1000;
int arc[N][N];
int Backpath(int n);
int creatGraph();

int main()
{
    int n = creatGraph( );
    int pathLen = Backpath(n);
    cout<<"最短路径的长度是："<<pathLen<<endl;
    return 0;
}
int creatGraph()
{
    int i, j, k;
    int weight;
    int vnum, arcnum;
```

```
    cout<<"请输入顶点的个数和边的个数：";
    cin>>vnum>>arcnum;
    for (i = 0; i < vnum; i++)
        for (j = 0; j < vnum; j++)
            arc[i][j] = MAX;
    for (k = 0; k < arcnum; k++)
    {
        cout<<"请输入边的两个顶点和权值：";
        cin>>i>>j>>weight;
        arc[i][j] = weight;
    }
    return vnum;
}
int Backpath(int n)
{
    int i, j, temp;
    int cost[N];
    int path[N];
    for(i = 0; i < n; i++)
    {
        cost[i]=MAX;
        path[i]=-1;
    }
    cost[0] = 0;
    for(j = 1; j < n; j++)
    {
        for(i = j - 1; i >= 0; i--)
        {
            if(arc[i][j]+cost[i]<cost[j])
            {cost[j]=arc[i][j]+cost[i];
            path[j]=i;
            }
        }
    }
    cout<<n-1;
    i = n-1;
    while (path[i] >= 0)
    {
        cout<<"<-"<<path[i];
        i = path[i];
    }
```

```
    cout<<endl;
    return cost[n-1];
}
```