软件设计实践

面向性能的设计

谢 涛　　马 郓

教学评估

- 网址：kcpg.pku.edu.cn
- 时间：5月26日—6月11日
- 完全匿名，教师及任何其他管理人员均无权查看学生的个人信息
- 结课后教师才可看到评估结果
  - 不必担心因为评估结果影响考试评分
- 评估内容
  - 课程
  - 两位授课教师
  - 助教
- 希望每位同学都积极参加评估，提出宝贵意见建议

期末笔试

- 时间：2023年6月23日（周五）下午14:00—16:00
- 地点：理教406
- 卷面共计100分，换算为30分计入总分
- 题型（后续公布题量及分值）：
  - 单项选择题
  - 分析简答题
- 考试范围：12—21讲的内容
- 答疑：
  - 邮件答疑：发问题给老师和助教
  - 在线答疑：微信群
  - 线下答疑：
    - 时间地点待定

本讲概览

- 性能度量指标
  - 时间性能
  - 空间性能
- 高效的算法设计
  - 枚举、递推与递归、动态规划、贪心
  - 二分、深度优先搜索、广度优先搜索
- 代码调优
  - 常用技巧
  - 面向性能的设计模式
    - 单例模式

性能

- 软件的性能是指软件在执行过程中的响应时间、资源利用率和吞吐量等方面的表现
- 软件只有在功能正确的前提下，性能才有意义
  - 对性能的关注要与其他质量属性进行折中
  - 过度的优化导致软件不再适应变化和复用
- “过早优化是万恶之源”

“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**”

— Donald Knuth

时间性能

- 执行时间
  - 每条指令、每个控制结构、整个程序的执行时间
  - 执行时间的分布：不同语句或控制结构执行时间的分布情况
  - 时间瓶颈
- 影响时间性能的因素
  - 算法
  - 数据结构
  - 内存管理
  - I/O

空间性能

- **内存占用**
  - 每个变量、每个复杂结构、整个程序的内存消耗
  - 内存占用的分布：不同变量、数据结构的相对消耗
  - 空间瓶颈
  - 内存随时间的变化
- **影响空间性能的因素**
  - 基本语句
  - 算法
  - 数据结构
  - I/O

性能分析

- 性能分析（Profiling）是一种动态程序分析形式，它可以测量程序的空间（内存）或时间复杂度、特定指令的使用情况、以及函数调用的频率和持续时间
  - 分析程序每个部分的执行时间
  - 获取程序执行的路径
  - 分析内存使用情况

- 性能分析器（Profiler）是用于收集程序执行信息并予以展示的工具

性能分析的主要手段

- 代码注入（Insertion）/代码插桩（Instrumentation）
  - 在原始程序中加入某些语句来收集运行时数据，这些语句不改变原程序的语义，但对原程序的性能有了轻微变化
  - 在源代码中注入、在目标代码中注入
- 采样（Sampling）
  - 以特定的频率观察程序执行的特定时刻所展现出的行为与状态，存储各时刻的快照（Snapshot）
- 系统级插桩
  - 在操作系统、解释器、框架中增加代码来收集上层程序的信息

本讲概览

- 性能度量指标
  - 时间性能
  - 空间性能
- 高效的算法设计
  - 枚举、递推与递归、动态规划、贪心
  - 二分、深度优先搜索、广度优先搜索
- 代码调优
  - 常用技巧
  - 面向性能的设计模式
    - 单例模式

算法的分类

- 根据应用分
  - 基本算法，数据结构相关算法，几何算法，图论算法，规划算法，数值分析算法，加密解密算法，排序算法，查找算法，并行算法，数值算法
- 根据确定性分
  - 确定性算法：有限时间内完成，得到结果唯一
  - 非确定性算法：有限时间内完成，得到结果不唯一，存在多值性
- 根据算法的思路分
  - 递推算法，递归算法，穷举算法，贪心算法，分治算法，动态规划算法，迭代算法

算法的性能评价

- 时间复杂度：执行算法所需要的计算工作量
  - 一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示。
  - 若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n))，称O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度
- 空间复杂度：执行算法所需要的内存空间
  - 程序保存所需要的存储空间，也就是程序的大小
  - 程序在执行过程中所需要消耗的存储空间资源，如程序在执行过程中使用的变量等

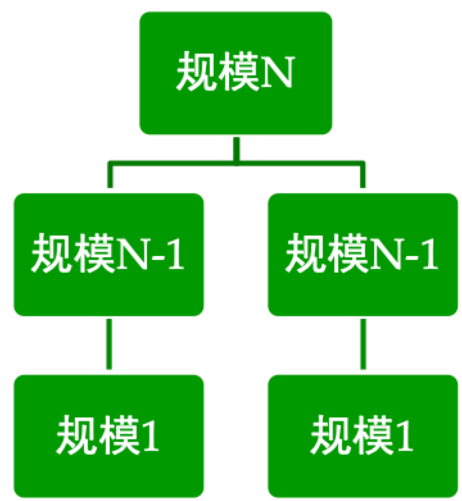
枚举

- 枚举算法是在分析问题时，逐个列举出所有可能情况，然后根据条件判断此答案是否合适，合适就保留，不合适就丢弃，最后得出一般结论
  - 枚举法是通过牺牲时间来换取答案的全面性
- 方法步骤
  - 确定枚举对象、枚举范围、判断条件
  - 循环验证每一个解
    - 对问题可能解集合的每一项
    - 根据问题给定的检验条件判哪些是成立
    - 使条件成立的即是问题解
- 枚举算法的优化
  - 建立简洁的数学模型（变量数尽可能少且相互独立）
  - 减少搜索的空间（减少循环次数和循环层数）
  - 采用合适的搜索顺序

递推与递归

- 递推是从已知的初始条件出发，逐步逼近目标结果，每次逼近均产生新的结果
- 递归是从目标结果（未知项）出发，逐步逼近预期的递归边界（已知项），每次逼近不产生结果，最后利用返回值逐层得出未知项

如果规模增长是单一路径，则均可递推与递归。此时区别在于从小到大还是从大到小



如果规模增长不是单一路径，则只能用递归。

动态规划

- 动态规划是求解包含重叠子问题的最优化方法
- 基本思想
  - 将原问题分解为相似的子问题
  - 在求解过程中通过保存子问题的解求出原问题的解
  - 只能应用于有最优子结构的问题（即局部最优解能决定全局最优解）
- 一般思路
  - 将原问题分解为子问题
  - 确定状态（将和子问题相关的各个变量的一组取值，称之为一个“状态”）
  - 确定一些初始状态（边界状态）的值
  - 确定状态转移方程

记忆递归型

- 优点：只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费
- 缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。空间优化方面比较困难。总体来说，比递推型慢

“我为人人”递推型

- 状态i的值F_i在被更新的时候, 依据F_i去更新和状态i相关的其他一些状态的值F_k，F_m，…，F_y
- 没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比“人人为我”型节省空间

“人人为我”递推型

- 状态i的值F_i由若干个值已知的状态值F_m，F_n，…，F_y推出
- 在选取最优备选状态的值F_m, F_n, . . . , F_y 时，有可能有好的算法或数据结构可以用来显著降低时间复杂度

- 在对问题求解时，总是做出在当前看来是最好的选择
  - 不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解
- 贪心算法要求所求问题的整体最优解可以通过一系列局部最优的选择
- 基本思路
  - 从问题的某一个初始解出发一步一步地进行
  - 根据某个优化测度，每一步都要确保能获得局部最优解。
  - 每一步只考虑一个数据，他的选取应该满足局部优化的条件；若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中
  - 直到把所有数据枚举完，或者不能再添加算法停止。

- 思考：
  - A 心里想一个1-1000之间的数，B 来猜，可以问问题，A 只能回答是或否。怎么猜才能问的问题次数最少？

- 二分算法，又称折半查找，即在一个单调有序的集合中查找一个解。每次分为左右两部分，判断解在哪个部分中并调整上下界，直到找到目标元素，每次二分后都将舍弃一半的查找空间

❑ 例：二分查找函数

➢ 写一个函数BinarySeach，在包含size个元素的、从小到大排序的int数组a里查找元素p，如果找到，则返回元素下标，如果找不到，则返回-1

```
int BinarySearch(int a[],int size,int p) {
    int L = 0;          //查找区间的左端点
    int R = size - 1; //查找区间的右端点
    while(L <= R) {  //如果查找区间不为空就继续查找
        int mid = L+(R-L)/2; //取查找区间正中元素的下标
        if(p == a[mid] )
            return mid;
        else if(p > a[mid])
            L = mid + 1; //设置新的查找区间左端点
        else
            R = mid - 1; //设置新的查找区间右端点
    }
    return -1;
}
```

❑ 例：二分法求方程的根

- 求下面方程的一个根：$f(x) = x^3 - 5x^2 + 10x - 80 = 0$
- 若求出的根是a，则要求$|f(a)| \le 10^{-6}$

❑ 思路：

- 对f(x) 求导，得$f'(x) = 3x^2 - 10x + 10 = 3(x - 5/3)^2 + 5/3 > 0$
- $f'(x)$恒大于0，所以f(x)是单调递增的
- 又由于$f(0) < 0$ 且$f(100) > 0$, 所以区间[0, 100] 内必然有且只有一个根
- 所以可以用二分的办法在区间[0, 100]中寻找根

例：二分法求方程的根

- 求下面方程的一个根：$f(x) = x^3 - 5x^2 + 10x - 80 = 0$
- 若求出的根是a，则要求 $|f(a)| \le 10^{-6}$

```
double EPS = 1e-6;
double f(double x) {
    return x*x*x - 5*x*x + 10*x - 80;
}
int main() {
    double root, x1 = 0, x2 = 100, y;
    root = x1 + (x2 - x1) / 2;
    int triedTimes = 1; //记录一共尝试多少次
    y = f(root);
    while (fabs(y) > EPS) {
        if (y > 0)
            x2 = root;
        else
            x1 = root;
        root = x1 + (x2 - x1) / 2;
        y = f(root);
        triedTimes ++;
    }
    printf("%.8f\n", root);
    printf("%d", triedTimes);
    return 0;
}
```

例：寻找指定和的整数对

- 输入n (n ≤ 100000) 个整数，找出其中的两个数，它们之和等于整数m（假定肯定有解）
- 题中所有整数都能用int 表示

解法1：用两重循环，枚举所有的取数方法

- 复杂度是$O(n^2)$，超时！

```
for (int i = 0;i < n - 1; ++i) {
    for (int j = i + 1; j < n; ++j) {
        if (a[i] + a[j] == m) break;
    }
}
```

例：寻找指定和的整数对

- 输入n (n ≤ 100000) 个整数，找出其中的两个数，它们之和等于整数m（假定肯定有解）
- 题中所有整数都能用int 表示

解法2

- 将数组排序，复杂度是O(n log n)
- 对数组中的每个元素a[i]，在数组中二分查找m−a[i]，看能否找到。复杂度log n，最坏要查找n – 2 次，所以查找这部分的复杂度也是O(n log n)
- 总的复杂度是O(n log n) 的

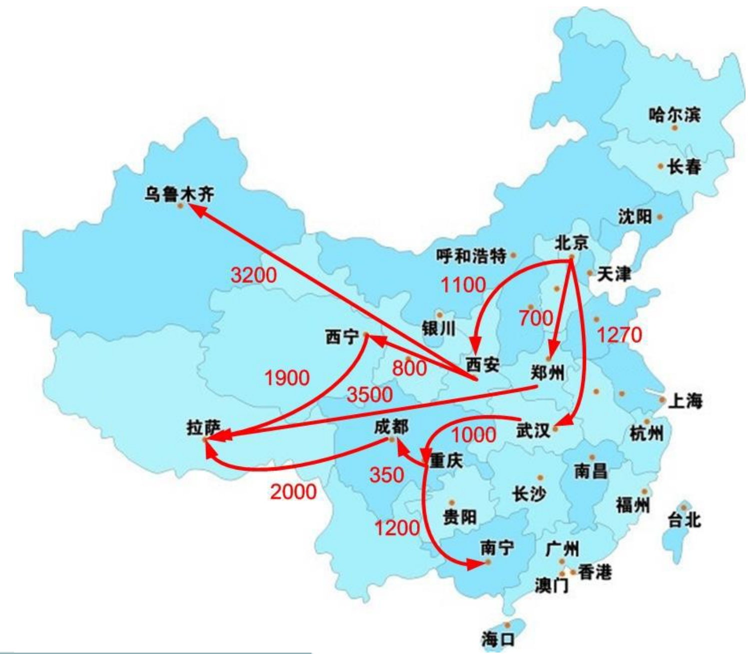
❑ 例：寻找指定和的整数对

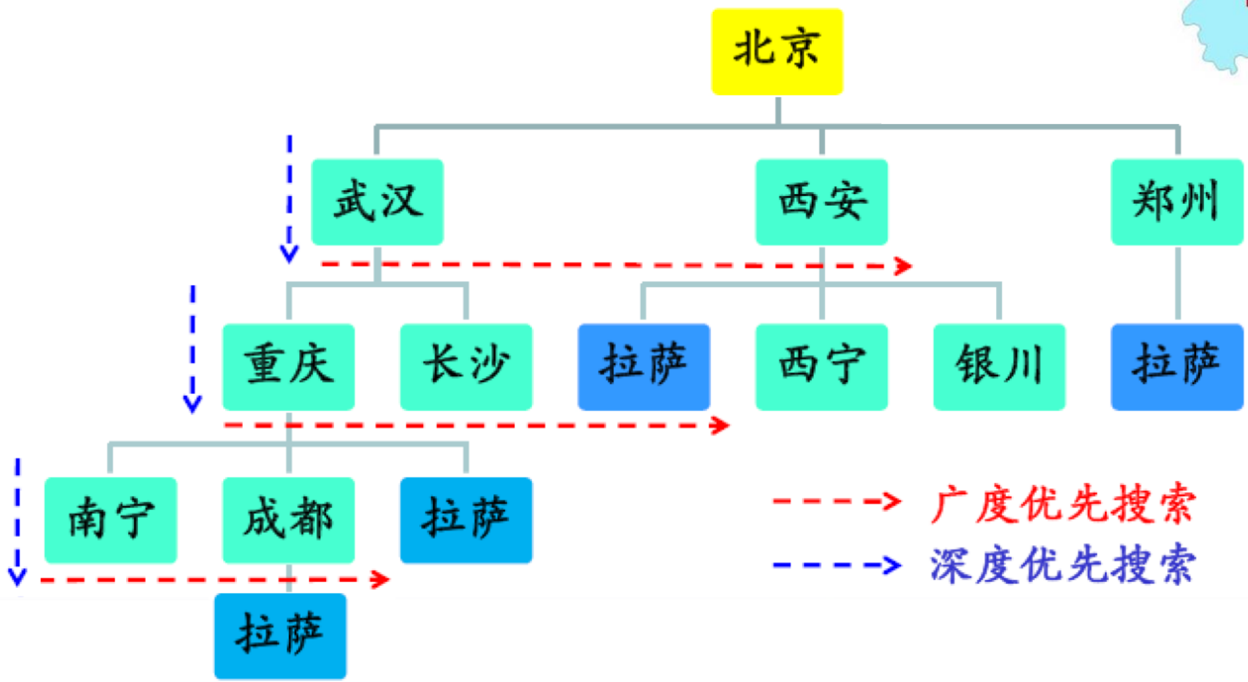
- 输入n (n ≤ 100000) 个整数，找出其中的两个数，它们之和等于整数m（假定肯定有解）
- 题中所有整数都能用int 表示

❑ 解法3

- 将数组排序，复杂度是O(n log n)
- 查找的时候，设置两个变量i和j， i 初值是0，j 初值是n−1
- 看a[i]+a[j]，如果大于m，就让j减1，如果小于m, 就让i加1，直至a[i] + a[j] = m
- 总的复杂度是O(n log n) 的

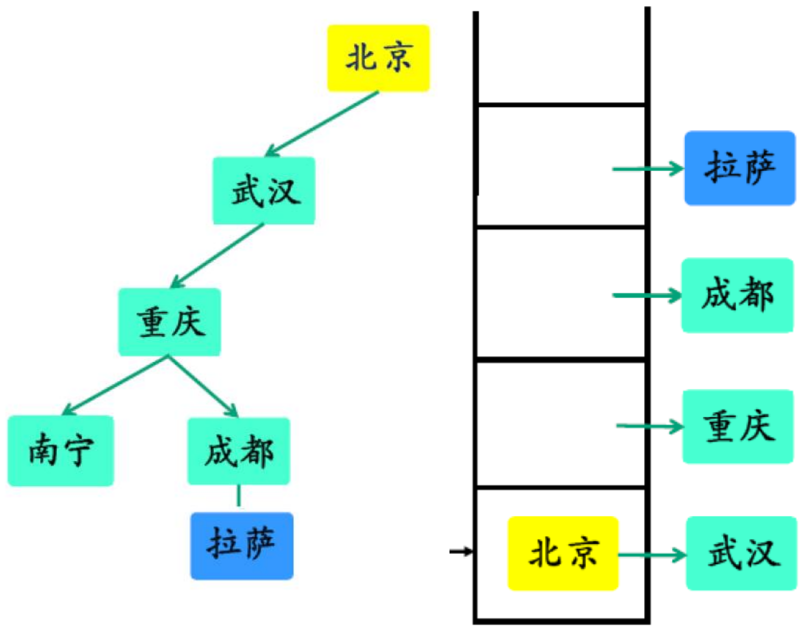
- 生活中的搜索问题
  - 预订从北京到拉萨的航班
  - 没有直飞的航班怎么办？
- 搜索树→状态空间
- 不同的搜索顺序





❑ 深度优先搜索（Depth-First-Search，DFS）

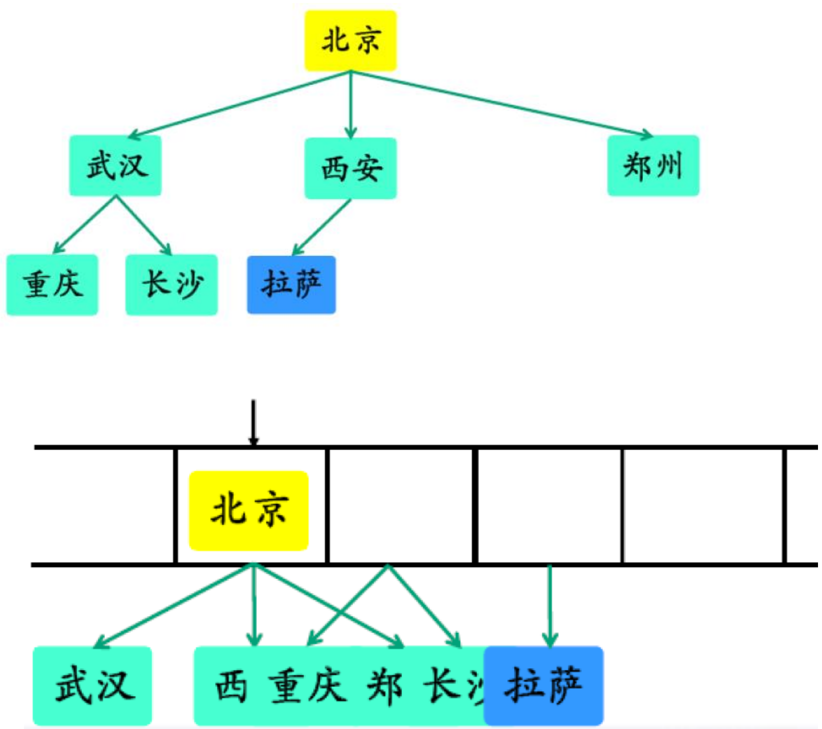
- 优先深入遍历靠前的结点
- 可以用栈实现
- 在栈中保存从起始节点到当前结点路径上的所有结点




```
DFS() {
  初始化栈;
  while (栈不为空&未找到目标结点) {
    取栈顶结点扩展，扩展出的结点放回栈顶;
  }
  ……
}
```

❑ 广度优先搜索（Breadth-First-Search， BFS）

- 优先扩展浅层结点，逐渐深入
- 可以用队列保存待扩展的结点
- 从队首取出结点，扩展出的新结点放入队尾，直到找到目标结点[问题的解]




```
BFS() {
  初始化队列;
  while (队列不为空&未找到目标结点) {
    取队首结点扩展,并将扩展出的结点放入队尾;
    必要时要记住每个结点的父结点;
  }
}
```

枚举与搜索

- 枚举：逐一判断所有可能的方案是否为问题解
  - 解空间中的每个元素是一动作的集合 F
- 搜索：高级枚举，有顺序策略地枚举状态空间中的结点
  - 解空间的每个元素是一动作的序列 F

采用递归的策略进行搜索

- 两个状态的集合
  - α: 未处理完的状态
  - β: 已处理的状态
- 状态的处理：有顺序的尝试备选动作，每一次的尝试都演化出另外一个状态
  - 已处理的状态：全部备选动作都已经尝试
- 树结构：状态之间的演化关系
- 递归的出口
  - α为空
  - 演化出目标状态 S*
  - 演化出的状态属于 α ∪ β

❑影响搜索效率的因素

- 状态空间
  - α: 未处理完的状态
  - β: 已处理的状态
- 判重：每次演化出一个状态s时，s 是否属于α 或者β
- 剪枝：状态s 的任意演化结果是否都属于β
- 演化出来的状态数量：$\alpha \cup \beta$ 的大小

❑ 例：城堡问题

- ➢ 右图是一个城堡的地形图。请你编写一个程序，计算城堡一共有多少房间，最大的房间有多大。
- ➢ 城堡被分割成m × n(m ≤ 50, n ≤ 50) 个方块，每个方块可以有0 ~ 4 面墙

```
   1   2   3   4   5   6   7
  #############################
1 #   |   #   |   #   |   |   #
  #####---#####---#---#####---#
2 #   #   |   #   #   #   #   #
  #---#####---#####---#####---#
3 #   |   |   #   #   #   #   #
  #---#########---#####---#---#
4 #   #   |   |   |   |   #   #
  #############################
             (图 1)

  #  = Wall
  |  = No wall
  -  = No wall
```

❑ 思路：

- ➢ 对每一个房间，扩展相邻的房间，从而给这个房间能够到达的所有位置染色
- ➢ 最后统计一共用了几种颜色，以及每种颜色的数量

1	1	2	2	3	3	3
1	1	1	2	3	4	3
1	1	1	5	3	5	3
1	5	5	5	5	5	3

例：Roads

- N 个城市，编号1到N。城市间有R条单向道路。
- 每条道路连接两个城市，有长度和过路费两个属性。
- Bob只有K块钱，他想从城市1到城市N。
- 问最短共需要走多长的路。如果到不了N，输出-1

$2 \le N \le 100$
$0 \le K \le 10000$
$1 \le R \le 10000$
每条路的长度 L, $1 \le L \le 100$
每条路的过路费 T , $0 \le T \le 100$

基本思路：

- 从城市1开始深度优先遍历整个图
- 找到所有能到达N的走法，选一个最优

```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;
int K,N,R;
struct Road {
    int d,L,t;
};
vector<vector<Road>> cityMap(110);//邻接表cityMap[i]表示i有路连到的城市集合
int minLen = 1 << 30; //当前找到的最优路径的长度
int totalLen; //正在走的路径的长度
int totalCost ; //正在走的路径的花销
int visited[110]; //城市是否已经走过的标记
```

- 例：Roads
- 基本思路：
  - 从城市1开始深度优先遍历整个图
  - 找到所有能到达N的走法，选一个最优

```
void dfs(int s) { //从s开始向N行走
    if (s == N) {
        minLen = min(minLen, totalLen);
        return;
    }
    for (int i = 0 ;i < cityMap[s].size(); ++i){
        int d = cityMap[s][i].d; //s有路连到d
        if (visited[d]) continue;
        int cost = totalCost + cityMap[s][i].t;
        if (cost > K) continue;
        totalLen += cityMap[s][i].L;
        totalCost += cityMap[s][i].t;
        visited[d] = 1;
        dfs(d);
        visited[d] = 0;
        totalCost -= cityMap[s][i].t;
        totalLen -= cityMap[s][i].L;
    }
}
```

```
int main() {
    cin >>K >> N >> R;
    for (int i = 0;i < R; ++i) {
        int s;
        Road r;
        cin >> s >> r.d >> r.L >> r.t;
        if( s != r.d ) cityMap[s].push_back(r);
    }
    memset(visited,0,sizeof(visited));
    totalLen = 0;
    totalCost = 0;
    visited[1] = 1;
    minLen = 1 << 30;
    dfs(1);
    if (minLen < (1 << 30))
        cout << minLen << endl;
    else
        cout << "-1" << endl;
    return 0;
}
```

- 例：Roads
- 最优性剪枝：
  - 如果当前已经找到的最优路径长度为L ,那么在继续搜索的过程中，总长度已经大于等于L 的走法，就可以直接放弃，不用走到底了
  - 保存中间计算结果用于最优性剪枝：
    - 用minL[k][m]表示：走到城市k时总过路费为m的条件下，最优路径的长度。
    - 若在后续的搜索中，再次走到k时，如果总路费恰好为m，且此时的路径长度已经不小于minL[k][m]，则不必再走下去了

❑ 例：Roads

❑ 最优性剪枝：

```
int minL[110][10100];
//minL[i][j]表示1到i点的、花销为j的最短路径长度
void dfs(int s) {
    if (s == N) {
        minLen = min(minLen, totalLen);
        return;}
    for (int i = 0 ;i < cityMap[s].size(); ++i) {
    int d = cityMap[s][i].d;
    if (visited[d]) continue;
    int cost = totalCost + cityMap[s][i].t;
    if (cost > K) continue;
    if (totalLen + cityMap[s][i].L >= minLen)
       continue;//剪枝1
    if (totalLen + cityMap[s][i].L >= minL[d][cost])
       continue;//剪枝2
    totalLen += cityMap[s][i].L;
    totalCost += cityMap[s][i].t;
    minL[d][cost] = totalLen;
    visited[d] = 1;
    dfs(d);
    visited[d] = 0;
    totalCost -= cityMap[s][i].t;
    totalLen -= cityMap[s][i].L;
    }
}
```

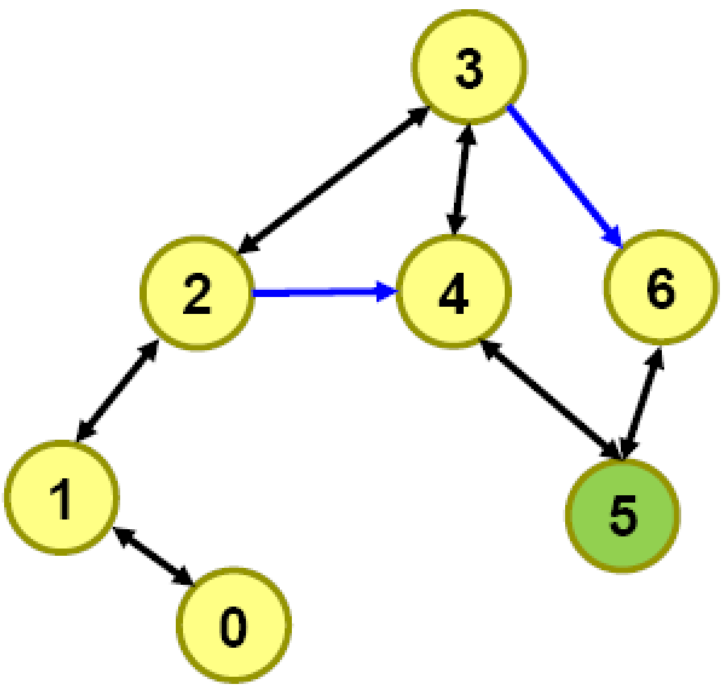
```
int main() {
    cin >>K >> N >> R;
    for (int i = 0;i < R; ++i) {
        int s;
        Road r;
        cin >> s >> r.d >> r.L >> r.t;
        if( s != r.d )
cityMap[s].push_back(r);
    }
    for (int i = 0;i < 110; ++i)
        for (int j = 0; j < 10100; ++j)
            minL[i][j] = 1 << 30;
    memset(visited,0,sizeof(visited));
    totalLen = 0;
    totalCost = 0;
    visited[1] = 1;
    minLen = 1 << 30;
    dfs(1);
    if (minLen < (1 << 30))
        cout << minLen << endl;
    else
        cout << "-1" << endl;
    return 0;
}
```

❑ 例：抓住那头牛

- 农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 $N(0 \le N \le 100000)$，牛位于点 $K(0 \le K \le 100000)$。农夫有两种移动方式
  - 从X 移动到X－1 或X＋1，每次移动花费一分钟
  - 从X 移动到2＊X，每次移动花费一分钟
- 假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

例：抓住那头牛

- 分析：假设农夫起始位于点3，牛位于5，最右边是6，如何搜索到一条走到5的路径？



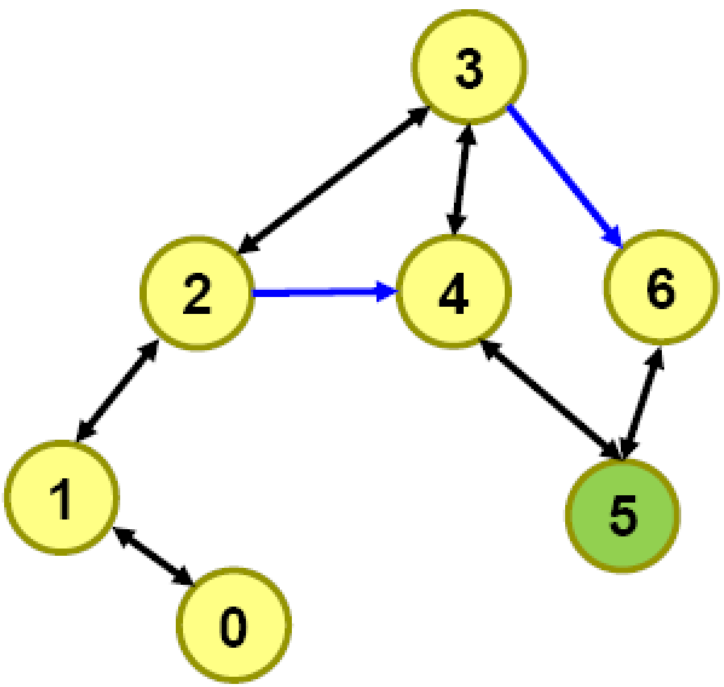
- 策略1：深度优先搜索
  - 从起点出发，随机挑一个方向，能往前走就往前走（扩展），走不动了则回溯。不能走已经走过的点（要判重）。

运气好的话：3->4->5 或3->6->5 问题解决！
运气不太好的话：3->2->4->5
运气最坏的话：3->2->1->0->4->5

  - 要想求最优解，则要遍历所有走法。可以用各种手段优化，比如，若已经找到路径长度为n 的解，则所有长度大于n 的走法就不必尝试

❑ 例：抓住那头牛

➢ 分析：假设农夫起始位于点3，牛位于5，最右边是6，如何搜索到一条走到5的路径？



➢ 策略2：广度优先搜索

- 给节点分层。起点是第0层。从起点最少需n步就能到达的点属于第n层

  第1层：2, 4, 6
  第2层：1, 5
  第3层：0
- 依层次顺序，从小到大扩展节点。把层次低的点全部扩展出来后，才会扩展层次高的点。扩展时，不能扩展出已经走过的节点（要判重）
- 可确保找到最优解，但是因扩展出来的节点较多，且多数节点都需要保存，因此需要的存储空间较大

例：抓住那头牛

```
#include <iostream>
#include <cstring>
#include <queue>
using namespace std;

int N, K;
const int MAXN = 100000;
int visited[MAXN + 10];
//判重标记visited[i]=1，表示i已被扩展过
struct Step {
  int x;      //位置
  int steps;  //到达x所需的步数
  Step(int xx, int s):x(xx), steps(s){}
};

queue<Step> q; //队列

int main() {
  cin >> N >> K;
  memset(visited, 0, sizeof(visited));
  q.push(Step(N, 0));
  visited[N] = 1;
```

```
  while (!q.empty()) {
      Step s = q.front();
      q.pop();
      if (s.x == K) { //找到目标
          cout << s.steps << endl;
          return 0;
      }
      if (s.x - 1 >= 0 && !visited[s.x - 1]) {
          q.push(Step(s.x - 1, s.steps + 1));
          visited[s.x - 1] = 1;
      }
      if (s.x + 1 <= MAXN && !visited[s.x + 1]) {
          q.push(Step(s.x + 1, s.steps + 1));
          visited[s.x + 1] = 1;
      }
      if (s.x * 2 <= MAXN && !visited[s.x * 2]) {
          q.push(Step(s.x * 2, s.steps + 1));
          visited[s.x * 2] = 1;
      }
  }
  return 0;
}
```

深搜与广搜的比较

- 广搜一般用于状态表示比较简单、求最优策略的问题
  - 优点：是一种完备策略，即只要问题有解，它就一定可以找到解。并且，广度优先搜索找到的解，还一定是路径最短的解。
  - 缺点：盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用的节点，因此其搜索效率较低。需要保存所有扩展出的状态，占用的空间大。空间换时间
- 深搜几乎可以用于任何问题
  - 只需要保存从起始状态到当前状态路径上的节点, 时间换空间

本讲概览

- 性能度量指标
  - 时间性能
  - 空间性能
- 高效的算法设计
  - 枚举、递推与递归、动态规划、贪心
  - 二分、深度优先搜索、广度优先搜索
- 代码调优
  - 常用技巧
  - 面向性能的设计模式
    - 单例模式

代码调优

- 代码调优（Code Tuning）不是为了修复bug，而是对正确的代码进行修改以提高其性能
  - 直到其他方面都已经无法再优化了，再考虑代码调优
    - 优先级顺序：需求→系统设计→类和方法的设计（算法）→与操作系统的交互→编译优化→硬件优化→代码调优
  - 代码调优不会减少代码行数
  - 不要边写程序边调优
  - 不要猜原因，而应有明确的优化目标
- 帕累托法则（Pareto Principle）
  - 又称“二八定律”(80/20 rule)
  - 在任何一组东西中，最重要的只占其中一小部分，约20%，其余80%尽管是多数，却是次要的
  - 一个典型的程序有80%的执行时间花费在20%的代码身上
  - 代码调优的目标是针对这20%的代码

调优的一般步骤

- 备份已有代码
- 通过度量发现热点瓶颈
- 分析原因，评判代码调优的必要性
- 调优
- 每次调优后都要重新度量
- 若无效果或负效果，则回滚
- 重复上述过程

常用技巧

- 尽早结束循环：当需要遍历寻找首个符合条件的元素或判断是否符合某种性质时，可通过break语句及时跳出循环

```
bool found = false;
for (int i = 0; i < iCount; i++ ) {
    if (input[i] < 0 ) {
        found = true;
    }
}
```

```
bool found = false;
for (int i = 0; i < iCount; i++ ) {
    if (input[i] < 0 ) {
        found = true;
        break;
    }
}
```

- 将判断移到循环外：如果循环内分支语句的判断结果不会因循环而发生改变，应该将分支语句移到循环外

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET )
      netSum = netSum + amount[ i ];
   else
      grossSum = grossSum + amount[ i ];
}
```

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ )
      netsum = netsum + amount[i];
}
else {
   for ( i = 0: i < count; i++ )
      grossSum = grossSum + amount[i];
}
```

查表法：将代码中复杂的if-else和switch-case语句从代码中分离出来，通过“查表”的方式完成

```
if ( ( a && !c ) || ( a && b && c ) )
    category = 1;
else if ( ( b && !a ) || ( a & c & !b ) )
    category = 2;
else if ( c && !a && !b )
    category = 3;
else
    category = 0;
```

```
int categoryTable[2][2][2] = {
  // !b!c !bc  b!c  bc
       0,   3,   2,   2,     // !a
       1,   2,   1,   1      // a
};
category = categoryTable[a][b][c];
```

直接访问表

根据下标直接查找值

```
int dPerMonth[] =
{ 31, 28, 31, 30, 31,
30, 31, 31, 30, 31,
30, 31 };

days=dPerMonth[mon-1];
```

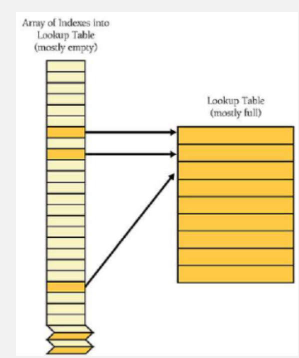
阶梯访问表

表条目对一定范围的数据有效，而不对特定值有效

```
int gradeLimits[] = {60, 70, 80, 90, 100};
string grades[] = {"F", "D", "C", "B", "A"};
string stuGrade = "A";
int gradeLevel = 0, maxGradeLevel = 5, score;
cin >> score;
while ((stuGrade == "A") && (gradeLevel <
maxGradeLevel)) {
    if (score < gradeLimits[gradeLevel])
        stuGrade = grades[gradeLevel];
    gradeLevel++;
}
```

索引访问表

类似哈希表，将一个大范围的值映射到一个小范围



- 惰性计算（Lazy Evaluation）：表达式在第一次使用之前不求值，而是将求值推迟到真正需要时
  - 引用计数：除非确实需要，否则不为任何东西制作拷贝

```
class String { ... };  // 一个String 类
String s1 = "Hello";
String s2 = s1;        // 调用String的复制构造函数，开销大
```

- 让s1和s2共享一个值，只须做一些记录以便知道谁在共享什么，就能够省掉调用new和拷贝字符的开销
- 仅仅某一个String的值被修改时，共享同一个值的方法才会造成差异

```
s2.convertToUpperCase();
```

- convertToUpperCase函数应该制作s2值的一个拷贝，在修改前把这个私有的值赋给s2

  - 区别对待读取和写入
    - 利用代理类推迟做出是读操作还是写操作的决定

```
String s = "Homer's Iliad";   // 假设是一个引用计数的String
cout << s[3];                 // 调用 operator[] 读取s[3]
s[3] = 'x';                   // 调用 operator[] 写入s[3]
```

- 惰性计算（Lazy Evaluation）：表达式在第一次使用之前不求值，而是将求值推迟到真正需要时
  - 惰性提取
    - 假如程序中使用了一些包含很多字段的大型对象。这些对象的生存期超越了程序运行期，所以它们必须被存储在数据库里。每一个对象都有一个唯一的对象标识符，用来从数据库中重新获取对象

```
class LargeObejct{                  //大型持久对象
public:
    LargeObejct(ObjectID id);       //从磁盘中恢复对象
    const string& field1() const;   // field1的值
    int field2() const;             // feild2的值
    double field3() const;
    const string& field4() const;
    ...
};
```

    - 下述程序只用到field2()的值，为此获取其他字段的努力都是浪费的

```
void restoreAndProcessObject(ObjectID id){
    LargeObject object(id);
    if(object.field2() == 0)
        cout << "Object " << id << ": null field2\n";
}
```

    - 解决方案：每个字段都用一个指向数据的指针来表示，初始化为空，被访问时再获取对应的值

- 惰性计算（Lazy Evaluation）：表达式在第一次使用之前不求值，而是将求值推迟到真正需要时
  - 惰性表达式计算
    - 考虑一个矩阵类和矩阵的求和运算

```
class Matrix { ... };
Matrix m1(1000, 1000);    // 一个1000*1000的矩阵
Matrix m2(1000, 1000);    // 同上
...
Matrix m3 = m1 + m2;      // 计算矩阵的和
```

    - 如果只需要计算结果的一部分

```
cout << m3[4];
```

    - 可以等m3被用的时候再进行计算，并且只计算被用到的部分
    - 解决方案：建立一个数据结构来表示m3的值是m1与m2的和

惰性计算（Lazy Evaluation）：表达式在第一次使用之前不求值，而是将求值推迟到真正需要时

惰性表达式计算

- 示例

```
using matrix = std::array<int, 2>;
struct matrix_add {
    matrix_add(const matrix& a, const matrix& b) :
a_(a), b_(b) { cout<<"do nothing\n"; }
    // 从matrix_add到plain matrix的隐式转换
    operator matrix() const {
        cout <<"add operation\n";
        matrix result;
        for (int i = 0; i < 2; ++i)
            result[i] = a_[i] + b_[i];
        return result;
    }
    // 计算一个元素的加法
    int operator ()(unsigned int index) const {
        cout << "calculate *just one* element\n";
        return a_[index] + b_[index];
    }
private:
    const matrix& a_, b_;
};
```

```
matrix_add operator + (const
matrix& a, const matrix& b) {
    return matrix_add(a, b);
}

int main() {
    matrix mat1 = {2, 3}
    matrix mat2 = {7, 8};
    auto ret = mat1 + mat2;
    cout << "....";
    matrix mat3(ret); // 类型转换
    cout << ret(1);
    return 0;
}
```

单例模式

- 某些类在应用运行期间只需要一个实例
  - 强制调用方只能创建一个实例，避免因为new操作所带来的时空性能的损失，也便于复用
- 单例（Singleton）模式是一个类只创建唯一的对象
  - 基本实现步骤：
    - 构造函数私有化
    - 增加静态私有的当前类的指针变量
    - 提供静态对外接口,可以让用户获得单例对象

```
class A {
public:
    static A* getInstace() {
        return a;
    }
private :
    A() {
        a = new A;
    }
    static A* a;
};
A* A::a = nullptr;
```

饿汉式实现

- 对象在程序执行时优先创建
- 问题：在对象过多时会造成内存浪费

```
class SingletonHungry {
public:
    static SingletonHungry* getInstance() {
        return pSingleton;
    }
    static void freeSpace() {
        if (pSingleton != nullptr) {
            delete pSingleton;
        }
    }
private:
    SingletonHungry() {}
    static SingletonHungry* pSingleton;
};
//以下语句将会在main函数运行前执行
SingletonHungry* SingletonHungry::pSingleton=new SingletonHungry;
```

懒汉式实现

- 对象的创建在第一次调用getInstance函数时创建
- 问题
  - 内存泄露，可使用智能指针
  - 线程不安全，可以通过mutex.lock()和mutex.unlock()来解决

```
class SingletonLazy {
public:
    static SingletonLazy* getInstance() {
        if (pSingleton == NULL) {
            pSingleton = new SingletonLazy;
        }
        return pSingleton;
    }
private:
    SingletonLazy() {}
    static SingletonLazy* pSingleton;
};
//在类外面进行初始化
SingletonLazy* SingletonLazy::pSingleton=nullptr;
```

```
class Singleton {
private:
    Singleton() { };
    ~Singleton() { };
    Singleton(const Singleton&);
    Singleton& operator=(const
Singleton&);
public:
    static Singleton& getInstance()
        {
        static Singleton instance;
        return instance;
    }
};
```

Meyers' Singleton

上机安排/作业九

- 6月3日/6月10日 13:00—15:00 理一1235机房

- 上机安排
  - 算法题目练习（不计分）
  - 大作业答疑与面测

- 作业九（最后一次作业）
  - 时间：6月3日 12:00—6月18日 23:59
  - 从教学网“课程作业”栏目下载“作业九”文档，完成后在教学网提交
    - 可提交手写拍照版本

谢谢

欢迎在课程群里填写问卷反馈