HW06 分支限界法 实验报告

日期：2024 年 5 月 25 日

1. 背景知识

本次作业涉及分支限界法的相关知识和应用。分支限界法类似于回溯法，也是一种在问题的解空间树 T 中搜索问题解的算法。分支限界法通常用广度优先的方式搜索问题的解空间树。在遍历的过程中，对已经扩展出的每一个节点根据限界函数估算目标函数的可能取值，从中选取使目标函数取得极值的节点优先进行广度优先搜索，从而不断调整搜索方向，尽快找到问题的解，适用于求解最优化问题。

分支限界法的基本思想：

① 首先，确定一个合理的限界函数，并根据限界函数确定问题的目标函数的界[down,up](具体问题可以只有下界 down，或上界 up)；

② 然后，按照广度优先策略遍历问题的解空间树：当搜索到达一个扩展结点时，一次性扩展它的所有孩子，估算每一个孩子结点的目标函数的可能取值(又称为耗费函数值)；将那些满足约束条件且耗费函数值不超过目标函数的界的孩子，插入活动结点表 PT 中；依次从 PT 表中取耗费函数值极大(小)的下一结点同样扩展，直到找到所需的解或 PT 表为空为止。

③ 对于 PT 中的叶子结点：其耗费函数值是极值(极大或极小)，则该叶子结点对应的解就是问题的最优解；否则，将问题目标函数的界([down,up])调整为该叶子结点的耗费函数值，然后丢弃 PT 表中超出目标函数界的结点，再次选取结点继续扩展。

分支限界法有两种实现方法，一种是队列式，一种则是优先队列式。它与回溯法不同，回溯法是找出满足约束条件的所有解，而分支限界法则是找出满足条件的一个解或者是某种意义下的最优解；且它使用广度优先或最小耗费优先；在分支限界法中，每一个活节点只有一次机会成为扩展节点。

2. 实验内容

2.1. 最小重量机器设计问题（6-4）

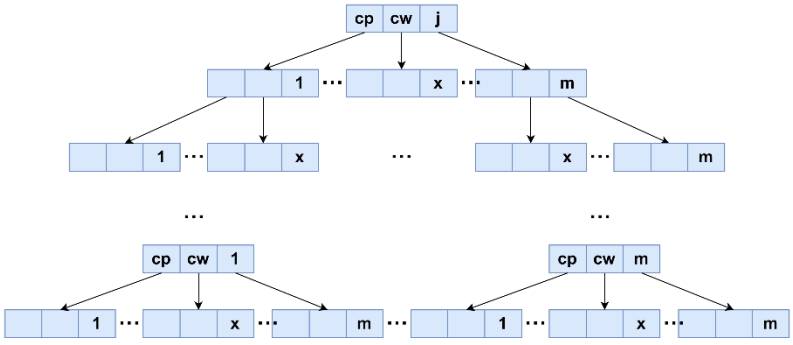
2.1.1. 问题描述

> 设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设$w_{\{ij\}}$是从供应商j处购得的部件i的重量，$c_{\{ij\}}$是相应的价格。试设计一个优先队列式的分支限界算法，给出总价格不超过d的最小重量机器设计。

2.1.2. 问题分析

由第一部分的分析可知，分支限界法和回溯法有些类似，都是通过解空间树来进行问题的搜索与求解。所以可以先分析问题的解空间树。实际上通过之前的练习，针对最小重量机器设计问题，我们可以假定一个一个选择零件，每个零件有 m 个选择。在解空间树中可以使用某层的所有节点来表示对第 i 个零件的所有可能选择，一层一层往下进行，表示对不同零件进行选择，直到达到叶节点，表示对所有零件完成选择。

当完成某个节点的选择后，其下一个零件也有 m 个选择，因此其子节点也会有 m 个，表示下一个零件可以在 m 个供应商中随机选择一个。因此解空间树就是从根节点到某一层节点的路径，记录下了到当前零件为止，前面所有零件的选择方案。



如上图所示，在零件选择时有两个约束条件，一个是零件价格总和需要不超过规定成本 c，这是约束条件；一个是零件重量总和，希望其越小越好，这是限界条件。所以对于每个节点需要存储以下信息，当前的花费（cp），以及当前的重量（cw），除此之外，还要记录一些基本信息，比如零件序号 i，零件厂商序号 j，其中 i 还是解空间树的层序号。

同时为了输出最优选择，也就是最优解，我们需要还原路径，这里有两种方法：

① 每个节点使用 path 数组存下迄今为止所有零件选择的序号，到达叶节点后直接输出即为解的选择方案

② 从叶节点往回遍历，回到根节点，中间每个节点记录了每个零件的选择方案，使用这种方式需要指针记录下父节点。

经过对比分析，虽然第一种在最后输出时效率较高，但是空间开销有些大，所以第二种好一些，每个节点只需要存储父结点的指针即可，在时间和空间上的开销可以承担。

另外为了减少搜索空间，在使用分支限界法时需要剪去已经不符合条件的搜索分支，和之前的回溯等算法思路差不多。具体做法为在将子节点加入队列或者优先队列时，除了约束条件的判断外，还需要额外判断是否满足限界条件，也就是最小重量，来决定是否将其加入队列，当子节点不满足限界条件时会被忽略，其所在子树也不再被搜索，即为剪枝操作。在 BFS 基础上加上限界函数，则会得到基于队列的分支界限函数，若在统一代价搜索 UCS 中加入限界函数，则会得到基于优先队列的分支限界算法，这道题要求基于优先队列，所以我采用优先队列的数据结构进行设计，在优先队列内部排序时，需要使用到重载的运算符来比较节点前后顺序，经过题目分析，可以按照重量先排序，若相等，则按照序号，否则按照厂商序号。

数据结构的设计：

```
struct node
{
	int cw;//当前重量
	int cp;//当前价值
	int i;//当前零件序号
	int j;//厂商序号
	node* parent;//父节点

	node(int cw, int cp, int i, int j, node* parent)
		: cw(cw), cp(cp), i(i), j(j), parent(parent) {}
	bool operator < (const node& a) const {
		if (a.cw != cw)
			return cw > a.cw;
		else if (a.i != i)
			return a.i > i;
		else
			return j > a.j;
	}
};
```

分支限界算法的关键在于设计限界函数，也就是剪枝函数来忽略不再可能产生最优解的分支。设计完这个也就完成了一大半，接下来将在算法设计中详细叙述剪枝函数的设计。

2.1.3. 算法设计

剪枝函数的设计：

当求最小重量的时候，当前可能的最小重量为 min_weight,如果我们根据剪枝函数求出某一分支的下界 lower_bound 值大于 min_weight，该分支能够得到的最小重量一定大于其下界，而当下界大于当前最小重量时，说明分支能够得到的最小重量大于当前最小的重量，那也就意味着这条分支不会再产生最优解，可以进行剪枝。

在剪枝函数设计时，需要考虑界限计算的时间，以及计算出的界限值与实际求解目标的接近程度，时间越短，接近程度越近越好。但两者往往如同时间和空间代价一样，无法兼得。针对本问题，需要计算最小重量，因此需要设计下界函数 lowerBound()，可以直接使用当前已选择零件的全部重量和作为该分支的下界，这样求解时间为 O(1)，较迅速，但是与该分支的真实最小重量接近程度太差。因此可以在已选择零件重量和的基础上，加上剩下未选择零件中最小重量的重量和，这一定不会大于真实的最小重量，为了减少后者的求解时间，可以将每一层剩下零件的最小重量和计算出来并且存在数组 lb 中，可以在

lowerBound 中直接调用，时间复杂度低，但是第一次计算要花费一些时间，可以接受。

```
/**
 * brief: 计算下界
 * \param n
 * \return 返回当前节点的重量加上剩余零件的最小重量
 */
int lowerBound(node tree) {
          //如果当前节点是最后一个节点，返回当前节点的重量
          if (tree.i == n - 1)
                    return tree.cw;
          //返回当前节点的重量加上剩余零件的最小重量
          return tree.cw + lb[tree.i + 1];
}
```

```
// 计算每一行的最小值，用于下界计算
for each row i from 0 to n-1 {
    minWeight = INT_MAX;
    for each column j from 0 to m-1 {
        read weights[i][j]; // 输入重量
        if (minWeight > weights[i][j]) {
            minWeight = weights[i][j]; // 找到最小重量
        }
    }
    rowBounds[i] = minWeight; // 将最小重量赋值给 rowBounds[i]
}

// 计算行下界的累加和
for i from n-2 down to 0 {
    rowBounds[i] += rowBounds[i + 1] + rowBounds[i];
}
```

计算出下界以后，就可以在分支限界函数中应用剪枝策略，具体如下：

① 将起始节点放入优先队列

② 重复以下步骤，直到优先队列为空：从优先队列中取出一个节点作为当前节点；如果当前节点不是叶节点，将当前节点的所有满足约束条件的子节点放入优先队列，并自动根据指定的排序函数进行优先级调整；如果当前节点是叶节点，更新最优解。

③ 当优先队列为空时，输出最优解。

在实现时还要注意要申请内存，否则临时变量每次循环结束都会被释放，导致节点丢失无法还原。分支限界函数的代码设计如下：

```
function BranchBound():
    pq = priority_queue<node> // 优先队列，用于存储待处理的节点
    root = node(0, 0, -1, 0, NULL)
    pq.push(root) // 将根节点压入优先队列
while pq is not empty:
        // 取出当前优先队列中的顶部节点（具有最小的估价值）
        curNode = pq.top()
        pq.pop()
        if curNode.i == n - 1: // 如果当前节点已经是最后一步
            // 如果当前节点的重量小于当前最优解且代价不超过限制
            if curNode.cw < bestw and curNode.cp <= d:
                bestw = curNode.cw // 更新最优解的重量
                // 从当前节点回溯到根节点，记录最优解的路径
                for k from curNode to root:
                    bestx[k.i] = k.j + 1 // 记录最优解的机器分配情况
        else: // 如果当前节点不是最后一步
            for j from 0 to m-1: // 遍历所有可选的机器
                child = node(curNode.cw + w[curNode.i + 1][j], curNode.cp + c[curNode.i +
                             1][j], curNode.i + 1, j, curNode) // 创建子节点
                // 如果子节点代价不超过限制且下界小于当前最优解
                if child.cp <= d and lowerBound(child) < bestw:
                    pq.push(child) // 将子节点加入优先队列以待处理
    if bestw == INT_MAX: // 如果找不到可行解
        print "No solution"
    else: // 否则
```

```
        print bestw // 输出最优解的重量
        for i from 0 to n-1: // 输出最优解的机器分配情况
            print bestx[i]
```

复杂度分析：

时间复杂度：

初始化函数$init_MinWeight$包含两个嵌套循环，既有数组的初始化，也有 $lowerBound$ 的计算，时间复杂度为$O(n*m)$；

$lowerBound$函数用于计算下界，主要调用已经计算好的 lb 数组，所以时间复杂度为$O(1)$，也没有空间复杂度。

最主要的$BranchBound()$分支限界函数，在最坏情况下，每个节点都要被遍历一次，并且优先队列的插入操作的时间复杂度为$O(logn)$，因此总的时间复杂度为$O(b\^d * logn)$，其中 b 为分支因子，d 为树的深度。

空间复杂度：

$init_MinWeight$除了存储输入数据的$vector\ w$ 和 c 之外，还有 $bestx$ 和 lb 向量，其大小均为 n，因此空间复杂度为$O(n)$；

除了优先队列外，还有节点对象的创建和存储，最坏情况下，优先队列中可能存储所有节点，因此空间复杂度为$O(b\^d)$。

总体来说，分支限界算法的时间复杂度主要取决于搜索树的大小，而空间复杂度主要取决于优先队列和节点的存储。

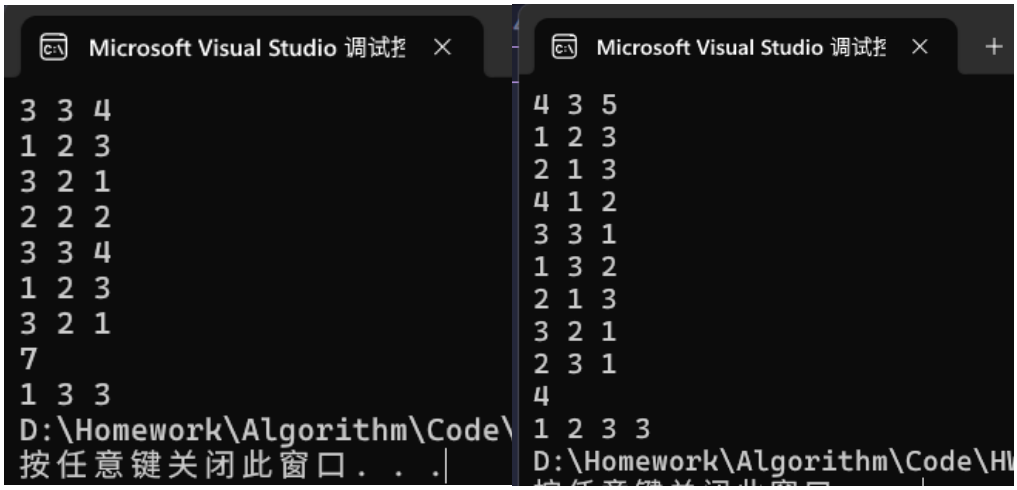
2.1.4. 算法代码

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
struct node
{
    int cw;//当前重量
    int cp;//当前价值
    int i;//当前零件序号
    int j;//厂商序号
    node* parent;//父节点
    node(int cw, int cp, int i, int j, node* parent)
        : cw(cw), cp(cp), i(i), j(j), parent(parent) {}
    bool operator < (const node& a) const {
        if (a.cw != cw)
            return cw > a.cw;
        else if (a.i != i)
            return a.i > i;
        else
            return j > a.j;
    }
};
int n, m, d;//n 表示零件数，m 表示厂商数，d 表示最大重量
int bestw;//最优重量
//lb[i]表示第 i 个零件到第 n 个零件的最小重量
vector<int>lb;
//bestx[i]表示第 i 个零件由第 bestx[i]个厂商加工
vector<int>bestx;
//w[i][j]表示第 i 个零件在第 j 个厂商的重量，c[i][j]表示第 i 个零件在第 j 个厂商的价值
vector<vector<int>>w, c;
void init_MinWeight() {
    cin >> n >> m >> d;
    bestx.resize(n);
    lb.resize(n);
    w.resize(n);
    c.resize(n);
    for (int i = 0; i < n; i++) {
        w[i].resize(m);
        c[i].resize(m);
    }
    for (int i = 0; i < n; ++i) {
```

```
        for (int j = 0; j < m; ++j) {
            cin >> c[i][j];//输入价值
        }
    }
    //计算 c[i][j]的累加和
    for (int i = 0; i < n; i++) {
        int minw = INT_MAX;     //初始化最小重量为最大值
        for (int j = 0; j < m; j++) {
            cin >> w[i][j];//输入重量
            if (minw > w[i][j])
                minw = w[i][j];//找到最小重量
        }
        lb[i] = minw;//将最小重量赋值给 lb[i]
    }
    //计算 lb[i]的累加和
    for (int i = n - 2; i >= 0; --i) {
        lb[i] += lb[i + 1] + lb[i];
    }
    bestw = INT_MAX;
}
/*
 * brief: 计算下界
 * \param n
 * \return 返回当前节点的重量加上剩余零件的最小重量
 */
int lowerBound(node tree) {
    //如果当前节点是最后一个节点，返回当前节点的重量
    if (tree.i == n - 1)
        return tree.cw;
    //返回当前节点的重量加上剩余零件的最小重量
    return tree.cw + lb[tree.i + 1];
}
/**
 * brief: 分支限界法
 */
void BranchBound()
{
    priority_queue<node> pq;//优先队列
    node root(0, 0, -1, 0, NULL);//初始化 root 节点
    pq.push(root);//将 root 节点压入优先队列

    while (!pq.empty())
    {
        //取出当前节点
        node* curNode = new node(pq.top());
        pq.pop();

        if (curNode->i == n - 1) {
            if (curNode->cw < bestw && curNode->cp <= d) {
                bestw = curNode->cw;
                for (node* k = curNode; k->i != -1; k = k->parent) {
                    bestx[k->i] = k->j + 1;//记录最优解
                }
            }
        }
        else {
            for (int j = 00; j < m; ++j) {
                node child(curNode->cw + w[curNode->i + 1][j], curNode->cp + c[curNode->i
+ 1][j], curNode->i + 1, j, curNode);
                if (child.cp <= d && lowerBound(child) < bestw)
                    pq.push(child);
            }
        }
    }
    //输出结果
    if (bestw == INT_MAX) {
        cout << "No solution" << endl;
        return;
```

```
    }
    cout << bestw << endl;
    for (int i = 0; i < n; i++) {
        cout << bestx[i] << " ";
    }
}
int main()
{
    init_MinWeight();
    BranchBound();
    return 0;
}
```

结果截图：



2.1.5. 调试分析

发现这些题和之前做过的题基本一致，但是可以用多种方法求解，通过这道题我也认识到了回溯法和分值边界法的区别，在写代码的过程中也有过将两者搞混，后来看文章发现一个是 DFS，一个是 BFS 或者是 UCS，另外这道题要求使用优先队列，所以还需要自己重载一下运算符。

2.2. 最佳调度问题（6-8）

2.2.1. 问题描述

> 假设有 n 个任务由 k 个可并行工作的机器完成，完成任务 i 需要的时间为 ti。设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。
>
> 对任意给定的整数 n 和 k，以及完成任务 i 需要的时间为 ti（i=1~n）。设计一个优先队列式分支界限界法，计算完成这 n 个任务的最佳调度。

2.2.2. 问题分析

通过上一题的分析，这一题就简单了很多，而且也是要求必须用优先队列式的分支界限法。N 个任务，k 个可并行工作的机器，初始状态是没有任务完成，终止状态是 n 个任务都完成了，而且每个任务只有 k 个状态，也就是在机器 1 或 2 或…或 k 中，参考上一题的思路，可以一个任务一个任务判断，每层代表一个任务的所有选择情况。根据上一题的思路，可以设计如下数据结构，用于存储分支限界索要用到的信息。

特别是 Node 结构体，他是该算法的核心，用于表示每个搜索节点的状态，time[MAX_K] 存储每台机器当前的完成时间，这个数组允许追踪每台机器在每个节点下的进度，d 则表示当前任务的序号，即已经分配的任务数量，是一个关键部分，用于确定当前节点在搜索树中的位置也就是层数，同时为了让剪枝函数可以快速从节点中判断是否需要剪枝，可以在节点中添加一个 maxtime，存储当前状态下所有机器得分最大完成时间，这个变量是分支限界法中剪枝决策的关键，可以通过这个来判断当前节点是否值得探索。

```
struct Node {
    int time[MAX_K]; // 当前各个机器的时间
    int d; // 当前任务序号
```

```
        int maxtime;// 当前最大时间
};
```

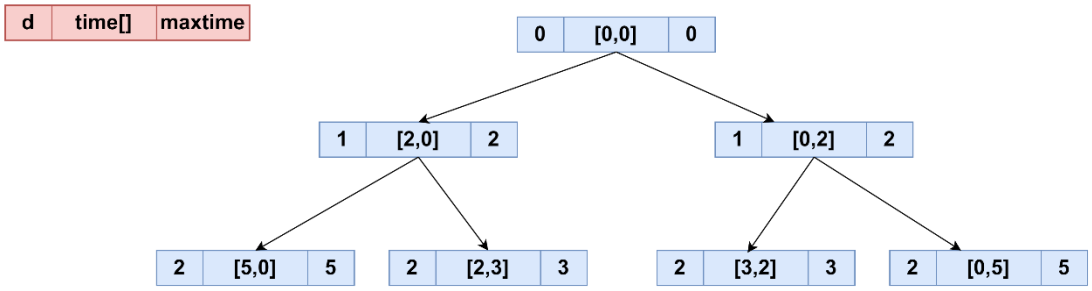
```
const int MAX_N = 1000;
const int MAX_K = 1000;

int n, k;//n 表示任务数，k 表示机器数
int Time[MAX_N];//每个任务的时间
int bestT = INT_MAX; // 最优时间
```

另外，由于本题要求使用优先队列，所以需要重载“<”运算符，使得 Node 对象可以被用作优先队列中的元素，具体逻辑应该是按照 maxtime 的升序进行排列，同时注意到这里应该是让 maxtime 越小越好，也就是优先队列最好是从 maxtime 小的节点开始探索，所以可以指定操作符判断逻辑是$maxtime > a.maxtime$，这样也就实现了最小堆的行为。

```
    // 重载<运算符，用于优先队列的比较
    bool operator<(const Node& a) const {
        return maxtime > a.maxtime;
    }
```

定义好解空间中的节点后，很容易将解空间树画出来。首先需要确定根节点，根节点表示初始状态，此时没有任务被分配，d=0,所有机器的当前时间都是 0；随后定义节点内容，每个结点包括 d（当前分配的任务编号），time[]（数组，表示每台机器当前的完成时间），maxtime（当前状态下所有机器的最大完成时间）；随后绘制分支，这里涉及到状态转移，从根节点开始，每个节点根据任务分配方案分叉成多个子节点，每个任务（从 1 到 n）可以分配给 k 个机器中的任意一台，因此每个节点将有 k 个子节点；然后再添加边代表从父结点到子节点的转变，即一个任务被分配给某台机器。

解空间可以表达为$\{(S1,S2,\dots,Sn)|Si \in [1,k]\}$，这里以简单情况为例，假设有 2 个任务(n=2)和 2 台机器(k=2)，并且任务的时间分别为[0, 2, 3]（数组的第 0 个元素未使用）。用图表示为：



解空间画出后，可以发现问题所求的最佳调度问题就是找所有机器完成时间最大值的最小值叶子结点。分支限界法的本质是对解空间树的 BFS 搜索，即将下一任务由每个机器完成的调度状态入队，然后依次将出队的每一个状态的子状态入队。具体的剪枝和算法逻辑在下一节详细叙述。

2.2.3. 算法设计

剪枝函数设计：

为了实现剪枝，我们需要一个处理时间上界的值，用于表示当前剩下任务全部由完成时间最早的机器处理之后，所有机器完成的最晚时间。当从优先队列中取出一个节点时，如果该节点的 maxtime 属性大于当前已知的最小完成时间 bestT，则该节点不会引导到一个更好的解，因此可以被剪枝。具体实现如下：

- 在每次从优先队列中取出节点后，立即检查 if (p.maxtime >= bestT)条件。
- 如果条件为真，使用 continue 语句跳过当前循环迭代，不执行任何进一步的操作，包括不将该节点的子节点加入队列。

剪枝在每次迭代中自然触发，因为优先队列始终首先处理具有最小 maxtime 的节点。一旦 bestT 被更新为一个更小的值，所有后续的节点在被处理之前都会与 bestT 进行比较。这也是使用优先队列的优点，通过自定义的优先队列容器所需的对象比较符号，可以是队列每次出队的元素都为最大完成时间最小的调度方案，有点类似于贪心策略。

伪代码实现：

```
定义结构体 Node 包含：
  整数数组 time[MAX_K] 表示每台机器的当前时间
  整数 d 表示当前任务序号
  整数 maxtime 表示当前最大时间

定义函数 pq()：
  开始
     创建一个最小堆 priority_queue 用于存储节点
     创建 Node 类型的初始节点 initialNode
     初始化 initialNode 的 time 数组的所有元素为 0
     初始化 initialNode 的 d 为 0
     初始化 initialNode 的 maxtime 为 0
     将 initialNode 推入 priority_queue

     当 priority_queue 不为空 时：
        从 priority_queue 中弹出具有最小 maxtime 值的节点 currentNode

        如果 currentNode 的 maxtime 大于当前最佳解 bestT：
            继续下一次循环（剪枝）

        如果 currentNode 的 d 等于任务总数 n：
            更新 bestT 为 currentNode 的 maxtime
            继续下一次循环（叶节点处理完毕）

        对于每台机器 i 从 0 到 k-1：
            创建新的 Node 节点 childNode 作为 currentNode 的副本
            将 childNode 的 d 增加 1（分配下个任务）
            将 childNode 的 time[i] 增加 Time[childNode.d]（更新机器时间）
            更新 childNode 的 maxtime 为 time[i] 和 currentNode 的 maxtime 中的较大值
            将 childNode 推入 priority_queue

     返回 bestT
  结束
```

复杂度分析：

时间复杂度：

在最坏的情况下，每个任务可以分配给 k 台机器中的任意一台，因此对于 n 个任务，总共有k^n种可能的分配方式，也就是解空间树可能有k^n个叶子节点。

在 pq 函数中，每次从优先队列中弹出一个节点的操作是$O(log\ m)$，其中 m 是优先队列中的元素数量。在最坏的情况下，每个叶子节点都会被访问，因此总的弹出操作次数接近k^n，但是剪枝减少了搜索空间，避免了对不可能产生最优解的分支的探索。剪枝的效果取决于问题的特定实例和 bestT 的更新频率。

所以总的来说，由于每个节点都有 k 个子节点，算法的总时间复杂度接近$O(k^n * log(m))$。然而，由于剪枝的存在，实际的时间复杂度可能会低得多。

空间复杂度：

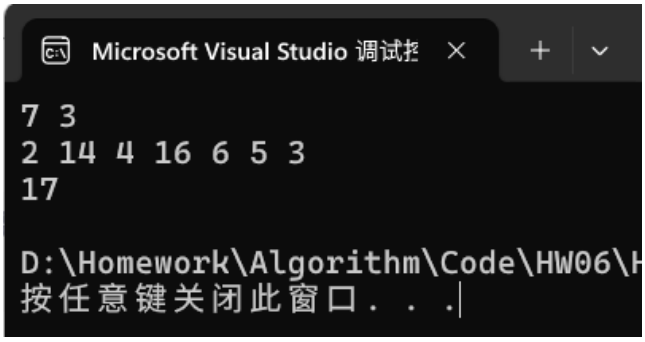
主要是优先队列和 Node 结构体的空间占用。优先队列在任何时候都可能包含从根节点到某一深度的所有节点。在最坏的情况下，如果没有任何剪枝，队列的大小可以达到 n * k（每层的节点数乘以层数），总的空间复杂度是$O(n * k)$

2.2.4. 算法代码

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
const int MAX_N = 1000;
const int MAX_K = 1000;
int n, k;//n 表示任务数，k 表示机器数
int Time[MAX_N];//每个任务的时间
int bestT = INT_MAX; // 最优时间
struct Node {
```

```
    int time[MAX_K]; // 当前各个机器的时间
    int d; // 当前任务序号
          int maxtime;// 当前最大时间
          // 重载<运算符，用于优先队列的比较
    bool operator<(const Node& a) const {
        return maxtime > a.maxtime;
    }
};
/**
 * brief: 优先队列解法
 */
int pq() {
    priority_queue<Node> pq;
    // 初始化
    Node p;
    for (int i = 0; i < k; i++)
        p.time[i] = 0;
    p.d = 0;
    p.maxtime = 0;
    pq.push(p);
          // 优先队列循环
    while (!pq.empty()) {
        p = pq.top();
        pq.pop();
                    // 剪枝
        if (p.maxtime >= bestT)
            continue;
                    // 到达叶节点,更新最优时间
        if (p.d == n) {
            bestT = p.maxtime;
            continue;
        }
                    // 递归，遍历所有分支
        for (int i = 0; i < k; ++i) {
            Node next = p;
                                next.d = p.d + 1;// 更新任务序号，下一层
                                next.time[i] += Time[next.d]; // 更新机器时间
                                next.maxtime = max(next.time[i], p.maxtime);// 更新最大时间
            pq.push(next);
        }
    }
    return bestT;
}
int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++)
        cin >> Time[i];
    cout << pq() << endl;
    return 0;
}
```

结果截图：



2.2.5. 调试分析

感觉这类题的关键在于数据结构的设计和剪枝函数的选取。需要选择合适的数据结构

来表示任务调度，在上一题需要输出顺序序列，所以需要使用 parent 指针往回走，而这道题只要最后的结果，不需要输出序列，所以节点里可以不放父结点指针。另外，针对这种问题，第一次读题读不明白的话，可以找一个简单情形，画一下图或者流程，就会清晰很多。

3. 实验总结

本次作业涉及了分支限界法的相关概念，最开始，其实发现这些问题情境都一样，分不太清每种算法思路的区别，但是真正在写代码的过程中，还是体会到了不同算法的特点，特别是这个分支限界法和回溯法的区别，这个算法好像是五大常见算法思想里最后一个，希望这次作业会后将这五个算法再看一下，对比记忆他们的区别以及适用情形。例如这篇所讲的 https://blog.csdn.net/vivian_ll/article/details/103253664。