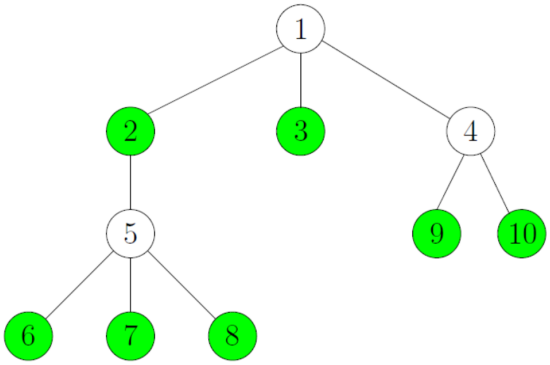
1.Given a tree T = (V,E), find the maximum independent set of the tree. For example, maximum independent set of the tree of following tree has size 7.



The green vertices shows that the maximum indpendent set of thetree has size 7.

(1) Design an efficient greedy algorithm to solve the problem.

(2) If each vertex has a weight such as benefit,wehther can you use a Greedy algorithm to get a maximum benefit where if one vertex has been selected then its adjacent verticex can not be selected.

问题分析（1）

最大独立集的知识在离散数学中就已经学过，其主要特点就是，独立集中的结点不可以相邻。所以如果我们要用贪心的算法去实现本题，则一定要立足于这个特点。

算法思路

一开始，我的想法是：

1.初始化一个空的最大独立集合 MIS

2.为树 T 中的每个节点 v，将其标记为未访问

3.选择一个未访问的节点 v 作为起始节点

4.将节点 v 添加到 MIS 中，并将其标记为已访问

5.对于节点 v 的每个相邻节点 u，将节点 u 标记为已访问

6.重复步骤 3-5，直到所有节点都被访问过

7.返回 MIS 作为最大独立集

在这个算法中，我们利用了最大独立集中结点互不相邻的特性，并且利用标志去判断该结点是否被标记来进行是否为邻接点的判断。尽管我们的算法为贪心，不一定能够找到最优解。但是经过我的模拟演算，我发现此算法的结果非常依赖于起始节点的选择，而且找不到最优解的概率非常大。因此我们需要改进。

我们的对象是树，由于这种结构的特殊性，我发现最下面的叶子结点一定在最大独立集中，因此我觉得在刚刚的算法中，结点的选取为叶子结点会更好。更改后的算法为：

1.初始化一个空的最大独立集合MIS，为树中每一个节点设置标志量0。

2.选择一个叶子节点加入MIS，并标记该节点为1和标记其父节点为2。

3.当目前叶子节点全部为1后，删除所有标志为2的节点。

4.重复23，直到树为空。

此算法的缺点为：运算过程中可能会产生森林（多个离散的部分），运算完原来的树结构被破坏。

```
def maximum_independent_set(tree):
    MIS = []  # 最大独立集合
    leaf_nodes = find_leaf_nodes(tree)  # 找到树中的叶子节点

    while leaf_nodes:
        node = leaf_nodes.pop()  # 选择一个叶子节点
        MIS.append(node)  # 将节点添加到最大独立集合
        node.flag = 1  # 将节点标记为1

        parent = node.parent  # 获取父节点
        if parent:
            parent.flag = 2  # 将父节点标记为2

        # 删除标记为2的节点
        remove_nodes = []
        for n in tree.nodes:
            if n.flag == 2:
                remove_nodes.append(n)
        for n in remove_nodes:
            tree.remove_node(n)

        leaf_nodes = find_leaf_nodes(tree)  # 更新叶子节点列表

    return MIS


def find_leaf_nodes(tree):
    leaf_nodes = []
    for node in tree.nodes:
        if len(node.neighbors) <= 1:
            leaf_nodes.append(node)
    return leaf_nodes
```

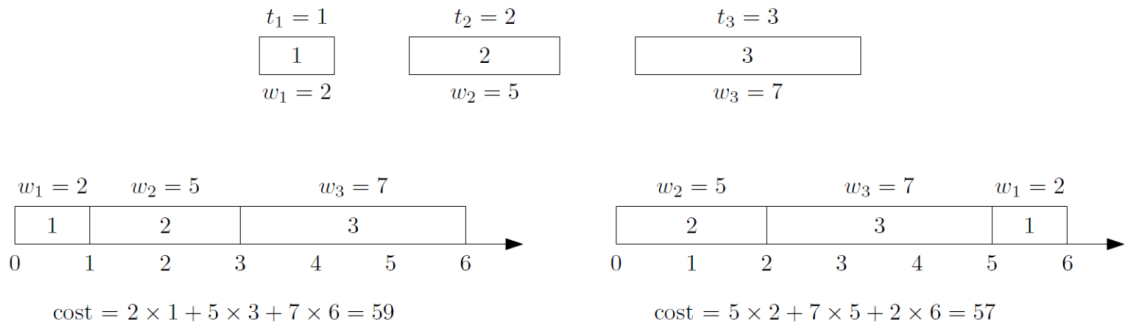
问题分析（2）

此问题只要求我们用贪心算法求得最大权值的独立集，而非最大独立集中最大权值的独立集。遵循贪心算法的规则，每一步只选当前最好的选择即可。

算法思路

开始前将所有结点设置为未访问，在当前未访问的结点中，选择权值最大的结点加入集合，并标记该结点和其相邻结点为已访问。再从剩余未访问的结点中选择权值最大的结点加入集合，一致循环此操作直到所有结点都访问。

Scheduling to Minimize Weighted Completion Time,Input: A set of n jobs [n] := {1, 2, 3, · · · , n}, each job j has a weight wj and processing time tj Output: an ordering of jobs so as to minimize the total weighted completion time of jobs



问题分析

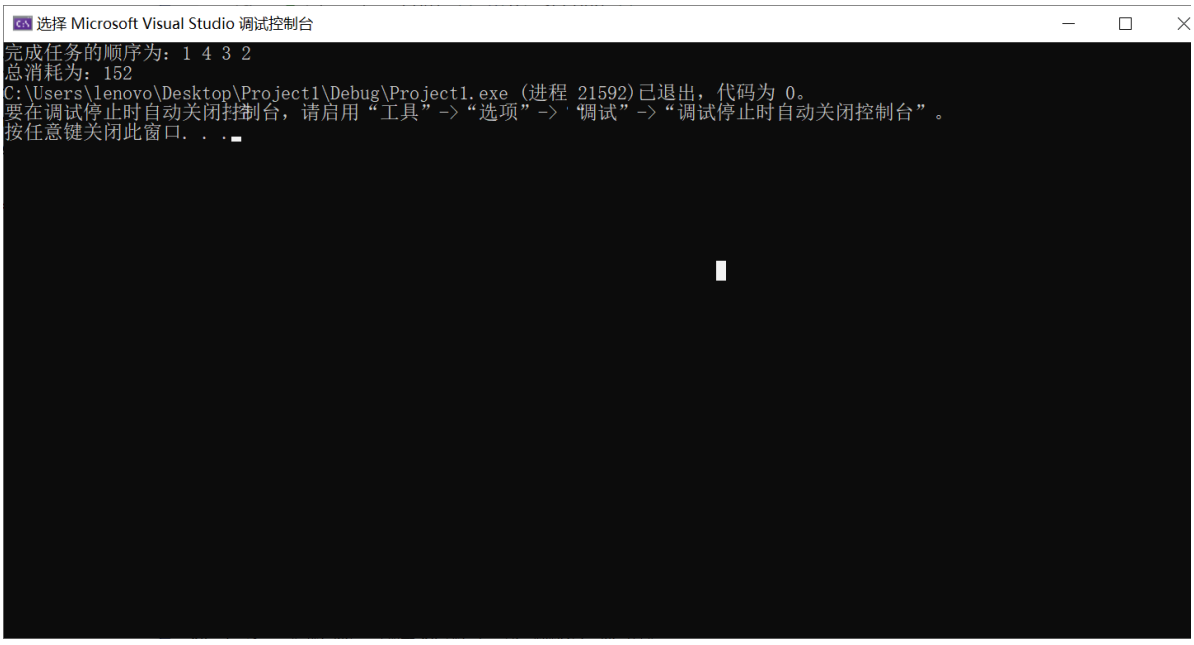
我们的目的是让总cost最低，而cost的计算方法为时间*权重，并且任务是一个一个完成的，所以我们可以采用贪心的算法，每次选取能使当前cost最小的工作去完成即可。

算法思路

假设集合中共有n个元素，则我们需要进行n次判断。在每次判断中，我们记录下每个任务选取时所消耗的consumption，然后将consumption进行比较，选取最小的工作，打印其工作序号并且将consumption加入到cost中，将该工作从集合中移除。一直循环此过程，直到集合中元素为空。

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

using namespace std;

// 树的节点定义
struct job {
    int id;
    int weight;
    int time;
};

void cost(std::vector<job> a) {
    std::vector<job> c = a;
    int cost = 0;
    int count = 0;
```

```
    int n = a.size();
    for (int i = 0; i < n; i++) {
        std::vector<std::pair<int,int>> b;
        for (int j = 0; j < a.size(); j++) {
            int consume = (count + a[j].time) * a[j].weight;
            b.push_back(std::make_pair(consume,a[j].id));

        }
        std::sort(b.begin(), b.end());
        cost += b[0].first;
        cout << b[0].second << " ";
        count += c[b[0].second-1].time;
        for (int i = 0; i < a.size(); i++) {
            if (a[i].id == b[0].second) {
                a.erase(a.begin() + i);
            }
        }
    }
    cout << endl;
    cout << cost;
}



int main() {
    job job1 = { 1,2,3 };
    job job2 = { 2,7,2 };
    job job3 = { 3,5,4 };
    job job4 = { 4,2,3 };
    std::vector<job> a = { job1,job2,job3,job4 };
    cost(a);
    return 0;
}
```



需要注意的是，该算法为贪心算法，所得到的结果是近似最优解，而并非最优解。