**算法分析第4次作业**

4-1 将k个活动按开始时间升序排序，用小顶堆存储已经安排的结束时间。如果当前活动的开始时间早于最早结束时间，则需要新开一个会场，否则将当前活动的结束时间加入小根堆中，更新最早结束时间。

以下为代码演示：

#include<bits/stdc++.h>

int getMinMeetingPlace(vector<pair<int,int>> *activities*)

{

    sort(*activities*.begin(),*activities*.end());

    priority\_queue<int,vector<int>,greater<int>> p;

    int res=0;

    for(auto activity:*activities*)

    {

        if(p.empty())

        {

            p.push(activity.second);

            res++;

            continue;

        }

        if(activity.first>=p.top())

        {

            p.pop();

            p.push(activity.second);

        }

        else

        {

            p.push(activity.second);

            res++;

        }

    }

    return res;

}

使用了排序算法，时间复杂度为O(n\*logn)，遍历的时间复杂度为O(n)，使用优先队列时间复杂度为O(logk)，最坏情况下可达到O(logn)，因此时间复杂度为O(n\*logn)

4-2 使用haffman算法，使用优先队列，每次将最小/最大的两个值进行合并。

int getMinCompareCounts(priority\_queue<int,vector<int>,greater<int>>& *pq*)//getMaxCompareCounts(priority\_queue<int> &pq)

{

    int res=0;

    if(*pq*.empty())

    {

        return 0;

    }

    while(*pq*.size()>1)

    {

        int x=*pq*.top();

*pq*.pop();

        int y=*pq*.top();

*pq*.pop();

        res+=x+y-1;

*pq*.push(x+y);

    }

    return res;

}

循环迭代次数为O(n)，优先队列的top()时间复杂度为O(1)，push()和pop()最坏情况下时间复杂度为O(logn)，所以时间复杂度为O(n\*logn)

4-4 贪心策略，检索概率最大的文件排在最中间，两边依次按概率从大到小顺序排序。假设存在一个最优排列，其中两个文件i、j，pi>pj但是i里中间位置远于j，如果交换i、j的位置，对于其他文件k，d(i,k)会减小，d(k,j)会增大，因为pi>pj，总期望时间减小，因此矛盾，贪心算法正确。

以下是算法实现：

double minProbability(int *f*[],int *n*)

{

    sort(*f*,*f*+*n*,greater<int>());

    int mid=*n*/2;

    int \*x=new int[*n*];

    x[mid]=*f*[0];

    for(int i=mid+1;i<*n*;i++)

    {

        x[i]=*f*[2\*(i-mid)];

    }

    for(int i=mid-1;i>=0;i--)

    {

        x[i]=*f*[2\*(mid-i)-1];

    }

    double res=0;

    int sum=0;

    for(int i=0;i<*n*;i++)

    {

        sum+=*f*[i];

        for(int j=i;j<*n*;j++)

        {

            res+=x[i]\*x[j]\*(j-i);

        }

    }

    return res/sum/sum;

}

使用排序算法，时间复杂度为O(n\*logn)，使用二重循环，时间复杂度为O(n^2)，故时间复杂度为O(n^2)。

4-6 贪心策略，优先服务服务时间短的顾客。假设存在最优序列，其中两个顾客i和j，i的服务时间比j长，但排在j的前面，交换两者顺序，i的等待时间变长，j的等待时间变短，两者前后等待时间减少ti-tj，i和j之间的等待时间变短ti-tj，所以总等待时间减少n\*(ti-tj)，n为后者下标减去前者下标，贪心算法正确。

以下是代码实现：

double averageServeTime(int *t*[],int *n*)

{

    sort(*t*,*t*+*n*);

    for(int i=1;i<*n*;i++)

    {

*t*[i]+=*t*[i-1];

    }

    double res=0;

    for(int i=0;i<*n*;i++)

    {

        res+=*t*[i];

    }

    return res/*n*;

}

使用排序算法，时间复杂度为O(n\*logn)，使用一重循环，时间复杂度为O(n)，故时间复杂度为O(n\*logn)。

4-8 从叶节点进行遍历，当出现该叶节点的父节点到父节点的距离和到叶节点距离之和大于d时，切割该叶节点的父节点，s的顶点数加1。用队列存储叶节点，每次遍历后当前节点出列，父节点出度减1，并将到叶节点的距离更新为到当前叶节点和原到叶节点距离的较大值，当父节点出度为0时入列。

代码实现如下：

struct Node

{

    int father;

    int distanceF;*//到父节点的距离*

    int distanceL;*//到叶节点的距离*

    int out;*//出度*

    bool cut;*//是否被切*

};

int dTree(Node *nodes*[],queue<int> *leaf*,int *d*)

{

    int res=0;

    while(!*leaf*.empty())

    {

        int t=*leaf*.front();

*leaf*.pop();

        int parent=*nodes*[t].father;

        int distance=*nodes*[t].distanceF;

        if(!*nodes*[parent].cut&&*nodes*[t].distanceL+distance>*d*)

        {

*nodes*[parent].cut=true;

            parent=*nodes*[parent].father;

            res++;

        }

        else if(!*nodes*[t].cut&&*nodes*[parent].distanceL<*nodes*[t].distanceL+distance)

        {

*nodes*[parent].distanceL=*nodes*[t].distanceL+distance;

        }

        if(--*nodes*[parent].out==0)

        {

*leaf*.push(parent);

        }

    }

    return res;

}

使用队列进行遍历，时间复杂度为O(n)，队列的操作都是O(1)，故时间复杂度为O(n)

4-9 优先选择从当前加油站所能到达的最远的加油站。假设汽车最远能到达j汽车站，但在汽车站i(i的下标<j的下标)就加了一次油，所能到达的最远距离就会更近，能达到的最远的加油站i’的下标<j’的下标，最终加油次数不超过在j加油站再加油的次数，故贪心算法正确。

以下是算法实现：

int minRefuelCount(int *fuel*[],int *n*,int *k*)

{

    int res=0;

    int sum=0;

    for(int i=0;i<=*k*;i++)

    {

        if(*fuel*[i]>*n*)

        {

            return -1;

        }

        sum+=*fuel*[i];

        if(sum>*n*)

        {

            res++;

            sum=*fuel*[i];

        }

    }

    return res;

}

使用一重循环，时间复杂度为O(k)

4-11 优先删除高位上的下降点。假设任意n位数a1a2...an，从高位起，首个下降点为ai>a(i+1)，删除后第i+1位数为a(i+1)，如果删除ai后面的点则为ai，因为ai>a(i+1)，所以删除ai后较小，如果删除ai前面的点aj，第j+1位为a(j+1)，而删除ai的第j+1位为aj，因为ai前均为上升点，故a(j+1)>aj，所以删除ai后较小，贪心算法正确。

以下是算法实现：

int\* stringToArray(string *a*)

{

    int n=*a*.length();

    int\* arr=new int[n];

    for(int i=0;i<n;i++)

    {

        arr[i]=*a*[i]-'0';

    }

    return arr;

}

int minNum(string *a*,int *k*)

{

    int\* arr=stringToArray(*a*);

    int n=*a*.length();

    int res=0;

    while(*k*--)

    {

        int i=0;

        for(i=0;i<n-1&&*a*[i]<=*a*[i+1];i++)

        ;

*a*[i]=-1;

    }

    for(int i=0;i<n;i++)

    {

        if(*a*[i]!=-1)

        {

            res=res\*10+*a*[i];

        }

    }

    return res;

}

嵌套循环中，外层循环为O(k)，内层循环最坏情况下为O(n)，故时间复杂度为O(k\*n)。

4-15 要使获得的乘积最大，就得分解出的自然数要尽可能多，且均大于2，因为∑ai=const，就要使分解的自然数尽可能小，又不能有重复，所以使用贪心优先大于2的较小数，当出现不得不重复的较小数时，优先将该较小数平均分配给较大数。

以下是算法实现：

int maxMultiply(int *n*)

{

    if(*n*<=2)

    {

        return *n*;

    }

    vector<int> arr;

    for(int i=2;i<=*n*;i++)

    {

        arr.push\_back(i);

*n*-=i;

    }

    int i=arr.size()-1;

    while(*n*>0)

    {

        arr[i]++;

*n*--;

        i--;

        if(i<0)

        {

            i=arr.size()-1;

        }

    }

    int res=1;

    for(int a:arr)

    {

        res\*=a;

    }

    return res;

}

每个循环遍历次数均小于n，时间复杂度应小于O(n)。