算法入门推荐：《算法图解》

介绍一本关于算法基础的入门级书籍，对于非科班出身的人来说，算法和数据结构的补充还是很有必要的，但是这些东西往往又是很枯燥以致于打消了很多人的积极性，《算法图解》用python为编程语言，对于一些基础性的算法介绍可以说很通俗易懂了，真的很适合入门，同时这篇文章我也结合了一些校招题目来进行一定程度的扩充，有一些用到了C++。

一、二分查找（O（log n））

```
def binary_search(list,item):
    low = 0;
    high = len(list)-1
    while low <= high :
        mid = int((low+high)/2)
        guess =list[mid]
        if guess == item:
            return mid
        if guess < item:
            low = mid+1
        else:
            high=mid-1
    return None
my_list =[1,3,5,7,9]
print (binary_search(my_list,5))
print (binary_search(my_list,-1))
```

> 输出结果：
>
> 2
> None

二分查找的考查有很多，例如有一年字节跳动的编程题就是用到了这个方法：

问题描述：

总共有n条长度不等的绳子，可以任意切割，不能拼接，要求切割后得到m条长度相等的绳子，求问得到的这些长度相等的绳子的长度最大值L。

输入： 绳子的条数n；n条绳子的长度；要求切成的绳子数量m

输出：切割成相同长度的m条绳子的最大的长度

思路解析：

这道题其实就是二分查找的题目，区别于动态规划（在接下来的那个板块我会讲到也是割绳子的另一道题），因为我们知道最长长度的绳子，可以确定最终将剪成的长度只能在0到这个数中间，我们对这个范围进行二分查找，然后对得到的guess值，我们构造一个函数来判断是否符合要求，这个函数就是遍历所有绳子算出能剪出多少段这样长度的绳子，这个数量大于m则说明短了，这个数量小于m则说明长了，这样的二分查找还是挺清晰的。

```
N = input()
N=int (N)
max_len = 0.0
min_len = 0.0
num={}
for i in range(N):
    num[i]=int (input())
    if num[i]>max_len:
        max_len=num[i]
M = input()
M=int (M)
def check(length):
    ans = 0
    for i in range(N):
        ans += (int)(num[i]/length)
    return ans
while max_len-min_len>=0.00001:
    mid = (max_len+min_len)/2
    if(M>check(mid)):
        max_len=mid
    else:
        min_len=mid
print(mid)
```

> 示例结果：
>
> 5
> 5
> 5
> 8
> 10
> 10
> 45
> 0.8333301544189453

这道题卡的是精度，要把精度提高才能Access。

二、排序算法

在这本书介绍的常见的排序算法是选择排序和快速排序，下面是选择排序的程序：

```
def findsmallest(arr):
    smallest = arr[0]
    smallest_index = 0
    for i in range(0,len(arr)):
        if arr[i]<smallest:
            smallest = arr[i]
            smallest_index=i
        return smallest_index
def selectionsort(arr):
    newArr=[0]*7
    for i in range(len(arr)):
        smallest=findsmallest(arr)
        newArr[i]=(arr.pop(smallest))
    return newArr
print(selectionsort([5,3,6,2,10,22,45]))
```

> 示例结果：
>
> [5, 3, 6, 2, 10, 22, 45]

按照书本的内容，在介绍完选择排序之后就讲了递归，相信很多人对于递归还是有一定的了解，书本的介绍基于栈原理的实现，感兴趣的可以去看一下，我就直接进入快速排序了，我将通过一道牛客网上的题目对它进行实现：

问题描述：

为了找到自己满意的工作，牛牛收集了每种工作的难度和报酬。牛牛选工作的标准是在难度不超过自身能力值的情况下，牛牛选择报酬最高的工作。在牛牛选定了自己的工作后，牛牛的小伙伴们来找牛牛帮忙选工作，牛牛依然使用自己的标准来帮助小伙伴们。牛牛的小伙伴太多了，于是他只好把这个任务交给了你。

输入：

每个输入包含一个测试用例。每个测试用例的第一行包含两个正整数，分别表示工作的数量N(N<=100000)和小伙伴的数量M(M<=100000)。接下来的N行每行包含两个正整数，分别表示该项工作的难度Di(Di<=1000000000)和报酬Pi(Pi<=1000000000)。接下来的一行包含M个正整数，分别表示M个小伙伴的能力值Ai(Ai<=1000000000)。保证不存在两项工作的报酬相同

输出：

对于每个小伙伴，在单独的一行输出一个正整数表示他能得到的最高报酬。一个工作可以被多个人选择

代码如下：

```
#include<iostream>
#include<algorithm>
using namespace std;
bool cmp(int a[] , int b[])
{
    if(a[1]==b[1])
    {
        return a[0]>b[0];
    }
    return a[1]>b[1];
}
//use the quicksort algorithm to solve the two dimension array
void quicksort(int **a,int left,int right)
{
    int pos1=left;
    int pos2=right;
    if(left<right)
    {
        int temp=a[left][0];
        int temp1=a[left][1];
        while(left<right)
        {
            while(left<right&&(a[right][0]>temp||(a[right][0]==temp&&a[right]
[1]>temp1)))
                right--;
            a[left][0]=a[right][0];
            a[left][1]=a[right][1];
            while(left<right&&(a[left][0]<temp||(a[left][0]==temp&&a[left][1]
<temp1)))
                left++;
```

```
            a[right][0]=a[left][0];
            a[right][1]=a[left][1];
        }
        a[left][0]=temp;
        a[left][1]=temp1;
        quicksort(a,pos1,left-1);
        quicksort(a,left+1,pos2);
    }


}
int main()
{
    int num_of_work,num_of_worker;
    cin>>num_of_work>>num_of_worker;
    int **work=new int*[num_of_work];
    for(int i=0;i<num_of_work;i++)
    {
        work[i]=new int[2];
        cin>>work[i][0]>>work[i][1];
    }
    //sort the array by the algorithm in C++,and we get the ascent order
    sort(work,work+num_of_work,cmp);
    //sort the array by the self function,and we get the descent order
    //quicksort(work,0,num_of_work-1);

    int *worker=new int[num_of_worker];
    int *money=new int[num_of_worker];
    for(int i=0;i<num_of_worker;i++)
    {
        cin>>worker[i];
        int cando=worker[i];
        for(int j=0;j<num_of_work;j++)
        {
            if(cando>=work[j][0])
            {
                money[i]=work[j][1];
                break;
            }
        }
    }
    for(int i=0;i<num_of_worker;i++)
        cout<<money[i]<<endl;




}
```

对于这道题虽然上述方法只能通过80%，因为它的复杂度太高了O(mn)，使用了暴力搜索的方法。但是上面的代码示范了怎么自己写二维数组的快速排序，同时也示范了怎么使用库函数进行二维数组的排序，下面是复杂度低的优化程序（Access程序）

```
#include<iostream>
#include<algorithm>
using namespace std;
```

```
struct job
{
    int dif;
    int money;
};
struct  people
{
    int index;
    int dif;
    int money;
};
bool cmp1(job a,job b)
{
    return a.dif<b.dif;
}
bool cmp2(people a,people b)
{
    return a.dif<b.dif;
}
bool cmp3(people a,people b)
{
    return a.index<b.index;
}
int main()
{
    int num_of_job,num_of_people;
    cin>>num_of_job>>num_of_people;
    job *Job=new job[num_of_job];
    people *People =new people[num_of_people];
    for(int i=0;i<num_of_job;i++)
    {
        cin>>Job[i].dif>>Job[i].money;
    }
    for(int i=0;i<num_of_people;i++)
    {
        cin>>People[i].dif;
        People[i].index=i;
    }
    sort(Job,Job+num_of_job,cmp1);
    sort(People,People+num_of_people,cmp2);
    int j=0;int maxmoney=0;
    for(int i=0;i<num_of_people;i++)
    {
        while(j<num_of_job)
        {
            if(Job[j].dif>People[i].dif)
                break;
            else
            {
                maxmoney=max(maxmoney,Job[j].money);
                j++;
            }
        }
        People[i].money=maxmoney;
    }
    sort(People,People+num_of_people,cmp3);
    for(int i=0;i<num_of_people;i++)
    {
```

```
        cout<<People[i].money<<endl;
    }
}
```

代码还是很容易理解的，具体就不讲了。

三、散列表

这一部分介绍了散列表的功能，对于基础实现没有过多的解释（其实更多依靠散列函数的实现），我也就简单讲一下，先举一个简单的例子，给定N个整数，再给定M个整数，要你检查出M个数中每个数中是否在N个整数里有出现过，可能你会直观地想到遍历查询，但对于N和M很大时，显然是无法承受的（O（MN）），所以可以用Hashtable这个bool数组来记录N个数里面每一个数是否出现过，如果出现了，就记为true，否则为false，那么查询的时候，我们可以直接把输入的数作为数组下标就可以得到是否出现的结果，这是一个最简单的例子，但是，如果我们的输入不是整数，而是一些字符串等等，那么就需要用到散列来映射到一个整数上，这就是我们所说的散列表。

散列函数可以有直接定址法，也就是恒等变化或者线性变化：H（key）=key，H（key）=a*key+b；也有除留余数法：H（key）%mod；

稍加思考便可以注意到：通过除留余数法可能会有两个不同的Key得到相同的hash值，他们无法占用相同的位置，这种情况叫冲突，解决冲突的方法有：线性探查法（当得到的值已经被占用了就顺移到接下来的位置直到有空位，但这种方法容易导致扎推，也就是如果连续若干个位置都被占用，一定程度会降低效率。）与之相似的有平方探查法，而相对不一样的就是链地址法，也就是如果计算得到的hash值相同，就把所有相同hash值的key连成一条链表。

四、广度优先搜索

```
def search(name):
    search_queue  =deque()
    search_queue +=graph[name]
    searched = []
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:
            if person_is_seller(person):
                return True
            else:
                search_queue +=graph[person]
                search.append(person)
    return False
search("you")
```

广度优先搜索通俗一点就是通过队列把你周围最近的元素都塞进去，然后一个一个pop出来检查，每检查一个，就把这个元素周围的没有访问过的元素塞进队列里面，继续检查，然后检查完了标记它为已检查。

对于广度优先搜索，比较常见的问题是迷宫问题，可以看看《算法笔记》的一些介绍，我就列举一道迷宫的题目补充一下：

问题描述：

给定一个迷宫m*n大小，"*"代表不可以通过的墙壁，而"."代表平地，S代表起点，T代表终点，移动过程中，当前位置只能上下左右移动，求最短路径。迷宫如下：（S坐标是（2，2））

. . . . .

.*.*.

.*S*.

.***.

...T*

思路解析：

可以用广度优先搜索通过层次的顺序来遍历，找到最小步数。

代码：

```
#include<iostream>
#include<queue>
using namespace std;
const int maxn=1000;
char maze[maxn][maxn];
bool visited[maxn][maxn];
struct node{int x,y;int step;}S,T,Node;
int n,m;
int x[4] = {0,0,1,-1};
int y[4] = {1,-1,0,0};
bool isok(int x,int y)
{
    if(x>=n||x<0||y>=m||y<0)
    {
        return false;
    }
    if(maze[x][y]=='*')
    {
        return false;
    }
    if(visited[x][y]==true)
    {
        return false;
    }
    return true;
}
int BFS()
{
    queue<node> q;
    q.push(S);
    while(!q.empty())
    {
        node top = q.front();
        q.pop();
        if(top.x==T.x&&top.y==T.y)
        {
            return top.step;
        }
        for(int i=0;i<4;i++)
        {
            int tmpx=top.x+x[i];
            int tmpy=top.y+y[i];
            if(isok(tmpx,tmpy))
            {
```

```
                Node.x=tmpx;
                Node.y=tmpy;
                Node.step=top.step+1;
                q.push(Node);
                visited[tmpx][tmpy]=true;
            }
        }
    }
    return -1;
}
int main()
{
    cin>>n>>m;
    for(int i=0;i<n;i++)
    {
        getchar();
        for(int j=0;j<m;j++)
        {
            maze[i][j]=getchar();
        }
        maze[i][m+1]='\0';
    }
    cin>>S.x>>S.y>>T.x>>T.y;
    S.step=0;
    cout<<BFS();
}
```

> 示例：
>
> 5
> 5
> .....
> .*.*.
> .*S*.
> .***.
> ...T*
> 2 2 4 3
>
> 输出结果：
>
> 11

五、Dijkstra算法

在这里举一个比较常见的例题：

问题简述：

给出N个城市，M条无向边，每个城市中都有一定数目的救援小组，所有比那的边权已知，现在给出起点和重点，求起点到终点的最短路径条数及最短路径上的救援小组数目之和，如果有多条路径，输出数目之和最大的。

思路：

本题在求解最短路径的同时需要求解另外两个信息，最短路径条数和最短路径上的最大点权值和，因此我们可以令w[u]表示从起点s到顶点u可以得到的最大的点权之和，初始化为0，令num[u]表示从起点s到顶点u的最短路径条数，初始化时只有num[s]为1，其余num[u]为0.然后在更新d[v]时同时更新两个数组，代码如下：

```
#include<iostream>
#include<cstring>
using namespace std;
const int MAXV = 510;
const int INF = 1000000000;

//n为顶点数，m为边数，st和ed分别为起点和终点
//G为邻接矩阵，weight为点权
//d【】记录最短距离，w【】记录最大点权之和，num【】记录最短路径条数
int n,m,st,ed,G[MAXV][MAXV],weight[MAXV];
int d[MAXV],w[MAXV],num[MAXV];
bool vis[MAXV]={false};

void Dijkstra(int s)
{
    fill(d,d+MAXV,INF);
    memset(num,0,sizeof(num));
    memset(w,0,sizeof(w));
    d[s]=0;
    w[s]=weight[s];
    num[s]=1;
    for(int i=0;i<n;i++)
    {
        int u=-1,MIN=INF;
        //找到未访问的顶点
        for(int j=0;j<n;j++)
        {
            if(vis[j]==false&&d[j]<MIN)
            {
                MIN=d[j];
                u=j;
            }
        }
        if(u==-1) return;
        vis[u]=true;
        for(int v=0;v<n;v++)
        {
            if(vis[v]==false&&G[u][v]!=INF)
            {
                if(d[u]+G[u][v]<d[v])
                {
                    d[v]=d[u]+G[u][v];
                    w[v]=w[u]+weight[v];
                    num[v]=num[u];
                }

                else if(d[u]+G[u][v]==d[v])
                {
                    if(w[u]+weight[v]>w[v])
                        w[v]=w[u]+weight[v];

                    num[v]+=num[u];
```

```
                }
            }
        }

    }
}

int main()
{
    cin>>n>>m>>st>>ed;
    for(int i=0;i<n;i++)
    {
        cin>>weight[i];
    }
    int u,v;
    fill(G[0],G[0]+MAXV*MAXV,INF);
    for(int i=0;i<m;i++)
    {
        cin>>u>>v;
        cin>>G[u][v];
        G[v][u]=G[u][v];
    }
    Dijkstra(st);
    cout<<num[ed]<<" "<<w[ed]<<endl;
}
```

> 示例：
>
> 5 6 0 2
> 1 2 1 5 3
> 0 1 1
> 0 2 2
> 0 3 1
> 1 2 1
> 2 4 1
> 3 4 1
>
> 输出：
>
> 2 4

六、动态规划

动态规划最经典的就是背包问题，包括01背包和完全背包问题，这里重点讲一下01背包问题，01问题涉及推导公式，可以参考《算法图解》里面的解释，但是最终我们可以得到一条普适性的公式（与完全背包问题的区别也在于这条公式），当然01背包问题也可以作为经典的深度优先搜索教案，我们先看一下公式法的写法，再看看深度优先搜索的方法：

问题描述：

有n件物品，每件物品的重量为w[i]，价值为c[i]。现有一个容量为V的背包，问如何选取物品放入背包，使得背包内物品价值最大，每种物品只有1件。

思路：

直接上公式：dp[v]=max(dp[v],dp[v-w[i]]+c[i])

代码：

```
#include<iostream>
using namespace std;
const int maxn=100;
const int maxv=1000;
int w[maxn],c[maxn],dp[maxv];
int main()
{
    int n,V;
    cin>>n>>V;
    for(int i=0;i<n;i++)
    {
        cin>>w[i];
    }
    for(int i=0;i<n;i++)
    {
        cin>>c[i];
    }
    for(int v=0;v<=V;v++)
    {
        dp[v]=0;
    }
    for(int i=1;i<=n;i++)
    {
        for(int v=V;v>=w[i];v--)
        {
            dp[v]=max(dp[v],dp[v-w[i]]+c[i]);
        }
    }
    int max=0;
    for(int v=0;v<=V;v++)
    {
        if(dp[v]>max)
        {
            max=dp[v];
        }
    }
    cout<<max<<endl;
}
```

> 输出结果：
>
> ```
> 5 8
> 3 5 1 2 2
> 4 5 2 1 3
> 10
> ```

如果采用深度优先搜索方法，可以看到如下：

```
#include<iostream>
using namespace std;
const int maxn = 30;
int n,V,maxValue = 0;
int w[maxn],c[maxn];
```

```
void DFS(int index,int sumW,int sumC)
{
    if(index==n)
    {
        if(sumW<=V&&sumC>maxValue)
            maxValue=sumC;
        return ;
    }
    DFS(index+1,sumW,sumC);
    DFS(index+1,sumW+w[index],sumC+c[index]);
}
int main()
{
    cin>>n>>V;
    for(int i=0;i<n;i++)
    {
        cin>>w[i];
    }
    for(int i=0;i<n;i++)
    {
        cin>>c[i];
    }
    DFS(0,0,0);
    cout<<maxValue<<endl;
}
```

> 示例：
>
> 5 8
> 3 5 1 2 2
> 4 5 2 1 3
> 10

总结：

总的来说，这篇文章是从《算法图解》这本书的整体框架进行一定的拓展，因为我之前对于算法有一定的了解，所以只花了一天时间看完这本书，但还是觉得书里面的讲解很有趣而且对于自己知识的巩固有着很好的作用，我也花了一两天总结这本书的相关算法，也写在这篇文章里面，包括一些校招、PAT的题目，因为我主要以c++为编程语言，所以出现了很多以c++为语言的代码，但是其实与python是一样的，我们主要理解里面的算法思想，同时我也没有提及书本里面的一些拓展知识，也留给读者们去阅读的机会。

书本的文件可以到我的GitHub上去下载，同时相应的程序我也打出来，想试一试的人可以到我GitHub上去看看。

GitHub地址：https://github.com/Brian-Liew/Algorithm/tree/master/%E4%B9%A6%E7%B1%8D%E7%AC%94%E8%AE%B0%EF%BC%9A%E3%80%8A%E7%AE%97%E6%B3%95%E5%9B%BE%E8%A7%A3%E3%80%8B