面试题目剖析

七月算法 邹博

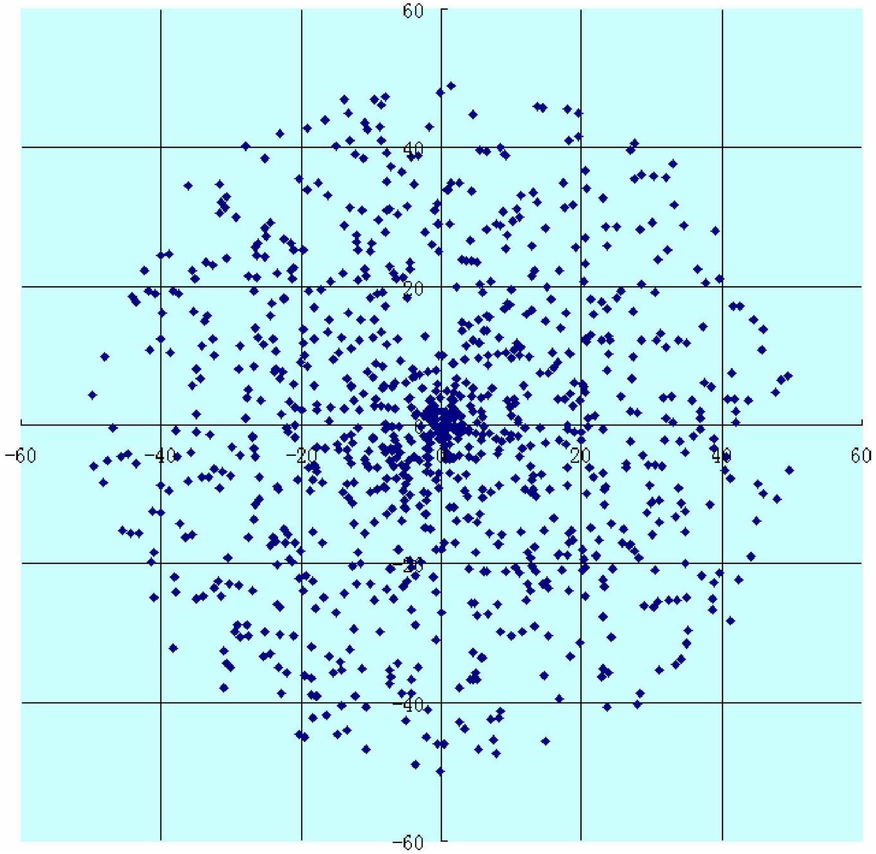
2015年6月27日

圆内均匀取点

- 给定定点$O(x_0,y_0)$和半径r，使得二维随机点(x,y)等概率落在圆内。
- 分析
  - 因为均匀分布的数据是具有平移不变性，生成半径为r，定点为圆心的随机数(x_1,y_1)，然后平移得到(x_1+x_0,y_1+y_0)即可。
  - 直接使用$x=r*\cos\theta$，$y=r*\sin\theta$是否可以呢？
    - 具体试验一下。

圆内均匀取点代码与效果

```
int rand50()
{
    return rand() % 100 - 50;
}

int _tmain(int argc, _TCHAR* argv[])
{
    ofstream oFile;
    oFile.open(_T("D:\\rand.txt"));
    double r,theta;
    double x,y;
    for(int i = 0; i < 1000; i++)
    {
        r = rand50();
        theta = rand();
        x = r*cos(theta);
        y = r*sin(theta);
        oFile << x << '\t' << y << '\n';
    }
    oFile.close();
    return 0;
}
```

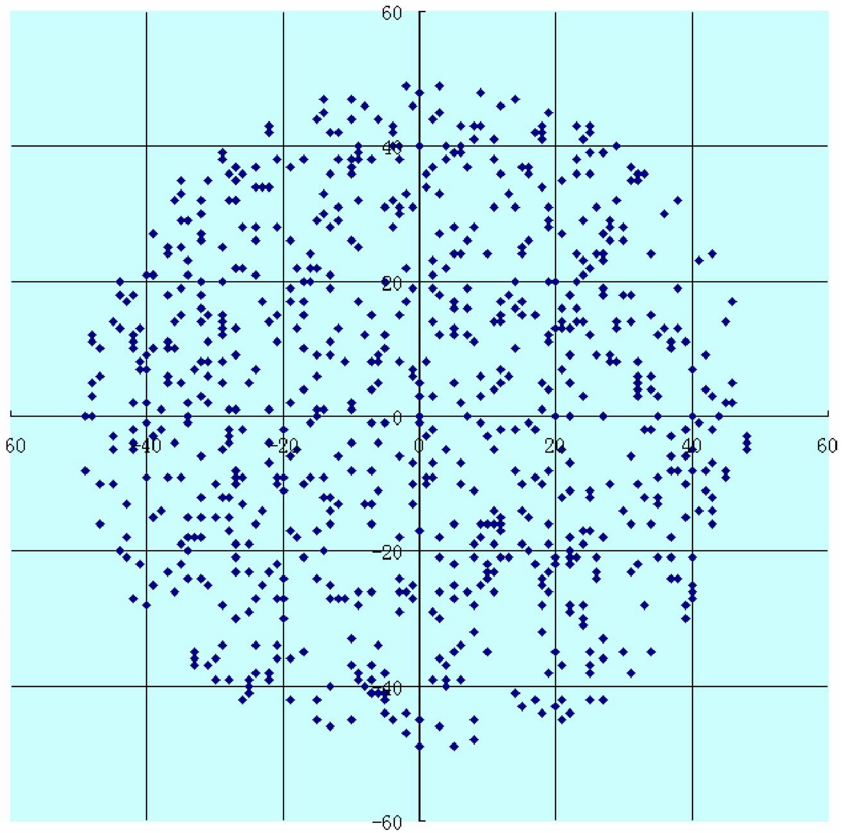


代码与效果

- 显然上述做法是不对的。但可以使用二维随机点的做法，若落在圆外，则重新生成点。结果如下。

代码与效果

```
int rand50()
{
    return rand() % 100 - 50;
}

int _tmain(int argc, _TCHAR* argv[])
{
    ofstream oFile;
    oFile.open(_T("D:\\rand.txt"));
    int x,y;
    for(int i = 0; i < 1000; i++)
    {
        x = rand50();
        y = rand50();
        if(x*x + y*y < 2500)
            oFile << x << '\t' << y << '\n';
    }
    oFile.close();
    return 0;
}
```



思考

- 不是每次生成随机数都能退出该算法
  - 有一定的接受率。
  - 请问：
    - 以多大的概率1次退出：接受率是多少？
    - 得到随机数的需要的平均次数(期望)是多少？
- 这个做法简洁、有效，值得推荐；
  - 许多相关问题，往往可以如此解决。

复习：一定接受率下的采样

- 已知有个rand7()的函数，返回1到7随机自然数，让利用这个rand7()构造rand10() 随机1~10。
- 解：因为rand7仅能返回1~7的数字，少于rand10的数目。因此，多调用一次，从而得到49种组合。超过10的整数倍部分，直接丢弃。

附：Code

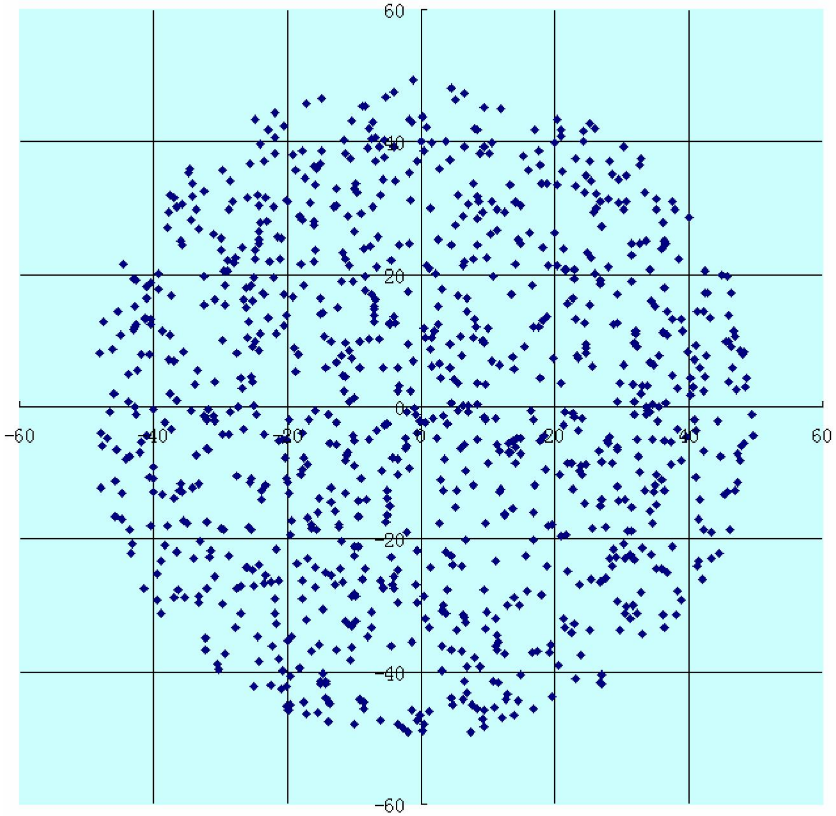
```
int rand10()
{
    int a1, a2, r;
    do
    {
        a1 = rand7() - 1;
        a2 = rand7() - 1;
        r = a1 * 7 + a2;
    } while (r >= 40);
    return r / 4 + 1;
}
```

圆内均匀取点的1次成功算法(朴素)

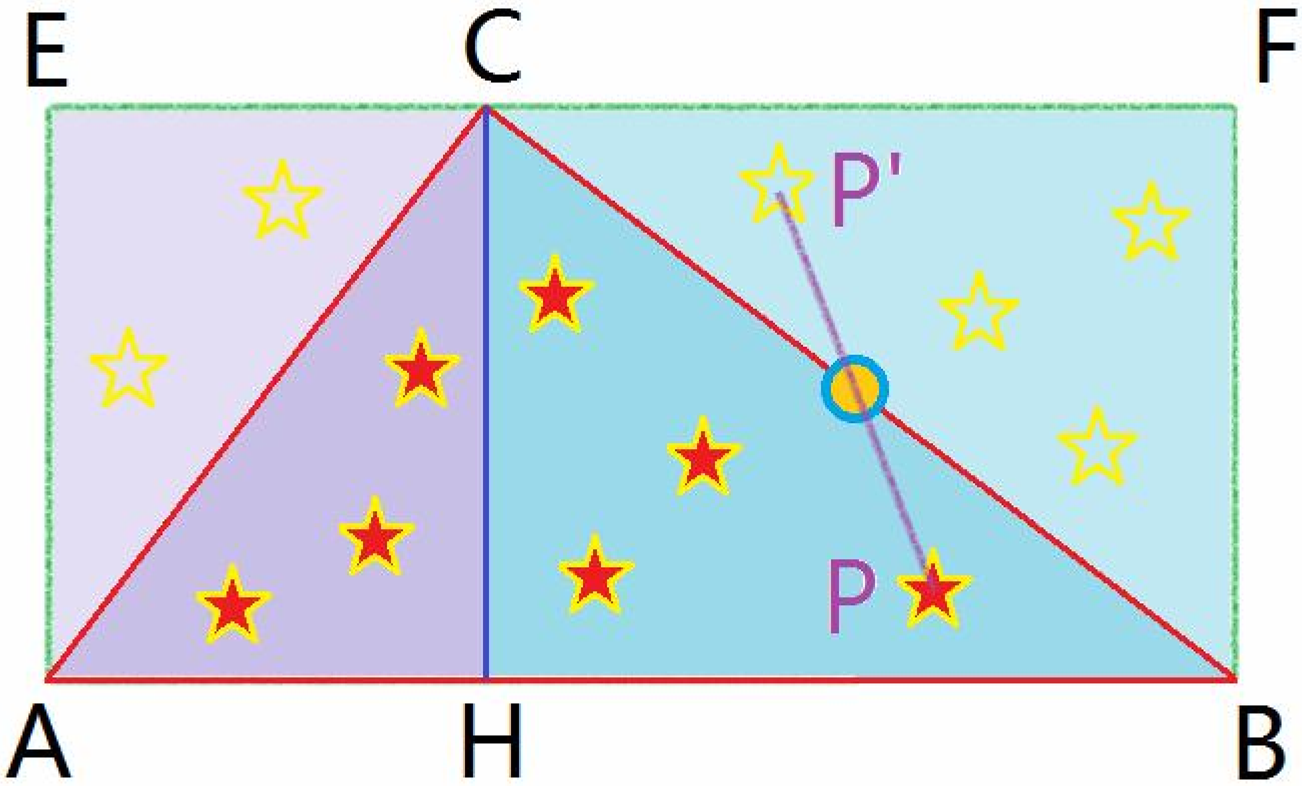
- 问题分析：把随机点看做面积很小的区域，圆内均匀取点意味着随机点P的面积与圆的面积成正比。
- $S_p=kS$
- $S=\pi r^2$，与半径的平方成正比
- 从而，$S_p(r)=k\pi r^2$
- 将均匀生成的随机数x取平方根赋值给r；则$S_p(r)$即为均匀分布。
- 同时，是与角度θ无关的，即：取均匀分布的随机数θ作为旋转角即可。

代码与效果

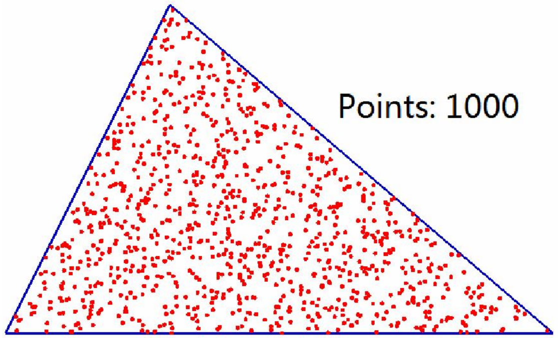
```
double rand2500()
{
    return rand() % 2500;
}

int _tmain(int argc, _TCHAR* argv[])
{
    ofstream oFile;
    oFile.open(_T("D:\\rand.txt"));
    double r,theta;
    double x,y;
    for(int i = 0; i < 1000; i++)
    {
        r = sqrt(rand2500());
        theta = rand();
        x = r*cos(theta);
        y = r*sin(theta);
        oFile << x << '\t' << y << '\n';
    }
    oFile.close();
    return 0;
}
```

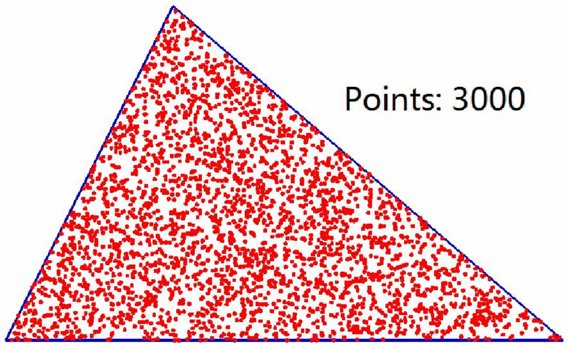


思考：将圆域换成三角形呢？

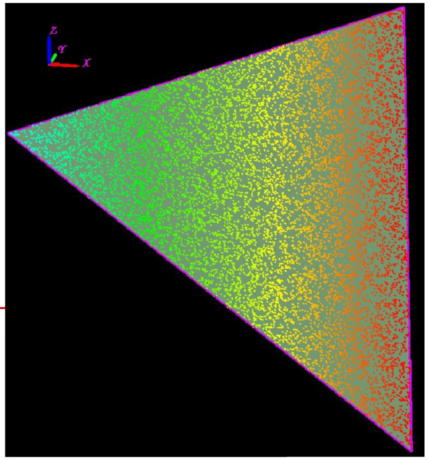


代码与效果






```
void CRandomTriangle::Random2(int nSize)
{
    CalcRotate();
    m_nSize = nSize;
    if(m_pRandomPoint)
        delete[] m_pRandomPoint;
    m_pRandomPoint = new CDelPoint[nSize];
    CDelPoint pt;
    for(int i = 0; i < nSize; i++)
    {
        pt.RandomInRectangle(m_ptExtend, m_ptHeight);
        if(m_tsBig.IsIn(pt))
        {
            pt += m_ptBase;
            m_pRandomPoint[i] = pt;
        }
        else if(m_tsLeft.IsIn(pt))
        {
            CDelPoint::MirrorPoint(pt, m_ptLeft0);
            pt += m_ptBase;
            m_pRandomPoint[i] = pt;
        }
        else if(m_tsRight.IsIn(pt))
        {
            CDelPoint::MirrorPoint(pt, m_ptRight0);
            pt += m_ptBase;
            m_pRandomPoint[i] = pt;
        }
    }
    CDelPoint::Save(m_pRandomPoint, m_nSize, _T("D:\\random.pt"), 0);
}
```



进一步思考

- 由于三点共面，所以三角形内的所有点必然在某平面上，因此，上述算法能够方便的推广到三维空间。
- 问题：请设计多边形内随机取点算法。
  - 圆内取点的思想：计算多边形的外包围矩形盒，生成外包围盒内的二维点，若点在多边形内，则退出；否则，继续探测。
  - 将多边形剖分成三角形集合，调用三角形内均匀取点算法。
- 算法2思路：
  - 按照面积为权重，选择某个三角形；
  - 生成该三角形内的随机点。
- 拓展
  - 每首歌有不同的分值，设计算法，根据分值随机推荐歌曲。
  - 如何将多边形快速剖分成三角形？注：Delaunay三角剖分

思考题

- 若某函数rand()以概率p(p≠0.5)返回数字0，以概率1-p返回数字1，如何利用该函数返回等概率的0和1？

复习：随机算法中的确定性问题

- 随机选词
- 如何随机选取1000个关键字？
- 给定一个数据流，其中包含未知数目的搜索关键字（比如，人们在搜索引擎中不断输入的关键字）。如何才能从这个未知数目的流中随机的选取k个关键字？

算法思路

- 开辟长度为k的缓冲区a[0…k-1]；
- 将前k个元素放置于a[0…k-1]中；
- 对于第i(i > k)个元素
  - 取随机数r∈[0,i-1]，(如：r= rand() % i)
  - 若r < k，则替换a[r]=a[i]

算法的合理性

- 第i (i≤k)个元素被选中的概率：

$$1\times\frac{k}{k+1}\times\frac{k+1}{k+2}\times\cdots\times\frac{N-1}{N}=\frac{k}{N}$$

- 第i (i > k)个元素被选中的概率：

$$\frac{k}{i}\times\frac{i}{i+1}\times\frac{i+1}{i+2}\times\cdots\times\frac{N-1}{N}=\frac{k}{N}$$

复习：任务安排

- 给定一台有m个储存空间的单进程机器；现有n个请求：第i个请求计算时需要占用R[i]个空间，计算完成后，储存计算结果需要占用O[i]个空间(其中O[i]<R[i])。问如何安排这n个请求的顺序，使得所有请求都能完成。
  - 如：m=14，n=2，R[1,2]=[10,8]，O[1,2]=[5,6]。可以先运行第一个任务，计算时占用10个空间，计算完成后占用5个空间，剩余9个空间执行第二个任务；但如果先运行第二个任务，则计算完成后仅剩余8个空间，第一个任务的计算空间就不够了。

算法分析

- 第k个任务的计算占用空间加上前面k-1个任务的空间占用量之和，越小越好。从而：

$$\begin{cases} O_1 + O_2 + \cdots + O_j + \cdots + O_{k-1} + R_k \\ O_1 + O_2 + \cdots + O_k + \cdots + O_{k-1} + R_j \end{cases}$$

$$\Rightarrow O_j + R_k \le O_k + R_j$$

$$\Rightarrow R_k - O_k \le R_j - O_j$$

- 得：将任务按照R[i]-O[i]降序排列即可。

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 2;    //任务数目
    const int M = 14;   //内存数目
    int R[N] = {10,8};
    int O[N] = {5,6};
    bool b = IsTaskable(N, M, R, O);
    cout << (b ? "Yes\n" : "No\n");
    return 0;
}
```

```
typedef struct tagTask
{
    int taskID;
    int RO;

    static bool Compare(const tagTask& t1, const tagTask& t2)
    {
        return t1.RO > t2.RO;
    }
}STask;

bool IsTaskable(int N, int M, const int* R, const int* O)
{
    STask* st = new STask[N];
    int i;
    for(i = 0; i < N; i++)
    {
        st[i].taskID = i;
        st[i].RO = R[i] - O[i];
    }
    sort(st, st+N, STask::Compare);

    int occupy = 0;
    bool bOK = true;
    int k;
    for(i = 0; i < N; i++)
    {
        k = st[i].taskID;
        if(occupy + R[k] > M)
        {
            bOK = false;
            break;
        }
        occupy += O[k];
    }
    delete[] st;
    return bOK;
}
```

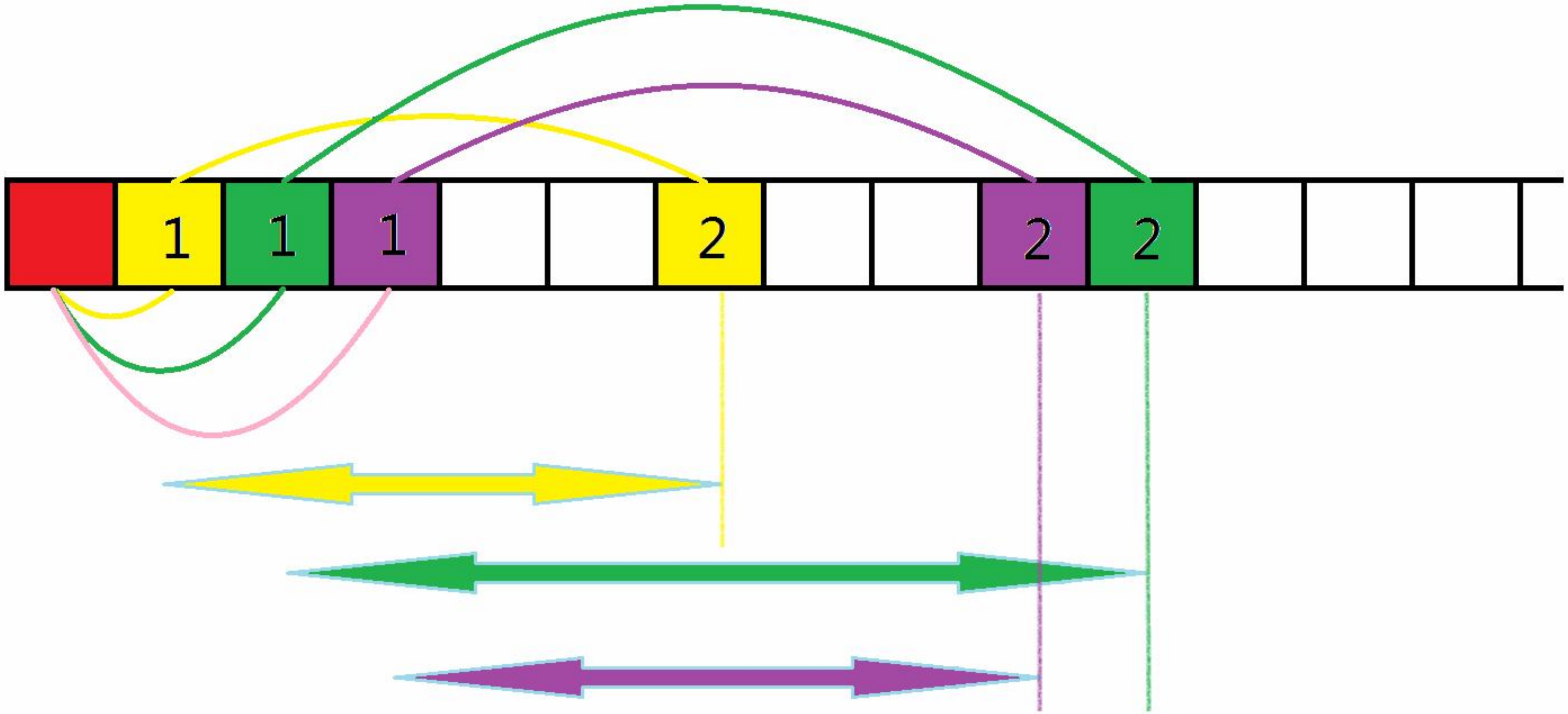
思考

- 本题可以看做是贪心法。
  - 在有限内存下，每次总是选择计算消耗(计算空间减去算后空间)最大的那个。
- 使用贪心法前需要经过严格的证明。

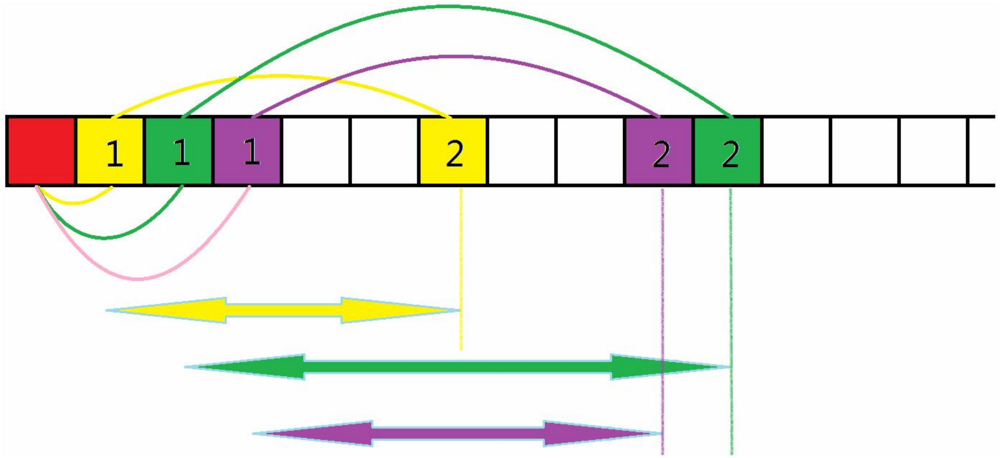
Jump

- 跳跃问题
- 给定非负整数数组，初始时在数组起始位置放置一机器人，数组的每个元素表示在当前位置机器人最大能够跳跃的数目。它的目的是用最少的步数到达数组末端。例如：给定数组A=[2,3,1,1,2]，最少跳步数目是2，对应的跳法是：2→3→2。
- 如：2,3,1,1,2,4,1,1,6,1,7，最少需要几步？

跳跃问题分析



跳跃问题算法步骤



- 初始步数step赋值为0；
- 记当前步的控制范围是[i,j]，则用k遍历i到j
  - 计算A[k]+k的最大值，记做j2；
- step++；继续遍历[j+1,j2]；

Code

```
int Jump(int A[], int n)
{
    if (n == 1)
        return 0;

    int step = 0; //最小步数
    int i = 0;
    int j = 0;  //[i,j]是当前能覆盖的区间
    int k,j2;
    while(j < n)  //覆盖区间尚未包含最后元素
    {
        step++;
        j2 = j;
        for(k = i; k <= j; k++)
        {
            j2 = max(j2, k + A[k]);
            if(j2 >= n-1)   //已经跳跃到最后一步
                return step;
        }
        i = j+1;
        j = j2;
        if(j < i)   //覆盖区间为负，说明无法跳到末尾
            return -1;
    }
    return step;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int A[] = {2,3,1,1,2,4,1,1,6,1,7};
    Jump(A, sizeof(A) / sizeof(int));
    return 0;
}
```

Jump问题“知识挖掘”

- 上述代码的时间复杂度是多少？
  - O(N) or $O(N^2)$
- 该算法能够天然处理无法跳跃到末尾的情况。
  - 若无法跳到莫问，则返回-1
- 该算法在每次跳跃中，都是尽量跳的更远，并记录j2——属于贪心法；也可以认为是从区间[i,j](若干结点)扩展下一层区间[j+1,j2] (若干子结点)——属于广度优先搜索。
  - 可见，贪心法是需要详细分析才能放心使用。
  - 回忆图论中的概要说明：
    - 广度优先搜索往往和“最少”、“最短”相关联。
- 思考：是否可以使用动态规划解决？
  - 记dp[i]为：到达A[i]时，还剩余多少步没有用。
  - 则：$dp[i+1]=\max(dp[i],A[i])-1$

找零钱

- 给定某不超过100万元的现金总额，兑换成数量不限的100、50、10、5、2、1的组合，共有多少种组合呢？

该问题的思考过程

- 此问题涉及两个类别：面额和总额。
  - 如果面额都是1元的，则无论总额多少，可行的组合数显然都为1。
  - 如果面额多一种，则组合数有什么变化呢？
- 定义dp[i][j]：使用面额小于等于i的钱币，凑成j元钱，共有多少种组合方法。
  - dp[100][500] = dp[50][500] + dp[100][400]
  - $dp[i][j] = dp[i_{small}][j] + dp[i][j-i]$
    - 不考虑j-i下溢出等边界问题

递推公式 $dp[i][j] = dp[i_{small}][j] + dp[i][j-i]$

- 使用dom[]={1,2,5,10,20,50,100}表示基本面额，i的意义从面额变成面额下标，则：
  - $dp[i][j] = dp[i-1][j] + dp[i][j-dom[i]]$
- 从而：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j \ge dom[i] \\ dp[i-1][j], & j < dom[i] \end{cases}$$

- 初始条件：

$$\begin{cases} dp[0][j]=1 \\ dp[i][0]=1 \end{cases}$$

Code

```cpp
int Charge(int value, const int* denomination, int size)
{
    int i;
    int** dp = new int*[size]; //dp[i][j]：用i面额以下的组合成j元
    for(i = 0; i < size; i++)
        dp[i] = new int[value+1];

    int j;
    for(j = 0; j <= value; j++) //只用面额1元的
        dp[0][j] = 1;

    for(i = 1; i < size; i++)  //先用面额小的，再用面额大的
    {
        dp[i][0] = 1;  //原因：添加任何一个面额，就是一个有效组合
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[i][j] = dp[i-1][j] + dp[i][j-denomination[i]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    int time = dp[size-1][value];

    for(i = 0; i < size; i++)  //清理内存
        delete[] dp[i];
    delete[] dp;
    return time;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int denomination[] = {1,2,5,10,20,50,100}; //面额
    int size = sizeof(denomination) / sizeof(int);
    int value = 200;
    int c = Charge(value, denomination, size);
    cout << c << endl;
    return 0;
}
```

滚动数组

- 将状态转移方程去掉第一维，很容易使用滚动数组，降低空间使用量。
- 原状态转移方程：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j \ge dom[i] \\ dp[i-1][j], & j < dom[i] \end{cases}$$

- 滚动数组版本的状态转移方程：

$$dp[j]=last[j]+dp[j-dom[i]],\quad (j \ge dom[i])$$

Code2

```
int Charge2(int value, const int* denomination, int size)
{
    int i;
    int* dp = new int[value+1]; //dp[j]: 凑成j元的组合数
    int* last = new int[value+1];

    int j;
    for(j = 0; j <= value; j++) //只用面额1元的
    {
        dp[j] = 1;
        last[j] = 1;
    }

    for(i = 1; i < size; i++)  //先用面额小的，再用面额大的
    {
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[j] = last[j] + dp[j-denomination[i]];
        }
        memcpy(last, dp, sizeof(int)*(value+1));
    }
    int chargeTimes = dp[value];

    delete[] last;
    delete[] dp;
    return chargeTimes;
}
```

总结与思考

- 请问：本问题的时间复杂度是多少？
- 在动态规划的问题中，如果不求具体解的内容，而只是求解的数目，往往可以使用滚动数组的方式降低空间使用量（甚至空间复杂度）
  - 由于滚动数组减少了维度，甚至代码会更简单
- 思考0-1背包问题和格子取数问题。

Word Break

- 分割词汇
- 给定一组字符串构成的字典dict和某字符串str，将str增加若干空格构成句子，使得str被分割后的每个词都在字典dict中。返回满足要求的分割str后的所有句子。如：
  - str="catsanddog",
  - dict=["cat","cats","and","sand","dog"]
  - 返回：["cats and dog","cat sand dog"]。

分割词汇问题分析

- 记长度为i的前缀串str[0…i-1]有至少一个可行划分，用布尔变量dp[i]表示，则：

$$dp[i] = \exists j\left(dp[j] \,\&\&\, str[j \cdots i-1] \in dict, 0 \le j \le i-1\right)$$

- catsanddog
- 初始条件dp[0]=true；
  - 若划分到最后是空串，则说明该划分是有效的；即默认空串即在字典中。
- 若只需要计算str是否可以划分成句子，直接返回dp[size]即可；该题目还需要返回所有的划分，所以，需要保存“前驱”。
  - 代码中将其记录为棋盘chess。

Code

```cpp
void AddAnswer(const string& str, const vector<int>& oneBreak, vector<string>& answer)
{
    int s = (int)oneBreak.size();
    int size = (int)str.length();
    answer.push_back(string());
    string& sentence = answer.back();
    sentence.reserve(size+s);   //申请足够的内容长度
    int start = 0, end = 0;
    for(int i = s-2; i >= 0; i--)   //oneBreak[size-1]==0, 特殊处理
    {
        end = oneBreak[i];  //别忘了, k=oneBreak[i]的值表示在string[k]的前面添加break
        sentence += str.substr(start, end-start);
        sentence += ' ';
        start = end;
    }
    sentence += str.substr(start, size-start);  //最后一个break
}

//计算str[0...cur-1]的wordbreak有哪些
void FindAnswer(const vector<vector<bool> >& chess, const string& str, int cur, vector<int>& oneBreak, vector<string>& answer)
{
    if(cur == 0)    //叶子
    {
        AddAnswer(str, oneBreak, answer);
        return;
    }
    int size = (int)str.length();
    for(int i = 0; i < cur-1; i++)
    {
        if(chess[cur][i])   //str[i...cur]在词典中
        {
            oneBreak.push_back(i);
            FindAnswer(chess, str, i, oneBreak, answer);
            oneBreak.pop_back();
        }
    }
}

void WordBreak(const set<string>& dict, const string& str, vector<string>& answer)
{
    int size = (int)str.length();
    //chess[i][j]: str[0...i-1]中, 是否可以在第j号元素的前面加break
    vector<vector<bool> > chess(size+1, vector<bool>(size));
    vector<bool> f(size+1); //f[i]: str[0...i-1]是否在词典中

    int i,j;
    f[0] = true;    //空串在词典中
    for(i = 1; i <= size; i++) //str[0...i-1]: 长度为i
    {
        for(j = i-1; j >= 0; j--)
        {
            if(f[j] && (dict.find(str.substr(j, i-j)) != dict.end()))   //str[j...i-1]
            {
                f[i] = true;
                chess[i][j] = true;
            }
        }
    }
    vector<int> oneBreak;   //一种可行的划分
    FindAnswer(chess, str, size, oneBreak, answer); //计算str[0...size-1]的wordbreak有哪些
}

void Print(const vector<string>& answer)
{
    vector<string>::const_iterator itEnd = answer.end();
    for(vector<string>::const_iterator it = answer.begin(); it != itEnd; it++)
        cout << *it << endl;
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("下雨天");
    dict.insert("留客");
    dict.insert("留客天");
    dict.insert("天留");
    dict.insert("留我不");
    dict.insert("我不留");
    dict.insert("留");
    dict.insert("dog");
    string str = "下雨天留客天留我不留";
    vector<string> answer;
    WordBreak(dict, str, answer);
    Print(answer);
    return 0;
}
```

Main Code

```cpp
//计算str[0...cur-1]的wordbreak有哪些
void FindAnswer(const vector<vector<bool> >& chess, const string& str, int cur,
                vector<int>& oneBreak, vector<string>& answer)
{
    if(cur == 0)    //叶子
    {
        AddAnswer(str, oneBreak, answer);
        return;
    }
    int size = (int)str.length();
    for(int i = 0; i < cur-1; i++)
    {
        if(chess[cur][i])   //str[i...cur]在词典中
        {
            oneBreak.push_back(i);
            FindAnswer(chess, str, i, oneBreak, answer);
            oneBreak.pop_back();
        }
    }
}

void WordBreak(const set<string>& dict, const string& str, vector<string>& answer)
{
    int size = (int)str.length();
    //chess[i][j]: str[0...i-1]中，是否可以在第j号元素的前面加break
    vector<vector<bool> > chess(size+1, vector<bool>(size));
    vector<bool> f(size+1); //f[i]: str[0...i-1]是否在词典中

    int i,j;
    f[0] = true;    //空串在词典中
    for(i = 1; i <= size; i++) //str[0...i-1]: 长度为i
    {
        for(j = i-1; j >= 0; j--)
        {
            if(f[j] && (dict.find(str.substr(j, i-j)) != dict.end()))   //str[j...i-1]
            {
                f[i] = true;
                chess[i][j] = true;
            }
        }
    }
    vector<int> oneBreak;   //一种可行的划分
    FindAnswer(chess, str, size, oneBreak, answer); //计算str[0...size-1]的wordbreak有哪些
}
```

Aux Code

```
void AddAnswer(const string& str, const vector<int>& oneBreak, vector<string>& answer)
{
    int s = (int)oneBreak.size();
    int size = (int)str.length();
    answer.push_back(string());
    string& sentence = answer.back();
    sentence.reserve(size+s);   //申请足够的内容长度
    int start = 0, end = 0;
    for(int i = s-2; i >= 0; i--)   //oneBreak[size-1]==0, 特殊处理
    {
        end = oneBreak[i];  //别忘了, k=oneBreak[i]的值表示在string[k]的前面添加break
        sentence += str.substr(start, end-start);
        sentence += ' ';
        start = end;
    }
    sentence += str.substr(start, size-start);  //最后一个break
}
```

下雨天．留客．天留．我不留

下雨天．留客天．留．我不留

下雨天．留客天．留我不．留

```
void Print(const vector<string>& answer)
{
    vector<string>::const_iterator itEnd = answer.end();
    for(vector<string>::const_iterator it = answer.begin();
        it != itEnd; it++)
        cout << *it << endl;
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("下雨天");
    dict.insert("留客");
    dict.insert("留客天");
    dict.insert("天留");
    dict.insert("留我不");
    dict.insert("我不留");
    dict.insert("留");
    dict.insert("dog");
    string str = "下雨天留客天留我不留";
    vector<string> answer;
    WordBreak(dict, str, answer);
    Print(answer);
    return 0;
}
```

Distinct Subsequences

- 子序列数目
- 给定文本串Text和模式串Pattern，计算文本串Text的子序列中包含模式串Pattern的个数——模式串Pattern以子序列的形式在文本串Text中出现过几次。
  - 如“rabbit”在“rabbbit”中出现过3次。
  - “ab”在“abacab”出现过4次。
    - Text[0,1]= Text[0,5]=Text[2,5]=Text[4,5]=“ab”

动态规划解决子序列数目问题

- 记Pattern[0...j]在Text[0...i]中出现次数为dp[i,j]，借鉴LCS的思想：
- 若Pattern[j]≠Text[i]
  - 则Text[0...i]和Text[0...i-1]对于Pattern[0...j]表达能力相同，即：dp[i,j]=dp[i-1,j]
- 若Pattern[j] = Text[i]
  - Text[0...i-1]表达Pattern[0...j-1]后，最后缀上Text[i]
  - 或者Text[0...i-1]直接表达Pattern[0...j]
  - 即： dp[i,j]=dp[i-1,j-1]+ dp[i-1,j]

状态转移方程和初值

- 写出状态转移方程，得：

$$dp(i,j)=\begin{cases} dp(i-1,j) & Text[i]\neq Pattern[j] \\ dp(i-1,j-1)+dp(i-1,j) & Text[i]=Pattern[j] \end{cases}$$

- 初值：dp(0,0)=1
  - 空串在空串中出现过1次。

滚动数组 $dp(i,j)=\begin{cases} dp(i-1,j) & Text[i]\neq Pattern[j] \\ dp(i-1,j-1)+dp(i-1,j) & Text[i]=Pattern[j] \end{cases}$

- dp(i,j)的更新只需要前面一行元素，同时不需要记录路径，所以，可以使用滚动数组(回忆LCS中的“进一步思考”)，得：

$$dp(j)=\begin{cases} dp(j) & Text[i]\neq Pattern[j] \\ dp(j-1)+dp(j) & Text[i]=Pattern[j] \end{cases}$$

- 注意：
  - dp(0)=1：空串在空串中出现过1次
  - 因为计算dp(j+1)要用到dp(j)，所以，更新dp(j+1)要在的dp(j)之前——即：要从后向前更新。

子序列数Code

```cpp
int DistinctSubsequence(const char* pText, const char* pPattern)
{
    int size1 = (int)strlen(pText);
    int size2 = (int)strlen(pPattern);
    if(size1 < size2)
        return 0;

    int* pSize = new int[size2+1];
    pSize[0] = 1;   //空串在空串中出现1次
    memset(pSize+1, 0, sizeof(int)*size2);

    int i,j;
    for(i = 0; i < size1; i++)
    {
        for(j = size2-1; j >= 0; j--)
        {
            if(pText[i] == pPattern[j])
                pSize[j+1] += pSize[j];
        }
    }
    int s = pSize[size2];
    delete[] pSize;
    return s;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char text[] = "abacab";
    char pattern[] = "ab";
    cout << DistinctSubsequence(text, pattern) << endl;
    return 0;
}
```

Longest Substring Without Repeating Characters

- 无重复字符的最长子串
- 对于给定的字符串，返回它最长的无重复字符的子串的长度，如：字符串“abcabcbb”的无重复最长子串是“abc”，长度为3；字符串“bbbb”的无重复最长子串是“b”，长度为1。
  - 假定字符只包含26个英文小写字母。

从暴力求解开始分析

- 既然计算子串，则设置两个索引i，j分别指向子串首尾，判断该子串是否有重复字符。
  - i，j从0到N-1，子串最长为N，时间复杂度$O(N^3)$ / $O(N^4)$
- 如何判断一个字符串str[i,i+1…j]是否有重复数字？
  - k从i+1到j遍历，判断str[k]是否在str[i…k-1]中出现？
    - 本身已经是$O(N^2)$。
  - 考虑到只有26个字母，所以，使用缓存exist[26]：
    - 初始化为-1
    - k从i+1到j遍历，若str[k]所在的缓存位置exist[str[k]-'a']为-1，表示str[k]未出现过，则标记exist[str[k]-'a']=k，继续k+1的考察；若exist[str[k]-'a']不为-1，表示str[k]出现过，则该子串不是无重复字符的字符串。
  - 以上"缓存"的思路，能否用来解决整个问题的优化？

继续分析无重复最大子串的优化方法

- 使用exist[‘A’~’Z’][N]：
  - exsit[‘a’]表示：字符’a’在字符串str中出现的位置。
- 事实上：只需要记录str[j]在str[0…j-1]最后一次出现的位置k，那么，对于子串str[i…j]：
  - 如果k＞i，则表示str[k]==str[j]，即子串不是无重复串。
- str[0…j-1]最后一次出现的位置不需要提前计算好，边向后查找边更新即可。
- 时间复杂度O(N)，空间复杂度O(1)。
  - 如果把exist[26]当成O(1)的话。

Code

```
const int CHARATER_MAX = 26;
int LongestSubstringUnique(char* str, int size)
{
    int last[CHARATER_MAX]; //记录字符上次出现过的位置
    int start = 0;          //记录当前子串的起始位置
    fill(last, last + CHARATER_MAX, -1);
    int nMax = 0;
    for(int i = 0; i < size; i++)
    {
        if(last[str[i] - 'a'] >= start)  //str[start...i]中出现重复，重新开始记录
        {
            nMax = max(i - start, nMax);
            start = last[str[i] - 'a'] + 1;
        }
        last[str[i] - 'a'] = i;     //记录str[i]最后出现的位置
    }
    return max(size-start, nMax);
}

int _tmain(int argc, _TCHAR* argv[])
{
    char string[] = "abcabcbb";
    LongestSubstringUnique(string, sizeof(string) / sizeof(char) - 1);
    return 0;
}
```

参考文献

- 余祥宣等，计算机算法基础[M]，华中科技大学出版社，2001
- 戴方勤,LeetCode 题解,2014

我们在这里

- 更多算法面试题在 7 七月算法
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月算法问答
  - @邹博_机器学习

感谢大家！

欢迎大家提出宝贵的意见！