算法杂谈

七月算法　邹博

2015年6月28日

再论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。

Code1

- DP

```
bool WordBreak1(const set<string>& dict, const string& str)
{
    int size = (int)str.length();
    vector<bool> f(size+1); //f[i]：str[0...i-1]是否在词典中

    f[0] = true;
    int i,j;
    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;
                break;
            }
        }
    }
    return f[size];
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("cat");
    dict.insert("cats");
    dict.insert("and");
    dict.insert("sand");
    dict.insert("dog");
    dict.insert("dog");
    string str = "catsanddog";
    if(WordBreak1(dict, str))
        cout << "Break is TRUE\n";
    else
        cout << "Break is FALSE\n";
    return 0;
}
```

Code2 递归

```
bool WordBreak2(const set<string>& dict, const string& str)
{
    int size = (int)str.length();
    if(size == 0)
        return true;

    for(int i = size-1; i >= 0; i--)
    {
        if(WordBreak2(dict, str.substr(0,i))
        &&(dict.find(str.substr(i, size-i)) != dict.end()))
            return true;
    }
    return false;
}
```

Code3

暂存空间的递归

```
int WordBreak3(const set<string>& dict, const string& str, vector<int>& f)
{
    int size = (int)str.length();
    for(int i = size-1; i >= 0; i--)
    {
        if(f[i] == 0)
            f[i] = WordBreak3(dict, str.substr(0,i), f);

        if((f[i] == 1) && (dict.find(str.substr(i, size-i)) != dict.end()))
            return 1;
    }
    return -1;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("cat");
    dict.insert("cats");
    dict.insert("and");
    dict.insert("sand");
    dict.insert("dog");
    dict.insert("dog");
    string str = "catsandsdog";
    int size = (int)str.length();
    vector<int> f(size+1);  //f[i]：str[0...i-1]是否在词典中
    memset(&f.front() + 1, 0, sizeof(int)*size);
    f[0] = 1;
    if(WordBreak3(dict, str, f) == 1)
        cout << "Break is TRUE\n";
    else
        cout << "Break is FALSE\n";
    return 0;
}
```

Last 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
}

//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

```
//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;
}
```

总结与思考

- 通过递推关系，很容易写出递归代码或动态规划代码。一般的说，动态规划即利用空间存放小规模问题的解，以期便于总问题的求解。递归的过程中，可以借鉴这种方案，保存中间解的结果，避免重复计算。
- 因为递归计算的中间结果必然是最终结果所需要的，有些情况下，可以避免动态规划中计算所有小规模解造成的浪费。
  - 思考：走迷宫问题，往往是从出口回溯。
- 如果需要通过计算具体解，则需要回溯；如果需要计算所有解，则需要深度/广度优先搜索。

下面这段代码，会输出什么？

- 5 or 6

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int c = 5;
    int* p = (int*)&c;
    *p = 6;
    cout << c;
    int x = c;

    int i = 8;
    int j = i;
    int* p2 = &i;
    return 0;
}
```

深层理解
const & pointer

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int c = 5;
    int* p = (int*)&c;
    *p = 6;
    cout << c;
    int x = c;

    int i = 8;
    int j = i;
    int* p2 = &i;
    return 0;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
0041BD60  push        ebp
0041BD61  mov         ebp,esp
0041BD63  sub         esp,108h
0041BD69  push        ebx
0041BD6A  push        esi
0041BD6B  push        edi
0041BD6C  lea         edi,[ebp-108h]
0041BD72  mov         ecx,42h
0041BD77  mov         eax,0CCCCCCCCh
0041BD7C  rep stos    dword ptr [edi]
    const int c = 5;
0041BD7E  mov         dword ptr [c],5
    int* p = (int*)&c;
0041BD85  lea         eax,[c]
0041BD88  mov         dword ptr [p],eax
    *p = 6;
0041BD8B  mov         eax,dword ptr [p]
0041BD8E  mov         dword ptr [eax],6
    cout << c;
0041BD94  push        5
0041BD96  mov         ecx,offset std::cout (457668h)
0041BD9B  call        ostream<char,char_traits<char> >::operator<<(4195D2h)
    int x = c;
0041BDA0  mov         dword ptr [x],5

    int i = 8;
0041BDA7  mov         dword ptr [i],8
    int j = i;
0041BDAE  mov         eax,dword ptr [i]
0041BDB1  mov         dword ptr [j],eax
    int* p2 = &i;
0041BDB4  lea         eax,[i]
0041BDB7  mov         dword ptr [p2],eax
    return 0;
0041BDBA  xor         eax,eax
}
```

从上述汇编分析得出结论

- 指针是汇编级直接支持的结构：
  - i=func();
    - 伪代码：mov dword ptr [i], func
  - p是指针：即p的值表示了某内存地址。
    - *p = i;
      - lea eax,[i]
      - mov dword ptr [p], exa
- const是高级语言在编译期间实现的内容：
  - const int c = 5;
    - move dword ptr [c], 5
  - int x = c;
    - mov dword ptr [x], 5

Code

- 这段代码有问题吗？

Kitty	Puppy
1	2

```
class animal
{
protected:
    int age;
public:
    virtual void MyAge(void) = 0;
};

class dog : public animal
{
public:
    dog()
    {
        age = 2;
    }
    ~dog() {}
    virtual void MyAge(void)
    {
        cout<< "Wang, my age = " << age <<endl;
    }
};

class cat: public animal
{
public:
    cat()
    {
        age = 1;
    }
    ~cat() {}
    virtual void MyAge(void)
    {
        cout<< "Miao, my age = " << age << endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    kitty = puppy;
    return 0;
}
```

使用指针的方法传递

```
int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    animal* pKitty = &kitty;
    animal* pPuppy = &puppy;
    *pKitty = *pPuppy;
    kitty.MyAge();
    return 0;
}
```

Kitty	Puppy
1	2

如果换成其他指针呢？

```
int _tmain(int argc, _TCHAR* argv[])
{
    cat kitty;
    dog puppy;
    int* pKitty = (int*)&kitty;
    int* pPuppy = (int*)&puppy;
    *pKitty = *pPuppy;
    kitty.MyAge();
    return 0;
}
```

Kitty	Puppy
1	2

Kitty	Puppy
virtual	virtual
1	2

思考

- 可以看出，所谓指针的类型，仅为高级语言加入的特性，在汇编层，都是变量的地址而已。
- 这段代码输出什么？

```
class animal
{
private:
    int age;
public:
    animal(int _age)
    {
        age = _age;
    }
    void PrintAge()
    {
        cout << age << endl;
    }

};

int _tmain(int argc, _TCHAR* argv[])
{
    animal a(1);
    int* p = (int*)&a;
    *p = 2;
    a.PrintAge();
    return 0;
}
```

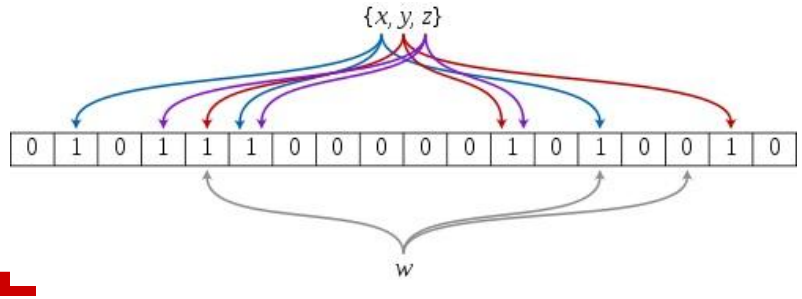
Bloom Filter

- 布隆过滤器(Bloom Filter)是由Burton Howard Bloom于1970年提出的，它是一种空间高效(space efficient)的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单、爬虫(Crawler)的网址判重等问题中经常被用到。
- 哈希表也能用于判断元素是否在集合中，但是BloomFilter只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。BloomFilter可以插入元素，但不可以删除已有元素。集合中的元素越多，误报率(false positive rate)越大，但是不会漏报(false negative)。

Bloom Filter

- 如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比对来判定是否在集合内：链表、树等数据结构都是这种思路。但是随着集合中元素数目的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。
- 可以利用Bitmap：只要检查相应点是不是1就知道可以集合中有没有某个数。这就是Bloom Filter的基本思想。

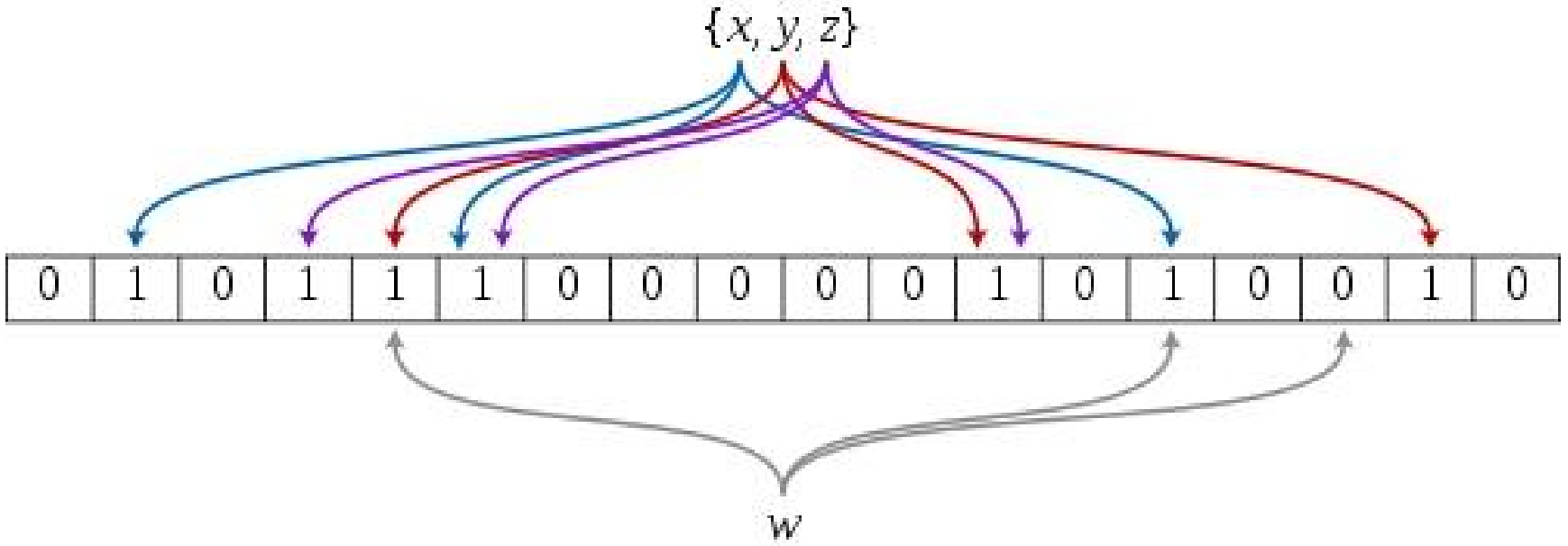
Bloom Filter算法描述



- 一个空的Bloom Filter是一个有m位的位向量B，每一个bit位都初始化为0。同时，定义k个不同的Hash函数，每个Hash函数都将元素映射到m个不同位置中的一个。
  - 记：n为元素数，m为位向量B的长度(位：槽slot)，k为Hash函数的个数。
- 增加元素x
  - 计算k个Hash(x)的值$(h_1,h_2\ldots h_k)$，将位向量B的相应槽$B[h_1,h_2\ldots h_k]$都设置为1；
- 查询元素x
  - 即判断x是否在集合中，计算k个Hash(x)的值$(h_1,h_2\ldots h_k)$。若$B[h_1,h_2\ldots h_k]$全为1，则x在集合中；若其中任一位不为1，则x不在集合中；
- 删除元素x
  - 不允许删除！因为删除会把相应的k个槽置为0，而其中很有可能有其他元素对应的位。

Bloom Filter 插入查找数据

- 插入x,y,z
- 判断w是否在该数据集中



BloomFilter的特点

- 不存在漏报：某个元素在某个集合中，肯定能报出来；
- 可能存在误报：某个元素不在某个集合中，可能也被认为存在：false positive；
- 确定某个元素是否在某个集合中的代价和总的元素数目无关
  - 查询时间复杂度：O(1)

Bloom Filter参数的确定

- 单个元素某次没有被置位为1的概率为：$1-\frac{1}{m}$
- k个Hash函数中没有一个对其置位的概率为：$\left(1-\frac{1}{m}\right)^{k}$
- 如果插入n个元素，仍未将其置位的概率为：$\left(1-\frac{1}{m}\right)^{kn}$
- 因此，此位被置位的概率为：$1-\left(1-\frac{1}{m}\right)^{kn}$

Bloom Filter参数的确定

- 查询中，若某个待查元素对应的k位都被置位，则算法会判定该元素在集合中。因此，该元素被误判的概率(上限)为：

$$q(k)=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$$

- 考虑到：

$$\left(1-\frac{1}{m}\right)^{kn}=\left(1+\frac{1}{-m}\right)^{-m\cdot\frac{-kn}{m}}=\left(\left(1+\frac{1}{-m}\right)^{-m}\right)^{-\frac{kn}{m}}\approx e^{-\frac{kn}{m}}$$

- 从而：

$$P(k)\approx\left(1-e^{-\frac{kn}{m}}\right)^{k}$$

Bloom Filter参数的确定

- P(k)为幂指函数，取对数后求导：

$$P(k)=\left(1-e^{-\frac{kn}{m}}\right)^k \xrightarrow{令b=e^{\frac{n}{m}}} \left(1-b^{-k}\right)^k$$

$$\Rightarrow \ln P(k)=k\ln\left(1-b^{-k}\right)$$

$$\xrightarrow{取关于k的导数} \frac{1}{P(k)}P'(k)=\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{1-b^{-k}}$$

$$\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{\left(1-b^{-k}\right)}=0$$

$$\Rightarrow \left(1-b^{-k}\right)\ln\left(1-b^{-k}\right)=b^{-k}\ln b^{-k}$$

$$\Rightarrow 1-b^{-k}=b^{-k} \Rightarrow b^{-k}=\frac{1}{2} \Rightarrow e^{-\frac{kn}{m}}=\frac{1}{2}$$

$$\Rightarrow k=\ln 2\cdot\frac{m}{n}\approx 0.693\cdot\frac{m}{n}$$

$$P(k)=\left(1-\frac{1}{2}\right)^k=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$

参数m、k的确定

- m的计算公式：
  - 由 $$P(k)=\left(1-\frac{1}{2}\right)^{k}=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$
  - 得 $$P=2^{-\ln 2\frac{m}{n}}\Rightarrow \ln P=\left(\ln 2\cdot\frac{m}{n}\right)\ln 2\Rightarrow m=\frac{\ln P^{-1}}{(\ln 2)^{2}}\cdot n$$
- 此外，k的计算公式：$$k=\ln 2\cdot\frac{m}{n}=\frac{\ln P^{-1}}{\ln 2}$$
- 至此，任意先验给定可接受的错误率，即可确定参数空间m和Hash函数个数k。

Bloom Filter参数的讨论

- 1.442695041
- 若接收误差率为10^{-6}时，需要位的数目为29*n。

$$\begin{cases} m = \dfrac{\ln P^{-1}}{(\ln 2)^2} \cdot n \\ k = \ln 2 \cdot \dfrac{m}{n} = \dfrac{\ln P^{-1}}{\ln 2} \end{cases}$$

p	m/n	k
0.5	1.442695041	1
2^-2	2.885390082	2
2^-3	4.328085123	3
2^-4	5.770780164	4
2^-5	7.213475204	5
2^-6	8.656170245	6
2^-7	10.09886529	7
2^-8	11.54156033	8
2^-9	12.98425537	9
2^-10	14.42695041	10
2^-20	28.85390082	11
2^-30	43.28085123	12
2^-40	57.70780164	13
2^-50	72.13475204	14

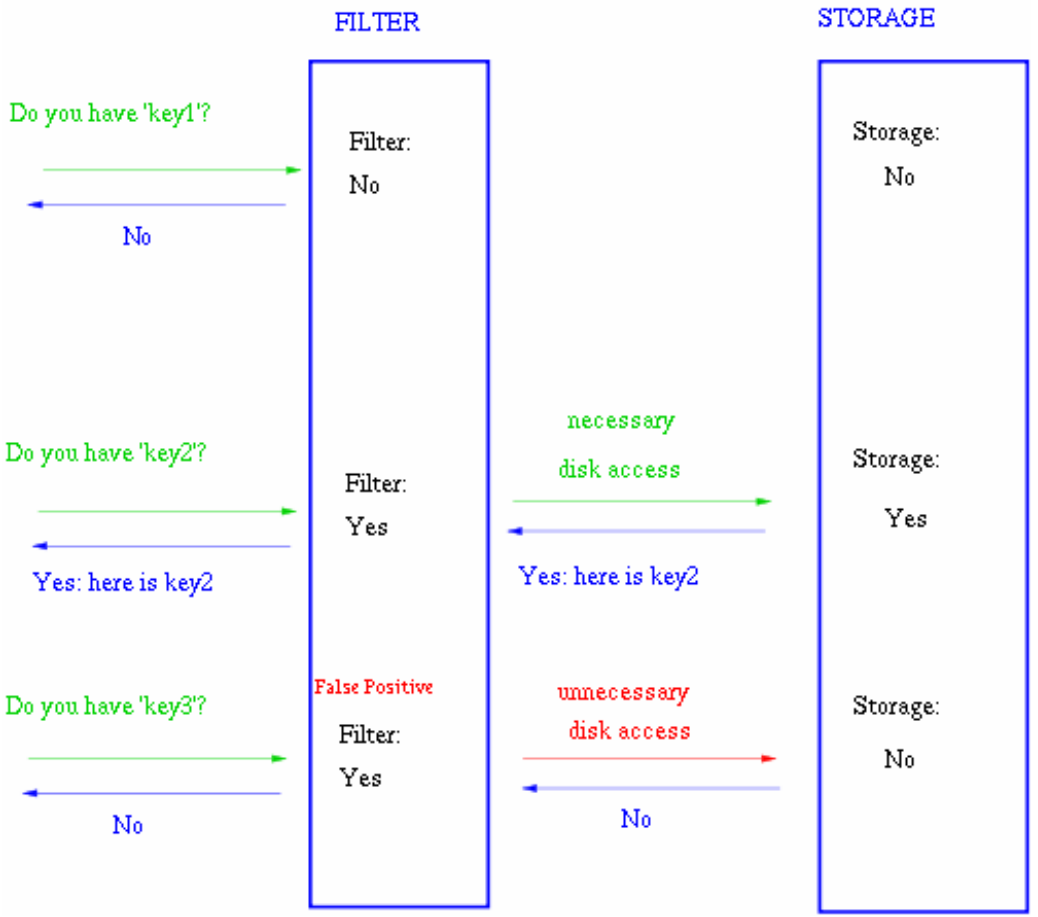
Bloom Filter的特点

- 优点：相比于其它的数据结构，Bloom Filter在空间和时间方面都有巨大的优势。Bloom Filter存储空间是线性的，插入/查询时间都是常数。另外，Hash函数相互之间没有关系，方便由硬件并行实现。Bloom Filter不存储元素本身，在某些对保密要求非常严格的场合有优势。
- 很容易想到把位向量变成整数数组，每插入一个元素相应的计数器加1，这样删除元素时将计数器减掉就可以了。然而要保证安全的删除元素并非如此简单。首先我们必须保证删除的元素的确在BloomFilter里面。这一点单凭这个过滤器是无法保证的。另外计数器下溢出也会造成问题(槽的值已经是0了，仍然执行删除操作)。

BloomFilter用例

- Google著名的分布式数据库Bigtable使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数；
- Squid网页代理缓存服务器在cachedigests中使用了BloomFilter；
- Venti文档存储系统采用BloomFilter来检测先前存储的数据；
- SPIN模型检测器使用BloomFilter在大规模验证问题时跟踪可达状态空间；
- Google Chrome浏览器使用BloomFilter加速安全浏览服务；
- 在很多Key-Value系统中也使用BloomFilter来加快查询过程，如Hbase，Accumulo，Leveldb。
  - 一般而言，Value保存在磁盘中，访问磁盘需要花费大量时间，然而使用BloomFilter可以快速判断某个Key是否存在，因此可以避免很多不必要的磁盘IO操作；另外，引入布隆过滤器会带来一定的内存消耗。

Bloom Filter + Storage结构



排序的目的

- 排序本身：得到有序的序列
- 方便查找
  - 长度为N的有序数组，查找某元素的时间复杂度是多少？
  - 长度为N的有序链表，查找某元素的时间复杂度是多少？
    - 单链表、双向链表
    - 如何解决该问题？

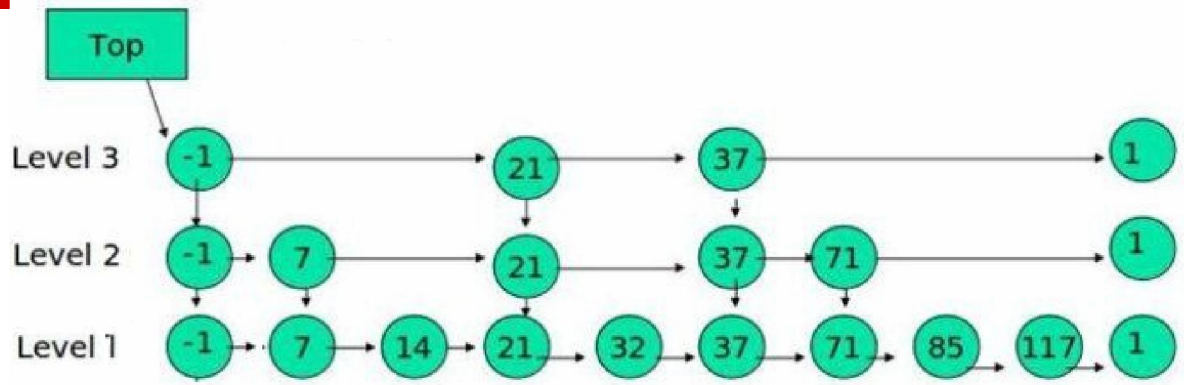
跳跃链表(Skip List)

- Treaps/RB-Tree/BTree
- 跳跃链表是一种随机化数据结构，基于并联的链表，其效率可比拟于二叉查找树(对于大多数操作需要O(logn)平均时间)。具有简单、高效、动态(Simple、Effective、Dynamic)的特点。
- 基本上，跳跃列表是对有序的链表附加辅助链表，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分结点(因此得名)。查找结点、增加结点、删除结点操作的期望时间都是logN的(with high probability≈1-1/(n^ α), W.H.P.)。

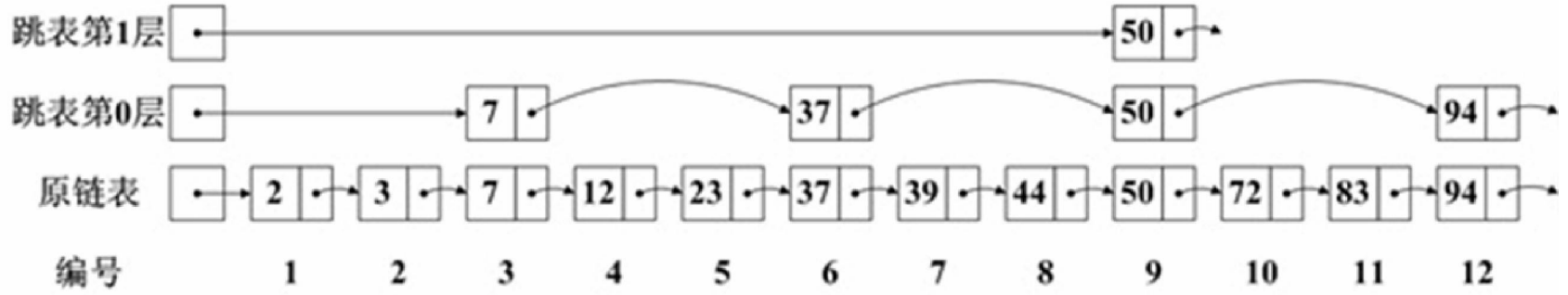
跳跃链表(Skip List)

- 跳跃列表在并行计算中也很有用，这里的插入可以在跳跃列表不同的部分并行进行，而不用全局的数据结构重新平衡。
- 跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面列表的“快速跑道”，这里在层i中的元素按某个固定的概率p出现在层i+1中。平均起来，每个元素都在1/(1-p)个列表中出现。
  - 思考：为什么是1/(1-p)?

跳跃表示例

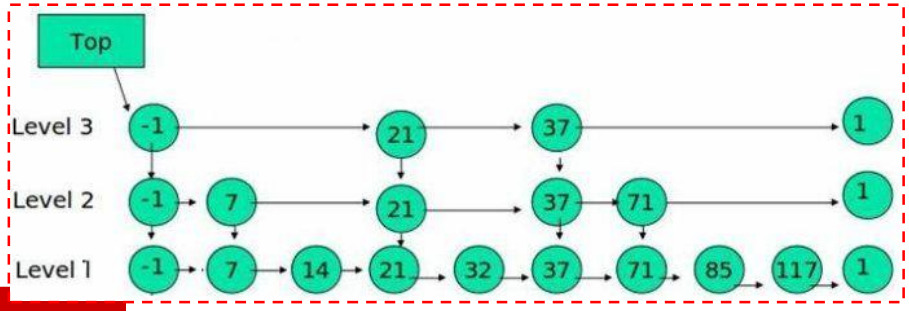


跳跃表：跳跃间隔(Skip Interval) 为 3，层次(Level)共2层



注：各个文献中对于“层”、“间隔”的定义略有差别

双层跳表时间计算

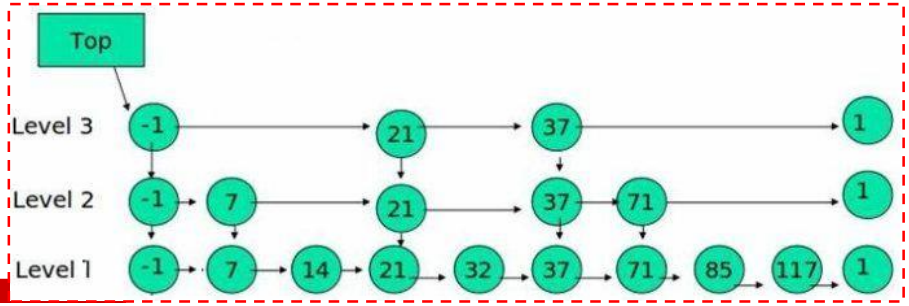


- 粗略估算查找时间：T=L1 + L2/L1(L1是稀疏层，L2是稠密层)
  - 在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度
  - L1：在稀疏层的最差查找次数
  - L1/L2：在稀疏层没有找到元素，跳转到稠密层需要找的次数
- 若基本链表的长度为n，即|L2|=n，|L1|为多少，T最小呢？
  - T(x)=x+n/x，对x求导，得到x=sqrt(n)
  - min(T(x))=2sqrt(n)

时间复杂度分析

- 粗略估算查找时间：T=L1 + L2/L1 + L3/L2(L1是稀疏层，L2是稠密层，L3是基本层)
  - 在L3上均匀取值，构成L2；在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度， L3/L2是L2上相邻两个元素在L3上的平均长度
- 若基本链表的长度为n，即|L3|=n，|L1|、|L2|为多少，T最小呢？
  - T(x,y)=x+y/x+n/y，对x,y求偏导，得到x= $\sqrt[3]{n}$
  - min(T(x,y))= $3*\sqrt[3]{n}$

跳表最优时间分析



- 建立k层的辅助链表，可以得到最小时间 $T(n)=k*\sqrt[k]{n}$
- 问题：在n已知的前提下，k取多大最好呢？
- 显然， 当k=logN时， T(n)=logN * n^(-logN)
- 问： n^(-logN)等于几？
- 一个很容易在实践中使用的结论是：
  - 当基本链表的数目为N时，共建立k=logN个辅助链表，每个上层链表的数码取下层链表的一半，则能够达到logN的时间复杂度！
- 理想跳表：ideal skip list

插入元素

- 随着底层链表的插入，某一段上的数据将不满足理想跳表的要求，需要做些调整。
  - 将底层链表这一段上元素的中位数在拷贝到上层链表中；
  - 重新计算上层链表，使得上层链表仍然是底层链表的1/2；
  - 如果上述操作过程中，上层链表不满足要求，继续上上层链表的操作。
- 新的数据应该在上层甚至上上层链表中吗？因为要找一半的数据放在上层链表(为什么是一半？)，因此：抛硬币！

插入元素后的跳表维护

- 考察待需要提升的某段结点。
- 若抛硬币得到的随机数p>0.5，则提升到上层，继续抛硬币，直到p>0.5；
  - 或者到了顶层仍然p>0.5，建立一个新的顶层

删除元素

- 在某层链表上找到了该元素，则删除；如果该层链表不是底层链表，跳转到下一层，继续本操作。

进一步说明的问题

- 若多次查找，并且相邻查找的元素有相关性(相差不大)，可使用记忆化查找进一步加快查找速度。
- 对关于k的函数$T(n)=k*\sqrt[k]{n}$求导，可计算得到k=lnN处函数取最小值，最小值是elnN。
  - 对于N=10000，k取lnN和logN，两个最小值分布是：25.0363和26.5754。
- 强调：编程方便，尤其方便将增删改查操作扩展成并行算法。
- 跳表用单链表可以实现吗？用双向链表呢？

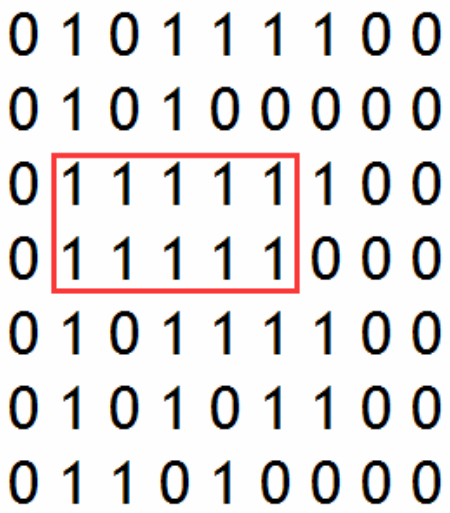
进行10^6次随机操作后的统计结果

P	平均操作时间	平均列高	总结点数	每次查找跳跃次数（平均值）	每次插入跳跃次数（平均值）	每次删除跳跃次数（平均值）
2/3	0.0024690 ms	3.004	91233	39.878	41.604	41.566
1/2	0.0020180 ms	1.995	60683	27.807	29.947	29.072
1/e	0.0019870 ms	1.584	47570	27.332	28.238	28.452
1/4	0.0021720 ms	1.330	40478	28.726	29.472	29.664
1/8	0.0026880 ms	1.144	34420	35.147	35.821	36.007

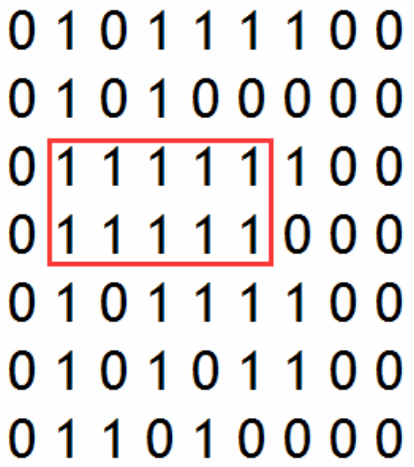
进行10^6次随机操作后的统计结果

Maximal Rectangle

- 最大全一矩形
- 给定二维布尔矩阵，元素只能取0或者1，找出只包含元素1并且面积最大的矩阵，返回它的面积。

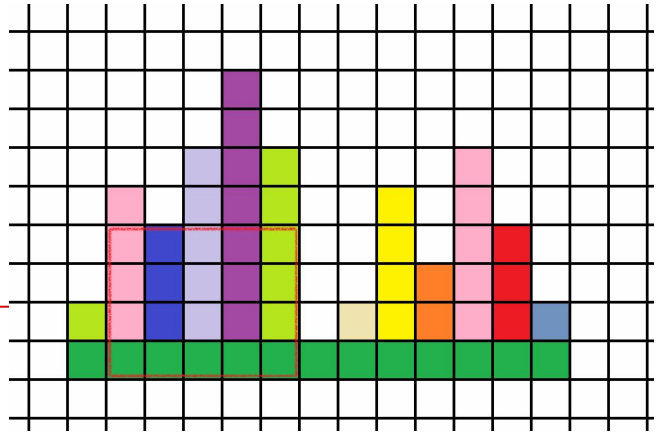


问题分析



- 首先想到的是动态规划。
- 记以(i,j)为右下角的矩形最大面积为dp(i,j)
  - 在向右、向下扩展时，发现状态信息“不够”。
- 记以(i,j)为右下角的矩形，其左上角为(x,y)，其最大面积记做dp(i,j,x,y)
  - 在向右、向下扩展时，发现状态信息仍然“不够”。

问题求解思路分析



- 分析以(i,j)为右下角的子矩阵：
- k从j-1到0遍历，直到chess[i][j]为0；
  - 在第i行，得到全一子数组chess[i][k+1...j]；
- r从j到0，计算chess[i][r]为底的最高的全1子数组。
- 考察chess[i][r]：
  - k从i-1到0遍历，直到chess[k][r]为0；
  - 在第r列，得到全一子数组chess[k+1...i][r]；
- 形成如下锯齿型结构，下面需要计算该结构的最大矩形面积。
- 结论：直方图最大矩形面积问题！

Code

```
int LargestRectangleArea(int* height, int N)    //height[N]==0
{
    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i <= N; )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}

const int N = 9;
const int M = 7;

int LargestRectangleArea2(int chess[M][N], int M, int N)
{
    int* height = new int[N+1];
    memset(height, 0, sizeof(int)*(N+1));
    //height[N] = 0：确保原数组height的最后一位能够得到计算
    //height[0...N-1] = 0：表示当前行没有1

    int i,j;
    int area = 0;
    int cur;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++) //每一行分别处理
        {
            if(chess[i][j] == 0)
            {
                height[j] = 0;
            }
            else
            {
                height[j] += chess[i][j];
            }
        }
        cur = LargestRectangleArea(height, N);
        area = max(cur, area);
    }
    delete[] height;
    return area;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int chess[M][N]
    ={
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,0,0,0,0},
        {0,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,1,0,0},
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,1,1,0,0},
        {0,1,1,0,1,0,0,0,0},
    };
    LargestRectangleArea2(chess, M, N);
    return 0;
}
```

Code Split

```
int LargestRectangleArea(int* height, int N)    //height[N]==0
{
    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i <= N; )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}
```

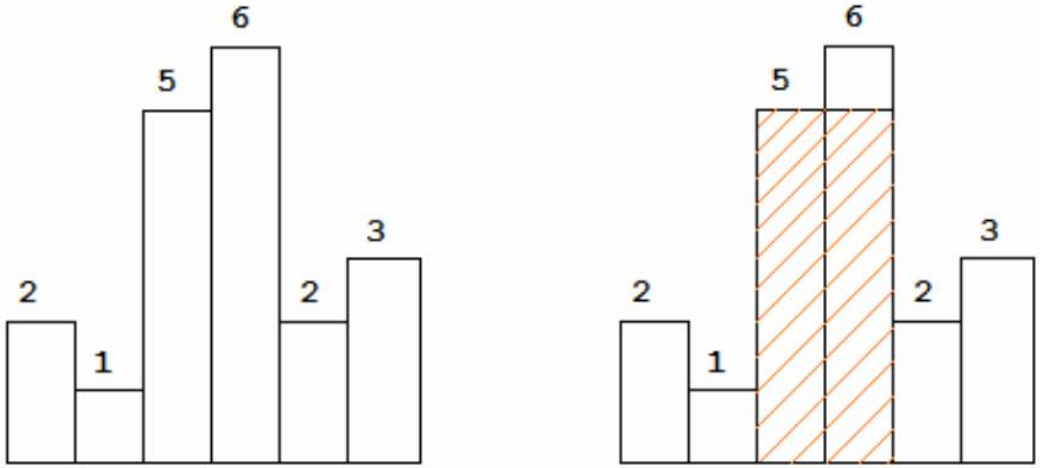
```
int _tmain(int argc, _TCHAR* argv[])
{
    int chess[M][N]
    ={
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,0,0,0,0},
        {0,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,0,0,0},
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,1,1,0,0},
        {0,1,1,0,1,0,0,0,0},
    };
    LargestRectangleArea2(chess, M, N);
    return 0;
}
```

```
int LargestRectangleArea2(int chess[M][N], int M, int N)
{
    int* height = new int[N+1];
    memset(height, 0, sizeof(int)*(N+1));
    //height[N] = 0：确保原数组height的最后一位能够得到计算
    //height[0...N-1] = 0：表示当前行没有1

    int i,j;
    int area = 0;
    int cur;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++) //每一行分别处理
        {
            if(chess[i][j] == 0)
            {
                height[j] = 0;
            }
            else
            {
                height[j] += chess[i][j];
            }
        }
        cur = LargestRectangleArea(height, N);
        area = max(cur, area);
    }
    delete[] height;
    return area;
}
```

PS. Largest Rectangle in Histogram

- 给定n个非负整数，表示直方图的方柱的高度，同时，每个方柱的宽度假定都为1；试找出直方图中最大的矩形面积。如：给定高度为：2,1,5,6,2,3，最大面积为10。

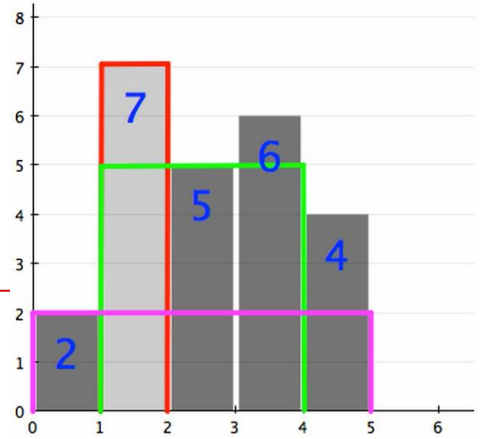


算法思想

- 从前向后遍历a[0...size](末尾添加了0)，若a[i]>a[i-1]，则将a[i]放入缓冲区；
- 若a[i]≤a[i-1]，则计算缓冲区中能够得到的最大矩形面积。

- 从a[i]>a[i-1]可以得出：
  - 缓冲区中放入的值是递增的
  - 每次只从缓冲区取出最后元素和a[i]比较——栈。

直方图最大矩形面积法分析



- 以2、7、5、6、4为例：
- 假设当前待分析的元素是4，由刚才的分析得知，栈内元素是2,5,6，其中，6是栈顶。
  - 此时，栈顶元素6＞4，则6出栈
    - 出栈后，新的栈顶元素为5，5和4的横向距离差为1：以6为高度，1为宽度的矩形面积是6*1=6
  - 此时，栈顶元素5＞4，则5出栈
    - 出栈后，新的栈顶元素为2，2和4的横向距离差为3：以5为高度，3为宽度的矩形面积是5*3=15
  - 此时，栈顶元素2≤4，则将4压栈，i++，同样的方法继续考察直方图后面的值。

Code

```
int LargestRectangleArea(vector<int>& height)
{
    height.push_back(0);     //确保原数组height的最后一位能够得到计算

    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i < (int)height.size(); )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}
```

结束语

- 算法的掌握速度是1,2,4,8,16……的过程；
  - 每天递增0.01：$1.01^{365}=?$
  - 设置适合自己的“学习率”。
- 掌握算法的根本途径是多练习代码。
  - 书读百遍，其义自见。
- 算法远远没有到此为止……
  - 下列属于算法范畴吗？
  - 网站开发/OA工作流
  - 操作系统资源调度/编译原理词法、语法、语义分析/数据库设计/计算机网络协议包解析
  - 机器学习/数据挖掘/计算机视觉
  - ……

我们在这里

- 七月算法 http://www.julyedu.com/
  - 精品视频
  - 直播课程
  - 问答社区
- 微博
  - @研究者July
  - @七月算法问答
  - @邹博_机器学习
- 微信公众号
  - julyedu

感谢大家！

恳请大家批评指正！