1 Different-sized sub-problems

$Akra-Bazzi$**公式**

对于形如

$$T(n)=\sum_{i=1}^{k}a_iT(n/b_i)+f(n)$$

这样递归表达式，$T(n)$的时间复杂度为

$$T(n)=\Theta(n^p(1+\int_1^n\frac{f(u)}{u^{p+1}}du))$$

p需满足

$$\sum_{i=1}^{k}a_i/b_i^p=1$$

所以对于$T(n)=T(n/2)+T(n/4)+T(n/8)+n$

首先由于$(1/2)^p+(1/4)^p+(1/8)^p=1$没有分析解，但是由于$(1/2)^x+(1/4)^x+(1/8)^x$为减函数，所以$0<p<1$

因此我们有

$$\int_1^n\frac{f(u)}{u^{p+1}}du=\int_1^n u^{-p}du=\frac{u^{1-p}}{1-p}|_{u=1}^{n}=\frac{n^{1-p}-1}{1-p}=\Theta(n^{1-p})$$

因此

$$T(n)=\Theta(n^p\cdot(1+\Theta(n^{1-p})))=\Theta(n)$$

2 What's wrong with this proof ?

(a)

(1),(2)是错的，(3)是对的

(b)

(1)是错的

$T(n)=O(n-5)+10n=O(n)$这一步是有错的

我们可以证明

$$\begin{gathered}\because T(n-5)=O(n-5)\\ \therefore T(n-5)\le c(n-5)\\ \because T(n)=T(n-5)+10n\\ \therefore T(n)\le c(n-5)+10n=(c+10)n-5c \qquad (2.1)\end{gathered}$$

根据式(2.1)我们并不能推出$T(n) = O(n)$，因为c是不确定的由$T(n-5)$中的n确定

(2)是错的

主定理中的b一定得是常数，而不能依赖于n

3

(a)

伪代码

```
Get_Random_euqal_probability()
  while(1)
    a = randP() ? 1 : 0
    b = randP() ? 1 : 0
    if(a + b != 1)
      continue
    if(a == 1)
      return true
    else
      return false
```

> a,b分别调用randP()函数，如果为true就等于1，如果为false就等于0
>
> 接着判断a + b
>
> 如果a + b不等于一就continue
>
> 如果a + b等于1且a == 1时返回true
>
> 如果a + b等于1且b == 1时返回false

(b)

a等于1的概率为p,等于0的概率为1 - p，同样

b等于1的概率为p,等于0的概率为1 - p

那么(a,b)等于(1,0)和(0,1)的概率分别是p(1-p)和(1-p)p

所以每次迭代能返回的概率为$2p(1-p)$，所以迭代的期望次数为$\frac{1}{2p(1-p)}$

所以调用randP的期望为$2 * \frac{1}{2p(1-p)} = \frac{1}{p(1-p)}$

(c)

a等于1的概率为p,等于0的概率为1 - p，同样，b等于1的概率为p,等于0的概率为1 - p

所以(a,b)组合的概率分布为

$(0,0) = (1-p)^2$

$(0,1) = (1-p)p$

$(1,0) = p(1-p)$

$(1,1) = p^2$

当遇到(0,0)或者(1,1)的时候我们就过滤掉，所以剩下的$(0,1)$和$(1,0)$的概率都是0.5,0.5

4

(a)

伪代码

```
输入：
A数组 表示排序好了的数组
B数组 表示排序好了的数组

Find_Median(A,B,length)
  if length==1
    return a[0]>b[0]?b[0]:a[0]
  mid = (length-1)/2
  if a[mid]==b[mid]
    return a[mid]
  else if a[mid] < b[mid]
    return Find_Median(A[length-mid-1:length-1],B[0:mid],mid+1)
  else
    return Find_Median(A[0:mid],B[length-mid-1:length-1],mid+1)
```

> 对于输入的长度相同的A,B两数组，如果长度为1，则返回较小的元素。如果两个数组的中位数相等就返回中位数，如果A的中位数小于B的中位数，那就返回A的中位数右边的数组和B的中位数左边的数组的递归结果，否则就返回A中位数左边的数组和B的中位数右边的数组的递归结果。

(b)

- Recurson invariant

  $Find_Median(A, B, length)$函数每次都会返回A,B两数组中排序好的中位数

- Base case("Initialization")

  初始化时A,B都是排序好了的长度一样的数组，length为A的长度

- Inductive step: (“Maintenance”)

  首先我们找到了A[n/2]和B[n/2]来比较

  1.如果他们相等，那么A[n/2]肯定就是两个数组的中位数

  2.如果B[n/2]>A[n/2]，那么这个中位数肯定在A[n/2:n]或B[0:n/2]的序列里。那么我们只要获取这两个数组中的中位数，这样肯定得到的中位数两边的数的个数肯定也是相等的。

  3.同理如果B[n/2]<A[n/2]，那么这个中位数肯定在A[0:n/2]或B[n/2:n]的序列里。那么我们只要获取这两个数组中的中位数，这样肯定得到的中位数两边的数的个数肯定也是相等的。

4.所以每次调用该函数都会返回该次迭代的A,B两数组的排序好的中位数

- Conclusion (“Termination”)

当递归到最后一次返回结果，并将结果返回到第一次调用时算法结束得到的就是A,B两个排序好的数组的中位数。

(c)

设$length = n$的时间复杂度表达式为$T(n)$

每次调用函数都会进行常数次比较和赋值

则表达式为$T(n) = T(n/2) + O(1)$，那么$T(n) = O(logn)$

5

(a)

伪代码

```
def binarysearch(value, low, high)
  if low >= high
    return low
  mid = (low + high) / 2
  ans = ask("数组中超过value的是否有mid个?")
  if ans is "yes":
    return binarysearch(value,mid+1,high)
  else:
    return binarysearch(value, low, mid)

def main():
  counts = []
  for i = 1 to k:
    counts.append(binarysearch(i,0,n))
  for i, count in enumerate(counts):
    for j = 1 to count:
      print i
```

> 首先我们遍历1. . . k,对于每个i值我们的目的是找出它在数组中的个数，我们可以通过二分查找找出每个i值在数组中出现的个数，比如我们可以问是否有超过n/2个元素都比i大，如果是我们就在(n/2,n)之间问，这样每次花的时间都是O(logn),所以总的时间复杂度就是O(klogn)

(b)

一个最多能问c个问题的算法最多能有2^c个输出因为每个答案的序列都只对应一个输出。对任意连续c个yes/no问题，都有2^c个可能的连续答案，所以如果有一个长度为N的元素互不相同，且排序好了的数组，在最坏的情况下任何算法都必须得问$logN$个问题

下界为:对每个值1,...,k-1,让它在数组中任何位置出现1到$\frac{k}{n}$之间次,然后让k出现直到填满数组为止。这样就会产生$(\frac{n}{k})^{k-1}$个不同的排序好的数组，因此便可以得到问题数量的最坏情况的算法复杂度的下界为:
$log((\frac{n}{c})^{k-1}) = (k-1)log\frac{n}{k} = \Omega(klog\frac{n}{k})$

6

(a)

伪代码

```
tree{
  int i //文件编号
  int j //第i个文件对应的密码编号
  set smaller_files //文件对应密码长度小于第i个文件的密码长度的文件编号集合
  set bigger_files  //文件对应密码长度大于第i个文件的密码长度的文件编号集合
  tree left_node  //左子树
  tree right_node  //右子树
}Tree

check(file,password) 检验函数，如果file的密码长度大于password返回1，相等返回0，小于返
回-1

输入
  tree:  Tree类型的node节点
   set:  当前搜索的文件集
     j:  当前匹配的密码编号
输出
  node:  Tree节点(node.i,node.j)表示第node.i个文件和第node.j个密码匹配
find_file_match_password(tree,set,j)
  Tree node = new Tree
  if tree == null
    for k = 0 to set.length
      if check(F[set[k]],P[j]) == 0
        node.i = set[k]
        node.j = j
      else if(check(F[set[k]],P[j]) < 0)
        node.smaller_files.add(set[k])
      else if(check(F[set[k]],P[j]) > 0)
        node.bigger_files.add(set[k])
     return node
  else
    if check(F[node.i],P[j] < 0)
      if tree.right_node == null
        tree.right_node = 
find_file_match_password(tree.right_node,tree.bigger_files,j)
        return tree.right_node
      else
```

```
        return find_file_match_password(tree.right_node,tree.bigger_files,j)
    else if check(F[node.i],P[j] > 0)
      if tree.left_node == null
        tree.left_node = 
find_file_match_password(tree.left_node,tree.smaller_files,j)
        return tree.left_node
      else
        return find_file_match_password(tree.left_node,tree.smaller_files,j)

输入
F:文件数组
P:密码数组
match(F,P)
  set all_files
  Tree node = null
  for i = 0 to F.length-1
    all_files.add(i)
  for j = 0 to P.length-1
    find_file_match_password(node,all_files,j)
```

算法描述

> 首先维护一棵二叉树
>
> - int i //文件编号
>   int j //第i个文件对应的密码编号
>   set smaller_files //文件对应密码长度小于第i个文件的密码长度的文件编号集合
>   set bigger_files //文件对应密码长度大于第i个文件的密码长度的文件编号集合
>   tree left_node //左子树
>   tree right_node //右子树
>
> 在寻找第j个密码对应的文件的时候，从根节点开始，如果当前密码大于根节点对应的文件密码长度，那就进入根节点的右子树搜索，如果当前密码小于根节点对应的文件密码长度，那就进入根节点的左子树搜索。如果递归到下一个节点为空时，就从可选集中遍历寻找与当前密码匹配的文件，赋给i,j。然后设置其smaller_files为小于当前匹配文件的那些文件编号，bigger_files为小于当前匹配文件的那些文件编号。最后返回该节点。

(b)

正确性证明

- Recurson invariant

$find\_file\_match\_password(tree, set, j)$ 函数肯定会返回当前tree节点及其子树中的与j匹配的文件节点

- Base case("Initialization")

第一次调用$find_file_match_password(tree, set, j)$时,tree为空节点null，且set的搜索集为所有F中的文件

- Inductive step: (“Maintenance”)

每次查询第j个密码对应的文件时，都会从我们维护的二叉树中去寻找包含它对应文件的那个字节点，直到找到一个叶子节点，然后在其搜索集中搜索到匹配的文件编号，同时记录它的smaller_files和bigger_files，然后返回结果节点。

- Conclusion (“Termination”)

当$j == n-1$调用$find_file_match_password(tree, set, j)$返回时，最后一个密码对应的文件也找到。算法结束

(c)

时间复杂度分析

由于我们在对n个密码进行匹配的过程中都是进行二叉树的搜索和生成，平均情况下，第一个密码进行匹配的时候会进行n次比较，第二个密码会进行n/2次比较,第三个密码会进行n/2(匹配的文件在第二层)次或n/4次比较(匹配的文件在第三层),接着第四个..第n个。最后会生成一颗二叉树，在随机算法中结果会接近满二叉树。

所以第i层进行的比较和搜索是$c*i*\frac{n}{i}=O(n)$，因此总的时间复杂度约为
$\sum_{i=1}^{k} c*i*\frac{n}{i}=O(nlogn), k=logn$

7

(a)

将CHOOSEPIVOT(A):中的

$Split\ A\ into m=\lceil\frac{n}{5}\rceil\ groups, of size\leq 5\ each$ 改为
$Split\ A\ into m=\lceil\frac{n}{3}\rceil\ groups, of size\leq 3\ each$

(b)

因为$n=3^k$,且整个数组被划分为大小为3的块，那么一共就有$n/3$个块，每个块中取中间元素，于是就一共选取了$n/3$个元素，因此在choosepivot中select-3的数组最多只有n/3个元素

(c)

$$
\begin{aligned}
&(n-1)-(2(\lceil\frac{m}{2}\rceil-1)+1)\\
&=n-2\lceil\frac{m}{2}\rceil\\
&\leq n-2(\frac{m}{2})\\
&=n-m\\
&\leq n-\frac{n}{3}\\
&=\frac{2n}{3}<\frac{2n}{3}+2
\end{aligned}
$$

(d)

1. 将我们的数组划分为块大小为3，并且找到每个块的中间元素(时间复杂度为常数),所以总的时间复

杂度为$\Theta(n)$,

2. 因为在选取pivot之后需要进行PARTITION(A,p)这个函数需要遍历将A数组分为比第p个元素小的L,A[p],和比第p个元素大的R，时间复杂度为$\Theta(n)$
3. 一旦我们PARTITION之后，我们就再选择哪一边进行递归，只需要1次判断，因此时间复杂度为$\Theta(1)$

所以综上一次单独的调用时间复杂度为O(n)

(e)

$$T(n) = T(n/3) + T(2n/3) + \Theta(n)$$

(f)

Select-3的时间复杂度上界不是$O(n)$

我们先尝试用代入法证明

先假设T(n)=O(n),所以当$n > n_0$时，有T(n) = kn

于是有

$$T(n) = kn/3 + 2kn/3 + cn = kn + cn = (k+c)n > kn$$

代入法证明失败，但是这并不能证明该时间复杂度就不是$O(n)$

我们采用$Akra-Bazzi$求解该递归式的复杂度，公示同第一题一样

$$\because \ (\frac{1}{3})^p + (\frac{2}{3})^p = 1 \quad \therefore \ p = 1$$

由

$$T(n) = \Theta(n^p(1 + \int_1^n \frac{f(u)}{u^{p+1}}du))$$

可计算得

$$\begin{aligned} T(n) &= \Theta(n(1 + \int_1^n \frac{1}{u}du)) \\ &= \Theta(n(1 + logn)) \\ &= \Theta(nlogn) \end{aligned}$$

所以$T(n)$并不等于$O(n)$