ADS-Review

目录

1. Advanced Data Structure

1.1 Tree

1.1.1 AVL Tree

- 根本目标：用二叉搜索树来**提高搜索的速度**
- AVL 树的定义
  - 空树是 height balanced 的
  - 对于任何一个节点，左右子树的高度差的绝对值不超过 1
    - 高度的定义：**空节点的高是 1，没有儿子的节点 (学名是叶节点) 高为 0**，否则就是儿子中高的最大值 +1
- **树的旋转 rotation**
  - 当高度差的约束被打破的时候就需要进行旋转，有 LL,RR,LR,RL 四种旋转方式
  - LL 型旋转

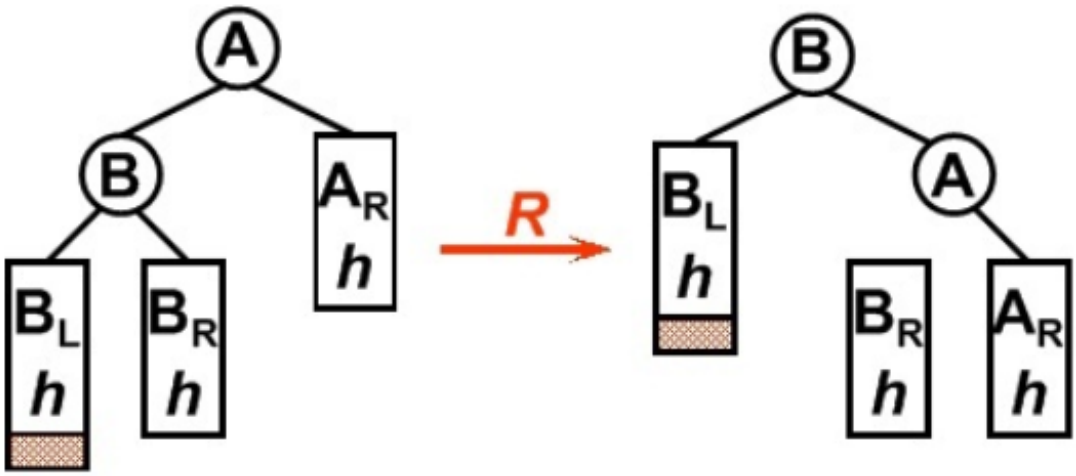


图 1: image-20200308230919340

  - RR 型旋转，基本就是 LL 型旋转的镜像对称

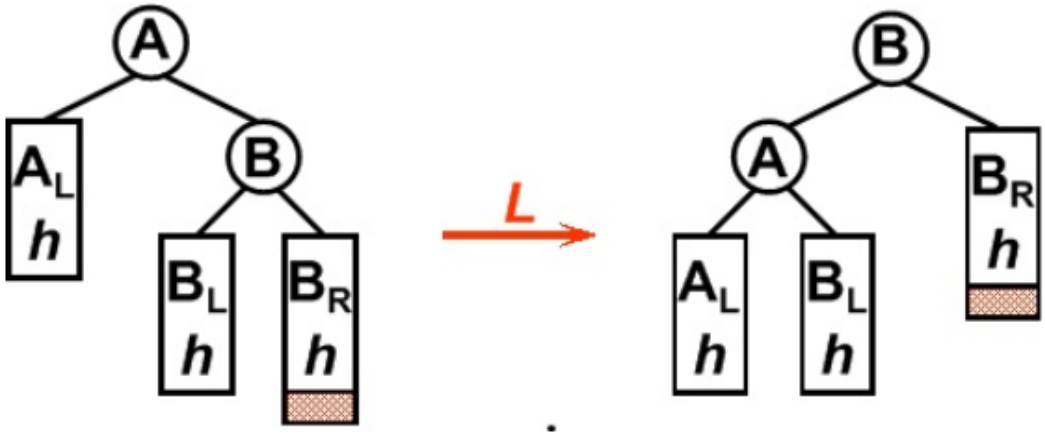


图 2: image-20200308230949370

– LR 型旋转

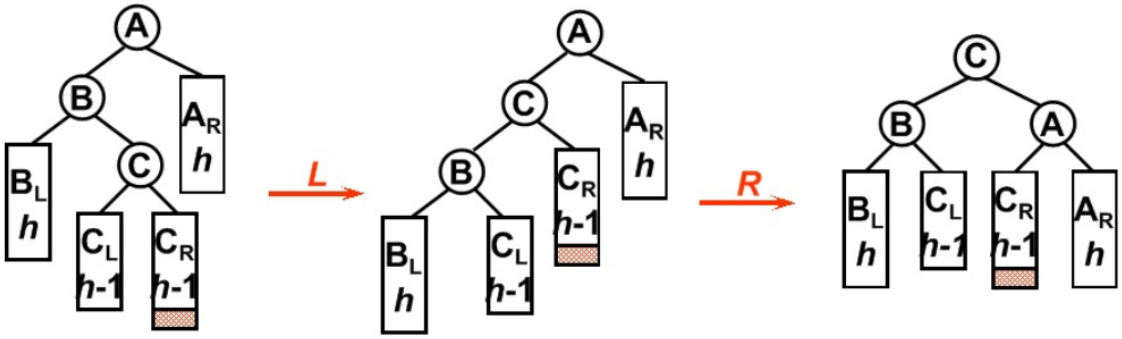


图 3: image-20200308231027057

– RL 型旋转

- 对于高度为 k(空树的高度为 0) 的 AVL 树，它的**最小节点个数**为 (最大的节点个数肯定是 K 层全满的时候)

$$n_k = n_{k-1} + n_{k-2} + 1 n_k = F_{k+2} - 1 so \quad k = \ln(n)$$

1.1.2 Splay Tree

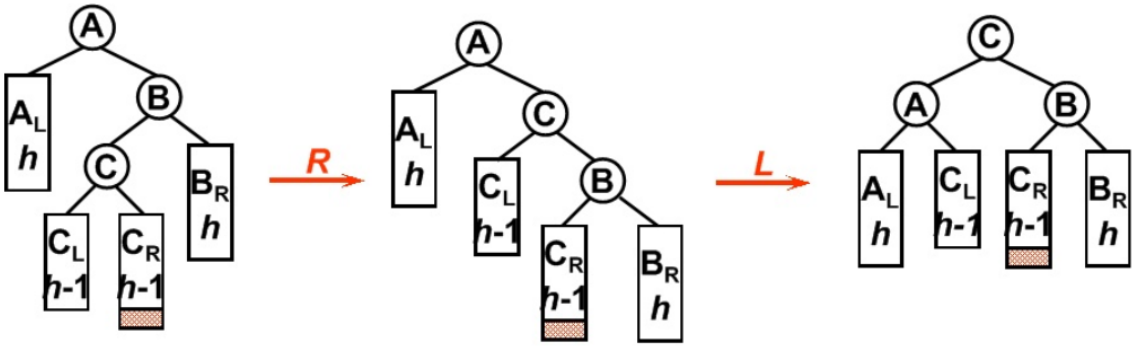


图 4: image-20200308231051693

- 目标：从**空树开始**，任何 M 次连续的操作一共最多消耗 $O(M \log N)$ 的时间
  - 必须要从一棵空树开始
  - 具体做法：在树中，每次有节点被访问到，就将其**旋转到根节点**
- 旋转的方法
  - 如果父节点是根节点，直接把父子旋转即可
  - 如果父节点不是根节点
    - zig-zag

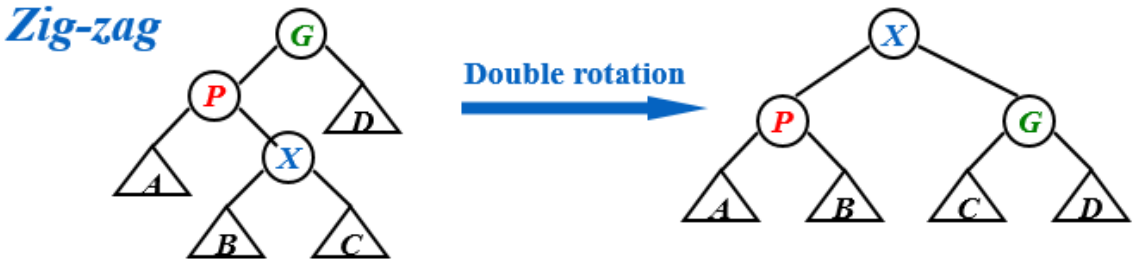


图 5:

    - zig-zig
  - 每次要将被访问的节点按照上述规则**旋转到根节点**上为止
- 删除的操作步骤

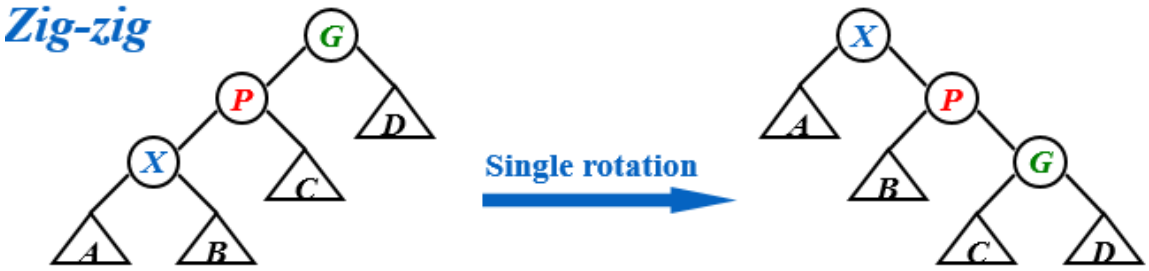


图 6:

- 找到要删除的节点 X，将其旋转到根节点 (这也是 access)
- 把根节点删除，产生左右两棵子树
- 在左子树上找到最大的节点旋转到根节点，并且把右子树接到左子树的右边

1.1.3 Red-Black Tree

- 红黑树的定义
  - 红黑树是一种二叉搜索树，并且节点的颜色为红色或者黑色
  - 红黑树的**根节点是黑色**
  - 叶节点都是黑色的 (叶节点是 NULL 的时候把 NULL 也视为黑色的节点)
  - 红色节点的两个儿子一定都是黑色节点
  - 对于每一个节点，所有的从该节点出发到达后代的叶节点的简单路径包含相同数量的黑色节点
- black height of node x: `bh(x)`
  - 节点的 black height 表示从 x **到叶节点的路径中黑色节点**的个数
  - 性质：
    - 有 N 个节点的红黑树的高度最多为 $2\ln(N+1)$
    - $bh(Tree) >= \frac{h(Tree)}{2}$

- 红黑树的操作
  - 插入 insert
    - **先当作普通的 BST 插入**并且将该**节点作为红色节点**，但是这么做会破坏红黑树的性质，需要进行旋转
    - 分情况判断怎么旋转
      - case1：插入的节点在父节点的右侧且叔叔节点是红色的：父节点和叔节点变成红色

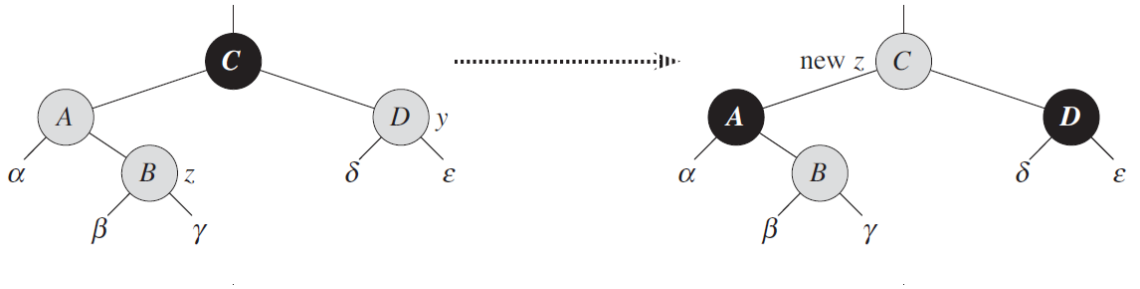


图 7:

      - case2：插入的节点在左侧而且叔叔是红色

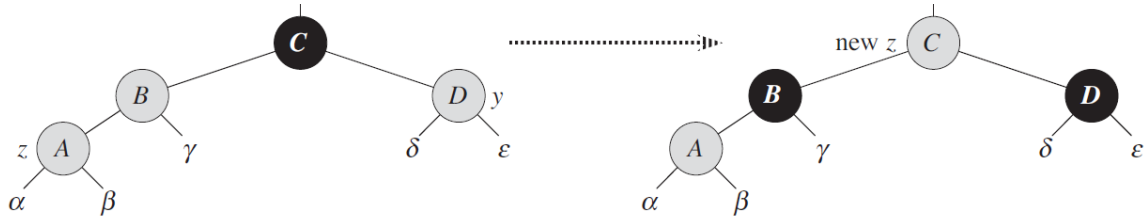


图 8:

      - case3：叔节点是黑色的，不论左右儿子都要进行**旋转成如下图所示之后换成一黑带两红**
    - 插入的时间复杂度是 $O(\log N)$ 插入之后需要从底下不断向上调整多次，直到满足红黑树的所有性质

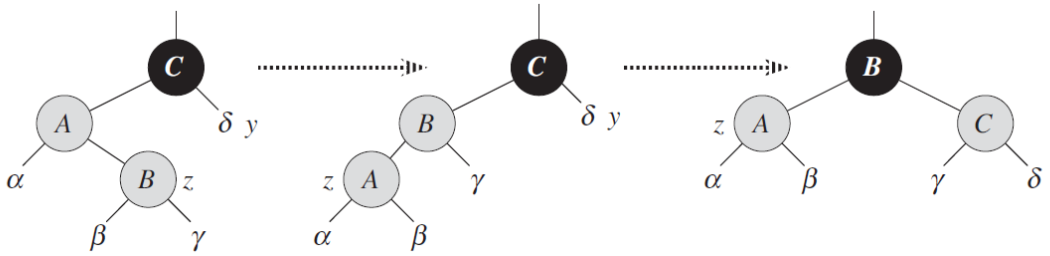


图 9:

- 删除 delete
  - 删除的节点是**叶节点**：让其父节点指向 NULL，保持不变
  - 删除的节点度数为 1：用孩子代替原本的节点
  - 删除的节点度数为 2：用最大的左儿子或者最小的右儿子代替节点
  - 当删除的节点是红色节点时性质不会被破坏，删除的节点是黑色的时候可能会破坏一些性质
- Number of rotations
  - 红黑树在 Insert 中旋转次数不超过 2，删除的过程中旋转次数不超过 3

1.1.4 B+ Tree

- M 阶的 B+ 树的定义
  - 根节点是叶节点或者根节点有 `2~M` 个儿子
  - 根以外的**非叶节点**有 `[M/2]~M` 个儿子，每个**叶节点**内含 `[M/2]~M` 个元素
    - B+ 树的儿子指的不是节点里的几个数据，而是节点向下指出来的新节点
    - 每个节点的 key 数是子节点数-1，画图的时候 key 分布在两个指向儿子的箭头之间
    - key 值的确定方法：等于 key 右边第一个指针对应的子节点的最左边的值
  - 所有的叶节点的深度相同

  - B+ tree of order 4 也被称为 2-3-4 树，order3 的被称为 2-3 tree

- B+ 树的插入算法
  - 对于 order M，有 N 个元素的 B+ 树而言 $T = O(\frac{M}{\log M} \log N)$
  - B+ 树的深度 $Depth(M, N) = O([\log_{[M/2]} N])$, 找到插入位置的事件复杂度是 log N
  - 对于 order3 的 B+ 树而言，非叶节点的索引个数在有三个儿子时需要两个，否则只需要一个索引

```
Btree  Insert ( ElementType X,  Btree T )
{
        Search from root to leaf for X and find the proper leaf node;
        Insert X;
        while ( this node has M+1 keys ) {
                    split it into 2 nodes with [(M+1)/2] and [(M+1)/2] keys, respectively;
                    if (this node is the root)
                        create a new root with two children;
                    check its parent;
        }
}
//算法的描述：找到合适的位置先插入，如果叶节点的 keys 数量超过了 M，则分裂成两个，然后向上继续合并和拆
```

- **Deletion is similar to insertion except that the root is removed when it loses two children** 删除和插入的做法相似，不过当一个根节点失去两个儿子时就要删除

1.2 Invert File Index

- Term-Document Incidence Matrix 文档关联矩阵
  - a matrix of the appearance of each word in each doc
  - 如果单词在某篇文章中出现，则对应的矩阵上位置的值为 1，否则为 0，若干篇文章的出现情况可以得到若干个二进制字符串
    - 只关注一个单词**出现与否**，并不关注出现的**频率**
  - 用**逻辑运算**可以考察单词在 doc 中的出现情况

- Inverted File Index 倒排文件索引
  - Index is a mechanism for locating a given term in a text 一种在文章中定位给定单词的方法
  - Inverted File contains a list of pointers to all occurrences of that term in the text
    - 索引的方式单词---< 次数；依次列出每一篇出现的 doc 的编号 >
    - 更 nb 的索引方式单词----< 次数；(出现的 doc 的编号；该 doc 中每一个出现的位置)>

Doc	Text
1	Gold silver truck
2	Shipment of gold damaged in a fire
3	Delivery of silver arrived in a silver truck
4	Shipment of gold arrived in a truck

Inverted File Index

No.	Term	Times; Documents
1	a	<3; 2,3,4>
2	arrived	<2; 3,4>
3	damaged	<1; 2>
4	delivery	<1; 3>
5	fire	<1; 2>
6	gold	<3; 1,2,4>
7	of	<3; 2,3,4>
8	in	<3; 2,3,4>
9	shipment	<2; 2,4>
10	silver	<2; 1,3>
11	truck	<3; 1,3,4>

图 10:

- Index generator 索引生成器

```
while(read a document D){
        while(read a term T in D){
                if(Find(Dictionary,T)==false)
                        Insert(T);
                Get T's position list;
                Insert a node to T's posting list
        }
}
write the inverted index to disk
```

- While accessing a term by hashing in an inverted file index, range searches are expensive.
- 读取时的简单处理
  - word stemming 碰到一个单词的多种形式和时态的时候只保留一种 root 形式
  - stop words 面对一些出现频率高但是 useless 的单词时不统计，比如 a，the 等
- 当存储空间不足时
  - using the memory block and merge them in the end 采用多块内存存储
  - distributed indexing 分布式索引
    - term-partitioned index 按词语来划分
    - document-partitioned index 按照文档的编号来划分
  - dynamic index 动态索引
    - 不常索引的 doc 将会被删除
    - 索引由 main index+auxiliary index 构成
- Thresholding 阈值
  - **document：只检索前面 x 个按权重排序的文档**
    - 对于布尔查询无效
    - 会遗漏一些重要的文档，因为有截断
  - **query：把带查询的 terms 按照出现的频率升序排序**
  - 要区分两种不同的阈值各自的作用
- 搜索引擎的评价标准
  - how fast does it index 索引有多快
  - how fast does it search 搜索有多快
  - Expressiveness of query language 查询语言的表现
- Relevance measurement

- 准确率 precision $P_R = R_R/(R_R + I_R)$
- 召回率 recall $R_R = (R_R + R_N)$

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

图 11:

1.3 Heap

1.3.1 Leftist Heap 左倾堆

- NPL(x) -- Null path length
  - 对于任意一个节点 x，通往**一个没有两个子节点的节点的最短路径长**称为 NPL(x)，空节点的 NPL 值为-1
  - 计算方法：NPL(x) = min { NPL(c) c is a child of X} +1
- 左倾堆的性质
  - 对于每个节点，**左儿子的 NPL 值不小于右儿子的 NPL 值**，所以整体看起来向左倾斜
    - npl 值看的是从自己出发的，但是左倾堆的性质是儿子的 npl 值满足条件
  - 左倾堆实际上是一种不平衡的二叉树，它的数据结构定义如下

```
struct TreeNode{
    ElementType Element;
        PriorityQueue Left;
        PriorityQueue Right;
```

```
        int         Npl;
};
```

- 对于一个右路径有 r 个节点的左倾堆，至少一共有 $2^r - 1$ 个节点
- 左倾堆的合并
  - 递归方法：
    - 每次合并需要比较两个左倾堆的根节点的大小，将大的合并在小的上面

```
PriorityQueue Merge(PriorityQueue H1,PriorityQueue H2)
{
    if(H1==NULL) return H2;
    if(H2==NULL) return H1;
    if(H1->Element<H2->Element)
        return Merge1(H1,H2);
    else
        return Merge1(H2,H1);
}

PriorityQueue Merge1( PriorityQueue H1, PriorityQueue H2 )
{
        if ( H1->Left == NULL )          /* single node */
                H1->Left = H2;         /* H1->Right is already NULL
                                       and H1->Npl is already 0 */
        else {
                H1->Right = Merge( H1->Right, H2 );      /* Step 1 & 2 */
                if ( H1->Left->Npl < H1->Right->Npl )
                        SwapChildren( H1 );         /* Step 3 */
                H1->Npl = H1->Right->Npl + 1;
        } /* end else */
        return H1;
}

//The time complexity is O(log N)
```

- 迭代方法
  - 将两个堆拆分成若干条 right paths，将它们从小到大合并成一个
  - 自上而下互换左右子节点，当子树不满足左倾堆性质的时候
- **Insertion is merely a special case of merging.**
  - 插入当作特殊的 merge 进行处理即可

- 左倾堆的 Delete Min
  - 删除根节点
  - 将左右子树作为两个左倾堆进行合并

1.3.2 Skew Heap 斜堆

- 斜堆是左倾堆的一种简单形式，它使得对于斜堆的连续 M 次操作最多消耗 $O(M \log N)$ 的时间
- 相比左倾堆的特点
  - **Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.** 不需要额外存储 NPL 值，交换左右子节点的时候不需要 test
  - It is an open problem to determine precisely the expected right path length of both leftist and skew heaps.
- 相比于普通的二项堆，Skew heaps are advantageous because of their ability to merge more quickly than balanced binary heaps. The worst case time complexities for Merge, Insert, and DeleteMin are all $O(N)$, while the amorited complexities for Merge, Insert, and DeleteMin are all $O(logN)$.
- 斜堆的 Merge
  - Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped.
  - 人类语言描述 merge 的过程

    - NULL 和 NULL 合并无事发生
    - NULL 和非空斜堆合并的结果是非空斜堆
    - 如果两个非空的斜堆合并，取根节点值比较小的作为新的根节点，让另一个堆和其右儿子进行合并后作为新的右儿子，合并完成之后交换左右子节点 (**不是镜像对称**，只要交换左右儿子)
  - **把 insert 当成是一个点和一个斜堆的 merge**
- 斜堆的摊还分析
  - 将操作后的根节点作为 D_i
  - 势能函数的选取：number of heavy nodes
    - heavy nodes：if the number of descendants of p's right subtree is at least half of the number of descendants of p, and light otherwise.
    - The only nodes whose heavy/light status can change are nodes that are initially on the right path
  - 对于斜堆而言，插入，合并，删除的**最坏情况的复杂度**都是 $O(N)$, 三种操作的**摊还代价**都是 $O(\log N)$

1.3.3 Binomial Queue 二项队列

- 定义：二项队列是**一系列** heap-order 的 tree，每一棵树是一个 binomial tree
- binomial tree 二项树
  - 高度为 0 的二项树是一个单节点的树
  - 高度为 k 的二项树是由两个高度为 k-1 的二项树拼接而成的，formed by attaching a binomial tree to the root of another binomial tree of height k-1;
  - B_k has k children and have 2^k nodes, the number of nodes in depth d is $C(k, d)$
- 二项队列
  - 不同高度的二项树只能存在一个，如果存在两个相同的就要进行合并, 数据结构的表示如下

```
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree;

struct BinNode
{
    int Element;
    Postion LeftChild;
    Postion NextSibling;
}

struct Collection
{
        int CurrentSize;  /* total number of nodes */
        BinTree        TheTrees[ MaxTrees ];
} ;
```

- 支持的 operation
  - FindMin
    - 找到每棵树的根节点中的最小值即可
    - 时间复杂度最多为 $O(\log N)$
    - 可以用维护一个变量来存储一个二项堆列中的最小值，每次更新的时候检查该值是否改变，这样一来 FindMin 消耗**常数时间**
  - Merge
    - 将两个高度相同的二项队列进行合并
    - 将两棵高度相同的二项树合并成新的二项树，取小的作为根节点，如果有三棵树 (进位产生了一棵) 则随机选两棵树合并

```
BinTree CombineTrees( BinTree T1, BinTree T2 )
{  /* merge equal-sized T1 and T2 */
        if ( T1->Element > T2->Element )
```

```
            /* attach the larger one to the smaller one */
            return CombineTrees( T2, T1 );
        /* insert T2 to the front of the children list of T1 */
        T2->NextSibling = T1->LeftChild;
        T1->LeftChild = T2;
        return T1;
}
```

· 时间复杂度为 $O(\log N)$

```
BinQueue  Merge( BinQueue H1, BinQueue H2 )
{        BinTree T1, T2, Carry = NULL;
        int i, j;
        if ( H1->CurrentSize + H2-> CurrentSize > Capacity )  ErrorMessage();
        H1->CurrentSize += H2-> CurrentSize;
        for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
            T1 = H1->TheTrees[i];
        T2 = H2->TheTrees[i]; /*current trees */
            switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
                case 0: /* 000 */
                 case 1: /* 001 */  break;
                case 2: /* 010 */  H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; br
                case 4: /* 100 */  H1->TheTrees[i] = Carry; Carry = NULL; break;
                case 3: /* 011 */  Carry = CombineTrees( T1, T2 );
                                    H1->TheTrees[i] = H2->TheTrees[i] = NULL; break
                case 5: /* 101 */  Carry = CombineTrees( T1, Carry );
                                    H1->TheTrees[i] = NULL; break;
                case 6: /* 110 */  Carry = CombineTrees( T2, Carry );
                                    H2->TheTrees[i] = NULL; break;
                case 7: /* 111 */  H1->TheTrees[i] = Carry;
                                    Carry = CombineTrees( T1, T2 );
                                    H2->TheTrees[i] = NULL; break;
            } /* end switch */
        } /* end for-loop */
        return H1;
```

```
}
```

* Insert
  - 是一种特殊的 Merge，时间复杂度为 $O(N)$
  - 从一个空的二项队列开始插入 N 个元素最多消耗 $O(N)$ 的时间，因此**均摊到每一个操作上**的时间复杂度是**常数时间**

* Delete Min 操作
  - 找到最小的根节点将其删除，将这颗二项树剩余的节点移除作为一个新的二项队列 (find min)，将剩下的若干二项树作为一个新的二项队列，一共得到两个二项队列
  - 将两个二项队列合并
  - 时间复杂度为 $O(\log N)$

```
ElementType  DeleteMin( BinQueue H )
{        BinQueue DeletedQueue;
        Position DeletedTree, OldRoot;
        ElementType MinItem = Infinity;  /* the minimum item to be returned */
        int i, j, MinTree; /* MinTree is the index of the tree with the minimum ite

        if ( IsEmpty( H ) )  {  PrintErrorMessage();  return -Infinity; }

        for ( i = 0; i < MaxTrees; i++) {  /* Step 1: find the minimum item */
            if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) {
                MinItem = H->TheTrees[i]->Element;  MinTree = i;    } /* end if */
        } /* end for-i-loop */
        DeletedTree = H->TheTrees[ MinTree ];
        H->TheTrees[ MinTree ] = NULL;   /* Step 2: remove the MinTree from H => H'
        OldRoot = DeletedTree;   /* Step 3.1: remove the root */
        DeletedTree = DeletedTree->LeftChild;   free(OldRoot);
        DeletedQueue = Initialize();   /* Step 3.2: create H" */
        DeletedQueue->CurrentSize = ( 1<<MinTree ) - 1;  /* 2MinTree - 1 */
        for ( j = MinTree - 1; j >= 0; j -- ) {
```

```
            DeletedQueue->TheTrees[j] = DeletedTree;
            DeletedTree = DeletedTree->NextSibling;
            DeletedQueue->TheTrees[j]->NextSibling = NULL;
        } /* end for-j-loop */
        H->CurrentSize  - = DeletedQueue->CurrentSize + 1;
        H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H" */
        return MinItem;
}
```

2. Algorithms

2.1 Backtracking

- 回溯法的基本思路：考虑所有可能的情况进行注意验证，在验证的过程中进行合理的剪枝 (pruning)

案例 1：八皇后问题

- 目标: 在棋盘中找到八个位置放置皇后，使得它们都不同行且不同列，也不能同时位于对角线上
- 使用 game tree 的方式来表示回溯的过程：对于 n 个皇后的问题有 n！种不同的情况需要验证

案例 2：加油站问题

- 目标：在一条直线上找到 n 个地方建立加油站，已知它们两两之间的距离，求出所有加油站的位置，假定第一个加油站的坐标是 0
- 解决方式
  - 有 $\frac{n(n-1)}{2}$ 个距离和 n 个加油站
  - 首先需要根据加油站的数量计算出 n 的大小
  - 先将第一个加油站和最后一个加油站的位置确定，并将已经可以计算出的距离从路径中删除

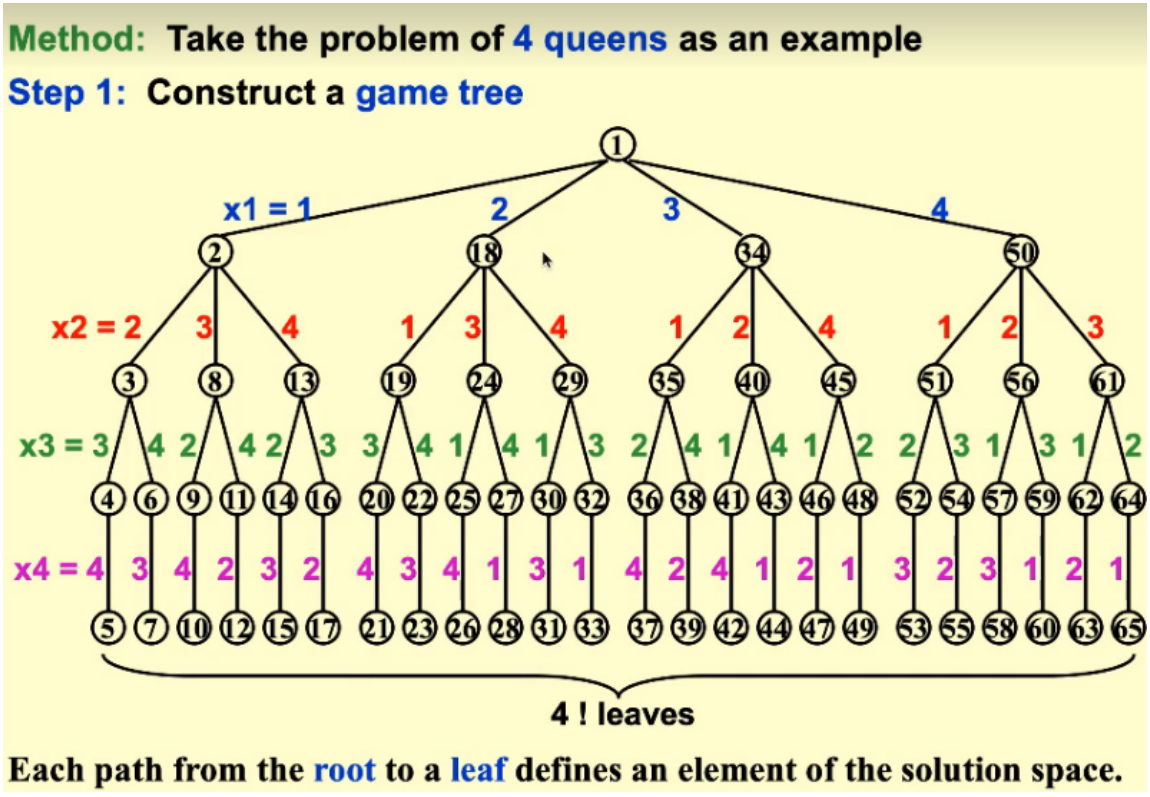

Method: Take the problem of 4 queens as an example
Step 1: Construct a game tree
x1 = 1
2
3
4
x2 = 2
x3 = 3
x4 = 4
4 ! leaves
Each path from the root to a leaf defines an element of the solution space.

图 12:

- 找到剩下的距离中最大的距离并检验，不断重复上述过程，如果检验失败则回到上一种情况，恢复原本被删除的距离再往下回溯
  - 每次检验分成靠近左边和靠近右边两种情况
- 代码实现

```
bool Reconstruct ( DistType X[ ], DistSet D, int N, int left, int right )
{ /* X[1]...X[left-1] and X[right+1]...X[N] are solved */
    bool Found = false;
    if ( Is_Empty( D ) )
        return true; /* solved */
    D_max = Find_Max( D );
    /* option 1: X[right] = D_max */
    /* check if |D_max-X[i]| in D is true for all X[i]' s that have been solved */
    OK = Check( D_max, N, left, right ); /* pruning */
    if ( OK ) { /* add X[right] and update D */
        X[right] = D_max;
        for ( i=1; i<left; i++ )  Delete( |X[right]-X[i]|, D);
        for ( i=right+1; i<=N; i++ )  Delete( |X[right]-X[i]|, D);
        Found = Reconstruct ( X, D, N, left, right-1 );
        if ( !Found ) { /* if does not work, undo */
            for ( i=1; i<left; i++ )  Insert( |X[right]-X[i]|, D);
            for ( i=right+1; i<=N; i++ )  Insert( |X[right]-X[i]|, D);
        }
    }
    /* finish checking option 1 */
        if ( !Found ) { /* if option 1 does not work */
        /* option 2: X[left] = X[N]-D_max */
        OK = Check( X[N]-D_max, N, left, right );
    if ( OK ) {
            X[left] = X[N] - D_max;
            for ( i=1; i<left; i++ )  Delete( |X[left]-X[i]|, D);
            for ( i=right+1; i<=N; i++ )  Delete( |X[left]-X[i]|, D);
            Found = Reconstruct (X, D, N, left+1, right );
```

```
                if ( !Found ) {
                    for ( i=1; i<left; i++ ) Insert( |X[left]-X[i]|, D);
                    for ( i=right+1; i<=N; i++ ) Insert( |X[left]-X[i]|, D);
                }
            }
            /* finish checking option 2 */
        } /* finish checking all the options */

        return Found;
    }
```

- 回溯算法的一种模板

```
bool Backtracking ( int i )
{
    bool Found = false;
    if ( i > N )
        return true; /* solved with (x1, ···, xN) */
    for ( each xi in Si ) {
        /* check if satisfies the restriction R */
        OK = Check((x1, ···, xi) , R ); /* pruning */
        if ( OK ) {
            Count xi in;
            Found = Backtracking( i+1 );
            if ( !Found )
                Undo( i ); /* recover to (x1, ···, xi-1) */
        }
        if ( Found ) break;
    }
    return Found;
}
```

- 回溯方式的选择：应该选择**从少到多**的回溯方式，这样**在剪枝的情况下可以排除更多的情况**

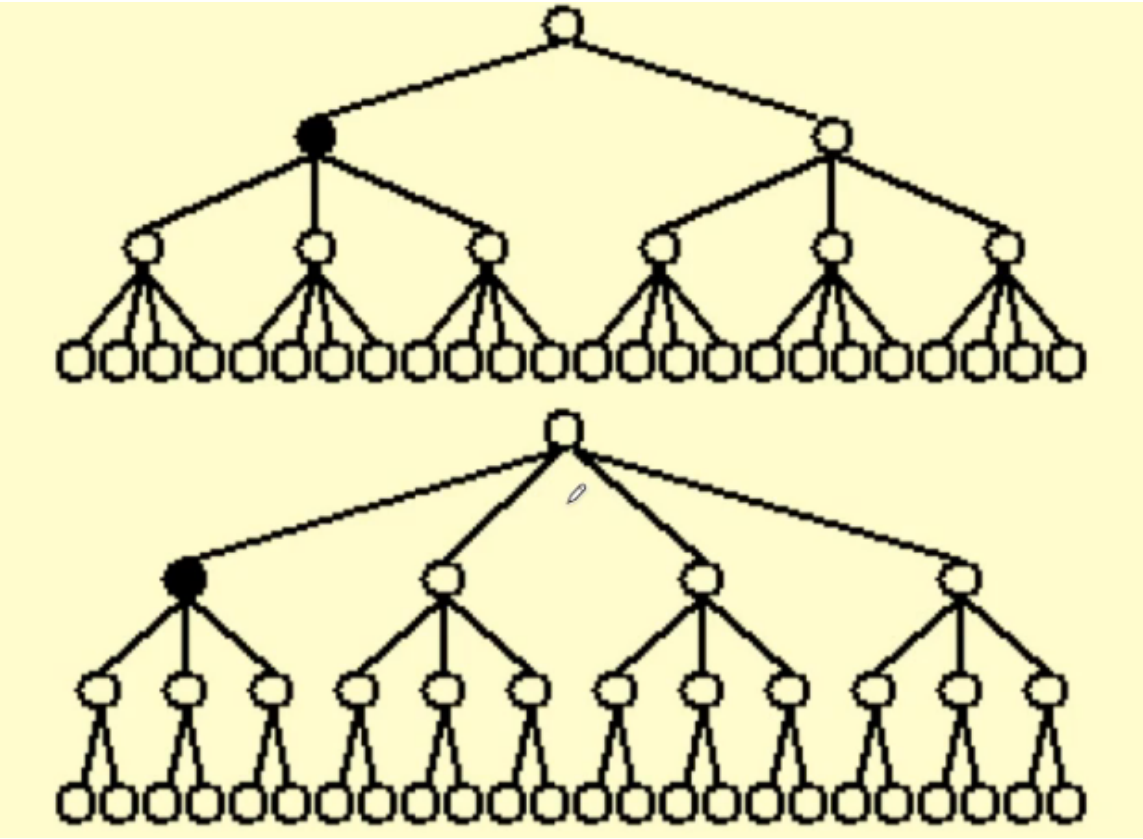


图 13: image-20200407214843754

案例 3：AI 下棋 Tic-tac-toe

- 需要推算出所有可能的情况并选择当前胜率最高的情况往下走
- Minimax Strategy 最大最小策略
  - 人需要最小化当前情况 P 的可能赢的情况，而 AI 要将它最大化
  - goodness 函数 $f(P) = W_{AI} - W_{Human}$，**W 是当前情况下某一方可能赢的所有结果**，不需要考虑另一方后面会怎么下，只要计算自己在当前局势下的任何可以赢的方

剪枝 Pruning

- 规则：每次在 max 里**向下取最大**的，在 min 里**向下取最小**的，并且上面的值是由下一层的取出来得到的，对于不会影响上一层取值的点就可以进行剪枝
  - α pruning-- max-min
  - β pruning-- min-max

2.2 Divide & Conquer

- A method to solve problem recursively 递归地解决问题
  - $T(N) = aT(N/b) + f(N)$
  - 一些基本的例子
    - 最大子列和问题 --- $O(N \log N)$
    - 树的遍历---$O(N)$
    - 归并排序和快速排序 $O(N \log N)$

案例 1：Closest Point Problem

- 对于 N 个点，最垃圾的方法就是搜索 $\frac{N(N-1)}{2}$ 次求出最短距离
- 分治法的解决思路
  - 将问题分成三个子问题，对于每个点，分别从同侧的左右两边和异侧来找到距离他最近的点

    - 需要按照 x 或者 y 坐标进行排序之后在进行分治，否则达不到 $O(N \log N)$ 的复杂度
  - $T(N) = 3T(N/3) + f(N)$，因此总的时间复杂度是 $O(N \log N)$

求解分治法时间复杂度的方法

- Substitution Method
  - 总结了一下就是猜出答案
- Recursion-Tree Method
  - 画一棵很奇怪的树，我也不知道这玩意和直接进行数学上的推导有啥区别
- Master Method
  - 好！只要记住公式就行
  - 一些细节
    - N/b 是不是证书
    - T(N)=O(1) 当 N 是比较小的数字的时候
  - 公式

1. If $f(N) = O(N^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(N) = \Theta(N^{\log_b a})$
2. If $f(N) = \Theta(N^{\log_b a})$, then $T(N) = \Theta(N^{\log_b a} \log N)$
3. If $f(N) = \Omega(N^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(N/b) < cf(N)$ for some constant $c < 1$ and all sufficiently large N, then $T(N) = \Theta(f(N))$

图 14:

  - 总结起来就是三种情况: 是 $N^{\log_b a}$ 的渐进上界，渐进下界和等价，证明可以通过取 $N = b^k$ 来完成
- 总结：分治法的时间复杂度可以通过下面的公式来总结

$$T(N) = a\,T(N/b) + \Theta(N^k \log^p N),$$
where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

图 15:

2.3 Dynamic Programing

- solve sub-problems **just once** and save the answer in a **table** 用存储空间记录子问题的结果，避免重复的运算
- Ordering Matrix Multiplications
  - Let bn = the number of different ways to compute M1-Mn, Then $M_{1n} = M_{1i}M_{i+1,n}$ so $b_n = \sum_{i=1}^{n-1} b_i b_{n-i}$ // Catalan number
  - Let M_{ij} be the cost of optional way to compute Mi-Mj, Then we have

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

图 16:

  - 这种算法的时间复杂度 $T(N) = O(N^3)$, 而传统算法 $b_n = O(\frac{4^n}{n^{1.5}})$
- Optinmal BST

Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{N} p_i \cdot (1 + d_i)$

图 17:

  - using **greedy method** 每次选取可选范围内概率最高的作为根节点，递归建树，但不一定是最优
  - 动态规划的解决办法 $T(N) = O(N^3)$，该算法可以继续优化成 $O(N^2)$

- All-Pairs Shortest Path
  - find the shortest path between all pairs of vertices in the graph

```
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )  /* Initialize D */
         for( j = 0; j < N; j++ )
         D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )  /* add one vertex k into the path */
         for( i = 0; i < N; i++ )
         for( j = 0; j < N; j++ )
            if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                /* Update shortest path */
                 D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

  - $T(N) = O(N^3)$ but faster in a dense graph

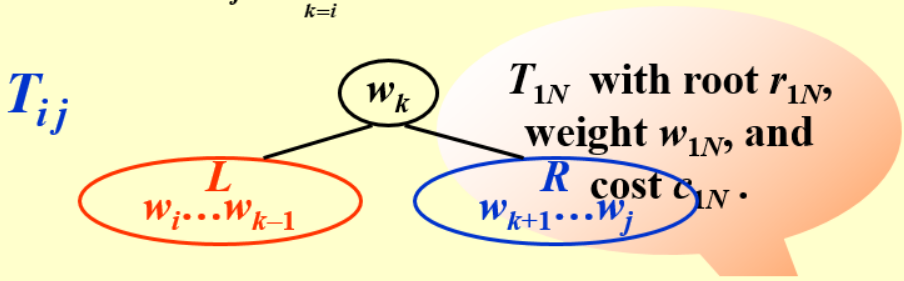
- 生产线问题
  - 两条生产线，每一步同时受到两条生产线前面一个步骤的影响，要求计算最短的加工时间

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0)$

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of T_{ij} $= \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

T_{ij} is optimal $\Rightarrow$ $r_{ij} = k$ is such that $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

图 18: image-20200415110219642

Method 2 Define

$$D^k[\,i\,][\,j\,] = \min\{\text{length of path } i \to \{\, l \le k \,\} \to j\}$$

and $D^{-1}[\,i\,][\,j\,] = \text{Cost}[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm Start from D^{-1} and successively generate $D^0, D^1, \ldots, D^{N-1}$. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \to \{\, l \le k \,\} \to j \;\Rightarrow\; D^k = D^{k-1}$; or

② $k \in$ the shortest path $i \to \{\, l \le k \,\} \to j$

$= \{$ the S.P. from i to $k \,\} \cup \{$the S.P. from k to $j \,\}$

$\Rightarrow\; D^k[\,i\,][\,j\,] = D^{k-1}[\,i\,][\,k\,] + D^{k-1}[\,k\,][\,j\,]$

$$\therefore\quad D^k[i][j] = \min\{D^{k-1}[i][j],\, D^{k-1}[i][k] + D^{k-1}[k][j]\},\ k \ge 0$$

图 19:

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;
      L[line][stage] = 1-line;
    }
  }
}
```

- 背包问题

- 一个背包的最大存储空间为 M，将已知大小的 N 个物品选择一些放入背包中，每种物品的大小是 w_i, 每种物品产生的收益是 x_i 目标是求解收益的最大值
  - 0-1 背包问题：背包问题的特殊情况，物品只能整个放入或者不放入
- 求解 0-1 背包问题的动态规划法
  - 分析：状态转移方程为 $F_j = max(F_j, x_i + F_{j-w_i})_{w_i \leq j}$ 也就是对于第 i 个物品，可以选择装上去也可以选择不装上去

```
for(i=0;i<N;i++)
{
    for(j=M;j>=w[i];j--)
            f[j]=max(f[j],x[i]+f[j-x[i]]);
}
```

2.4 Greedy

- 优化问题
  - **Given a set of constraints and an optimization function**. Solutions that satisfy the constrains are called **feasible solutions**. A feasible solution for which the optimization function has the best possible value is called an **optimal solution**.
- 贪心算法
  - **Make the best decision at each stage, under some greedy criterion****. A decision made in one stage is **not changed in a later stage, so each decision should assure feasibility**
  - 坚持局部最优直到全局最优，但**不一定能达到全局最优**, 贪心算法的结果不一定是真正的最优解
  - Greedy algorithm works **only if** the local optimum is equal to the global optimum 只在局部最优和全局最优等价的时候可以使用
- Activity Selection Problem 活动安排问题
  - 动态规划方法:n 个事件 a1-an, Cij 表示第 i 和第 j 个事件之间能安排的最多的事件数目 $c_{ij} = max(c_{ik} + c_{kj}) + 1$，时间复杂度 $T = O(N^2)$

* 另一种动态规划方法 $c_{1j} = max(c_{1,j-1}, c_{1,k(j)} + 1)$, 这里 k(j) 是距离 aj 最近的不冲突活动，并且要在 aj 之前完成
* 有权重的模式 $c_{1j} = max(c_{1,j-1}, c_{1,k(j)} + w_{ij})$

– 贪心方法

1. 总是选择开始最早的---错
2. 选择持续时间最短的活动---错
3. 跟别的活动冲突最少---错
4. 选择尽早结束的活动--对
5. 选择最迟开始的活动--对
    * 规律：每个子问题中活动结束最早的一定会包含在最优解当中

– Consider any nonempty subproblem Sk, and let am be an activity in Sk with the earliest finish time. Then am is included in **some maximum-size subset** of mutually compatible activities of Sk

– corectness

* algorithm gives non-overlapping intervals
* the result is optimal
* 需要证明按照这样的方法选取**不会使得结果变差**

- 可以进行贪心的基本条件
  – **the local optimum is equal to the global optimum**
  – 可以在做出一次贪心选择之后转化成一个子问题
  – 证明用贪心的方法总能存在最优解
  – 存在**最优子结构** (optimal substructure)，在做了贪心选择之后还可以在子问题中找到最优解
- Huffman Code 哈夫曼编码--用于文件路径压缩

- 把出现频率高的用短的 01 串来编码，以达到压缩路径的目的
- 需要保证没有一个字符是另一个字符的前缀，否则存在多种解码方式
- 等长编码：N 个字母需要 $\log_2 N$ 位的 01 字符串进行编码

3. CS Theory

3.0 Amortized Analysis 摊还分析

- **最坏 bound>= 摊还 bound>= 平均 bound(不是 cost，而是 bound)**
  - " 摊还 cost 一定小于平均 cost"这样的表述是**错**的
- 摊还分析**不涉及概率**上的分析，而可以保证**最坏情况下每个操作的平均性能**

案例：栈的 multipop

```
while(!isEmpty(s)&&k>0){
    pop(s);
    k--;
}
//The time cost T=min(sizeof(s),k)
```

- 方法一：Aggregate analysis **聚合分析**
  - 基本思路：n 个连续的操作最差情况下需要消耗的时间为 T(n)，则每个操作的摊还代价为 T(n)/n
  - 对于此题，一次 multipop 最差的时间复杂度为 O(n)，但是 pop 的次数不能超过 push 的次数，而 push 最多有 n 次，因此 n 次的 push，pop 和 multipop 最多消耗 O(n) 的时间，摊还之后的时间复杂度就是常数时间
- 方法二：Accounting method **核算法**
  - 基本思路：对不同的操作赋予不同的费用，可能会多于实际的代价，当一个操作的摊还代价超出实际的代价时就可以将多出来的存储邮与支付后续代价不够时的情况，

称为**信用** credit，但是我们要**始终保证信用不能是负数**，也就是信用可以累计，但不能透支

操作	实际代价	摊还代价
push	1	2
pop	1	0
multipop	min(k,s)	0

  - 我们可以把摊还代价设置成这样是因为 pop 次数之和肯定不会超过 push 的次数，因此可以保证任何情况下信用值不是负数，而此时摊还代价总和的上界为 2n 所以平均的摊还代价是 O(1)

- 方法三：Potential method 势能法
  - 基本思路：对信用进行更加定量的分析，定义势能函数使得 $c'_i = c_i + \Theta(D_i) - \Theta(D_{i-1})$，计算得到，$\sum_{i=1}^{n} c'_i - \sum_{i=1}^{n} c_i = \Theta(D_n) - \Theta(D_0)$，因此需要保证势能函数最终值大于初始值
  - 本题中，定义 $\Theta(D_n)$ 表示 stack 中的元素的个数，对于 push，实际代价为 1，引起的势能变化也是 1，因此代价为 2，而对于 pop 类型的操作，pop 了 K 次的实际代价为 K，引起势能的变化为-K，因此代价都是 0，求和可得摊还代价为 O(1)
    - 这类方法需要保证势能函数的初始值不能大于最终值
    - **In general, a good potential function should always assume its minimum at the start of the sequence.** 一个好的势能函数需要让势能函数的初始值是最小值

3.1 P and NP

- Recall
  - Euler circuit problem 欧拉回路问题: find a path that touches **every edge exactly once** 找到一条路径经过每条边恰好一次，一笔画问题
  - Hamilton cycle problem: find a single cycle that contains every vertex 找到一个回路经过每个点一次

  - Halting problem 停机问题: Is it possible to have your C compliler detect all infinite loops?---No 编译器不能发现所有的无限循环

- 图灵机 Turing Machine
  - 组成：Infinite Memory && Scanner 无限的内存和扫描头
    - scanner 上有若干 head，每一个扫描头一次只能指向一个 state，并且一次只能左右移动一格
  - 可以执行的操作：
    - change the finite control state 改变
    - erase the symbol in the unit currently pointed by head and write a new symbol in 清除并写入
    - Head moves on unit to left or right or stays at its current position 左右移动或保持不动
  - A **deterministic turing machine** executes one instruction at each point in time. Then depending on the instruction and it goes to the next unique instruction **确定性**图灵机
    - 对于给定的输入**每一步的执行都是唯一的**图灵机
  - A nondeterministic turing machine is free to choose its nect step from a finite set and is one of the steps leads to a solution, it will always choose the correct one 不确定性图灵机
    - 对于给定的输入可以自由选择执行的下一步，并且会选择正确的 solution

- NP problem
  - A problem is NP if we can prove any solution is true in polynomial time 可以在**多项式的时间内验证**问题的**任意解**是对的
  - Not all the decidable problems are in NP. 可描述的问题不全是 NP 问题

- NP-Complete Problems -- the hardest NP 完全问题
  - 性质：any problem in NP can be polynomially reduced to it 一个 NPC 问题可以从任何 NP 问题通过多项式规约得到

- NP-hard + NP 可以推出是 NPC
- 如果我们可以在多项式时间内解决**任何一个 NPC 问题**，那么我们就可以在**多项式时间内解决所有 NP 问题**
- 例如，如果哈密顿回路问题是 NPC 问题，我们可以推断出旅行商人问题 (**简称 TSP 问题**) 也是 NPC 问题
  - 一个问题可以被归约成 HCP 问题，然后被归约成 TSP 问题，也就是说归约具有传递性
- 第一个被证明为是 NPC 问题的是 **Circuit Satisfiability 问题** (Circuit SAT)
- SAT 问题，顶点覆盖问题，哈密顿回路问题都是 NPC 问题，**停机问题不是 NPC 问题**

- P，NP，NPH，NPC 的关系总结
  - P 是可以在多项式时间内解决的问题，NP 是可以在多项式时间内验证一组特定的解是否正确的问题，NP-hard 问题是 NP 问题可以通过多项式归约得到的一个较为复杂的问题，NPC 问题就是 NP 问题归约之后得到的，NPC 问题既是 NP 又是 NP-hard
  - 几个问题的**包含关系**是，$P \subset NP, NP \cap NP - hard = NPC$

- 形式化的语言描述
  - Abstract Problem：a binary relation on a set I of problem instances and a set S of problem solutions.
  - 形式化语言的描述
  - A verification algorithm is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate.
  - SAT 问题就是给定 n 个布尔变量 x_i，从他们本身和他们的否中选出 K 组，每组 m 个变量，小组取并集，大组取交集，要求让最后的结果是 1，这样的叫做 `m-SAT` 问题
  - 如果最大团问题是 NPC 问题，则顶点覆盖问题也是 NPC 问题
    - 转化方式是一个图 G 如果有大小为 K 的最大团当且仅当 G 的补图有一个大小为 `|V|-K` 的顶点覆盖
  - `co-NP` 是所有满足 L 的补和 L 本身都是 NP 问题的形式化语言 L 所构成的集合

- An *alphabet* Σ is a finite set of symbols $\{0, 1\}$
- A *language* L over Σ is any set of strings made up of symbols from Σ $L = \{ x \in \Sigma^* : Q(x) = 1 \}$
- Denote *empty string* by ε
- Denote *empty language* by $\emptyset$
- Language of all strings over Σ is denoted by Σ^*
- The *complement* of L is denoted by $\Sigma^* - L$
- The *concatenation* of two languages L_1 and L_2 is the language $L = \{ x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2 \}$.
- The *closure* or *Kleene star* of a language L is the language $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$,

  where L^k is the language obtained by concatenating L to itself k times

图 20:

* 这些概念最有可能的关系是，NP 和 co-NP 有交集，P 包含在这个交集当中
* $L \in NP \cap co-NP$ 如果 L 是一个 P 问题

3.2 Approximation algorithm

- Approximation Ratio 近似率
  - 对于任何规模为 n 的输入，C 为算法的 cost，C* 为优化后的算法的 cost，则 $max(\frac{C}{C'}, \frac{C'}{C}) \leq \rho(n)$, 如果一个算法的近似率达到了 $\rho(n)$ 则该算法可以被称为一个 $\rho(n)$-近似算法
  - **approximation scheme**：除了 n 以外还收一个参数 ϵ 影响
  - PTAS：polynomial-time approximation scheme 关于 n 成多项式复杂度的算法 (对于特定的 ϵ)
  - FPTAS：fully polynomial-time approximation scheme 关于 n 和 ϵ 都成多项式复杂度的算法

案例 1：Bin Packing

- Next Fit 方法：策略是**尽可能先把箱子填满**
  - 性质：如果最优的方法需要 M 个箱子，那么 next fit 方法使用的箱子不会超过 `2M-1`，可以用反证法证明该结论，最重要的条件是**相邻的两个箱子内的和肯定大于 1**，否则就会放到一个箱子里
    * 策略是当前箱子放不下直接到下一个去

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
        if ( item2 can be packed in the same bin as item1 )
place item2 in the bin;
        else
create a new bin for item2;
        item1 = item2;
    } /* end-while */
```

```
}
```

- First Fit 方法：策略是**找到第一个能放下的箱子**
  - 可以用 $O(N \log N)$ 的时间复杂度来实现这个算法
  - 如果最优的情况需要 M 个箱子，那么这个算法所需要的箱子不会多余 1.7M

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
place item in that bin;
        else
create a new bin for item;
    } /* end-while */
}
```

- Best Fit 方法：策略是找到**能放下这个当前物品并且剩余空间最小的箱子**
  - 时间复杂度也是 $O(N \log N)$, 所需要的箱子个数不会超过最优解的 1.7 倍
  - 以上三种实际上都是 **Online Algorithm**，可以证明只要是 Online 的算法所需要的箱子的个数不会少于最优解的 5/3 倍
- OffLine Algorithm
  - 在开始装箱之前先检查一次所有的箱子并按照非递增的顺序排序，然后使用 First Fit 算法，这种情况下需要的箱子的个数不会超过 (11M+6)/9
  - 简单的贪心方法会给出一些比较好的结果

案例 2：Knapsack Problem 0-1 背包问题

- 目标：将 N 个物品装入容量为 M 的背包里，每个物品有自己的重量 Wi 和收益 Pi，最终目标是要让收益最大化
- 使用贪心算法的近似率是 2，证明如下

- 注意到这些关系: $p_{max} \leq P_{opt} \leq P_{frac}, p_{max} \leq P_{greedy}, P_{opt} \leq P_{greedy} + p_{max}$ 因此可以得到 $\frac{P_{opt}}{P_{greedy}} \leq 1 + \frac{p_{max}}{P_{greedy}} \leq 2$

- 使用动态规划的方法求解
  - $W_{i,p}$ 代表仅考虑前 i 个物品利益最大化的最小的重量，则状态转移方程为 $W_{i,p} = W_{i-1,p}$(如果 pi>p) $W_{i,p} = min(W_{i-1,p}, W_{i-1,p-p_i} + w_i)$
  - 这种动态规划算法的事件复杂度是 $O(n^2 p_{max})$

- 背包问题是 NP-hard 问题，但是如果对于规模为 n 的背包问题，没有一个物品的大小超过多项式复杂度则不是 NP-hard 问题

案例 3：K-center problem

- 问题描述: 在给定的 N 个点的平面中选择 K 个点 (不一定是已有的点) 作为中心作圆覆盖所有的点，要求使得这些圆中的最大半径 (distance) 取得最小值
  - 关于距离的定义：需要运算函数满足同一性，对成性和三角不等式
  - 一种贪心的思路: 将第一个中心放在尽可能好的地方，然后不断加入点使得覆盖半径减小
  - 如果知道最优的解 C*

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r )
{   Sites  S' [ ] = S[ ]; /* S'  is the set of the remaining sites */
    Centers  C[ ] = empty;
    while ( S' [ ] != empty ) {
        Select any s from S'  and add it to C;
        Delete all s'  from S'  that are at dist(s' , s) <= 2r;
    } /* end-while */
    if ( |C| <= K ) return C;
    else ERROR(No set of K centers with covering radius at most r);
}
```

  - 如果不知道最优解 r(C*): 采用**二分法**，将已知的最大半径和 0 作为起点进行二分

```
Centers  Greedy-Kcenter ( Sites S[ ], int n, int K )
```

```
{   Centers  C[ ] = empty;
    Select any s from S and add it to C;
    while ( |C| < K ) {
        Select s from S with maximum dist(s, C);
        Add s it to C;
    } /* end-while */
    return C;
}
```

- 这种算法的**近似率是 2**，因此是个 2-approximation
- **除非 P=NP，否则 K-center 问题不存在近似率小于 2 的逼近算法**

3.3 Local Search 局部搜索

- 和贪心算法是有区别的，贪心算法不是局部搜索的特例
- 局部搜索的框架 Local Search Framework
  - Local：a local optimum is a best solution in a neighborhood
    - 定义邻居关系：S ~ S′: S′ is a neighboring solution of S – S′ can be obtained by a **small modification** of S.
    - N(S): neighborhood of S
  - Search：在 neighborhood 中从一个点出发，找到局部最优的解
    - **gradient descent** —— 梯度下降法，找梯度下降最快的方向来优化
- 局部搜索算法的框架

```
SolutionType Gradient_descent()
{
    Start from a feasible solution S in FS ;
    MinCost = cost(S);
    while (1) {
        S'  = Search( N(S) ); /* find the best S'  in N(S) */
        CurrentCost = cost(S');
```

```
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;
            S = S' ;
        }
        else  break;
    }
    return S;
}
```

案例 1：顶点覆盖问题

- 问题描述：在无向图 G 中找出最小的点集 S，对于 G 的每一条边，至少有一个顶点在 S 中
  - 快速得到可行解：直接取所有的顶点，此时的 cost(S)=|S|
  - S′ 是 S 的邻居，如果 S′ 可以从 S 中去掉一个点得到
  - 搜索方法：从 S=V 开始，每次删除一个点并且检查是否为最小覆盖
    - 梯度下降法不一定 work，改进后的算法：Metropolis Algorithm
    - 梯度下降法是定向选择邻居，而 metropolis 是**随机选择邻居**的算法

```
SolutionType Metropolis()
{
    Define constants k and T;
    Start from a feasible solution S in FS ;
    MinCost = cost(S);
    while (1) {
        S'  = Randomly chosen from N(S);
        CurrentCost = cost(S' );
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S' ;
        }
        else {
            With a probability e^{-dcost/KT}  , let S = S' ;
            else  break;
```

```
            }
        }
        return S;
    }
```

案例 2：Hopfield Neural Network

- 无向图 G 中的边带有权重 wi, wi 是正数时两个节点状态不同，是负数时节点状态相同。**There may be no configuration that respects the requirements imposed by all the edges.**
  - 定义好边和坏边 (好边就是满足条件的边) $w_e s_u s_v < 0$
  - 对于一个顶点，如果其好边数大于坏边数则称之为满足的，如果所有点都是满足的，那么这个图就是稳定的
  - **state-flipping** 算法

```
ConfigType State_flipping()
{
    Start from an arbitrary configuration S;
    while ( ! IsStable(S) ) {
        u = GetUnsatisfied(S);
        su = - su;
    }
    return S;
}
```

  - State-flipping 算法最多在 $W = \sum |w_e|$ 次循环后达到稳定情况，所以一定会停止

案例 3：Max Cut Problem

- 在无向图 G 中找到一个分割 (A,B), 使得 $w(A, B) = \sum_{u \in A, v \in b} w_{uv}$ 最大
  - 可行解：任何一种分割方案
  - 邻居的定义：S′ 可以由 S 在两个分割 AB 之间移动一个点得到，**实际上是 Hopfield Neural Network 的一种特殊情况**
  - 定理：局部最优解的权重和不会低于全局最优解的一半 $w(A, B) \geq \frac{1}{2} w(A*, B*)$

- **big-improvement-flip** 算法：当新的局部最优解的增长的幅度小于 $\frac{2\epsilon}{|v|}w(A,B)$ 的时候就停止，为了让算法可以在多项式时间内结束
  - 这样以来就有 $(2+\epsilon)w(A,B) \geq w(A*,B*)$
  - 最多 $O(\frac{n}{\epsilon}\log W)$ 次 flips 之后就可以停下来
- 最大分割问题存在近似于为 1.1382 的逼近算法，但是没有近似率低于 $\frac{17}{16}$ 的

3.4 Random Algorithm 随机算法

- 用随机的决策来处理最坏的情况
- 基本的性质
  - **efficient randomized algorithms that only need to yield the correct answer with high probability** 用非常高的概率给出正确的答案
  - **randomized algorithms that are always correct, and run efficiently in expectation** 总是正确的，并且在期望中运行的很有效率

案例 1：雇佣问题

- 总的 cost= 天数 N * 每天面试的花费 + 雇佣的人数 M* 雇佣费用
- 一种简单的算法
  - 但这种算法在参加者的质量递增的时候会不 work

```
int Hiring ( EventType C[ ], int N )
{   /* candidate 0 is a least-qualified dummy candidate */
    int Best = 0;
    int BestQ = the quality of candidate 0;
    for ( i=1; i<=N; i++ ) {
        Qi = interview( i ); /* Ci */
        if ( Qi > BestQ ) {
            BestQ = Qi;
            Best = i;
            hire( i );  /* Ch */
        }
```

```
    }
    return Best;
}
```

- 如果参加面试者的顺序是随机的，那么最后的总 cost 是 $O(C_h \ln N + NC_i)$，但是需要进行随机的排列，将上述算法中的数组 C 进行随机的排列之后在进行就可以达到这种效果
- Randomized Permutation 随机排序算法
  - 一种比较简单的实现方式：生成**随机数来代表元素的优先级**

```
void PermuteBySorting ( ElemType A[ ], int N )
{
    for ( i=1; i<=N; i++ )
        A[i].P = 1 + rand()%(N3);
        /* makes it more likely that all priorities are unique */
    Sort A, using P as the sort keys;
}
```

案例 2：雇佣问题 (online)

- 新的解决思路：先面试前 k 个不录取，在后面的 N-K 个人中选出第一个比前 K 个最高分要高的

```
int OnlineHiring ( EventType C[ ], int N, int k )
{
    int Best = N;
    int BestQ = - INF ;
    for ( i=1; i<=k; i++ ) {
        Qi = interview( i );
        if ( Qi > BestQ )   BestQ = Qi;
    }
    for ( i=k+1; i<=N; i++ ) {
        Qi = interview( i );
        if ( Qi > BestQ ) {
            Best = i;
            break;
```

```
        }
    }
    return Best;
}
```

- 记 Si 为第 i 个人被录取的概率，则需要满足第 i 个人是得分最高的并且第 k+1 到 i-1 个人都没有被录取

$$P(S_i) = \frac{k}{N(i-1)}P(S) = \sum_{i=k+1}^{N} \frac{k}{N(i-1)} = \frac{k}{N}\sum_{i=k}^{N-1}\frac{1}{i} \frac{k}{N} \ln\frac{N}{k} \leq P(S) \leq \frac{k}{N}\ln\frac{N-1}{k-1}$$

- 用求导的方法可以得到最合适的 k 值，应该是 N/e(e 是自然对数的底数)

案例 3：快速排序

- 确定性的算法中，快排**最差的时间复杂度**是 $O(N^2)$，
- 平均的时间复杂度是 $O(N \log N)$ 要求是 **every input permutation is equally likely**
- 随机选择一个位置作为 pivot
  - central splitter：将数组分成两段的 pivot 并且每段至少是总长度的 1/4
  - Modified Quicksort：在开始递归之前选择出一个中心分割点
- **The expected number of iterations needed until we find a central splitter is at most 2.**
- 最终随机选择 pivot 的快速排序的复杂度是 $O(N \log N)$

3.5 Parallel Algorithm 并行算法

- 两种并行算法的模型
  - Parallel Random Access Machine(PRAM)
    - 使用**共享的内存**，每个进程消耗 unit time access
    - 解决访问冲突的办法
      - EREW：不能同时读写一个位置

· CREW：可以同时读不能同时写

· CRCW：可以同时读写

* 计算 n 个数的和

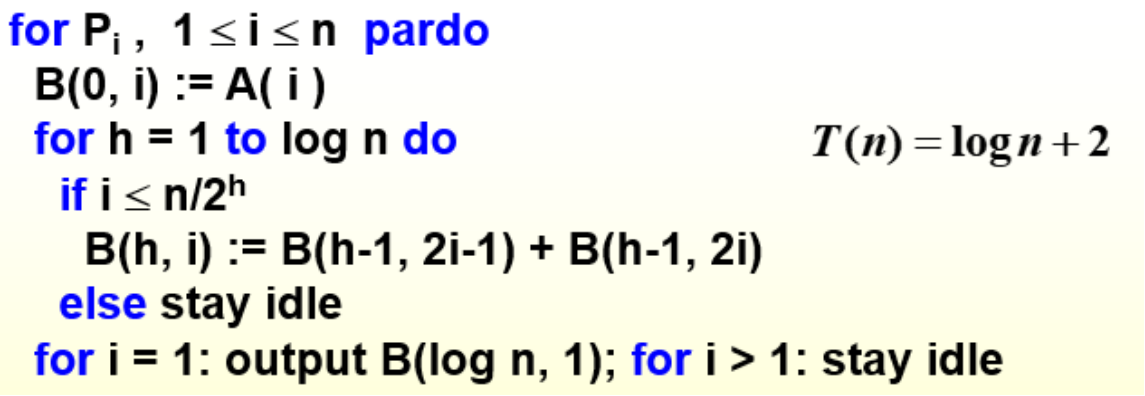


图 21:

* 缺点

· 不能知道当处理器个数发生变化的时候会如何影响我们的算法

· **Fully specifying the allocation of instructions to processors requires a level of detail which might be unnecessary**

– Work-Depth(WD) 算法

* In Work-Depth presentation, each time unit consists of a sequence of instructions to be performed concurrently; the sequence of instructions may include **any number**

– 计算 n 个数的和，操作总数变少了

- 评定平行算法优劣的规则

– Work Load 工作总量 W(N)

– 最坏的运行时间 T(N)

* P(N)=W(N)/T(N) processors and T(N) time(on a PRAM)

* W(N)/p time using any number of p(less than W(N)/T(N)) processors(on a PRAM)

```
for P_i , 1 ≤ i ≤ n pardo
  B(0, i) := A( i )
for h = 1 to log n
   for P_i, 1 ≤ i ≤ n/2^h pardo
      B(h, i) := B(h-1, 2i-1) + B(h-1, 2i)
for i = 1 pardo
  output B(log n, 1)
```

图 22:

  - W(N)/P+T(N) time using any number of p processors(on a PRAM)

- **WD-presentation Sufficiency Theorem**
  - **An algorithm in the WD mode can be implemented by any P(n) processors within O(W(n)/P(n) + T(n)) time, using the same concurrent-write convention as in the WD presentation.**

案例 1：Prefix Sum

- 伪代码如下所示
- T(N)=O(log N) W(N)=O(N)

案例 2：数组归并

- 将两个单调不减的数组 A,B 合并成两个
  - 简化一下问题，m=n，并且 A,B 中的元素各不相同，log n 是整数
  - 分割法 Partitioning Paradigm
    - 分割成若干个小问题，并行执行

```
for P_i , 1 ≤ i ≤ n pardo
  B(0, i) := A(i)
for h = 1 to log n
  for i , 1 ≤ i ≤ n/2^h pardo
    B(h, i) := B(h - 1, 2i - 1) + B(h - 1, 2i)
for h = log n to 0
  for i even, 1 ≤ i ≤ n/2^h pardo
    C(h, i) := C(h + 1, i/2)
  for i = 1 pardo
    C(h, 1) := B(h, 1)
  for i odd, 3 ≤ i ≤ n/2^h pardo
    C(h, i) := C(h + 1, (i - 1)/2) + B(h, i)
for P_i , 1 ≤ i ≤ n pardo
  Output C(0, i)
```

$$T(n) = \mathrm{O}(\log n) \qquad W(n) = \mathrm{O}(n)$$

图 23: image-20200527104505558

* 引入 RANK(j,A) 表示 B[j] 元素位于 A 的哪两个元素之间，分别计算 A 和 B 的所有 RANK，然后采用如下方法归并

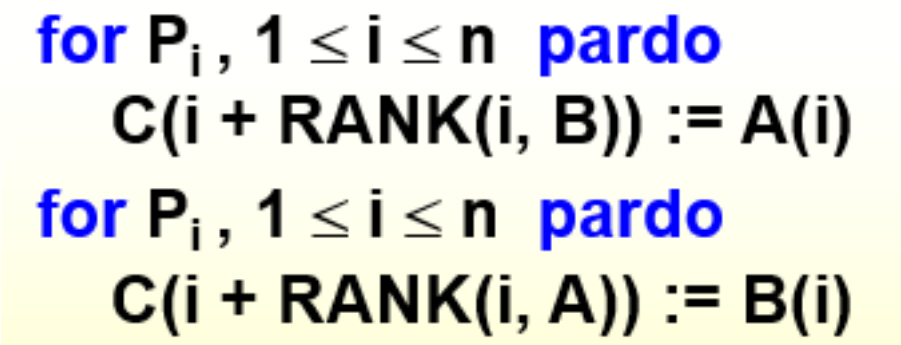


图 24:

* 计算 RANK 使用二分查找，可以使时间复杂度变成 log n 级别
* 本算法的时间复杂度为 `log n`，work load 的大小是 `n log n`

– 传统的串行算法：时间复杂度和 work load 都是 `O(n+m)` 级别

- 新算法：parallel ranking
  - 假设 n=m 并且数组严格递增
  - 步骤一：令 $p = \frac{n}{\log n}$，把两个数组分别等分成 P 组，**间隔为 log n**，并计算每个小组中的 RANK
  - 步骤二：Actual Ranking 有 2p 个规模为 O(log n) 的子问题，所以总的时间复杂度还是 log n，但是 work load 变成了 O(n) 级别

案例 3：找最大值

- 把求和算法中的 + 改成 max 即可使用
- 第一种算法：时间 O(1), work load 是 n^2
- 第二种：双对数算法
  - $h = \log\log n$，将问题按照 $\sqrt{n}$ 的规模划分成 $\sqrt{n}$ 个子问题，对每个子问题用时间和 work load 都为 $O(\sqrt{n})$ 的算法解决，最后用第一种算法求出 $\sqrt{n}$ 个结果中的最大值

```
for P_i , 1 ≤ i ≤ n  pardo
    B(i) := 0
for i and j, 1 ≤ i, j ≤ n  pardo
    if ( (A(i) < A(j)) || ((A(i) = A(j)) && (i < j)) )
          B(i) = 1
    else B(j) = 1
for P_i , 1 ≤ i ≤ n  pardo
    if B(i) == 0
      A(i) is a maximum in A
```

$$T(n) = O(1), \quad W(n) = O(n^2)$$

图 25:

        * 最后的结果是 $T(n) = O(\log \log n)$ $W(n) = n \log \log n$
    - 另外的划分方式：照规模为 h 进行划分，最后的 Work Load 变成了 O(n)
- 第三种：随机算法，高概率使得 $T(n) = O(1)$ 并且 $W(n) = O(n)$
    - 【Theorem】The algorithm finds the maximum among n elements. With very high probability it runs in O(1) time and O(n) work. The probability of not finishing within this time and work complexity is $O(1/n^c)$ for some positive constant c.

```
while (there is an element larger than M) {
     for (each element larger than M)
         Throw it into a random place in a new B(n7/8);
     Compute a new M;
}
```

3.6 External Sorting 外部排序

- 为什么不直接在磁盘上进行快排？
    - 磁盘 I/O 的效率太低了，需要找到磁道和扇区
    - 解决方法：使用 tapes 进行排序
        * tape 的特点：只能顺序地访问，不能像内存中的数组一样直接寻址
        * **至少需要 3 条 tapes**
- 假如内存里最多可以保持 M 条记录，对 N 个元素进行排序
    - 每次读出 M 个数据放进内存进行排序，然后放在 tapes 中
    - 一共需要进行的循环的次数 (passes) $1 + [\log_2(N/M)]$ 向上取整 (2 路归并)
- 优化的目标
    - 减少循环的次数
    - run merging
    - 并行操作的缓冲处理

- run generation 生成更好的 run

- **减少循环的次数**的方法
  - 使用 K 路归并排序，需要的 pass 数变成了 $1 + [\log_k(N/M)]$
  - 缺点是**需要 2k 个磁带**，比较消耗磁带

- 用更少的磁带来进行排序
  - 用 3 个 tape 来进行二路归并排序
    - 不要进行对半拆分，采用不对等的拆分来做
    - **合并的次数相比于对半分变多了，但是不需要进行磁带的复制，会更加节约时间**, 不对称分割效率反而高的原因是减少了磁带复制的时间消耗
  - 当 run 的次数是斐波那契数 Fn 的时候最好的拆分的办法是把它拆成 Fn-1 和 Fn-2
  - 对于 K 路 merge

Claim: For a *k*-way merge, $F_N^{(k)} = F_{N-1}^{(k)} + \cdots + F_{N-k}^{(k)}$

where $F_N^{(k)} = 0 \ (0 \le N \le k-2), \ F_{k-1}^{(k)} = 1$

图 26:

  - 如果不是斐波那契数，可以**增加一些空的 runs** 来凑到斐波那契数
  - K 路的 merge 最少需要 k+1 个 tapes

- **缓冲区的优化 (这里还是没搞明白)**
  - 并行的实际上是对 buffer 的读和写
  - 对于一个 K 路归并，需要 2k 个输入 buffer 和 2 个输出 buffer 来进行并行操作
  - 事实上 K 不是越大越好，因为如果 K 增大，就会导致 input buffer 的数量需求增加，导致 buffer size 减少，导致磁盘中一个 block 的 size 减少，导致访问磁盘的 seek time 增加，因此最优的 K 值取决于磁盘的参数和外部 memory 的规模

- 如何获取更加长的 run
  - 使用堆的结构来进行排序操作，规则是一直取出堆中现存的可以放在现在所在的 run 后面的最小的数，直到堆中的数据都放不进当前 run 了再更换一个 run
  - 如果内存可以容纳 M 个元素，则这种方法生成的 run 的平均长度为 2M
  - 再输入的元素接近已经排好序的状态时非常 work
- 最小化 merge 的时间 (这个比较简单)
  - 使用哈夫曼树，每次把最短的两个 run 进行合并
  - T=O(the weighted external path length)

4. 新增内容：Project 知识点的考察

- 突然说道考试要考 project 中出现过的内容，心血来潮整理一下这学期七个 project 中的相关内容

4.1 Project 1：Shortest path with heaps

- 这个 project 是唯一一个我们小组没有做过的 project
- 斐波那契堆：本题中引入的一种新的堆数据结构，具体的性质如下
  - 斐波那契堆和二项队列类似，是一组最小堆有序树构成的，相比于二项队列有更好的摊还性能
  - 堆中的每一棵树都有根但是无序，每个节点 x 包含指向父节点的指针和指向任意一个子节点的指针，x 的所有子节点都用双向的循环链表链接，称为 x 的子链表中的节点都有指向左右兄弟，所有的根节点之间也用一个双向的循环链表连接起来
  - 需要维护一个**指向斐波那契堆最小元素的指针**
  - 各种操作的时间复杂度的分析
    - FindMin：因为维护了一个指向最小元素的指针，所以时间复杂度是 $O(1)$
    - DeleteMin: 需要先 FindMin 之后在进行合并，总的时间复杂度是 $O(\log N)$
    - Decrease Key: 可能需要进行位置的调整，时间复杂度是 $O(\log N)$

* Insert：时间复杂度是 $O(\log N)$，但是摊还代价是 $O(1)$

- 总的空间复杂度是线性的

4.2 Project 2：Safe fruit

- 这个感觉也没什么好讲的，是一个非常具体的算法题，用回溯法实现，感觉直接拿来考没有什么必要

4.3 Project 3：Beautiful Sequence

- 这个题也是一个非常具体的算法题，主要就是定义了一种 Beautiful Sequence 的结构
  - 一个序列被称为是漂亮的，如果它包含两个相邻的元素，并且两个元素的差的绝对值小于一个给定的数字 m
  - 这个 project 就是需要计算所有可能的**漂亮子序列**的个数
- 如果采用传统的方法逐一验证，那么对于长为 n 的序列，其子序列个数为 2^n，则验证所需要的时间复杂度是 $O(2^N N)$，是非常离谱的，但是我们这里可以使用动态规划解决
  - 建立一个数组 `dp[ ]`，`dp[k]` 表示由子序列中的前 K 个数中的漂亮子序列个数，我们考虑从 k 到 k+1 时候的变化，对于前 K 个中的所有漂亮子序列，新加入的第 K+1 个数可有可无，因为不管怎么样都是漂亮的子序列，因此这一部分产生的漂亮子序列个数为 `2dp[k]`，另一部分漂亮子序列则是由于第 K+1 个数的加入而产生的，这就需要前 K 个数中存在一个数 i，这个数和第 K+1 个数的差的绝对值不超过 m，此时产生的新的漂亮子序列的个数等于前 i 个数产生的不漂亮子序列的个数
  - 所以这个题目的状态转移方程为 $dp[k+1] = dp[k] + \sum_{j<=k, |a[j]-a[k]|\leq m} (2^{j-1} + dp[j-1] - dp[j])$ 此时的时间复杂度是 $O(N^2)$

4.4 Project 4：Huffman Code

- 主要是模拟哈夫曼树来进行哈夫曼编码的正确性校验，好像也没什么东西，因为哈夫曼编码是 PPT 上的内容，主要要记住一点就是哈夫曼编码的各个编码之间**不能存在两个有前缀关系**

4.5 Project 5：Bin Pack

- 二维装箱问题，是一维的装箱问题的二位拓展，需要考虑箱子的长宽和物品的长宽
- First-Fit 算法
  - 先将物体按照高度排序，然后从底部开始，按照高度从大到小遍历所有物体，找到第一个可以放得下的位置把物体放进去
  - 排序消耗的时间复杂度是 $O(N \log N)$ 装箱的时间复杂度是 $O(N^2)$
  - 逼近率约为 2.7，是一个不太好的数字

4.6 Project6：Skip List

- 主要是引入了一种叫做跳表的数据结构，采用随机算法来降低时间复杂度，本质上是用时间来换空间
- 跳表的性质如下
  - 一个跳表有多个层级 (level)，每一层都是一个普通的链表，第一层包含全部节点
  - 每上一层会有**一定概率保留上一层的部分节点**，直到最高的一层只剩下一个
  - 每一层中的节点按照递增顺序排序
  - 在代码实现中就会给每个节点有多个 next 指针 (按照层数构成一个数组)，代码实现应该如下

```
struct SNode
{
        int key;
        SNode *forword[MAXN_LEVEL];
};

struct SkipList
{
        int nowLevel;
        SNode *head;
};
```

- level 的期望值是 $O(\log N)$ 级别的，空间复杂度依然是 $O(N)$ 级别
- 由于随机算法的加持，跳表的查询，插入，删除的平均时间复杂度都是 $O(\log N)$ 但是最坏的时间复杂度依然是线性级别的

4.7 Project 7：MapReduce

- 主要介绍了一种并行算法的框架也就是 MapReduce，核心步骤分为 map 和 reduce 两个部分，其中 map 是并行地对大规模的数据进行处理，reduce 是将处理的结果进行合并