ADS

ch1 AVLTree

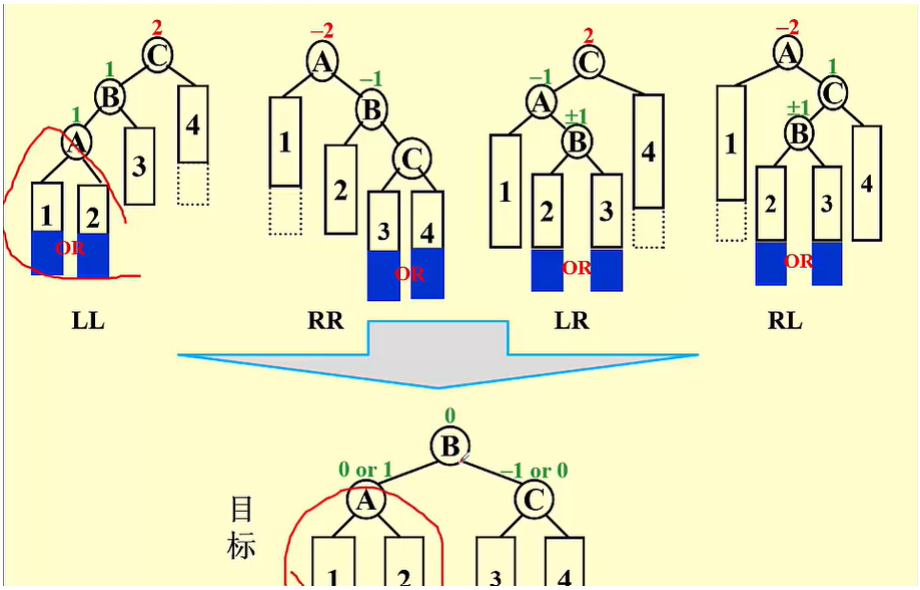
插入最多旋转2次，删除最多旋转O(logN)次

aim: to solve dynamical-search problem

1. rotation

focus: trouble maker

1.1 LL ROTATION&&RR Rotation



2 analysis

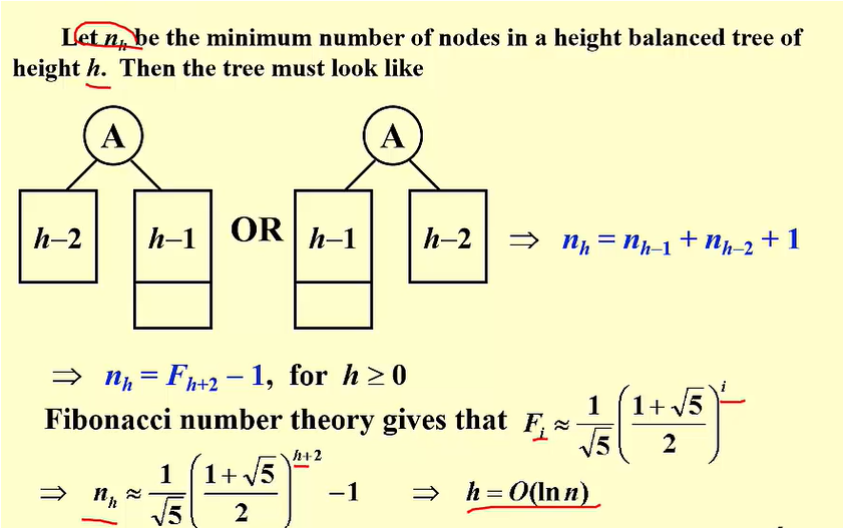
performance : height of tree

- complete binary tree--least height--hard
- 放松要求：高度、宽度、结点数量偏差小

Insert:

maintain: remain search tree, balance

induction:



3 code

```
int getheight(AVLT root)
{
    if(!root) return -1;
    else return root->height;
}
AVLT RRRotation(AVLT root)
{
    AVLT temp = root->right;
    root->right = temp->left;
    temp->left= root;
    root->height =getheight(root->left)>getheight(root->right)?getheight(root-
>left)+1 :getheight(root->right)+1;
    temp->height =getheight(temp->left)>getheight(temp->right)?getheight(temp-
>left)+1 :getheight(temp->right)+1;
    return temp;
}
AVLT LLRotation(AVLT root)
{
    AVLT temp = root->left;
    root->left = temp->right;
    temp->right = root;
    root->height =getheight(root->left)>getheight(root->right)?getheight(root-
>left)+1 :getheight(root->right)+1;
     temp->height =getheight(temp->left)>getheight(temp->right)?getheight(temp-
>left)+1 :getheight(temp->right)+1;

    return temp;
}
AVLT RLRotation(AVLT root)
{
    root->right = LLRotation(root->right);
    return RRRotation(root);
}
AVLT LRRotation(AVLT root)
{
    root->left = RRRotation(root->left);
    return LLRotation(root);
}
```

```
AVLT Insert(AVLT root, int x)
{
    if(!root)
    {
        AVLT newnode = (AVLT)malloc(sizeof(struct AVLTree));
        newnode->val = x;
        newnode->left = newnode->right = NULL;
        newnode->height = 0;
        return newnode;
    }
    if(root->val > x)
    {
        root->left = Insert(root->left, x);
        if(abs(getheight(root->left)-getheight(root->right))==2)
        {
            if(root->left->val > x) root = LLRotation(root);
            else root = LRRotation(root);
        }
    }
    else if(root->val < x)
    {
        root->right = Insert(root->right, x);
        if(abs(getheight(root->left)-getheight(root->right))==2)
        {
            if(root->right->val < x) root = RRRotation(root);
            else root = RLRotation(root);
        }
    }
     root->height =getheight(root->left)>getheight(root->right)?getheight(root->left)+1 :getheight(root->right)+1;
    return root;
}
```

4 exercise:

4.1 n_h and fibonacci

$n_h = F_{h+2} - 1$

If the depth of an AVL tree is 6 (the depth of an empty tree is defined to be -1), then the minimum possible number of nodes in this tree is D(**be careful of depth and height**)

A. 13

B. 17

C. 20

D. 33

最坏都是logn

ch 2 splay tree

splay 的search也是amortized logN

aim: Any M **consecutive tree operations** starting from an **empty tree** take at most O(M log N) time.

只推到树根错误做法：

worse case: $T(N) = O(N^2)$

均摊时间比平均时间大，比最坏时间小

势能函数是后继节点的个数

splay tree 的amortized time complexity：

$$\Phi(T) = \sum_{i \in T} \log S(i) \quad \text{where } S(i) \text{ is the number of descendants of } i \ (i \text{ included}).$$

zig：

$$\begin{aligned} \hat{c}_i &= 1 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\le 1 + R_2(X) - R_1(X) \end{aligned}$$

zig-zig:

$$\begin{aligned} \hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\le 3\left(R_2(X) - R_1(X)\right) \end{aligned}$$

zig-zag:

$$\begin{aligned} \hat{c}_i &= 2 + R_2(X) - R_1(X) \\ &\quad + R_2(P) - R_1(P) \\ &\quad + R_2(G) - R_1(G) \\ &\le 2\left(R_2(X) - R_1(X)\right) \end{aligned}$$

$$\hat{c}_i \le 1 + 3\left(R_2(X) - R_1(X)\right)$$

总分析

【Theorem】 **The amortized time to splay a tree with root T at node X is at most 3(** R(**T) – R (X)) + 1 =** O(logN).

以上其实只是splay的time cost, 但是find ， insert，delete等操作实际上是

均摊分析的最终目的是求actual cost的上界，只不过因为通常情况下amortized cost = actual cost + \fai(n)-\fai(0)，而后半部分大于等于0，所以求amortized 上界也就是求actual cost上界

ch3 Black Red Tree

树的任何一个分支最长的不会大于最小的那个的两倍

■ 红黑树：任何两分枝长度差不超过一倍

1. 每个节点是红色or黑色
2. 根节点黑色
3. 每个叶节点（NIL）黑色
4. 如果一个节点红色，其儿子都是黑色
5. 对每个节点，该节点到子孙节点的所有路径上包含的黑节点相同

插入最多旋转2次，删除最多旋转3次

一个有N个节点的红黑树，树高最多为2lg（N+1）

sizeof(x)>=$2^{bh(x)}$ -1(递推)

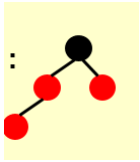
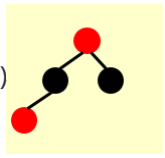
bh(Tree)>h(Tree)/2

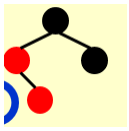
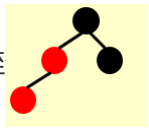
Since for every red node, both of its children must be black, hence on any simple path from *root* to a leaf, at least half the nodes (*root* not included) must be black.

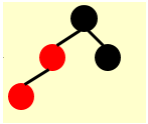
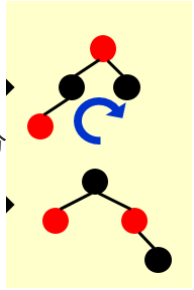
$$Sizeof(root) = N \geq 2^{bh(Tree)} - 1 \geq 2^{h/2} - 1$$

$$\text{i } h \leqslant 2\lg(n+1)。$$

Insert:考虑：父辈祖辈，叔叔红顶点红，叔叔黑顶点黑

1：  （自己爸爸叔叔红， 祖父变红父辈黑）  （顶点不变，祖父辈颜色互换）红色结点个数减一

2：  近亲叔叔和爷黑，我变父亲爸远侄  （顶点不变，爸爸和我互换辈分）红色结点个数不变

3：  祖辈远叔都是黑，我和爸爸升辈分  （顶点变爸，爸爸和爷爷换颜色）

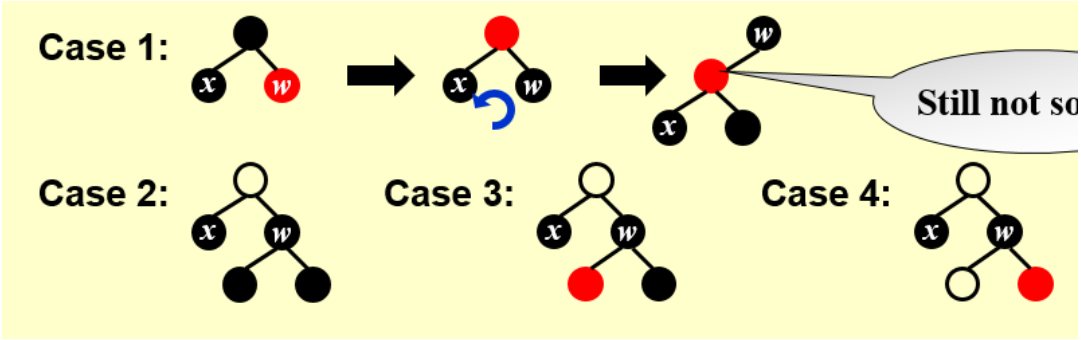
红色节点个数不变

T(h) = O(logN), 调三代四个节点，自己爸爸叔叔爷爷

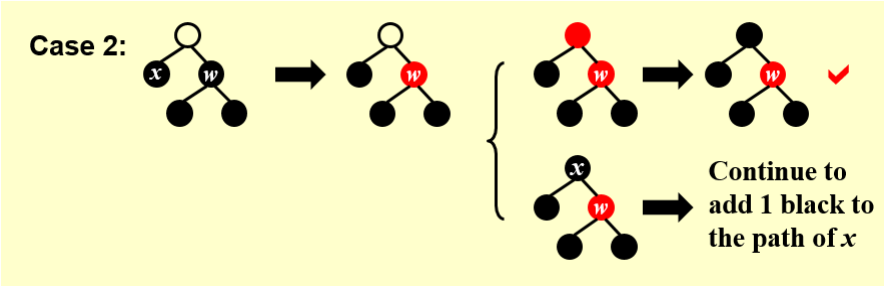
黑色给了树的平衡，红色给了自由度

delete：考虑：父辈我辈侄子辈

delete：在做删除的时候如果删除红节点是ok的

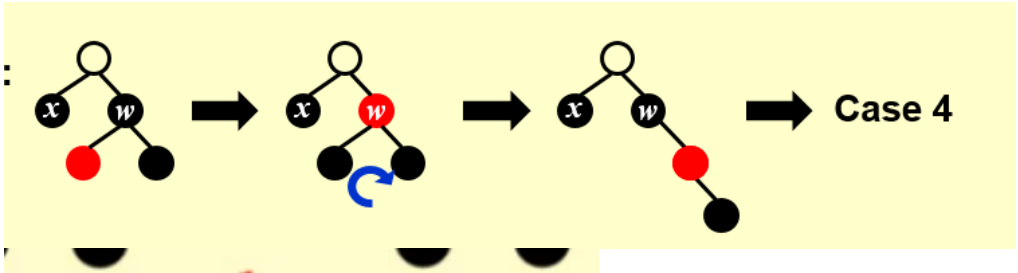


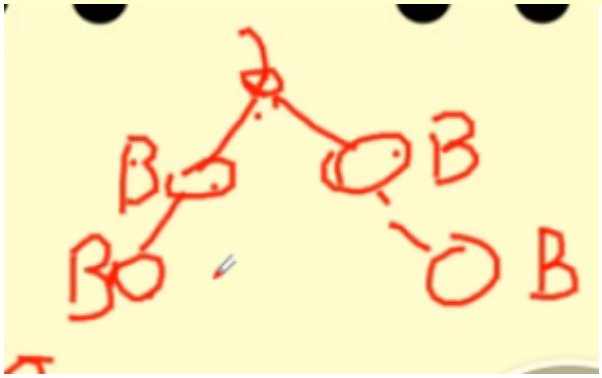
1. 兄弟红， 兄弟最高父变红---->红色节点未变少
2. 兄弟黑，侄子都黑----->把自己和兄弟的黑都给父亲，父亲黑，父亲加一黑，父亲红，父亲变黑

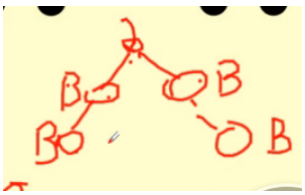


3. 兄弟黑，近侄子红，近侄子变远侄子

   近侄子黑给了兄弟，然后旋转--->红色节点个数未变





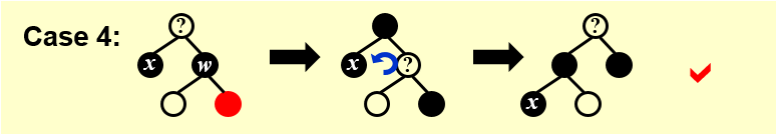


注意观察最终目标，近侄子红大字型，顶点颜色不变其他都黑

4. 兄弟黑，远侄子红

   红色节点少1

   我自己在的那个分支多一个节点，

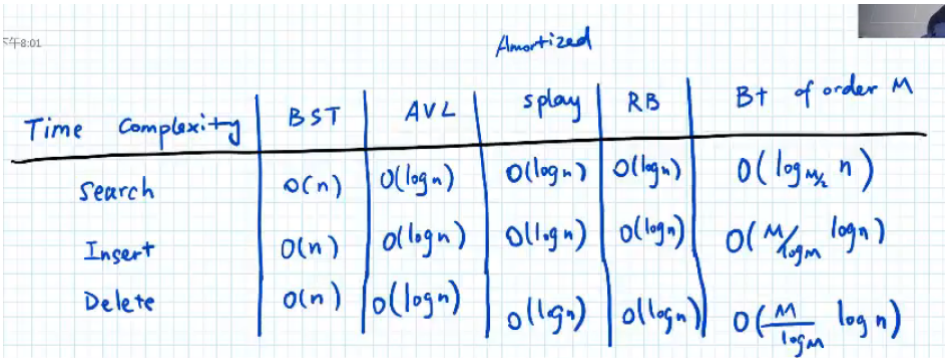


远侄子红huffman型，顶点和近侄子颜色不变其他都黑

如果把红黑树的红节点都挤到父亲节点，就变成一个B树

ch4

Number of *rotations*		
	AVL	**Red-Black Tree**
Insertion	≤ 2	≤ 2
Deletion	$O(\log N)$	≤ 3



B+树的性质

B+树深度：$\mathbf{Depth}(M, N) = O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

B+树查找：$\log_{M/2} N = \log M \log_M N$--->每个节点中找*树高

B+树插入：$M \log_M N = M/\log M \log N$

B+树删除：$M \log_M N = M/\log M \log N$

M不是越大越好

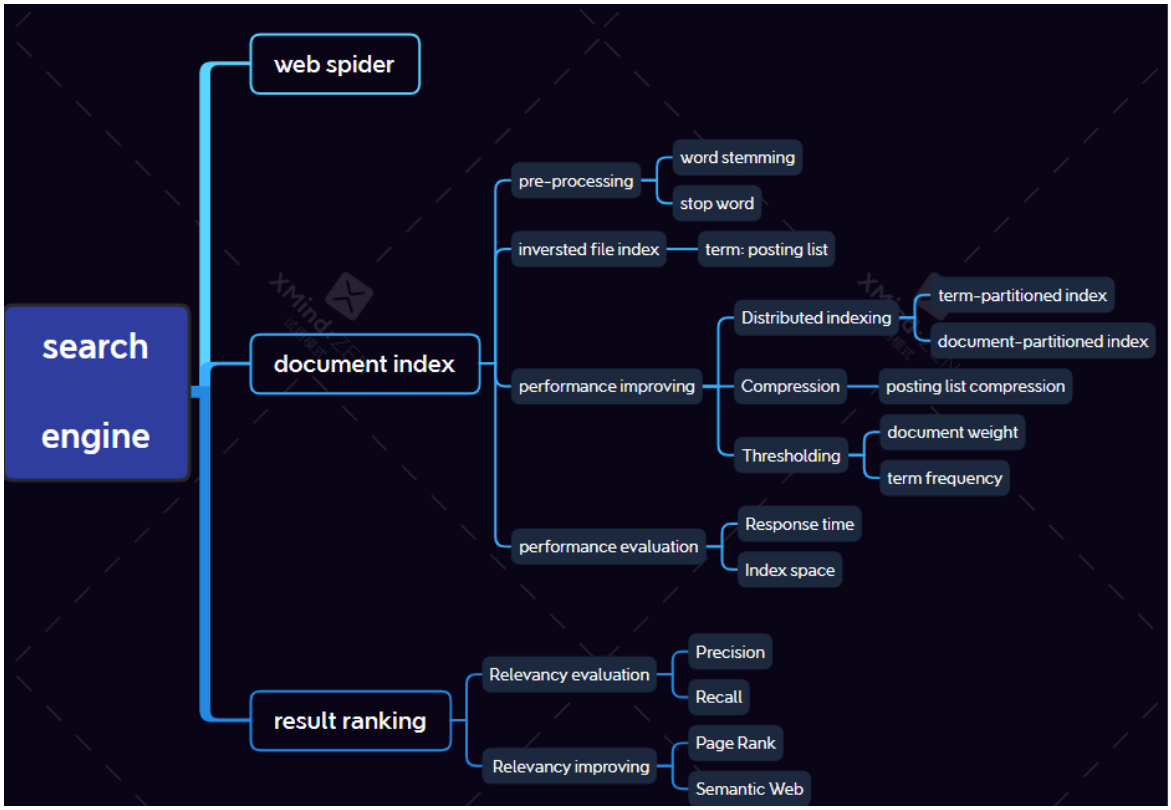
AVL is faster for search

RB is faster for insert and delete

RB 是B树的特殊情况

在n个节点的RBtree中连续m次进行插入，均摊时间复杂度O(m+n)，实际mO(logN)

ch5 Inverted File Index 倒排索引



法1：Term-Document Incidence Matrix too big，用一个矩阵去存每个词在每个网页出现了几次

法2：Compact Version - Inverted File Index

有一个独特的数据结构，每个词对应一个链表，表头是出现了几次，节点中存储对应页码

- Index is a mechanism for locating a given term in a text.
- **Inverted file contains a list of pointers (e.g. the number of a page) to all occurrences of that term in the text.**

优化：

1. find the term
    - word stemming-->词根
    - stop words-->频率太高的词不做索引
2. access the term
    - binary search tree
    - hash table-->一次到位（faster），但是空间要求高，且无法范围查询
3. when don't have enough memory

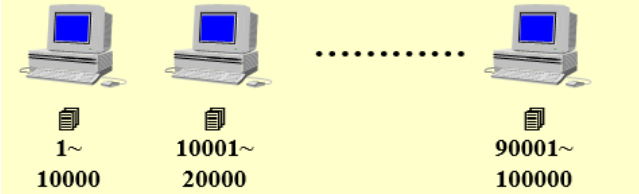
    一个block放不下就放多个block然后再merge，参考external sort

```
BlockCnt = 0;
while ( read a document D ) {
  while ( read a term T in D ) {
    if ( out of memory ) {
      Write BlockIndex[BlockCnt] to disk;
      BlockCnt ++;
      FreeMemory;
    }
    if ( Find( Dictionary, T ) == false )
      Insert( Dictionary, T );
    Get T's posting list;
    Insert a node to T's posting list;
  }
}
for ( i=0; i<BlockCnt; i++ )
  Merge( InvertedIndex, BlockIndex[i] );
```

some techniques to improve performance

1. Distribute indexing
   - Term-partitioned index---->根据关键词来

   - document-partitioned index---->根据序号来



2. compression--->相对位置

   record respective position instead of fact position

3. thresholding

在document的选取上，only retrieve the top x documents where documents are ranked by weight

- Not feasible for Boolean queries
- Can miss some relevant documents due to truncation

在query的选取上，Sort the query terms by their **frequency** in ascending order; search according to only some percentage of the original query terms， 按照频率升序

Measure of search engine

1. how fast does it index-->单位时间里面处理多少文档
2. how fast does it search
3. Expressive of query language

   Ability to express complex information needs

   Speed on complex queries

   对复杂查询的支持

Users' happiness

Data Retrieval Performance Evaluation (after establishing correctness)处理速度-->搜索引擎的性能

> Response time处理反应时间

> Index space空间代价

- Information Retrieval Performance Evaluation-->

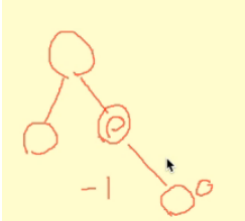
> + How relevant is the answer set?

	Relevant	**Irrelevant**
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

$$\textbf{Precision } P = R_R / (R_R + I_R)$$
$$\textbf{Recall } \quad R = R_R / (R_R + R_N)$$

BFS 最适合有限时间内下载最多的网页，dfs等有时间继续深搜

ch6 leftist heap



上图不是一个左偏树，第三个节点的左儿子的npl=-1，右儿子npl=0

原来的heap: merge 时用O(N)因为buildheap线性复杂度

Leftist heap:为了**加速merge**的过程

property：**unbalanced**

for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

如果一个左倾堆右路径有r个点，至少有2^r -1, 即为有N个点的左倾堆最多有log（N+1）个点在右路径上

insert 和delete都是merge的特例

merge（iteratively)

```
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) //left is null,right must also be null
       H1->left = H2;
    else
    {
       H1->right = Merge(H1->right, H2);
       if(H1->right->Npl > H1->left->Npl)
          Swapchildren(H1);
       H->Npl = H1->right->Npl + 1;//最重要
    }
    return H1;
}
```

merge（recursively)

ch7 skewheap

势能函数是重结点个数

Any M consecutive operations take at most O(M**log N**) time.

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

F It is an open problem to determine precisely the expected right path length of both leftist and skew heaps.

Amortized analysis

if light node is not on the right path, then it doesn't change

右路径一个heavy node一定变成light node

右路径一个light node 可能会变成heavy node

potential function 是可以变为right path上的heavy node

均摊代价的要求是要最大，所以 $$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i \leq 2\,(l_1 + l_2)$$

a tree with k light nodes on its right path has at least 2^k-1 nodes

logn

ch8 Binomial Queue

儿子从大到小 排

FindMin：logN

merge：logN

Insert： average：1

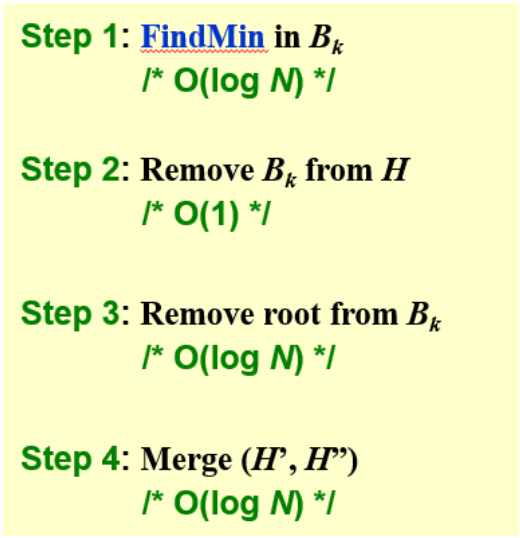
DeleteMin: logN

If the smallest nonexistent binomial tree is Bi **, then**

Tp **= Const · (**i **+ 1).**

Performing N Inserts on an initially empty binomial queue will take O(**N) worst-case time. Hence the average time is constant**.

DeleteMin



树的表示：左儿子右兄弟， 指向最大的儿子

Combine:O（1）

DeleteMin:注意for循环中的DeleteQueue->TheTrees[j]->NextSibling = NULL

代码

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

struct BinNode
{
   ElementType Element;
   Position LeftChild;
   Position NextSibling;
}
struct Collectiion
{
   int CurrentSize;
   BinTree TheTrees[MaxTrees];
}
BinTree CombineTrees(BinTree T1, BinTree T2)
{
   if(T1->Element > T2->Element)
      return CombineTrees(T2,T1);
   T2->Sibling = T1->LeftChild;
   T1->LeftChild = T2;
   return T1;
}
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];
      switch(4*!!Carry + 2*!!T2 + !!T1)
      {
            case'0':
            case'1': break;
            case'2':H1->TheTrees[i] = T2;H2->TheTrees[i] = NULL;break;
            case'3':Carry = CombineTrees(T1,T2); H1->TheTrees[i] = H2-
>TheTrees[i]=NULL;
            case'4':H1->TheTrees[i] = Carry;Carry = NULL;break;
            case'5':Carry = CombineTrees(T1,Cary); H1->TheTrees[i]  =NULL;
            case'6':Carry = CombineTrees(T2,Cary); H2->TheTrees[i]  =NULL;
            case'7':Carry =CombineTrees(T2,Cary); H2->TheTrees[i]  =NULL;
      }
   }
   return H1;
}
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 item
*/
   if ( IsEmpty( H ) )  {  PrintErrorMessage();  return -Infinity; }
   for(i = 0; i < MaxTrees;i++)
      if(H->TheTrees[i]&&H->TheTrees[i]->Element < MinItem)
      {
         MinItem = H->TheTrees[i]->Element;  MinTree = i;
      }
   DeletedTree = H->TheTrees[MinTree];
    H->TheTrees[MinTree] = NULL;
   OldRoot = DeletedTree;
   DeletedTree = DeletedTree->LeftChild;
   free(OldRoot);
   DeletedQueue = Initialize();
   DeletedQueue->CurrentSize = 1<<MinTree-1;
   for(j = MinTree-1;j>=0;j--)
   {
      DeletedQueue->TheTrees[j] = DeletedTree;
      DeletedTree = DeletedTree->NextSibling;
      DeletedQueue->TheTrees[j]->NextSibling = NULL;
   }
   H->CurrentSize -=DeletedQueue->CurrentSize +1;
   H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H" */
   return MinItem;


}
```

时间分析：

势能函数是树的个数

势能函数：number of trees after the ith insertion

每一步的均摊代价：2

Tworst = O(logN), Tamortized = 2

Proof 2: An insertion that costs c units results in a net increase of $2 - c$ trees in the forest.

C_i ::= cost of the ith insertion

Φ_i ::= number of trees *after* the ith insertion ($\Phi_0 = 0$)

$C_i + (\Phi_i - \Phi_{i-1}) = 2$ for all $i = 1, 2, \ldots, N$

Add all these equations up $\Rightarrow \sum_{i=1}^{N} C_i + \Phi_N - \Phi_0 = 2N$

$$\sum_{i=1}^{N} C_i = 2N - \Phi_N \le 2N = O(N)$$

$\blacksquare$

$$T_{worst} = O(\log N), \text{ but } T_{amortized} = 2$$

ch9 backtracking

1 八皇后问题：初始可n!（深度搜索）

nstrains: ① $S_i = \{1,2,3,4,5,6,7,8\}$ for $1 \le i \le 8$

② $x_i \ne x_j$ if $i \ne j$ ③ $(x_i - x_j) / (i - j) \ne \pm 1$

For the problem with n queens, there are $n!$ candidates in the solution space.

八皇后有92个解

四皇后有两种解法：2，4，1，3

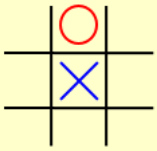
2 The Turnpike Reconstruction Problem

3 Tic-tac-toe

Use an evaluation function to quantify the "goodness" of a position. For example:

$$f(P) = W_{Computer} - W_{Human}$$

where W is the number of potential wins at position P.



$$f(P) = 6 - 4 = 2$$

The human is trying to *minimize* the value of the position P, while the computer is trying to *maximize* it.

α pruning：在确定maximial 层时发生

β pruning：在确定minimial 层时发生

α-β pruning: when both techniques are combined. In practice, it limits the searching to only

nodes, where N is the size of the full game tree.

***At* MAX node:**	***At* MIN node:**
Update α	**Update β**
Prune if $\alpha \geq \beta$	**Prune if $\alpha \geq \beta$**

α-β 一定能得到一个正确的解

- Are we guaranteed a correct solution?

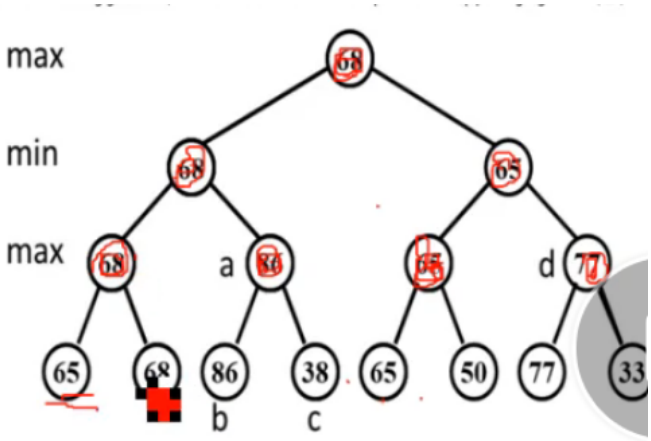
  Yes! Alpha-beta does not actually change the minimax algorithm, except for allowing us to skip some steps of it sometimes

α-β不一定能得到更快的解法

- Are we guaranteed to get to a solution faster?

  No! Even using alpha-beta, we might still have to explore all nodes. The success of alpha-beta $O(\sqrt{N})$ depends on the ordering in which we explore different nodes.

先知道叶子节点，再去根据博弈确定内部节点，后序遍历



ch10 Divide&Conquer

- The maximum subsequence sum – the O($N \log N$) solution
- Tree traversals – O(N)
- Mergesort and quicksort – O($N \log N$)

1 Closest Points Problem

2 substitution method

inductive method

recursion method

b:决定树高

a+b:决定叶子节点数

时间复杂度由每一层divide和combine花的时间和最后一层解决的时间

master method

不覆盖所有情况

法一：

【Master Theorem】Let $a \geq 1$ and $b > 1$ be constants, let $f(N)$ be a function, and let $T(N)$ be defined on the nonnegative integers by the recurrence $T(N) = aT(N/b) + f(N)$. Then:

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 $T(N) = \Theta(N^{\log_b a} \log N)$ regularity condition
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 $T(N) = \Theta(f(N))$

法二：以f(N)为标准，即为我这一层和我儿子这一层花的时间比较

☞ **Master method – *another form***

【Master Theorem】 The recurrence $T(N) = aT(N/b) + f(N)$ can be solved as follows:

1. If $af(N/b) = \kappa f(N)$ for some constant $\kappa < 1$, then $T(N) = \Theta(f(N))$
2. If $af(N/b) = Kf(N)$ for some constant $K > 1$, then $T(N) = \Theta(N^{\log_b a})$
3. If $af(N/b) = f(N)$, then $T(N) = \Theta(f(N)\log_b N)$

〖Example〗 $a = 4, b = 2, f(N) = N\log N$

$af(N/b) = 4(N/2)\log(N/2) = 2N\log N - 2N$?

$f(N) = N\log N \qquad O(N^{\log_b a - \varepsilon}) = O(N^{2-\varepsilon})$

$\rightarrow T = O(N^2)$

法三：

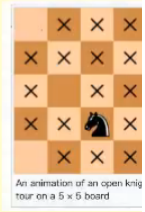
【Theorem】 The solution to the equation

$T(N) = aT(N/b) + \Theta(N^k \log^p N)$,

where $a \geq 1$, $b > 1$, and $p \geq 0$ is

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

- $O(N\log N)$ Mergesort, Closest Pair Points – $T(N) = 2T(N/2) + N$
- $O(N^{1.585})$ Integer multiplication $T(N) = 3T(N/2) + N$
- $O(N^{2.81})$ Matrix multiplication $T(N) = 7T(N/2) + N$
- $O(N)$ Find a median $T(N) = T(N/5) + T(7N/10) + N$
- $O(N^2)$ Knight tour $T(N) = 4T(N/2) + O(1)$



ch11 Dynamic Programming

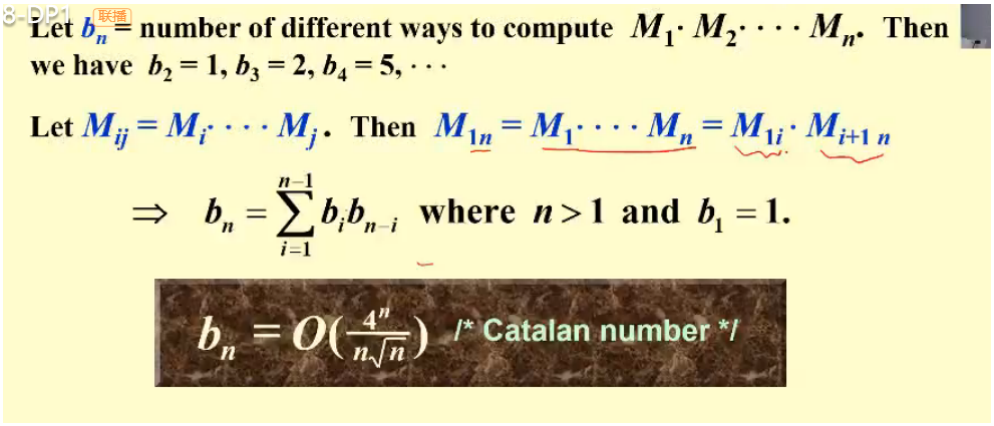
bottom-up

In a nutshell, dynamic programming is recursion without repetition

1 Matrix Multiplications

普通两矩阵相乘：$O(N^3/ mnl)$、

多矩阵相乘：



子问题：Mi....Mj

转移方程：

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

时间复杂度:O（N^3)

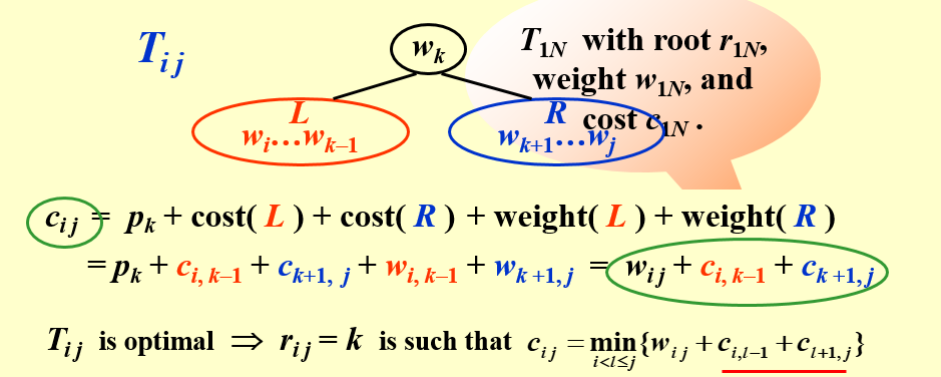
代码

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix(const long r[], int N, TwoDimArray M)
{
   int i,j,k,L;
   long thisM;
   for(i = 1; i <=N; i++) M[i][i] = 0;
   for(k = 1; k<=N; k++)
   {
      for(i = 1; i <= N-K;i++ )
      {
         j = i+K;
         M[i][j] = INFINITY;
         for(L = i; L < j;L++)
         {
            ThisM = M[i][L]+M[L+1][j]+r[i-1]*r[L]*r[j];
            if(ThisM < M[i][j])
            M[i][j] = thisM;
         }
      }
   }
}
```

2 Optimal Binary Search Tree

di是深度

$$T(N)=\sum_{i=1}^{N} p_i \cdot (1+d_i)$$



T（N) = O(N^3)

3 Pair-path

single source algoritm : O（E+NlogN)

法一：**1 Use single-source algorithm for |V| times. T = O(|V|^3)** – works fast on sparse graph.

法二：

定义：

Define

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

转移方程：

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

If there exists a negative-cost cycle, the recursive formula is still valid.

0. F

1. T

```
void AllPairs(TwoDimArray A, ToDimArray D, int N)
{
   int i,j,k;
   for(i=0; i < N; i++)
      for(j = 0; j < N; j++)
         D[i][j] = A[i][j];
   for(k = 0; k < N; k++)
      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 ];
         }
```

```
}
```

时间复杂度：$O(N^3)$,但faster in dense graph

若问题是一个有向无圈图，动态规划可解，如果是个圈，动态规划不可解

ch12 Greedy

不能保证获得最优解，只有当局部最优等价于全局最优才是最优解

optimazation

Optimization Problems:

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**

The Greedy Method:

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**

1 Activity Selection Problem

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

T（N）：O(logN)

2 Huffman Code

设计不等长编码比等长编码有更好的效果，都在叶节点上

ch13 NP

1. If we want to prove that a problem X is NP-Hard, we take a known NP-Hard problem Y and reduce Y to X in polynomial time.
2. The first problem that was proved to be NP-complete was the circuit satisfiability problem.
3. NP-complete is a subset of NP-Hard

欧拉回路问题：edge,P问题

哈密尔顿问题：vertices， 不是P问题

single source最短： P问题

single source最长：非P问题

Hilbert：可判定问题

哥德尔不完备定理：有些问题证不出来

Halting problem——不可判定问题

不存在一种算法能解决这个问题

计算模型

递归函数法

拉姆达演算

图灵机模型——自己定义一个计算过程：

有计算的存储单元，有磁头可以左右移动

无穷带（符号集合）+磁头（读，写，左移，右移）+有穷控制器

初始状态->判断适用规则->执行后的状态

确定性状态图灵机和非确定性状态图灵机

1Deterministic Turning Machine

A Deterministic Turing Machine executes one instruction at each point in time. Then depending on the instruction, it goes to the next unique instruction.

确定状态图灵机->在每个点都有确定的规则，更新状态达到下一个状态。

确定状态图灵机：一个问题在多项式时间内可解

2 Nondeterministic Turning Machine

A Nondeterministic Turing Machine is free to choose its next step from a finite set. And if one of these steps leads to a solution, it will always choose the correct one

不确定状态图灵机-->判定问题

不确定图灵机上多项式能解的问题是NP问题

eg: 哈密尔顿回路问题, 随便猜，即使猜测2^N 次，但每次是多项式时间猜测的，所以也是多项式，always can find the correct answer

一个图是否不存在哈密尔顿问题是非NP问题

3 NPC问题

any problem in NP can be polyminial reduced to NPC

旅行商问题没有多项式时间的算法

给定一个模型M，一个问题X，那么M需要多长时间来解决X?

如何分类complexity class

不同complexity class之间的关系?

decision problem：判定问题

如果能解决优化问题，那么判定问题一定也能解决

language 和problem 的关系

a problem belongs to a complexity class

an algorithm has a time complexity

random access model

讲一个算法复杂度：用random access model

将一个问题属于哪一类：用图灵机

确定性图灵机：若x属于L，M停在accept状态，若x不属于L，M停在reject状态

非确定性图灵机多一个witness

Language包含strings, ana algorithm accept a string if

A language is decided by an algorithm if all the strings in this language is accepted by algorithm A

to accept a language, an algorithm only needs to concern about strings in L, but to decide a lanuage, must ensure that all strings that are not in Language are rejected

co-NP

L属于NP，L的补集也属于NP

reduction

if(A能够多项式规约到B，A<=pB)，B至少和A一样难

多项式时间图灵规约(cook reduction)

A不断调用B，若B是多多项式时间可解，那么A也是，而且B的答案和A相同

Karp 归约-->一定是判定问题

f是在多项式时间构造的

A language L1 **is polynomial-time reducible to a language L**2 **(L**1 **≤P** L**2) if there exists a polynomial-time computable function f : {0, 1}*** → **{0,1}* such that for all x {0,** 1}*, **x** 属于 **L**1 iff **f (**x**)** 属于 **L**2.

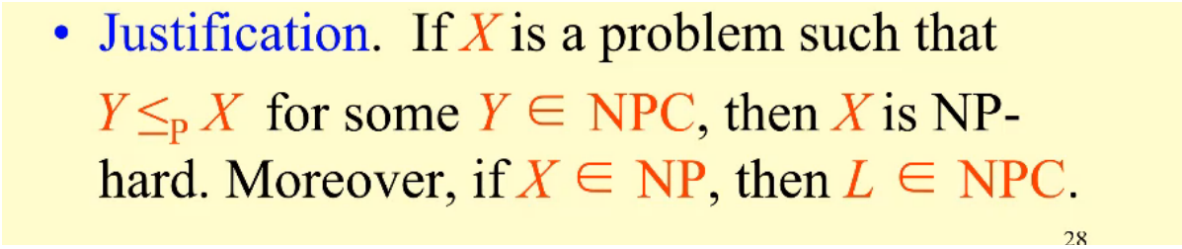
Karp reduction只调用b一次，cook调用多次

NP hard

To prove that problem A is NP-hard, reduce a known NP-hard problem to A

NP hard 和NPC的区别：NPC需要证明 所有的NP问题多项式归约到这个问题而且这个问题属于NP

NP hard问题只需要所有的NP问题小于等于这个问题即可



3-SAT---NPC

3-SAT 是NPC问题，2-SAT是P问题

circult SAT是NPC问题‘

clique----NPC

给一个无向图G和整数k，G有一个至少为K个顶点的子图

最大团问题，原来为optimization 问题，要先转化为decision问题with 一个k

要证明clique问题是一个NPC问题

首先证明是一个NP问题，其次证明3-SAT问题可以规约到clique问题

顶点覆盖问题 ----NPC

选择k个顶点是否可以把图上所有的边都覆盖住

首先2^n 一定可以

如验何证vertex cover是NPc

clique 和vertex cover问题的转化：补图

支配集----NPC

vertex cover可以归约到支配集

ch14 Approximation

近似算法一般要求running time是多项式的

在进行近似比计算时，

alg/opt-->如果用最优解

最优解下界：

bin packing :所有物品总和

TSP：minimal spanning tree

vertex cover：number of mutual disjoint edge--->相互之间没有边的端点

knapsack：

scheduling：

maximal cut：所有边权相加

通常情况下e和时间反比

放弃寻找最优解，在多项式时间内找到次优解，一般近似解deal with 最优化问题而不是判定问题

a polynomial-time approximation scheme (PTAS) if for any fixed e > 0, the scheme runs in time polynomial in the size n of its input instance.多项式近似算法 $O(n^{2/\varepsilon})$

fully polynomial-time approximation scheme(FPTAS): $O((1/\varepsilon)^2 n^3)$ (1/e和n都是多项式级的)

1 Bin Packing problem——NPhard

用尽量少的箱子

Next Fit方法-->2

```
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 */
}
```

最坏情况做到2M-1个箱子

if the next fit algorithm uses 2M/2M+1个箱子，那么最优解至少是M+1个箱子

prove:任何两个相邻的箱子所装的东西之和>1

S(1)+S(2)>1,S(3)+S(4)>1.....

S(1)+...+S(2M)>2M/2=M

First Fit方法---1.7

从头开始一个个看能不能装下，普通情况下需要O（N^2), 数据结构好的话用NlogN

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 */
}
```

Can be implemented in O(N log N)

Best Fit---<1.7

从头找到最好的那个箱子，<=1.7M,经过巧妙设计达到NlogN

Online alogrithm

可能根本不存在最优解**Theorem**There are inputs that force any on-line bin-packing algorithm to use at least**5/3 the optimal number of bins.**

Offline--->最好11/9

排序，从大到小然后采用first/best fit

11/9 M+6/9

The Knapsack Problem — fractional version

用greedy可以解决，p问题

The Knapsack Problem — 01 version （NPhard 问题）近似比可以做到2

完全按照贪心无法做到最优解，但近似比可以做到2

pmax<=Poptimal<=Pmax

pmax<=Pgreedy

Poptimal<pgreedy+pmax

Poptimal/Pgreedy = 1+pmax/Pgreedy<=2

用动态规划

take i Wi,p = wi+Wi-1,p-pi

skip i: Wi,p = Wi-1,p

i = 1,2...n

p = 1,...npmax

O(n^2pmax)

K-center 问题除非p = np， 否则rou不会小于2,即为rou>=2

任意的距离和metric不一样

1. 距离需要满足的条件：

   distance(self) = 0;

   对称性

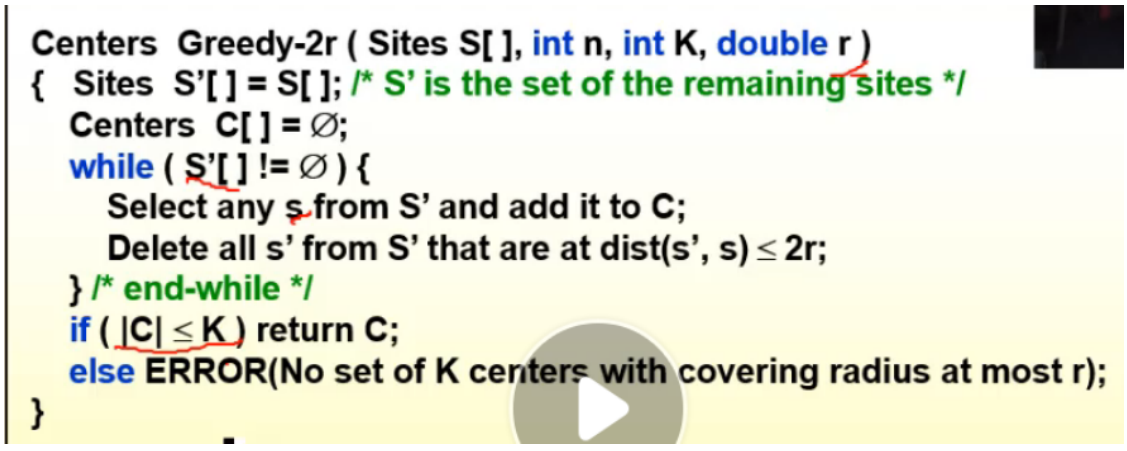
   三角不等式

2. 一个贪心算法：**Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.**

3. 另一种贪心算法

   假设知道一个最优解，那么我们去找近似解

   半径是最优解的两倍



   按照这个算法，如果r>=2r*,可以在k次停下来，如果k次不够说明r选的太小，所以这个算法可以用来卡r

   如何来选r呢？--->用research

   2-approximation

除非p = np， 否则rou不会小于2--->转换成dominateset问题-->顶点覆盖问题在P有解

算法问题的考虑

1. 最优解
2. 效率
3. all instances

同时满足这三个问题是困难的， 1+2：exactly and fast for some instances, 2+3:近似算法 1+3: exact for all instances

ch15 local search

有的问题连近似比都找不到或者近似比非常差

全局最优解找不到-->找局部最优解

凸函数：一般局部最优能找到全局最优

local：定义邻居，local optimal is a best solution in a neighborhood

local search 可以证明近似比但很难

梯度下降首先需要找到一个可行解，然后再邻居中找更好的解，将更好的邻居作为中心找他的邻居，找不到就放弃了

1 vertex cover--->NPComplete

顶点覆盖问题一般分为optimazation 问题和decidable problem

local search比较关键的是去找反例，local到某个地方绝对不可以找到近似解了

2The Metropolis Algorithm

Simulated Annealing

1. 不是从邻居里挑最好的而是随机挑一个
2. 如果有改进就更新，没改进就一定概率保留坏情况，想办法跳出局部最优

With a probability $e^{-\Delta \text{cost}/(kT)}$, let S = S';
else break;

如果没有改进那么▵cost >=0, T越大，P越大->上坡， T越小， P越小->接近梯度下降

2 Hopfield Neural Networks--->NPC

若边权重小于0，那么两端点同号，边权重大于0， 那么两端点不同号

对于某些图来说，可能并没有满足条件的一个配置，比如三边都为正的三角形

定义good = we* su * sv<0

对于每个端点的每条边计算total 的good值 weight of good >= weight of bad->satisfied

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \le 0$$

对于一个不满意的node，flip他的状态

State flipping algorithm

☞ State-flipping Algorithm

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

State flipping algorithm**肯定可以停下来

因为 $\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:e=(u,v)\in E \\ e \text{ is good}}} |w_e|$$

Φ是递增的，且Φ有上界，所以肯定可以停下来

近似算法分析：

maximizeΦ

feasible solution：a configuration

添加邻居：**S' can be obtained from S by flipping a single state**

尚未证明是P

Any local maximum in the state-flipping algorithm to maximize Φ is a stable configuration.

W不知道是多少--input size

3 maximal cut---NPhard--- a special case of hopfield neural network--->有比较好的近似解，但除非P=NP,否则不会小于17/16

Maximal flow， minimal cut--P

total weight of edges crossing the cut is maximial

☞ Problem: To *maximize* $w(A, B)$.

☞ Feasible solution set $\mathcal{FS}$: any partition (A, B)

☞ S ~ S': S' can be obtained from S by moving one node from A to B, or one from B to A.

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

这个算法也一定可以停止下来,不是多项式可解，非多项式可解也可以找到近似比

近似比 -->权重至少是近似解的一半

Proof: Since (A, B) is a local optimal partition, for any $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

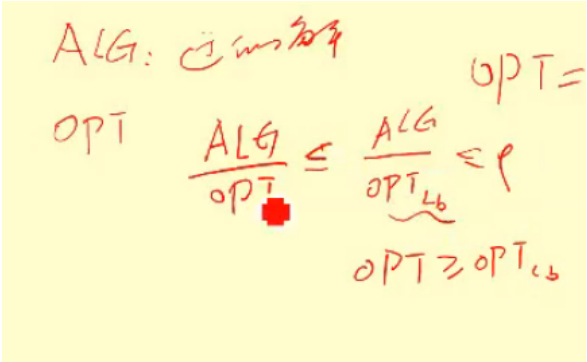
Summing up for *all* $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

在进行localsearch的时候，很难确定最优解-->找一个上界或者下界



右边放缩--->所有边的权重之和

近似比

2--->1.1382--->no smaller than17/16

并不一定是多项式内可解，因为跳的次数太多了

可以限制他一个条件，

Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least

$$\frac{2\varepsilon}{|V|} w(A,B)$$

Claim: Upon termination, the big-improvement-flip algorithm returns a cut (A, B) so that

$$(2+\varepsilon)\, w(A, B) \geq w(A^*, B^*)$$

Claim: The big-improvement-flip algorithm terminates after at most $O(n/\varepsilon \log W)$ flips.

2e/|V|, 近似比：2+e, terminates after at most O（n/elogW)

根据时间简单描述证明：
1. 每次flip至少增加(1+epsilon/n)倍，其实是（1+2*epsilon/n)倍
2. n/epsilon次flip之后，总增长至少是2倍。利用(1+1/x)^x >= 2, 如果x>=1
3. 总量不超过W，而cut翻倍的次数不能超过logW

在定义local的范围时，

解决方案的邻域应该足够丰富，以免我们倾向于陷入不良的局部最优； 但解决方案的邻域不应太大，因为我们希望能够有效地搜索邻居集中可能的本地移动。

把single flip 改成 k flip，每次改变k 个item的状态， 但需要$O(n^k)$search time

K-L heuristic---->$O(N^2)$

step1:从一个点出发，找他的neighbour，找最好的make 1-flip as good as we can--O(N)

step2:把移过去的顶点mark，以后不要再移动这个点了

不一定是在local里面找到的最优解

不一样的地方：

1 找的是neighbour里面的最好的，即使不比当前状态好，

2 把移过去的顶点mark，以后不要再移动这个点了，时间复杂度是$O(^N)$

旅行商问题：从一个点出发，覆盖所有点，回到这个点，生成树近似比<=2

ch17 Random search

想让算法有一定的随机性，针对一个确定的算法，根据不同的随机数，有不同的输出

对一个确定过程的一个或几个部分进行随机化，不一定是全部随机化

目标1：随机算法为了把最坏的时间复杂度降低，但往往问题的输入不具有随机性的特征，把输入数据进行随机处理，使得最坏情况出现的概率降到最低， ()

目标2：通过随机化使得问题求解的效率提高

两种情况：

随机算法不能保证一定正确，是以很高的概率保证正确，随机次数越多，准确率越高

保证一定正确，但效率要高

1 hiring problem

可能会出错，找一个high probility 得到正确答案

成本：面试的成本+雇佣的成本

Assume M people are hired.

Total Cost: $O(NC_i + MC_h)$

目标：最小化cost

naive solution

定义一个best， 如果当前大于best，雇佣当前，更新best-->容易被tricked

$$O(NC_h)$$

假定：**any of first i candidates is equally likely to be best-qualified so far**

Xi = 0, i不被雇佣，Xi=1，i被雇佣， P（Xi=1） = 1/ i

期望：ΣE(Xi) = Σ1/i = ln N

$$\Rightarrow O(C_h \ln N + NC_i)$$

改进：在进行for之前随机化输入

no longer need to assume that candidates are presented in random order

2 如何产生随机序列

N个P，P是随机生成数%N^3 + 1;

排序p，将A按照对应p排序，即为A的一个随机序列

```
void PermuteBySorting ( ElemType A[ ], int N )
{
   for ( i=1; i<=N; i++ )
      A[i].P = 1 + rand()%(N^3);
      /* makes it more likely that all priorities are unique */
   Sort A, using P as the sort keys;
}
```

3 online hiring-hire only once

先看k个，从k个里面选一个作为参照，继续面试，一旦找到一个人比参照好久录取他，后面的都不看了

所以k应该取多少呢

第i个人正好被选中需要满足两个条件：

1. k+1--->i-1中没有比参照好的----->前i-1个人里面最好的落在k个人里面---->k/i-1
2. i是N里面最好的---->1/N

1，2是独立事件，概率相乘

P(i是最好的) = 1/N*k/i-1

要求k使得期望最大，那就Σk+1 -->N

$$\Pr[S_i] = \Pr[A \cap B] = \underset{=1/N}{\Pr[A]} \cdot \underset{=k/(i-1)}{\Pr[B]} = \frac{k}{N(i-1)}$$

$$\Pr[S] = \sum_{i=k+1}^{N} \Pr[S_i] = \sum_{i=k+1}^{N} \frac{k}{N(i-1)} = \frac{k}{N} \sum_{i=k}^{N-1} \frac{1}{i}$$

$$\frac{k}{N} \ln\left(\frac{N}{k}\right) \le \Pr[S] \le \frac{k}{N} \ln\left(\frac{N-1}{k-1}\right)$$

使得Pr[S]最大，下界最大，求导可得

☞ What is the best value of k to maximize the above probability?

$$\frac{d}{dk}\left[\frac{k}{N}\ln\left(\frac{N}{k}\right)\right] = \frac{1}{N}(\ln N - \ln k - 1) = 0 \Rightarrow \ln k = \ln N - 1 \Rightarrow k = \frac{N}{e}$$

4 quicksort

结果是确定的，运行时间是一个随机变量

总是给正确答案，running time是random

☞ $\Theta(N^2)$ worst-case running time

☞ $\Theta(N \log N)$ average case running time, *assuming every input permutation is equally likely*

目的：尽量避免最坏情况--->用随机策略避免不均匀的情况

所以规定1/4, 3/4，没有任何一堆元素总个数小于1/4或者大于3/4

The expected number of iterations needed until we find a central splitter is at most 2.

期望挑的次数为2

每次选取有1/2的概率成功

则最坏情况就是1/4，3/4,时间复杂度为O(NlogN)

ch18 并行算法

并行分为机器并行和算法并行

算法并行：

1 Parallel Random Access Model

读/写是unit time pardo

解决冲突办法：EREW，CREW，CRCW

eg

B(h,i) = B(h-1, 2i-1)+B(h-1, 2i)

idle -->仍然开着相当于仍然在工作

```
PRAM model
{
   for Pi, 1<=i<=n pardo
   B(0,i) = A(i)              O(1)
   for h = 1 to logn pardo
      if i <= n/(2^h)         O(logn)
         B(h,i) = B(h-1,2i-1)+B(h-1,2i)
      else stay idle
   for i =1:output B(logn,1) O(1)
   for i >1 stay idle
}
```

T（N) = O(logN)

W(N) = O(N)

所有机器一直在运行

2 Work Depth

让空闲的就不要干了

```
WD model
{
   for Pi, 1<=i<=n pardo
   B(0,i) = A(i)
   for h = 1 to logn
      for Pi, 1<=i<=n/(2^h) pardo
         B(h,i) = B(h-1, 2i-1)+B(h-1, 2i)
   for i = 1 pardo
   output B(logn, 1)
}
```

机器的workload不一样

3 评价标准

1. workload--->total number of operations W(n)
2. worst case time T(n)
3. 并行度P(n) --> W(n)/T(n)

4. 若用少于所需处理器的个数的p台处理器，用时会是多少？ W(n)/p, any number of p <= W(n)/T(n)
5. 一般用W(n)/p + T(n)来衡量， 如果p>W(n)/T(n), 那么T(n)占主导，否则W(n)/p占主导

【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.**

4 prefix sum

```
prefix sum
{
   for Pi, 1<=i<=n pardo
   B(0,i) = A(i)
   for h = 1 to logn
      for Pi, 1<=i<=n/(2^h) pardo
         B(h,i) = B(h-1, 2i-1)+B(h-1, 2i)
   for h = logn 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%2 !=0, i!=1
         C(h,i) :=C(h + 1, (i - 1)/2) + B(h, i)
      for Pi , 1 <= i <= n pardo
       Output C(0, i)
}
```

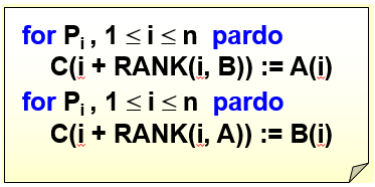
T(n) = O(logn)

W(n) = O(n)

Merging--未并行的时候是O（N)

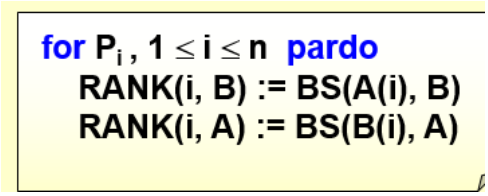
找出每个元素在对方所处位置，即知道对方有几个比我小，计算rank

O(1)time, O(m+n)work



计算rank有两种方法：

Binary Search：T(n) = O(logN) W(n) = O(nlogn)



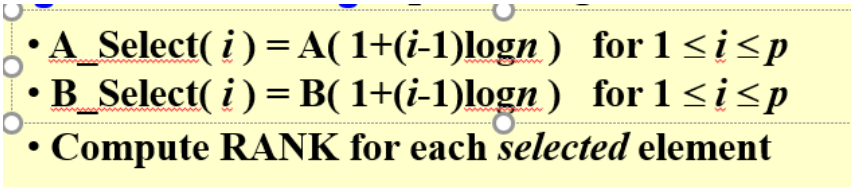
Serial： T(n) = O(n) W(n) = O(n)

```
i = j = 0;
while ( i ≤ n || j ≤ m ) {
   if ( A(i+1) < B(j+1) )
      RANK(++i, B) = j;
   else RANK(++j, A) = i;
}
```

$$T(n) = W(n) = O(n+m)$$

另一种更好的方法：paralell ranking

1. partitioning:分为p个小块， p = n/logn



   T = O(logn) W = O(plogn) = O(n)

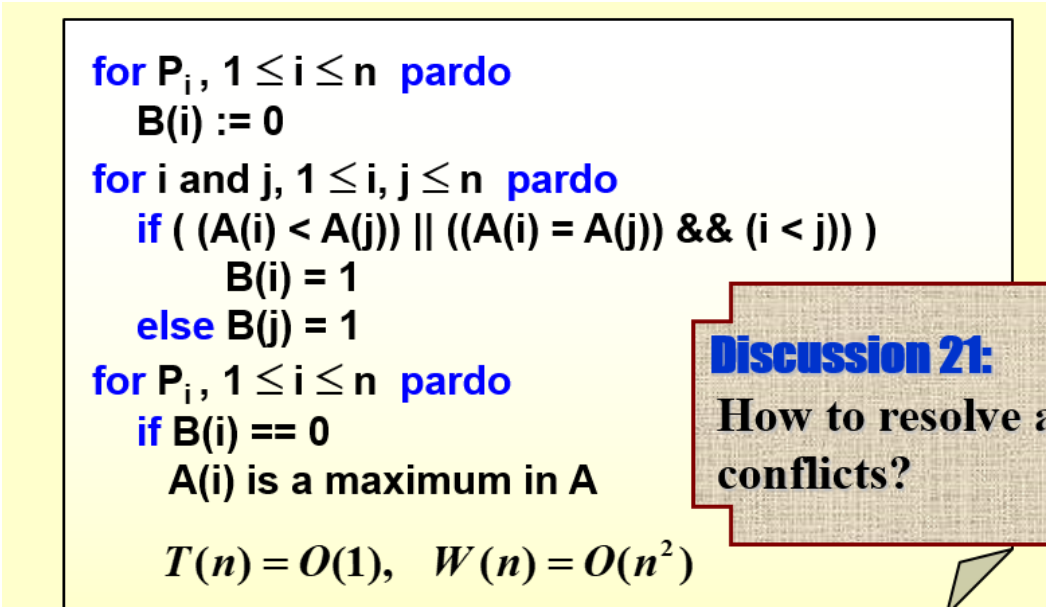
2. 最多有2p个size为O(logn)的问题

   最终T(n) = O(logn) W(n) = O(n)

Maximum finding

如果用原来求sum的方法，可以有T (n) = logn, W(n) = n的算法

Compare all pairs算法T(n)=1, W（n) = n^2，两两比较



Doubly-logarithmic Paradigm

1. 每组√N

   分为根号N一组，那么一共分为p=loglogN组，每个组内分别去求最大值

   那么根据线性的算法，每组内W(n) = T(n) = √N

   那么总的T(N) = O(loglogn),W(n) = nloglogn

   然后再求这p个里面的最大值，就用之前的compare all pairs算法

   T(n) = O(1), W(n) = (n)

   所以总的算法

T(n) =T（√n)+c1, W(n) = √n(W(√n))+n

T(n) = O(loglogn)， W(n) = (nloglogn)

2. 每组h = loglogn

每组里面 T(n) = O(loglogn)，W(n) = O(loglogn)

然后对于 这些最大值，在进行partition

T(n) = O(h+loglog(n/h)) T(n) = O(loglogn)

W(n) = O(n/h* O(h)+(n/h)*loglog(n/h)) = O(n)

Radom Sampling

以很高的概率找到最大值

从n个元素中抽出$n^{7/8}$, 然后分组， 每组$n^{1/8}$, 则一共有$n^{3/4}$ 组，

每组 中用 all -pairs的方法找出最大值，每组的T(n) = 1, W(n) = $n^{1/4}$

那么这些组一共的时间是T(n) = O(1), W(n) =$n^{3/4}$ W(n) = O(n)

这样就找到了$n^{3/4}$个最大值

然后再对这$n^{3/4}$ 个最大值进行partion， 每组$n^{1/4}$ 个， 则一共$n^{1/2}$ 组

组内用all-pairs的方式，T(n) = 1, W(n) = $n^{1/2}$

一共T(n) = 1, W(n) = O(n)

这样就找到了$n^{7/8}$ 个值中的最大值

Summary of Maximum Finding:

0. Replace "+" by "max" in the summation algorithm

T(n)=O(logn), W(n)=O(n)

1. Compare all pairs（find B(i) == 0）

T(n)=O(1), W(n)=O(n*n)

2. Doubly-logarithmic Paradigm: Partition by $n^{1/2}$:

T(n)=O(loglogn), W(n)=O(nloglogn),

3. Partition by h = log log n:

T(n)=O(loglogn), W(n)=O(n)

4. Random Sampling:

$n^{7/8}=n^{3/4}*n^{1/8}$ $n^{3/4}=n^{1/2}*n^{1/4}$

$M(n^{7/8}) \sim T = O(1), \quad W = O(n)$

ch 19 external sorting

数据量足够大，没办法放在内存里面

在硬盘上访问：（过程相当缓慢）

1.**find the track;** -

2.**find the sector;**

find a[i] and transmit

硬盘不适合随机访问，适合顺序访问，merge sort比较适合

mergesort可以至少使用3个tape完成排序

如果只有一个tape，那么每次访问操作都需要Ω(N)

1 最普通的 方法

$$1+\lceil \log_2(N/M) \rceil$$

2 优化

seek time---> number of passes---->**reduction of passes**

time to read one block of records----->**run generation**

time to internally sorty M records---->***Ø Run merging****

time to merge N records from input buffers to output buffers--->**Buffer handling for parallel operation**

- **Seek time** — $O($ *number of passes* $)$
- **Time to read or write one block of records**
- **Time to internally sort** M **records**
- **Time to merge** N **records from input buffers to the output buffer**

1 reduce passes

k-way merge instead of 2-way merge

require k tapes,

$$Number\ of\ passes = 1+\lceil \log_k(N/M) \rceil$$

如果用3tape进行2-way merge

evenly：less merge， more copies

unevenly：no copy， more merge

phibonacii

Polyphase Merge k + 1 tapes only

2 merge 过程平行化

2个input， 2个 output

In general, for a k-way merge we need **2k input buffers and 2 output buffers for parallel operations.**

2k+2是固定的，k越大，block size越小，访问次数越多，所以不是k越大越好

k的最佳值取决于磁盘参数和可用于缓冲区的内部内存量。

3 generate A longer run

replacement

4 minimize the merge time

huffman tree

Total merge time = O (the weighted external path length)

ch20 Clique问题

最大团：在无向图中找到一个点数最多的完全图

ch 21 Fibonacci heap

fibonacci heap :

1. Composed of min-heap ordered trees
2. Mergeable
3. Do not require subtrees to be in specific order.

structure of the node:

left, right, child, parent, key ,degree,mark

structure of the tree

•The root of every tree is stored in a **double-linked** list.

•The pointer “H.min” always points to the node containing the smallest key.

•For every node, **the children of which is stored in a double-linked list** as well.

•The operations on Fibonacci Heap are** based on operations on double-linked lists**.

operations:

add:O(1)

主要是指针的连接和变换

remove:O(1)

主要是指针的连接和变换

Push

Pop

Decreaseky

Complexity

insert:O(1)

merge two heaps:O(1)

decreaseKey:O(1)

actual:O(c)

before: t(H) + 2m(H)

after:t(H)+c+2(m(H)-c+2)

t(H)+c+2(m(H)-c+2)-(t(H) + 2m(H))+O(c) = O(c)

findmin:O(1)

deleteMin:O(logN)

actual cost:O(logN)+t(H)

before is t(H) + 2m(H)

after is logn+1+2m(H)

logn+1+2m(H)-(t(H) + 2m(H))+O(logN)+t(H) = O(logN)

deletNode:O(logN)

The upper bound on the degree of any node of an n-node Fibonacci heap is O(logn)

t(H)： the number of trees in the root list of H

m(H)： the number of marked nodes in H

Φ(H) = t(H)+2m(H).

For every node X in Fibonacci Heap, assume thai c_i is the i^th youngest child, then Rank(c_i)≥i -2

Given that F_0=0, F_1=1,F(i-1)+F*(i-2)(i>2) a tree with rank R has at least F*(R+1)descendants (including the root)

The rank of any node in Fibonacci Heap is O(log N)

ch22 Skip list

时间复杂度：

插入: O(logn)

删除:O(logn)

查找：O(logn)

空间复杂度：O(n)

问题集锦

1.

Recall the amortized analysis for Splay Tree and Leftist Heap, from which we can conclude that the amortized cost (time) is never less than the average cost (time). (2分)

Tworst>=Tamortized>=Taverage---->bound,这里讲的是每个操作

2.

Which of the following is NOT a step in the process of building an inverted file index? (1分

A. Read in strings and parse to get words
B. Use stemming and stop words filter to obtain terms
C. Check dictionary with each term: if it is not in, insert it into the dictionary
D. Get the posting list for each term and calculate the precision

3. greedy可以不管local

Greedy method is a special case of local search. (1分)

○ T ◉ F

4. 如果装箱问题扩大容量，箱子不一定变少

5. next fit 近似比2，

The potential of a skew heap is defined to be the total number of right heavy nodes. The weight of a node, $w(x)$, is defined to be the number of descendants of x (including x). A non-root node is said to be **heavy** if its weight is greater than half the weight of its parent.

Lemma 1: At most one child is heavy, of all children of any node.

Lemma 2: On any path from node x down to a descendant y, there are at most $\lfloor log_2 \frac{w(x)}{w(y)} \rfloor$ light nodes, excluding x. In particular, any path in an n-node tree contains at most $\lfloor log_2 n \rfloor$ light nodes.

Define the following functions:

- *makeheap* : create a new, empty heap
- *findmin* : return the minimum key in the heap
- *insert* : insert a key in the heap
- *deletemin* : delete the minimum key from the heap
- *merge* : merge two heaps

Then for any n-node skew heaps, which of the following is **FALSE**? (3分)

○ A. The amortized time of a merge operation is $O(logn)$
◉ B. The amortized time of a makeheap operation is $O(1)$
○ C. The amortized time of a findmin is $O(1)$
○ D. The amortized time of a deletemin operation is $O(1)$

1. 哈密尔顿回路问题：NPC问题

2. 证明：NP 中的任何语言都可以用一个运行时间为 $2^{O(n^k)}$（其中 k 为常数）的算法来加以判定。

3. NP≠co-NP，则 P≠NP

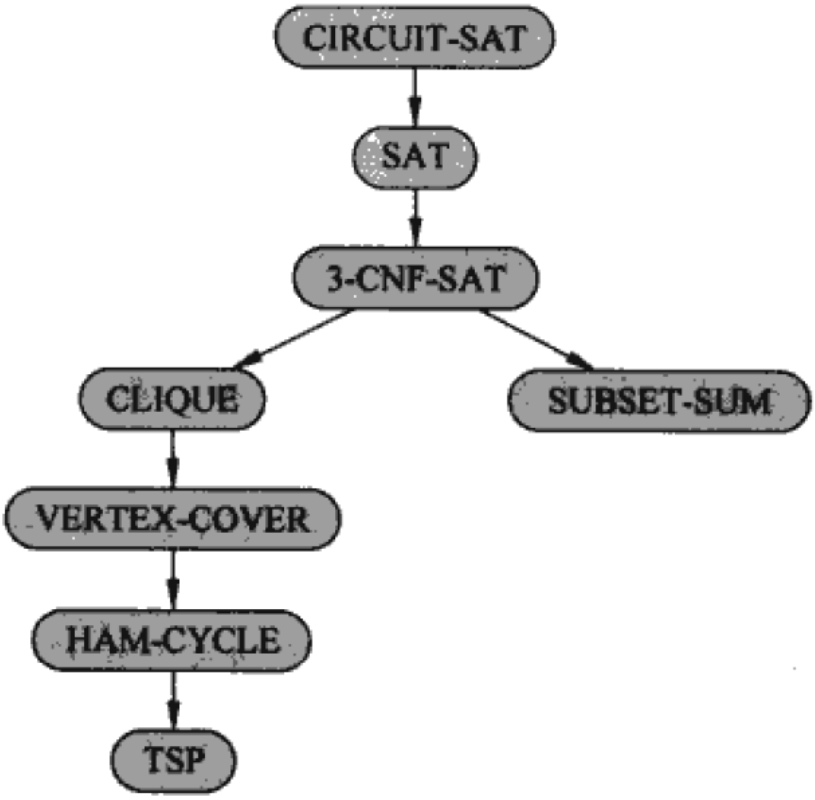
4. 图中的哈密顿路径是一种简单路径，它经过图中每个顶点一次。证明：语言 HAM-PATH＝{⟨G，u，v⟩：图 G 中存在一条从 u 到 v 的哈密顿路径}属于 NP。
不成回路也是np， 有向无回路图中p可解

5. $L \leqslant_P \bar{L}$，当且仅当 $\bar{L} \leqslant_P L$。

6. L 对 NP 是完全的，当且仅当 $\bar{L}$ 对 co-NP 是完全的。

7. SAT--->NPC

8. 越下面越难



9. clique-->NPC:3-SAT-->Clique
10. vertex cover NPC: clique->vertex cover
11. ham-cycle: vertex cover->Ham-cycle
12. TSP

    tsp分为cost满足三角不等式的tsp和一般tsp
13. subset-sum NPC:3SAT->subset-sum
14. 子图同构问题

    子图同构问题(subgraph-isomorphism problem)取两个图 G_1 和 G_2，要回答 G_1 是否与 G_2 的一个子图同构这一问题。证明：子图同构问题是 NP 完全的。
15. set-partion问题NPC
16. 在二分图和度都为2 的图中，独立集问题是p
17. Bonnie和Clyde问题

    a)共有 n 个硬币，但只有两种不同的面值：一些面值 x 美元，一些面值 y 美元。他俩希望平分掉这笔钱。

    b)共有 n 个硬币，它们有着任意数量的不同面值，但每一种面值都是 2 的非负整数次幂，亦即，可能的面值为 1 美元、2 美元、4 美元，等等。他俩希望平分掉这笔钱。

    c)共有 n 张支票，十分巧合的是，这些支票恰好是支付给“Bonnie 和 Clyde”的。他俩希望平分掉这些支票，从而可以分得同样数目的钱。

    d)与 c)一样，共有 n 张支票，但这一次，他俩愿意接受这样的一种支票分配方案，即两人所分得的钱数差距不大于 100 美元。

    a,d是p问题 b,c不p
18. 图着色问题
19. 顶点覆盖问题
    1. 两个端点都去掉，近似比2

2. Nixon 教授提出了以下的启发式方法来解决顶点覆盖问题：重复地选择度数最高的顶点，并去掉所有邻接边。给出一个例子，说明这位教授的启发式方法达不到近似比 2。（提示：可以考虑一个二分图，其中左图中顶点的度数一样，而右图中顶点的度数不一样。）

20. 旅行商问题

定理 35.2 APPROX-TSP-TOUR 是一个解决满足三角不等式的旅行商问题的、多项式时间的 2 近似算法。

似游程路线，除非 P=NP。

定理 35.3 如果 P≠NP 则对任何常数 $\rho \geqslant 1$，一般旅行商问题不存在具有近似比 ρ 的多项式时间近似算法。

考虑以下的用于构造近似旅行商游程的最近点启发式：从只包含任意选择的某一顶点的平凡回路开始。在每一步中，找出一个顶点 u，它不在回路中，但与回路上任何顶点之间的距离最短。假设回路上距离 u 最近的顶点为 v。将回路加以扩展以包含顶点 u，即将 u 插入在 v 之后。重复这一过程，直到所有的顶点都在回路上时为止。证明：这一启发式返回的游程总代价不大于最优游程代价的两倍。

21. 最小权值顶点覆盖问题

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C ← ∅
2  compute x̄, an optimal solution to the linear program in lines(35.15)—(35.18)
3  for each v∈V
4       do if x̄(v)≥1/2
5            then C ← C∪{v}
6  return C
```

近似比为2

22.