ADS06 Backtracking

每一步的所有可能情况为 S_i 集合.

找到问题答案的一个可靠方法是列出所有可能的情况(candidates)，逐一检查，在对所有（或部分）答案进行检查后，排除错误情况，得到其中哪些是正确答案。

回溯使我们能够eliminate对大量已知不可能情况的显式检查。

↓ pruning

基本方法：设已有部分答案 $(x_1, x_2, \cdots x_i)$，其中 $x_k \in S_k$.

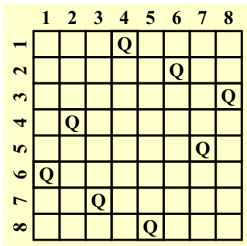
首先将 $x_{i+1} \in S_{i+1}$ 加入到已有答案中，检查 $(x_1, x_2, \cdots x_i, x_{i+1})$ 是否可能(满足constains)

- 若可能 则继续加 x_{i+2}
- 若不可能则删去 x_{i+1}，回到 $(x_1, x_2, \cdots x_i)$

回到 $(x_1, \cdots x_i)$ 后将会试验 S_{i+1} 中的下一个 x_{i+1}.

若 S_{i+1} 的所有 x_{i+1} 都不可能，返回 false

1. Eight Queens



Q_i ::= queen in the i-th row

x_i ::= the column index in which Q_i is

Solution = $(x_1, x_2, \ldots, x_8)$ = (4, 6, 8, 2, 7, 1, 3, 5)

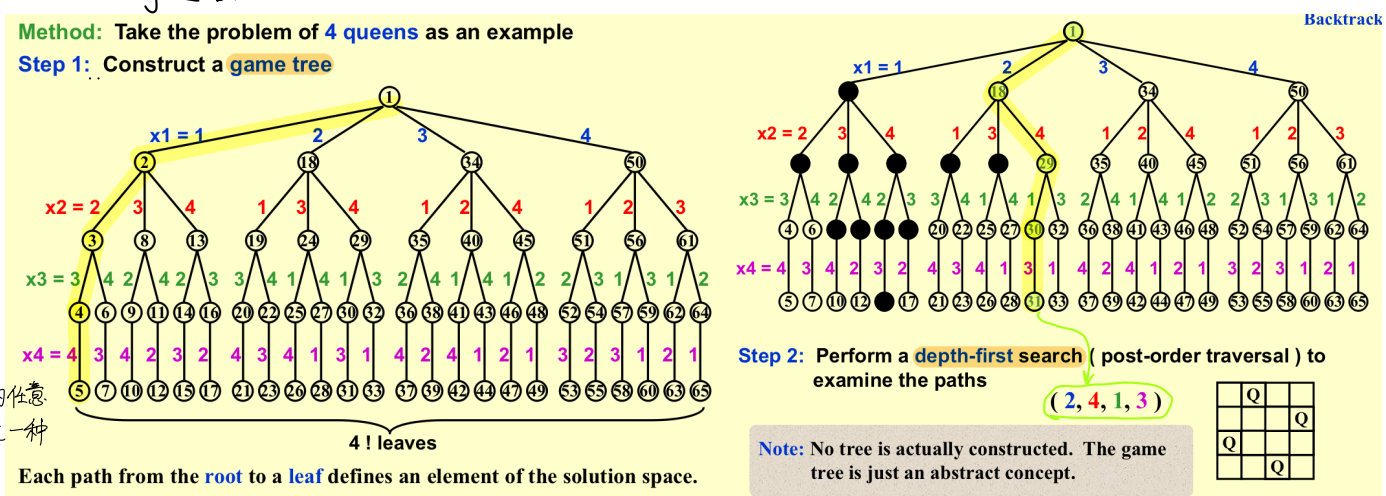
Constrains: ① $S_i = \{1,2,3,4,5,6,7,8\}$ for $1 \le i \le 8$

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

↓ 这一条件能将 8^8 种 candidate 缩减到 8! 种

对于 n×n 棋盘上的 n 皇后，candidates 有 n! 种.

backtracking 过程：



从根走叶的任意路径都是一种 candidate

数据结构上不用建 game tree

P1 是相当于 BFS.

(收费公路)

2. Turnpike Reconstruction Problem

Given N points on the x-axis with coordinates $x_1 < x_2 < \ldots < x_N$. Assume that $x_1 = 0$. There are $N(N-1)/2$ distances between every pair of points.

Given $N(N-1)/2$ distances. Reconstruct a point set from the distances.

从长向短试树

〖Example〗 Given $D = \{ 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10 \}$

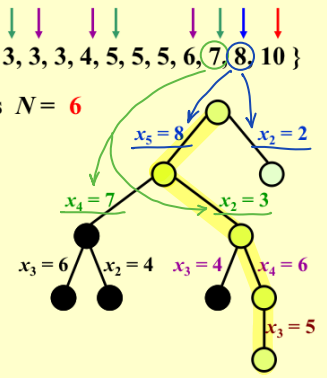
由距离段数求 points 数

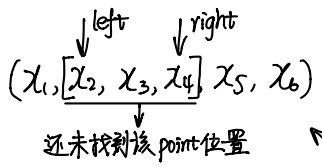
Step 1: $N(N-1)/2 = 15$ implies $N = 6$

Step 2: $x_1 = 0$ and $x_6 = 10$

Step 3: find the next largest distance and check

(0, 3, 5, 6, 8, 10)






```
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]|∈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);
      }
   }
   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;
}
```

新加入的点产生的新dist是否都在集合D中(且未被用过)

左子树

从D中删去新加入的点产生的新dist

将D中被删去的dist加回来(撤消)

(左路行不通)

右子树

从D中删去新加入的点产生的新dist

将D中被删去的dist加回来(撤消)

```
bool Backtracking ( int i )
{  Found = false;
   if ( i > N )
      return true; /* solved with (x1, …, xN) */
   for ( each xi ∈ 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;
}
```

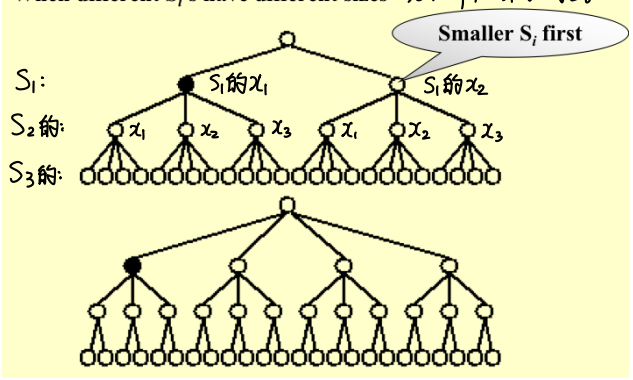
当前第 i 层中所有可能的情况集 S_i 中的每种情况 x_i

x_i 加入后，是否满足 constrains

若满足 constrains 则尝试加入 S_{i+1} 中的各 x_i

若加入 S_{i+1} 中的各 x_i 没有能成功的，撤消回 $x_1 \sim x_{i-1}$

When different S_i's have different sizes 先枚举 $|S_i|$ 小的 S_i



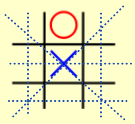
3. Tic-tac-toe

Tic-tac-toe: Minimax Strategy

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

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

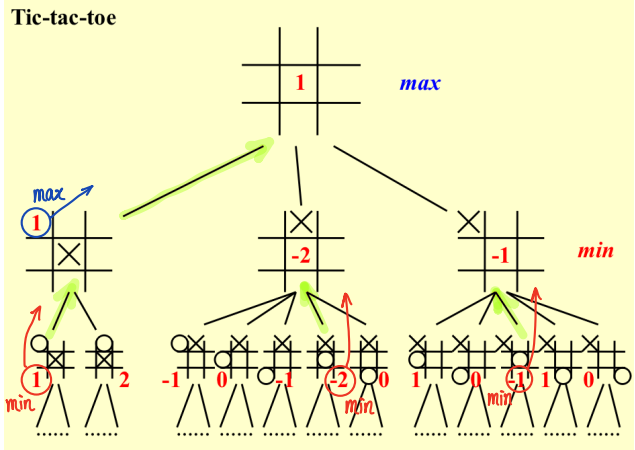
where W is the number of potential wins at position P.



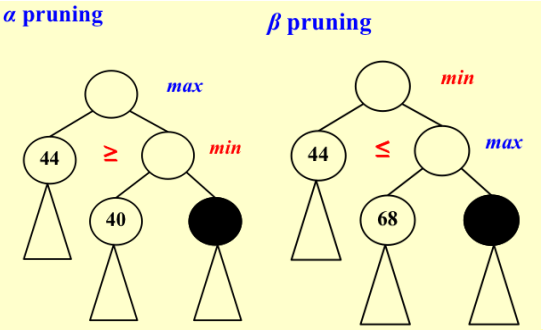
$f(P) = 6 - 4 = 2$

还有6种赢的可能布局

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



4. pruning



α-β pruning: when both techniques are combined.

pruning是说根本不用碰到它.

α-β pruning 能将要查找节点数减少至 $O(\sqrt{N})$.

ADS07 Divide and Conquer

Recursively:

- Divide the problem into a number of sub-problems
- Conquer the sub-problems by solving them recursively
- Combine the solutions to the sub-problems into the solution for the original problem

用分治思想的算法：① maximum subsequence sum 最大子串和 $O(N\log N)$

② tree traversals 树遍历 $O(N)$

③ merge sort & quick sort $O(N\log N)$

1. Closest Points Problem

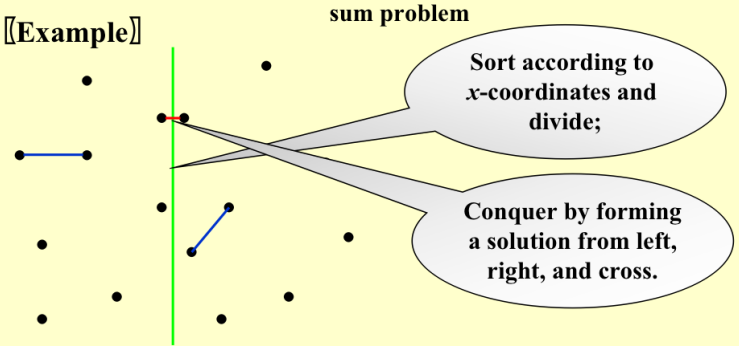
Given N points in a plane. Find the closest pair of points. (If two points have the same position, then that pair is the closest with distance 0.)

➤ **Simple Exhaustive Search**

Check $N(N-1)/2$ pairs of points. $T = O(N^2)$.

➤ **Divide and Conquer** – similar to the maximum subsequence sum problem

〖Example〗



采用分治算法：

在寻找 横跨中轴的最短点距时，可以使用类似剪枝的思想

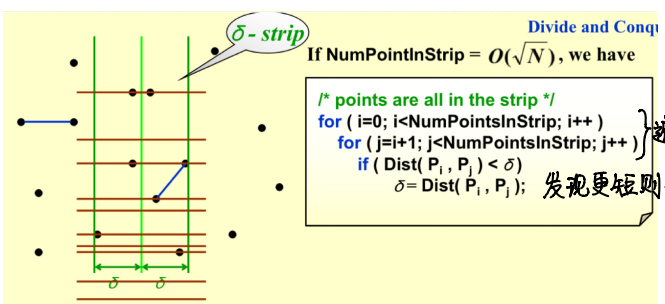
不必 $\frac{N}{2} * \frac{N}{2}$ 次遍历.

↓

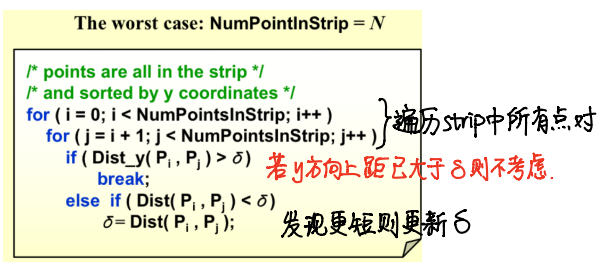
仅考虑strip中点对.

Delta是最小距离。在开始研究跨越中轴的点距之前，delta已经为都在左边/右边的最短点距。

在研究跨越中轴的点距时，只需考虑delta-strip内中轴左右的点对的距离

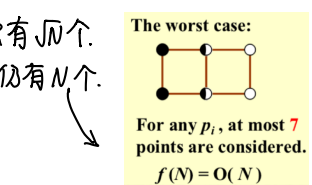


进一步简化：且对于左边/右边的一个点，只需考虑其下方delta距离内的点到它的距离（上方将由另一边考虑）



平均情况下 strip中点数只有 $\sqrt{N}$ 个.

worst case 时 strip中点数仍有 N 个.



2. 求解递推式 $T(N) = aT(N/b) + f(N)$

忽略细节：N/b 是否整数不影响分析. 当 n 小时，$T(n)$ 是 $\Theta(1)$ 的.

(1) 法一：Substitution Method

先猜对答案，再用数归证明.

[Example] $T(N) = 2T(\lfloor N/2 \rfloor) + N$

Guess: $T(N) = O(N \log N)$

Proof: Assume it is true for all $m < N$, in particular for $m = \lfloor N/2 \rfloor$.

Then there exists a constant $c > 0$ so that $T(\lfloor N/2 \rfloor) \le c\lfloor N/2 \rfloor \log \lfloor N/2 \rfloor$

Substituting into the recurrence:

$$T(N) = 2T(\lfloor N/2 \rfloor) + N$$
$$\le 2c\lfloor N/2 \rfloor \log \lfloor N/2 \rfloor + N$$
$$\le cN(\log N - \log 2) + N$$
$$\le cN \log N \quad \text{for } c \ge 1$$

[Example] $T(N) = 2T(\lfloor N/2 \rfloor) + N$

Wrong guess: $T(N) = O(N)$

Proof: Assume it is true for all $m < N$, in particular for $m = \lfloor N/2 \rfloor$.

$T(\lfloor N/2 \rfloor) \le c\lfloor N/2 \rfloor$

Substituting into the recurrence:

$$T(N) = 2T(\lfloor N/2 \rfloor) + N$$
$$\le 2c\lfloor N/2 \rfloor + N$$
$$\le cN + N = O(N) \times$$

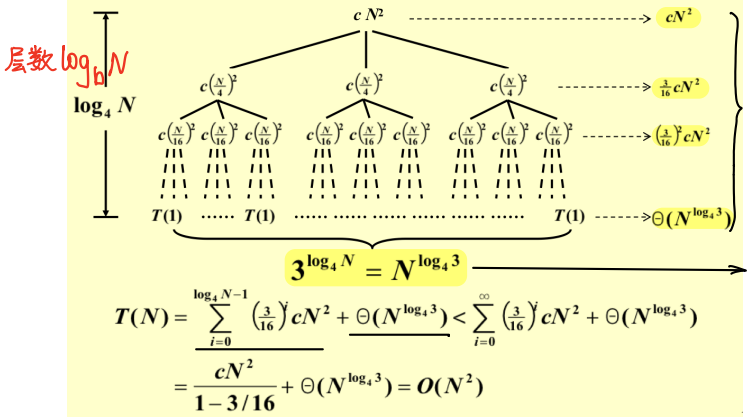
How to make a good guess? Must prove the *exact form*

(2) 法二：Recursion-tree Method

Recursion-tree method

[Example] $T(N) = 3T(N/4) + \Theta(N^2)$

$a^{\log_b N} = N^{\log_b a}$.



合并开销 $= \sum_{i=0}^{\text{层数}}$ 每层合并开销.

叶节点开销 $= O(1) *$ 叶数 $=$ 叉数$^{\text{层数}} = a^{\log_b N} = N^{\log_b a}$

$$3^{\log_4 N} = N^{\log_4 3}$$

$$T(N) = \underbrace{\sum_{i=0}^{\log_4 N - 1} \left(\frac{3}{16}\right)^i cN^2}_{} + \underbrace{\Theta(N^{\log_4 3})}_{} < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3})$$

$$= \frac{cN^2}{1 - 3/16} + \Theta(N^{\log_4 3}) = O(N^2)$$

(3) 法三：Master Method

$\log N < N^k$，

不论 k 多小 N^k 都是多项式级别

(a) Master Method 1:

$f(N) = O(N^{\log_b a - \varepsilon}),\ \varepsilon > 0 \Rightarrow T(N) = \Theta(N^{\log_b a})$

$f(N) = \Theta(N^{\log_b a}) \Rightarrow T(N) = \Theta(N^{\log_b a} \cdot \log N)$

$f(N) = \Omega(N^{\log_b a + \varepsilon}),\ \varepsilon > 0$ 且 $af(N/b) < cf(N),\ c < 1 \Rightarrow T(N) = \Theta(f(N))$

(b) Master Method 2:

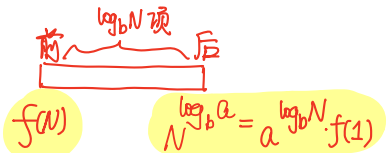
$af(N/b) = kf(N),\ k < 1 \Rightarrow T(N) = \Theta(f(N))$

$af(N/b) = kf(N),\ k > 1 \Rightarrow T(N) = \Theta(N^{\log_b a})$

$af(N/b) = f(N) \Rightarrow T(N) = \Theta(f(N) \cdot \log_b N)$

$af(N/b) \ne kf(N)$

后项 前项



前项占主导

后项占主导

项数 $\log_b N$ * 前/后

(C) Master Method 3: 对于 $T(N) = aT(N/b) + \Theta(N^k \log^p N)$ $(a \geq 1, b > 1, p > 0)$

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

ADS08 Dynamic Programming

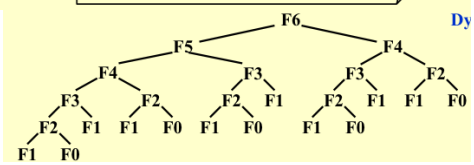
$T \geq O(N^{dim})$

在递归过程中避免多次解同一个子问题，用表记录下来其结果.

1. Fibonacci

1. Fibonacci Numbers: $F(N) = F(N-1) + F(N-2)$

```
int Fib( int N )
{
   if ( N <= 1 )
      return 1;
   else
      return Fib( N - 1 ) + Fib( N - 2 );
}
```



原本复杂度 $O(N)$

递归复杂度 $O(2^N)$ ← 有大量重复计算.

Solution

DP法一：在过程中用表记录计算结果，算过的直接查表

DP法二：不用递归，正向计算记录下来最近两个数，自底向上.

2. Ordering Matrix Multiplications 矩阵乘法的顺序

不同计算顺序下，计算量会有显著不同. 如何确定最优 order？

【Example】 Suppose we are to multiply 4 matrices

$M_{1\,[10\times20]} * M_{2\,[20\times50]} * M_{3\,[50\times1]} * M_{4\,[1\times100]}$.

If we multiply in the order

$M_{1\,[10\times20]} * (M_{2\,[20\times50]} * (M_{3\,[50\times1]} * M_{4\,[1\times100]}))$

Then the computing time is

$50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125{,}000$

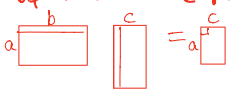
If we multiply in the order

$(M_{1\,[10\times20]} * (M_{2\,[20\times50]} * M_{3\,[50\times1]})) * M_{4\,[1\times100]}$

Then the computing time is

$20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2{,}200$

a*b 阵与 b*c 阵相乘需要 a*b*c 次计算



(1) n 矩阵相乘 有多少种 order？

设有 b_n 种 order. 令 $M_{ij} = M_1 * \cdots * M_j$

$\therefore M_{1n} = M_{1i} * M_{i+1,n}$

$$\Rightarrow b_n = \sum_{i=1}^{n-1} b_i \cdot b_{n-i} = O\left(\frac{4^n}{n\sqrt{n}}\right)$$ 卡特兰数 此问题穷举将 $O(4^n)$

(2) 矩阵相乘一定化为 2 个矩阵的相乘.

该问题的最优方案的 2 个子方案一定是最优方案（最优子结构）

设 $M_1 * \cdots M_n$ 的 M_i 为 $r_{i-1} \times r_i$ matrix，设 m_{ij} 为 $M_i * \cdots * M_j$ 的最优计算量.

有：

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

[递推关系]

↓ 要得到长度为 k 的答案，只涉及调用长度小于 k 的答案. [DP关键]

↓ 自底向上依次计算.

```
/* 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++ ) /* k = j - i */ 对所有的长度
      for( i = 1; i <= N - k; i++ ) {
         j = i + k;    M[ i ][ j ] = Infinity;}对长度为k的所有段
         for( L = i; L < j; L++ ) {      L将i,j分为两段
            ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                  + r[ i - 1 ] * r[ L ] * r[ j ];
            if ( ThisM < M[ i ][ j ] )  /* Update min */
                  M[ i ][ j ] = ThisM;
         } /* end for-L */                          } 计算min
      } /* end for-Left */
}
```

$m_{1,N}$

$m_{1,N-1}$ $m_{2,N}$

$m_{1,2}$ $m_{2,3}$ $\cdots$ $m_{N-1,N}$

$m_{1,1}$ $m_{2,2}$ $\cdots$ $m_{N-1,N-1}$ $m_{N,N}$

$T(N) = O(N^3)$

3. Optimal Binary Search Tree

最优二叉搜索树 (最优的静态搜索方法)

给定将插入的一堆词和搜索频率，应如何构建 BST 使总 access 代价的期望最小？

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

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

该问题的最优方案的2个子方案一定是最优方案 (最优子结构)

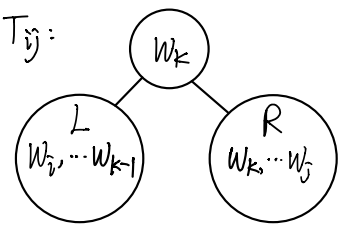
⇒ 一棵OBST的两个 subtree 也必是 OBST. 发现递推关系

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



[递推关系]

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

根结点 cost　左子问题 cost　右子问题 cost

L,R从单独的问题变为子问题使其所有结点d+1，因此

子问题 cost $= \sum p_i(1+d_i+1) = \sum p_i(1+d_i) + \sum p_i$

单独问题cost　单独问题cost　问题中words总权重

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

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

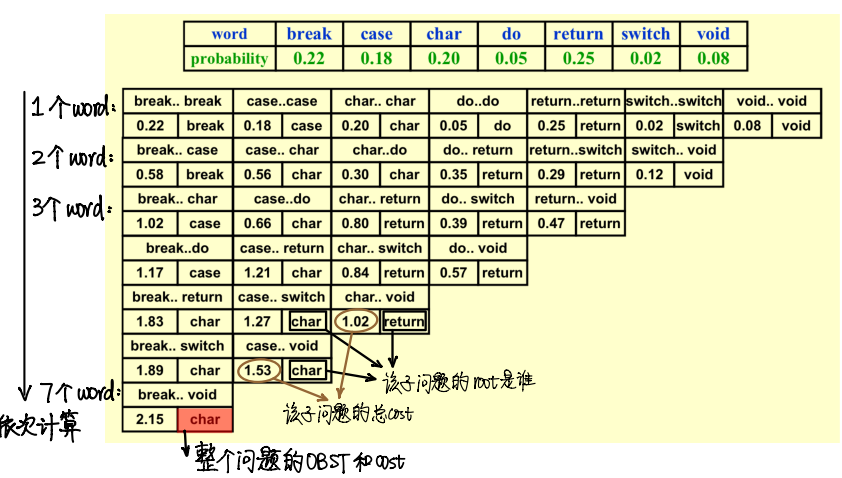
∴ 完成2个子问题后，根结点 r_{ij} 的选取应使 $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

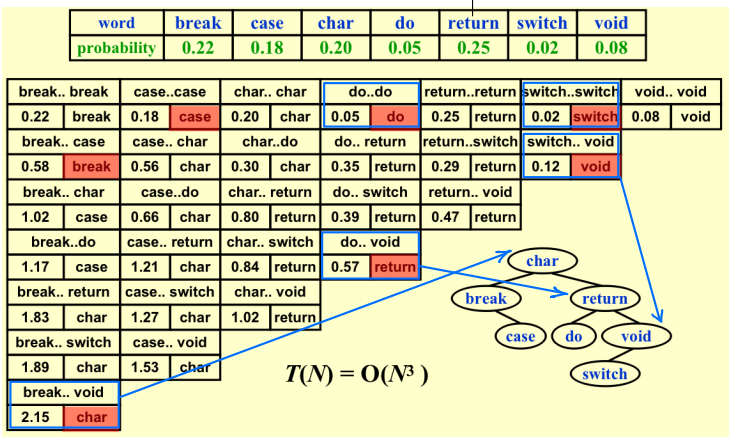
DP计算过程：

如何据计算结果构建OBST.

break~case | char | do~void

do~do　return　switch~void





4. All-Pairs Shortest Path 任意两点间最短路径

(1) 用dijkstra等single-source最短路径，对每个结点用一次. $T = O(|V|) \cdot O(|V|^2)$

$T = O(|V|^3)$ 稀疏图更快.

(2) 每次考虑加入一个点k，若k对最短路有影响就体现其影响. 当所有k都被考虑加入后即得最短路

该问题的最优方案的2个子方案一定是最优方案（最优子结构）

⇒ 若从v_i到v_k的最短路中经过了v_k，其上$v_i \to v_k$和$v_k \to v_j$段也一定是最短路.

设$D^k[i][j]$为从v_i到v_j在只经过1~k这些点条件下的最短路长. 则$D^{N-1}[i][j]$即为答案. （k是0~N-1）

$D^{-1}[i][j]$即为v_i与v_j间原始边长.

适用于有负权边（但不能负环）

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

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

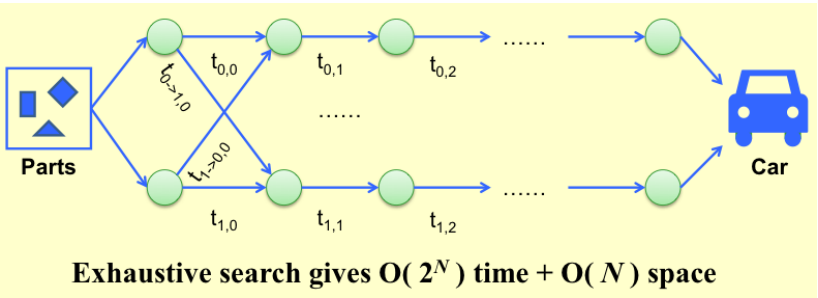
[递推关系] 若k不在i~j的实际最短路中 / 若k在i~j的实际最短路中

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
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++ )                    加入v_k
       for( i = 0; i < N; i++ )                对∀v_i v_j 间最短路作更新
           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 ];  更新
}
```

5. Product Assembly 产品线问题

可以切换 line 但不能跳工序

如何确定最优生产过程？



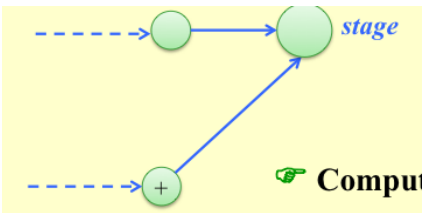
Exhaustive search gives O(2^N) time + O(N) space

该问题的最优方案的2个子方案一定是最优方案（最优子结构）

⇒ 从0~N的optimal solution一定对于中途的每个工序都是optimal.

对过程中 ∀ stage，要么从 line 0来，要么从 line 1来

到达 stage 的 optimal path 由 到达 stage-1 的 optimal path 得来.



[递推关系]

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){ 对0~N的stage
  for(line=0; line<=1; line++){ 对该stage的2个line
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];  若从stage-1到stage没切换
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];  若从stage-1到stage有切换
    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;
    }
  }
}
```

将 f[line][stage] = min{…}

L 记录路线.

```
line = f[0][n]<f[1][n]?0:1;
for(stage=n; stage>0; stage--){
  plan[stage] = line;
  line = L[line][stage];
}
```

plan为结果

How to design a DP method?

- Characterize an optimal solution
- Recursively define the optimal values
- Compute the values in some order
- Reconstruct the solving strategy

DP步骤：

① 由优化子问题结构 寻找递推关系

② 由递推关系 求解问题（但不用递归代码，而是自底向上）

③ (if necessary) 由结果构造出解的方案

必要条件：最优子结构、重叠的子问题

ADS09 Greedy Algorithm

optimization problem：优化问题：

满足限制集的解称 feasible solution（constraints）。在FS中求使优化函数取最好值的解（optimization function）。

Greedy Algorithm 未必能解决所有优化问题。

①仅在 local optimum = global optimum 时可用。

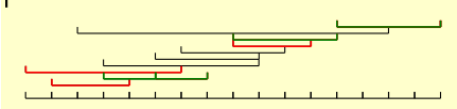
②它不保证给出最优解，但它给出的解近似于最优解，并且开销不大。

1. Activity Selection Problem 选择尽多的相容活动

活动集 $S=\{a_1, a_2, \cdots a_n\}$

每个活动占用时间 $[s_i, f_i)$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



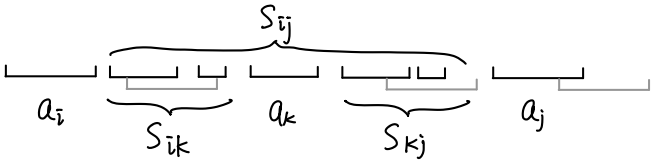
表中不妨设 $f_1 \le f_2 \le \cdots \le f_{n-1} \le f_n$.

Solution 1: DP

设 S_{ij} 是完全处于"a_i 结束之后 ~ a_j 开始之前"区间内的活动集合

设 c_{ij} 是 S_{ij} 中最多相容活动数.

最优子结构：从 S_{ij} 中选出一个 a_k，则 S_{ik}, S_{kj} 也一定要取最多相容活动

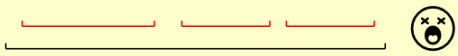


如果 Greedy 比 DP 的 $O(N^2)$ 还复杂，那就没意义了，就不是好的 Greedy.

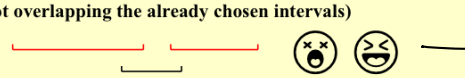
记 a_i~a_j 之中最优活动数为 $c_{ij} = \begin{cases} 0 & , S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & , S_{ij} \neq \emptyset \end{cases}$ $\Rightarrow O(N^2)$

Solution 2: Greedy

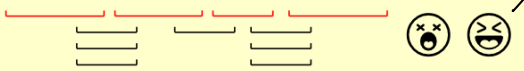
Greedy Rule 1: Select the interval which *starts earliest* (but not overlapping the already chosen intervals)



Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)

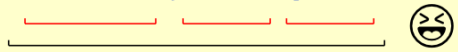


Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)



Greedy Rule 4: Select the interval which *ends first* (but not overlapping the already chosen intervals)

Resource become free as soon as possible



对于一种贪心策略
否定它的好方法是
找一个反例

总是留下了最多的资源

$O(N \log N)$

Correctness:
① Algorithm gives non-overlapping intervals
② The result is optimal

【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.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.
If a_m and a_{ef} are the same, we are done! Else ……
replace a_{ef} by a_m and get A_k'.
Since $f_m \le f_{ef}$, A_k' is another optimal solution. ∎

看上去策略是对的，但最优性仍需数学证明.
算法本身不保证最优性.

※ DP再思考

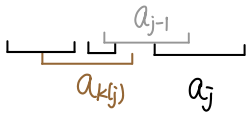
(1) 若不是二分而是从左向右

假设已得到 $a_1 \sim a_{j-1}$ 这些活动中的最多相容数，现考虑将 a_j 加入"已考虑集合".

则一定有2种情况 { ① 放入 $a_{k(j)}$ $C_{1,j} = C_{1,k(j)} + 1$ 从 $a_1 \sim a_{k(j)}$ 的最优 $+ a_j$; ② 不放入 $a_{k(j)}$ $C_{1,j} = C_{1,j-1}$ } 取max

会导致一些已放入活动被踢出

$a_{k(j)}$ 是在 a_j 开始前最晚结束的活动



$$C_{1,j} = \begin{cases} 1 & \text{, if } j=1 \\ \max\{C_{1,j-1},\ C_{1,k(j)}+1\} & \text{, if } j>1 \end{cases}$$

(2) 若各活动有权

$$C_{1,j} = \begin{cases} W_1 & \text{, if } j=1 \\ \max\{C_{1,j-1},\ C_{1,k(j)} + W_j\} & \text{, if } j>1 \end{cases}$$

DP仍可以，但刚才的Greedy就不行了，因为"最早结束"顾不上权重

※ 在每个 Greedy Algorithm 背后，总存在一个更复杂的 DP 算法.

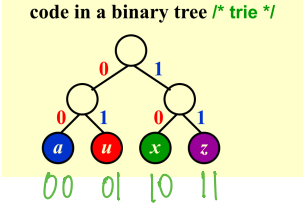
2. Huffman Codes (最优编码方式)

字符等长编码 → 空间浪费.

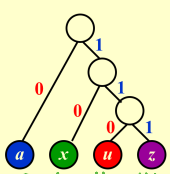
字符不等长编码 → 节省空间.

用字典树表示：

Representation of the original code in a binary tree /* trie */



Representation of the optimal code in a binary tree



➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010010 11)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

要求：没有哪个编码是另一个编码的前n位 ⟺ 字符只出现在字典树的叶子上.

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{  consider the C characters as C single node binary trees,
   and initialize them into a min heap;
   for ( i = 1; i < C; i++ ) {
      create a new node;
      /* be greedy here */
      delete root from min heap and attach it to left_child of node;
      delete root from min heap and attach it to right_child of node;
      weight of node = sum of weights of its children;
      /* weight of a tree = sum of the frequencies of its leaves */
      insert node into min heap;
   }
}
```

① 创建字典树的一个新 node

② 从堆中 pop min 两次，作为其左右子树

③ 新node weight = 左右儿子 weight

④ 新 node 作为子树 插入堆.

〖Example〗

C_i	a	e	i	s	t	sp	nl
f_i	10	15	12	3	4	13	1

在在12、13两个比18小的，会先从堆中拿出12,13.

a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

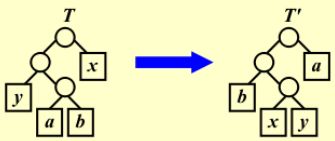
$Cost = 3\times10 + 2\times15 + 2\times12 + 5\times3 + 4\times4 + 2\times13 + 5\times1 = 146$

Huffman Code 最优性证明：

Correctness:

① The greedy-choice property

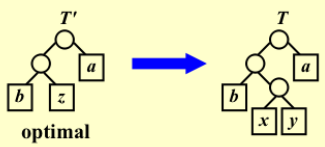
【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Cost(T') ≤ Cost(T)

② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.



By contradiction.

ADS 10 NP-Completeness

一、算法的复杂度

- Easiest: $O(n)$
- 易处理的: $\leq O(n^k)$ 多项式时间
- 不易处理的: $> O(n^k)$
- Hardest: undecidable problems 计算机不论多久均无法解决

Halting problem

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program, then surely it could be used to check itself.

```
Loop( P )
{
/* 1 */ if ( P(P) )
/* 2 */ else infinite_loop();
}
```

Impossible to tell

LOOP函数终止.

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops → /* 1 */ is true → Terminates

设程序P可判断另一个程序是否是死循环，若死循环返回 true，否则返回 false

构造一个程序 LOOP 对 P 进行调用，则 LOOP 的功能是

(i) 若 P 判断 P 死循环，则 LOOP 终止.

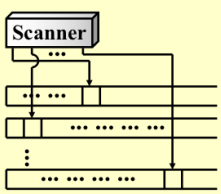
(ii) 若 P 判断 P 终止，则 LOOP 死循环.

LOOP 能判断一个能判断程序是否停止的程序在判断自己时的结果

二. Turing machine

Components: Infinite Memory and Scanner

Operations



① Change the finite control state.

② Erase the symbol in the unit currently pointed by head, and write a new symbol in.

③ Head moves one unit to the left (L), or to the right (R), or stays at its current position (S).

- Deterministic Turing Machine: 下一条执行的指令确定 (下一条由当前指令跳转)
  - ↓ 无穷并行，选最优
- Nondeterministic Turing Machine: 下一条执行的指令不确定，机器会从有限集中选一步执行. 并且总会选择正确的一步. (但 undecidable 仍为 undecidable)

NP: Non-deterministic Turing Machine polynomial-time 能解决

⇕

Deterministic Turing Machine polynomial-time 能验证

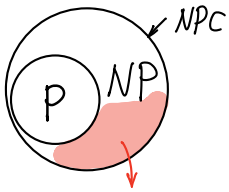
三、P、NP、NPC、NP-hard

1. P class：已知多项式时间算法的问题

2. NP class：给定一个答案，可以在 polynomial-time 验证它是否正确. 人们仅知道 poly-time 可验证，但不知道是否存在 poly-time 解法，称为 NP.

Nondeterministic polynomial-time.

※. P≠NP 是否所有能在 poly-time 验证的算法都能在 poly-time 解决？



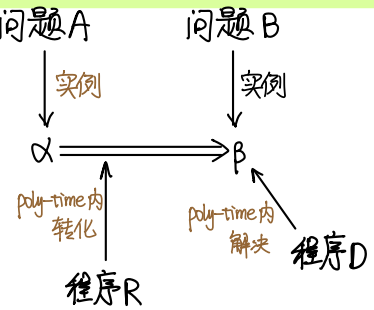
若 P⊂NP，则存在NP问题，不存在它们的poly-time算法.
若 P=NP，则NP类问题都存在poly-time算法.

若P⊂NP，则存在一些问题它们一定没有poly-time算法.

问题的归化：若Q2的算法也能解决Q1，称 Q1 可 reduce 为 Q2.

说是归化，但Q1 →(归化) Q2是问题变难/保持的过程.

标准定义：



若左图可以实现，则称问题A可 polynimially reduced 为问题B

3. NP-Complete Problem：将所有的NP向上归化，得到NP中最难的那些问题，称为 NP-Complete

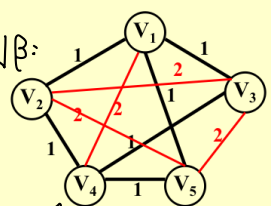
NPC是NP问题的边界.所有 NPC 问题在难度上等价，只要解决了任何一个 NPC，则边界上的 NPC 将全部被解决. 自然 NP 也将全部被解决.

证明一个问题是 NPC：① 问题∈NP ②一个已知的NPC问题 ≤p 该问题 (归化)

〖Example〗 Suppose that we already know that the Hamiltonian cycle problem is NP-complete. Prove that the traveling salesman problem is NP-complete as well.

Proof: ① TSP is obviously in NP, as its answer can be verified polynomially.

TSP问题实例β：



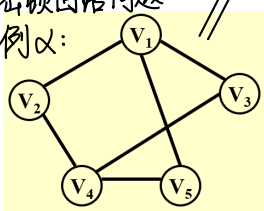
K = |V|

G has a Hamilton cycle iff G' has a traveling salesman tour of total weight |V|. ✔

- Hamiltonian cycle problem: Given a graph G=(V, E), is there a simple cycle that visits all vertices?
- Traveling salesman problem: Given a complete graph G=(V, E), with edge costs, and an integer K, is there a simple cycle that visits all vertices and has total cost ≤ K?

证明 ≤p 的核心是：
(i) 制造一个α，找到 α⇒β 的过程，找到β
(ii) 证明 α 和 β 答案相同

哈密顿回路问题实例α：



(i) α⇒β 过程：加边补成完全图，加权，设一个K

(ii) 证明α与β答案相同

即证：某回路是α的答案 ⇔ 是β的答案
⇒：在α上的遍各点的回路必在β上，且权和为|V|满足≤K
∴也一定是β的答案.

⇐：在β上的遍各点的权和 ≤ K=|V|的回路，必只过权为1的边 必在以上
∴也一定是以的答案.

※ 如果NPC是这么证的，那第一个 NPC 问题是？

是 Circuit-SAT 问题. 它是用"Nondeterministic Turing Machine 在 poly-time 解决"证明的

4. NP-hard：一个已知的NPC问题 $\leq_P$（约化）该问题.（但该问题未必是NP）

四. Formal-language Theory

1. Abstract Question Q：问题实例集合I ⟶ 问题实例的解的集合S 的映射/二元关系

eg:

〖Example〗 For SHORTEST-PATH problem
I = { <G, u, v>: G=(V, E) is an undirected graph; u, v ∈ V };
S = { <u, w_1, w_2, …, w_k, v>: <u, w_1>, …, <w_k, v> ∈ E }.
For every *i* ∈*I*, SHORTEST-PATH(*i*) = *s* ∈ *S*.

2. Encodings：将集合I 编码为 0,1 位串

3. Formal-language Theory

empty string ε
empty language $\emptyset$

(1) alphabet Σ 字符集 {0,1}

(2) language L (over Σ) 语言：由Σ中字符组成的字符串的集合. $L = \{x \in (0,1)^* : Q(x)=1\}$

x是字符串. Q(x)能判定它是否有实际含义/描问题实例.

语言是有实际含义字符串的集合
问题是 问题实例 的集合

(3) Σ^*：由Σ中字符可以组成的所有字符串的集合.

- complement of L 语言的补：所有字符串 − 语言. $\Sigma^* - L$.
- concatenation of L_1 and L_2 语言的连接：L_2串接在L_1串后面.
- closure of L 语言的闭包：$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cdots$

(1) Algorithm A
- accept a string $x \in \{0,1\}^*$ 算法接受一个字符串：字符串输入算法，输出1. 即 A(x)=1
- reject a string $x \in \{0,1\}^*$ 算法拒绝一个字符串：字符串输入算法，输出0. 即 A(x)=0

证明 NPC eg:

〖Example〗 Suppose that we already know that the clique problem is NP-complete. Prove that the vertex cover problem is NP-complete as well.

Proof: ① VERTEX-COVER ∈ NP

Given any x = <G, K>, take V' ⊆ V as the certificate y.

Verification algorithm: check if |V'| = K; check if for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$.

$O(N^3)$

Proof (con.): ② CLIQUE $\leq_P$ VERTEX-COVER

G has a clique of size K iff $\overline{G}$ has a vertex cover of size |V| − K.

⇒ G has a clique V' ⊆ V of size K

Let (u, v) be any edge in $\overline{E}$ ➡

At least one of u or v does not belong to V'

At least one of u or v does belong to V-V'

Every edge of $\overline{G}$ is covered by a vertex in V-V'

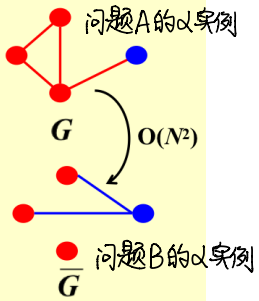
Hence, the set V-V', which has size |V| − K forms a vertex cover for $\overline{G}$

⇐ $\overline{G}$ has a vertex cover V' ⊆ V of size |V| − K

For all $u, v \in V$, if $(u, v) \notin E$, then $u \in V'$ or $v \in V'$ or both.

For all $u, v \in V$, if $u \notin V'$ AND $v \notin V'$, then $(u, v) \in E$.

V-V' is a clique and it has size |V|−|V'| = K. ∎



18

❖ Clique problem: Given an undirected graph G = (V, E) and an integer K, does G contain a complete subgraph (*clique*) of (at least) K vertices?

CLIQUE = { <G, K> : G is a graph with a clique of size K }.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

VERTEX-COVER = { <G, K> : G has a vertex cover of size K }.

(i) 制造一个x，找到α⇒β的过程，找到β

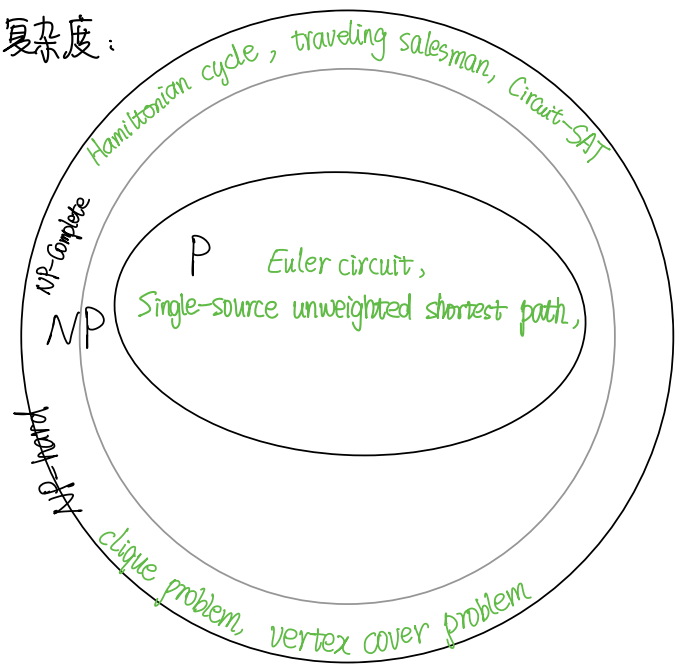
求其补图

设 $K_1 = |V|$, $K_2 = |V| - K_1$

(ii) 证明α与β答案相同

即证：G存在size为K的团 ⇔ $\overline{G}$存在size为|V|-K的V'是cover

常见问题的复杂度：



目前还不知道两个黑圈是不是一个圈

或者说不知道是否存在Q，Q∉P，Q∈NP

ADS II Approximation

面对不好的问题（> poly-time）可以用近似算法，以精度换时间.

若一个最优化问题的最优值为c*，一个近似算法的近似解的函数值为c，则将该近似算法的approximation ratio 定义为max(c/c*, c*/c)。

通常该ratio不超过输入规模n的一个函数ρ(n)，即 max(c/c*, c*/c) <= ρ(n)。

$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$ 若一个近似算法的近似比达到 ρ(n)，称其为 ρ(n)-approximation algorithm

- approximate scheme：

  是一个近似算法，它读入一个问题和 $\forall \varepsilon > 0$，它总能确保给出解的 ratio $\le 1+\varepsilon$

- poly-time approximate scheme (PTAS)：

  是一个近似算法，它读入一个问题和 $\forall \varepsilon > 0$，它总能确保给出解的 ratio $\le 1+\varepsilon$，且在poly-time解决 $O(n^{\frac{2}{\varepsilon}})$

- fully poly-time approximate scheme (FPTAS)：

  是一个近似算法，它读入一个问题和 $\forall \varepsilon > 0$，它总能确保给出解的 ratio $\le 1+\varepsilon$，且在poly-time解决. $O((\frac{1}{\varepsilon})^2 \cdot n^3)$

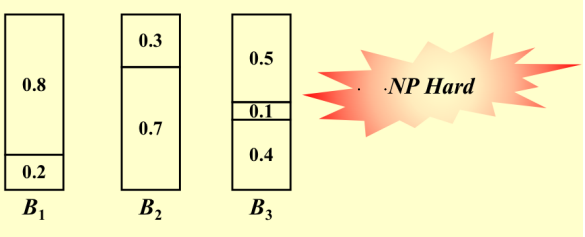
  且关于 $\frac{1}{\varepsilon}$ 也是多项式级别

1. Approximate Bin Packing

Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.

【Example】 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$



An Optimal Packing

On-line algorithm 在读入数据过程中就要决策、处理，且之后不能反悔

(1) 近似1：Next Fit ratio = 2

❖ Next Fit

```
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 */
}
```

只要当前bin剩余空间不够放，就开新bin

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *next fit* never uses more than $2M - 1$ bins. There exist sequences such that *next fit* uses $2M - 1$ bins.

(2) 近似2：First Fit ratio = 1.7

❖ First Fit

```
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)

已开的bin可放的中，第一个bin去放

若已开的bin都不够放，再开新bin

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

Let $S(B_i)$ be the size of the ith bin. Then we must have:

$S(B_1)+S(B_2)>1$
$S(B_3)+S(B_4)>1$
.....
$S(B_{2M-1})+S(B_{2M})>1$

假设NextFit用掉2M个bin

$\sum_{i=1}^{2M} S(B_i) > M$

总物品size > M

// optimal #bins

都为1.7

(3) 近似3: Best Fit ratio = 1.7

❖ **Best Fit**

Place a new item in the tightest spot among all bins.
$T = O(N \log N)$ and bin no. ≤ $1.7M$

已开的bin可放的中，放入之后能占得尽满的那个bin，去放
若已开的bin都不够放，再开新bin

online algorithm 永远不可能全局最优，因为它不知道之后会不会停，会有什么size，甚至可以被放出快的人整蛊
可以证明，对∀ on-line algorithm 总是存在 inputs 使其 ratio ≥ $\frac{5}{3}$

Off-line algorithm 读完全部输入数据后一起处理

(4) 近似4: First Fit / Best Fit Decreasing first fit 用至多 $\frac{11}{9}M+\frac{6}{9}$ bins

Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

先按递减顺序排列，之后使用 first fit / best fit

2. Knapsack Problem (只在最后 p_{max} round 时是近似算法)

(1) Knapsack Problem – fractional version

背包容量为M，有N个物体，第i个物体(金沙堆、银沙堆)重 w_i 值 p_i，允许把物体的 x_i 比例(金灰银沙)装入，(利益为 $p_i x_i$) 求利益最大装法。

在 $\sum_{i=1}^{n} x_i w_i \le M$ 条件下使 $\sum_{i=1}^{n} x_i p_i$ 最大

Greedy 依据：
~~①~~ 先把 p_i 最大的沙堆尽可能多装，再到 p_i 第二大的…… [maximum profit]
~~②~~ 先把 w_i 最小的沙堆尽可能多装，再到 w_i 第二大的…… [minimum weight]
③ 先把 $\frac{p_i}{w_i}$ 最大的沙堆尽可能多装，再到 $\frac{p_i}{w_i}$ 第二大的…… optimal! [maximum profit density]

三种 greedy 都是最终拆分一个物品的。

(2) Knapsack Problem – 0-1 version NP-hard

若仅用 Greedy 依据①③，approximation ratio 为 2.

证明：$P_{0\text{-}1,opt} \le P_{frac,opt} = P_{frac,Greedy} \le P_{0\text{-}1,Greedy} + p_{max}$

$$\frac{P_{0\text{-}1,opt}}{P_{0\text{-}1,Greedy}} \le 1+\frac{p_{max}}{P_{0\text{-}1,Greedy}} \le 1+1 = 2$$

假设至少能放下一个物体

可以用 dynamic programming：

近似算法：让 p 长度 s 减小（rounding）

$W_{i,p}$ = the minimum weight of a collection from {1, …, i} with total profit being exactly p

设 $W_{i,p}$ 是考虑完第 i 物品后利益恰为 p 时的最轻质量.

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$ 若 $W_{i,p}$ 由"放入第 i 物品"得到

② skip i : $W_{i,p} = W_{i-1,p}$ 若 $W_{i,p}$ 由"不放入第 i 物品"得到

③ impossible to get p : $W_{i,p} = \infty$

状态转移方程

目标状态都为 $W_{i,p}$，想下标时要按回退想.

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & otherwise \end{cases}$$

∞：在没考虑过任何书时想达到 p 要 ∞ 重.

$W_{i-1,p}$：若放入 i

$w_i + W_{i-1,p-p_i}$：若不放入 i

仅当 $p > p_i$ 时才有意义.

$O(n^2 p_{max})$

有 i 和 p 两个维度 $\begin{cases} i = 1, \cdots n \\ p = 1, \cdots np_{max} \end{cases}$

复杂度 $O(n^2 p_{max})$

↓

伪多项式时间复杂度，p_{max} 可以是 n 的指数级

What if p_{max} is LARGE?

Item	Profit	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

M = 11

Item	Profit	Weight
1	2	1
2	7	2
3	19	5
4	223	6
5	284	7

M = 11

可以用 round 方法缩小 p_{max} 规模. 但算法也将变成近似算法.

Round all profit values up to lie in smaller range!

$(1+\varepsilon) P_{alg} \geq P$ for any feasible solution P

（算法解 ≥ 最优解）

precision parameter

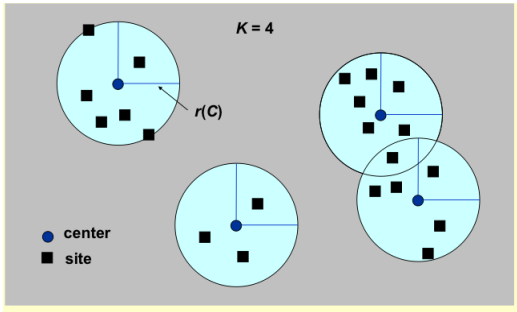
设 p_{max} 长为 s，则 $p_{max} = O(2^s)$ 复杂度为 $O(n^2 \cdot 2^s)$.

因此复杂度为指数级（果然是 NP-hard）.

3. K-center Problem

平面上给定 n 个点，要求找出不超过 K 个圆心的位置使圆覆盖所有点，且 radius 最小.

这个问题非常复杂，且没法遍历，搜索空间为 ∞



Input: Set of n sites $s_1, \ldots, s_n$

Center selection problem: Select K centers C so that the maximum distance from a site to the nearest center is minimized.

设 C^* 与 $r(C^*)$ 是最优解. 我们只考虑将点作为圆心.

一种判定当前 r 与 $r(C^*)$ 关系的方法：

若 ∀ 一个点作为中心，以 2r 为半径画圆，覆盖掉其中一些点.

对剩下的点再 ∀ 中心以 2r 为半径画圆，覆盖掉其中一些点. …… 不断往复，直至所有点都被覆盖.

此时圆心数 若 > K 则说明当前 $r < r(C^*)$

若 < K 则说明当前 $r > r(C^*)$

依据是：

若某个 C^* 的 $r(C^*)$ 的圆能覆盖一些点，则 ∀ 其中一个点为圆心以 $2r(C^*)$ 的圆一定也能覆盖.

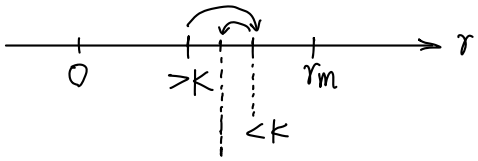
(Greedy algorithm)

对当前 r 进行试验

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r )
{  Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
   Centers  C[ ] = ∅;
   while ( S'[ ] != ∅ ) {
      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);
}
```

不断对点为圆心删其2r内点.

有了这种判定方法后. $r = \frac{\max\{dist(u,v)\}}{2}$ 开始二分查找 $r(C^*)$ 2-approximation



※ 另一种更好的 Greedy 方案:

第一个圆心任选,之后每次取离当前圆心们最远的点为圆心. 直至取到 K 个. (没有用到 r)

2-approximation

```
Centers  Greedy-Kcenter ( Sites S[ ], int n, int K )
{  Centers  C[ ] = ∅;
   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;
}
```

K-center 问题拓展:

定理: 除非 P=NP, 否则就不存在 ratio < 2 的解法

简证: 若 K-center 存在 (2-ε)-approximation, 则它也可以用于精确求解 Dominating Set 问题 (NP Complete)

这相当于用 poly-time 解了 NPC, 即 P=NP.

※ Approximation 总结.

Three aspects to be considered:	Researchers are working on
A: Optimality -- *quality of a solution*	A+C: Exact algorithms for all instances
B: Efficiency -- *cost of computations*	A+B: Exact and fast algorithms for special cases
C: All instances	B+C: Approximation algorithms
	Even if P=NP, still we cannot guarantee A+B+C .

ADS12 Local Search

有时不需要全局解，只要求局部最优，作为一种近似.

- local optimum：在可行集FS的某邻域 neighborhood 中最优的解
- search：从某个可行解S开始，在基础上进行小修改，使仍为可行解且更优. 当无法继续修改则 search 结束.
  - (S∈FS)
  - S~S'：由S生成S'的操作
  - S'的集合为N(S). S'叫S的neighborhood
  - cost(S')<cost(S)

对于某个实际问题，
要定义其 Feasible Set，cost(S)，S~S'的操作(Search 的方法)

local search 的梯度下降算法

```
SolutionType Gradient_descent()
{  Start from a feasible solution S ∈ FS ;
   MinCost = cost(S);
   while (1) {                          S'为N(S)里cost最小的
     S' = Search( N(S) ); /* find the best S' in N(S) */
     CurrentCost = cost(S');
     if ( CurrentCost < MinCost ) {
        MinCost = CurrentCost;   S = S';
     }
     else  break;                       若cost(S')仍比当前cost大，
   }                                    说明当前已到最低点.
   return S;
}
```

1. Vertex Cover Problem:

❖ **Vertex cover problem:** Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

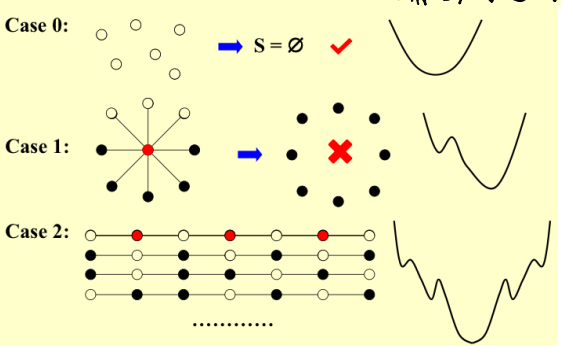
? Feasible solution set *FS* : all the vertex covers.
? cost(S) = | S |
? S ~ S':

针对问题，先定义 FS、cost(·)

Each vertex cover S has at most |V| neighbors.

? Search: Start from S = V; delete a node and check if S' is a vertex cover with a smaller cost.

确定本问题的 search 方法.



Try to improve …

? The Metropolis Algorithm

$e^{-\frac{\Delta cost}{KT}}$ T→∞ P→1 T→0 P→0 温度越高概率越大

```
SolutionType Metropolis()
{  Define constants k and T;
   Start from a feasible solution S ∈ 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^(-Δcost/(kT)) , let S = S';
       else  break;
     }
   }
   return S;
}
```

Adding is allowed

若cost(S')仍比当前cost大
说明当前已到最低点.

但在当前最低点，有一定概率会向外跳一步并继续循环
对图于局部最优的情况产生一定破除

7

T不是固定的，会慢慢下降.

simulated annealing 模拟退火算法：赋予搜索过程一种时变且最终趋于零的概率突跳性，从而避免陷入局部极小，使得算法最终趋于全局最优，的优化算法。

2. Hopfield NN

Graph G = (V, E) with integer edge weights w (positive or negative).
If $w_e < 0$, where $e = (u, v)$, then u and v want to have the *same state*;
if $w_e > 0$ then u and v want *different states*.
The absolute value $|w_e|$ indicates the *strength* of this requirement. ± 1

Output: A *configuration S* of the network –
an assignment of the state s_u to each node u

输入一个 graph 有 weight
若某 edge $w_e<0$，希望它两端点同号
若某 edge $w_e>0$，希望它两端点异号
} 每个 edge 希望 $w_e s_u s_v < 0$

输出一种各 node 的 state assignment

也许有无法全满足的情况，我们去找充分好的.(sufficiently good.)

定义:

设 node state S_u 为 ±1.

全局最优：所有 edge 都 good. $\Leftrightarrow w_e S_u S_v < 0$ 对 ∀edge 成立
但未必能做到，因此优化目标为 good edge Σw 最大.

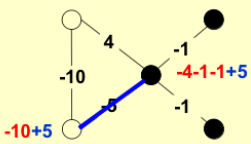
〖Definition〗 In a configuration S, edge $e = (u, v)$ is *good* if $w_e s_u s_v < 0$; otherwise, it is *bad*.

〖Definition〗 In a configuration S, a node u is *satisfied* if the weight of incident good edges ≥ weight of incident bad edges.

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

与 u 相连的 good edge weight 和 ≥ bad edge weight 和

〖Definition〗 A configuration is *stable* if all nodes are satisfied.



Does a Hopfield network always have a stable configuration, and if so, how can we find one?

stable 状态对任何 graph 一定存在.
因为我们总可以获得它

State-flipping Algorithm 状态置反算法

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

因为每一次都使 Σ good edge weight < Σ bad edge weight 的一组 edges 的 good ↔ bad. 将使这组 edge 的 Σ good edge weight ↑.
而组外 edge 不变
} 所有 Σ good edge ↑

每次至少 +1. 上界为 $\Sigma|w_e|$

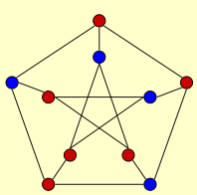
Claim: The state-flipping algorithm terminates at a stable configuration after *at most* $W = \Sigma_e|w_e|$ iterations.

该算法一定能得到 stable 的 assignment，在 $\Sigma_e|w_e|$ 次循环内
记为 W

3. Maximum Cut Problem

❖ **Maximum Cut problem: Given an undirected graph G = (V, E) with positive integer edge weights w_e, find a node partition (A, B) such that the total weight of edges crossing the cut is maximized.**

$$w(A, B) := \sum_{u\in A, v\in B} w_{uv}$$ 要最大



Related to Local Search

- ☞ **Problem: To *maximize* $w(A, B)$.**
- ☞ **Feasible solution set 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.**

A: state = +1. B: state = −1.
若设所有边 weight>0，则 cut edge $\Leftrightarrow S_u \cdot S_v < 0 \Leftrightarrow$ good edge
优化目标 Σ cut edge 最大 $\Leftrightarrow$ Σ good edge 最大.

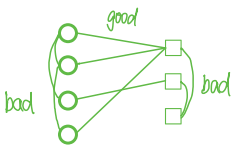
该问题是 Hopfield NN 的特例 (所有 $w_e>0$)

? A special case of Hopfield Neural Network – with w_e all being positive!

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

在W次内终止.
伪 poly-time 复杂度

该问题本质上，最后把图"二分"了
并且使 Σw_{good} 最大



Maximum Cut 这种近似的比率为2.(更低)

- **How good is this local optimum?**

Claim: Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

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* $u \in A$

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

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

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

? [Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.

$\min_{0 \leq \theta \leq \pi} \frac{\pi}{2} \frac{1-\cos\theta}{\theta}$

? [Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.

? [Håstad 1997] Unless P = NP, no 17/16 approximation algorithm for MAX-CUT.
↑ 1.0625

Big-improvement-flip: MAX-CUT 加速

将 flip 条件改为：若 flip 边 (A,B) 至少使 Σcut edge 增加 $\frac{2\varepsilon}{|V|} w(A,B)$ 才去 flip
则在 $O(\frac{n}{\varepsilon} \log W)$ 次 flip 内终止. 代价是 ratio $\leq 2+\varepsilon$.

- **May NOT terminate in polynomial time** 从 W 优化为 $\frac{n}{\varepsilon} \log W$

? **stop the algorithm when there are no "*big enough*" improvements.**

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.

- **Try a better *local* ?**

? **The neighborhood of a solution should be rich enough that we do not tend to get stuck in bad local optima; but ? the neighborhood of a solution should not be too large, since we want to be able to efficiently search the set of neighbors for possible local moves.**

Single-flip ➡ *k*-flip ➡ $\Theta(n^k)$ for searching in neighbors

[Kernighan-Lin 1970] *K-L heuristic*

Step 1: make 1-flip as good as we can – $O(n)$ ➡ (A_1, B_1) and v_1

Step *k*: make 1-flip of an ***unmarked*** node as good as we can – $O(n-k+1)$ ➡ (A_k, B_k) and $v_1 \ldots v_k$

Step *n*: $(A_n, B_n) = (B, A)$

Neighborhood of $(A, B) = \{ (A_1, B_1), \ldots, (A_{n-1}, B_{n-1}) \}$ $O(n^2)$

ADS13 Randomize Algorithm

非随机算法 R 是 randomized algorithm 的特例.

高效的 randomized algorithm 只需要以 high probability 产生正确结果即可. 它在 expectation 上是正确的

复习概统表达式:

Pr[A] := the *probability* of the event A

$\bar{A}$:= the *complementary* of the event A (A did not occur)

$$\Pr[A] + \Pr[\bar{A}] = 1$$

E[X] := the *expectation* (the "average value") of the random variable X

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$$

1. Hiring Problem

N天内每天面试一个人, 比现有的好则用它替换现有的.

? **Interviewing Cost = C_i << Hiring Cost = C_h**

? **Analyze interview & hiring cost instead of running time**

Assume M people are hired.

Total Cost: $O(NC_i + MC_h)$

Naïve Solution

```
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;
}
```

Worst case: The candidates come in increasing quality order

$N(C_i + C_h) \approx O(NC_h)$

(i) 若 candidate 递增, cost 很大.

(ii) 若 candidate 随机分布:

random: i个candidates中 ∀一个为最优的可能性均为 $\frac{1}{i}$.

⇕

在面试第i个人时, 他比之前都优被雇佣的概率为 $\frac{1}{i}$.

X = number of hires

$$E[X] = \sum_{j=1}^{N} j \cdot \Pr[X = j] \quad ?$$

$$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is NOT hired} \end{cases} \quad \rightarrow E(X_i) = \frac{1}{i}$$

第i个人被雇佣的期望是 $\frac{1}{i}$

总雇佣期望为 $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}$

➡ $X = \sum_{i=1}^{N} X_i$

➡ $E[X] = E[\sum_{i=1}^{N} X_i] = \sum_{i=1}^{N} E[X_i] = \sum_{i=1}^{N} 1/i = \ln N + O(1)$ 封皮公式

➡ $O(C_h \ln N + NC_i)$

Radomized Algorithm

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

无论输入是什么分布, 先随机打乱

? takes time

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

随机打乱一个数组的算法:

? **Assign each element A[i] a *random priority* P[i], and sort**

随机权范围为 $1 \sim N^3$

```
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;
}
```

Online Hiring Algorithm:

假设1~N总共雇佣一次后结束, 雇完不再向后看.

Online Hiring Algorithm – hire only once

```
int OnlineHiring ( EventType C[ ], int N, int k)
{
   int Best = N;
   int BestQ = - ∞ ;
   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;
}
```

之后只要有比1~k中的best还优的就雇佣.

- What is the probability we hire the best qualified candidate for a given k?
- What is the best value of k to maximize above probability?

k应取几最好?

S_i: 第i个人最优且被雇佣. (希望事件S_i概率最大)

What needs to happen for S_i to be TRUE?

{ A:= the best one is at position i } ← independent

∩ { B:= no one at positions $k+1 \sim i-1$ are hired }

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

k+1~i-1都没1~k中最好的好.
即: 1~i-1中最好的在1~k中.

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

$$\leq \frac{k}{N}\int_{k-1}^{N-1}\frac{1}{x}dx = \frac{k}{N}\ln\left(\frac{N-1}{k-1}\right)$$

对k求导即知k应选几

2. Modified QuickSort

- **Deterministic Quicksort**
  - $\Theta(N^2)$ worst-case running time
  - $\Theta(N \log N)$ average case running time, *assuming every input permutation is equally likely*

体现了算法性能与输入数据分布有关.

(Deterministic Qsort)

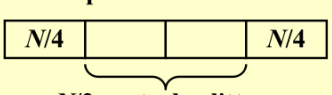
之前的QSort是首中尾取中位数, 现在采用central splitter

Central splitter := the pivot that divides the set so that each side contains at least $n/4$

Modified Quicksort := always select a central splitter before recursions

找到 central splitter 的复杂度期望为2. O(1)

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



Pr[*find a central splitter*] = 1/2 ✔

对于一个 Type j 的子问题 S 它的复杂度 |S|:

Type j : the subproblem S is of *type j* if $N\left(\frac{3}{4}\right)^{j+1} \leq |S| \leq N\left(\frac{3}{4}\right)^{j}$ (每次都划分成 $\frac{1}{4}:\frac{3}{4}$)

Claim: There are at most $\left(\frac{4}{3}\right)^{j+1}$ subproblems of *type j*.

$E[T_{type\ j}] = O(N\left(\frac{3}{4}\right)^{j}) \times \left(\frac{4}{3}\right)^{j+1} = O(N)$

Number of different types = $\log_{4/3} N = O(\log N)$

$O(N \log N)$

worst-case $O(N \log N)$

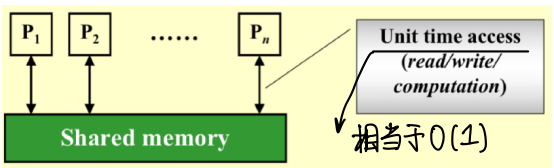
共有 $\log_{\frac{4}{3}} N$ 种 type

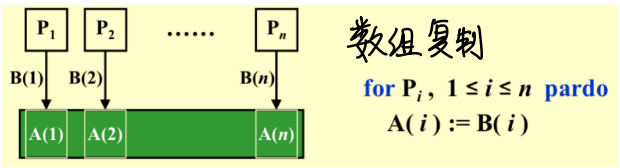
ADS 14 Parallel Algorithm

处理器并行：Machine parallelism： processor parallelism / pipeline / 长指令集

算法并行：Parallel algorithm 两个模型：{ Parallel Random Access Machine (PRAM)
Work-Depth (WD)

1. PRAM 模型





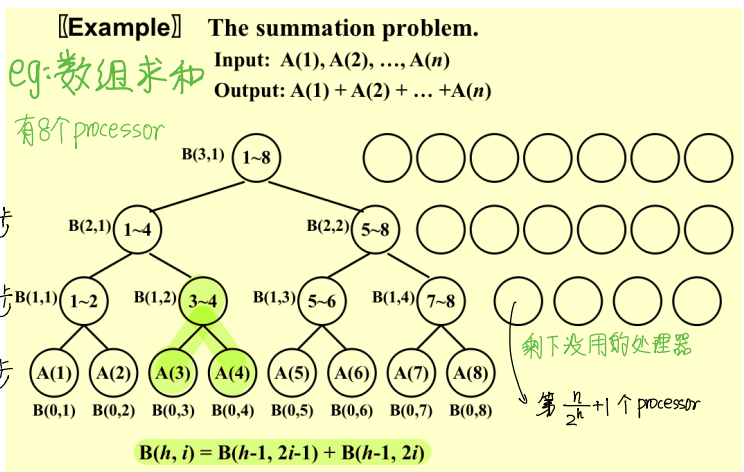
为避免内存 conflict，可以使用：① Exclusive-Read, Exclusive-Write (EREW) 排它读，排它写

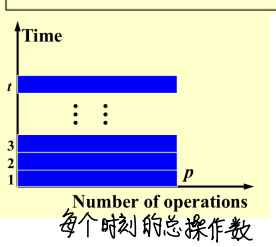
② Concurrent-Read, Exclusive-Write (CREW) 并发读，排它写

③ Concurrent-Read, Concurrent-Write (CRCW) 并发读，并发写

↓

内存冲突解决方案 { (i) arbitrary rule：随机选择一个 processor 让它写
(ii) priority rule：优先级高的 processor 让它写
(iii) common rule：只允许在都写相同值的时候并发写.





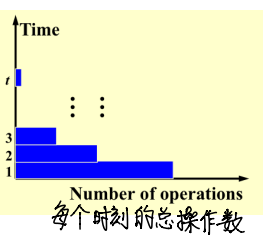
PRAM 两个缺点：

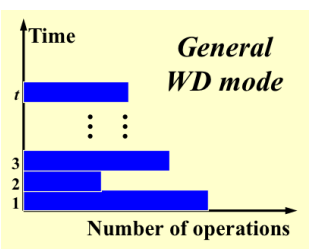
① 处理器数量变化时可迁移性差

② 所有处理器指令都要由程序员逐一分配，细节太多.

stay idle 也算是一种操作，也需要指令

2. WD 模型





2. Measuring the Performance

- Work load：总共多少步 $W(n)$ 吃得少
- Worst-case running time：运行时间 $T(n)$ 跑得快

① $W(n)$ 次操作，花费 $T(n)$ 时间

PRAM：

② $P(n)=\frac{W(n)}{T(n)}$ 个 processor，花费 $T(n)$ 时间

③ $\forall p(\leq \frac{W(n)}{T(n)})$ 个 processor，花费 $\frac{W(n)}{p}$ 时间　当 processor 数量不足 $\frac{W(n)}{T(n)}$（算法所需 processor 数），用时决定于 $\frac{W(n)}{p}$

④ $\forall p$ 个 processor，花费 $\frac{W(n)}{p}+T(n)$ 时间　当 processor 数量冗余

（以上4个渐近等价）

eg:

Work-Depth (WD) Presentation

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

$T(n)$	$W(n)$
1	n
$\log n$	$\frac{n}{2}+\frac{n}{4}+\cdots+1$
1	1

$\therefore T(n)=\log n+2$

$W(n)=2n$

[WD-presentation Sufficiency Theorem]

任何 WD 模型的 algorithm，用 $P(n)$ 个 processor，运行时间都至多为 $O(\frac{W(n)}{P(n)}+T(n))$

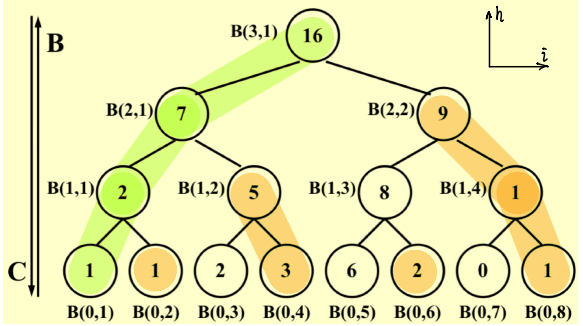
3. Example：Prefix-Sums 的算两趟算法.

：该坐标下变文：h+1，i/2

Input: A(1), A(2), …, A(n)

Output: $\sum_{i=1}^{1} A(i), \sum_{i=1}^{2} A(i), \cdots, \sum_{i=1}^{n} A(i)$

Technique: Balanced Binary Trees



每个节点 (h,i) 的 { B值：其子点之和（自底向上算）；C值：首个～其子树中最右边的叶子 之和（自顶向下算）

这种 balanced tree 有如下性质：

① 若 $i==1$，$C(h,i):=B(h,i)$

② 若 i 为偶数，$C(h,i):=C(h+1,\frac{i}{2})$

③ 若 i 为奇数且不为1，$C(h,i):=C(h+1,\frac{i-1}{2})+B(h,i)$

$:=C(h,i-1)+B(h,i)$

left path 上 $i==1$，C值 := B值

等比子树 right path 上 i 为偶数，C值 := 其父结点.C值

其他：C值 := B值 + 左边节点.C值 := B值 + 左边节点.的父节点.C值

算两趟之后，叶子的 C 值即为所求.

算两趟算法的代码：

$T(n)$　$W(n)$

```
for Pi , 1 ≤ i ≤ n pardo
  B(0, i) := A(i)
for h = 1 to log n 对每一层的节点
  for i , 1 ≤ i ≤ n/2^h pardo 用需要数目的Pi计算
    B(h, i) := B(h - 1, 2i - 1) + B(h - 1, 2i)
for h = log n to 0 对每一层的节点
  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 Pi , 1 ≤ i ≤ n pardo
  Output C(0, i)
```

向上算B

向下算C

用需要数目的Pi计算

1 n

log n $\frac{n}{2}+\frac{n}{4}+\cdots+1$

log n 还是每层每个都要算 与上方同

1 n

∴ $T(n)=O(\log n)$

$W(n)=O(n)$

4. Example: Merging

给定两个递增数组 A[ ]、B[ ]，将其 merge 成一个 C[ ].

Solution: Partitioning：将问题分成 p 个子问题，每个子问题并行求解

先将 Merging 问题转为 Ranking 问题:

本质是要知道 A[ ]中每个元素在 B[ ]中的排序位置，之后可直接放入C.
B[ ]中每个元素在 A[ ]中的排序位置，之后可直接放入C.

↓ Ranking 问题

Merging 问题 ⟺ 求出 RANK(*, B) 和 RANK(*, A)

假设 Ranking 可解，则 Merging 可在 $T(n)=O(1)$ 和 $W(n)=O(n+m)$ 求解

用此法直接放入

```
for Pi , 1 ≤ i ≤ n  pardo
   C(i + RANK(i, B)) := A(i)
for Pi , 1 ≤ i ≤ n  pardo
   C(i + RANK(i, A)) := B(i)
```

记：B[j] 在A中时，在 A[i] 和 A[i+1] 之间 ⟶ RANK(j,A) = i 数组自1起

在 $-\infty$ 和 A[n] 之间 ⟶ RANK(j,A) = 0

在 A[n] 和 $+\infty$ 之间 ⟶ RANK(j,A) = n

那么，Ranking 问题解法：

法一：

Binary Search

```
for Pi , 1 ≤ i ≤ n  pardo
   RANK(i, B) := BS(A(i), B)
   RANK(i, A) := BS(B(i), A)
```

n个binary search并行

$T(n)=O(\log n)$ < 遍历的 O(n)

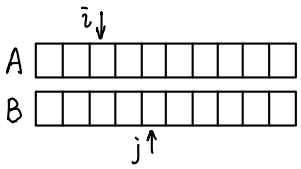
$W(n)=O(n\log n)$ > 遍历的 O(n)

跑得快但吃得也多，不够好.

法二：

Serial Ranking

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



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

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

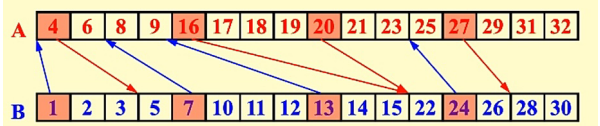
法三：Parallel Ranking

① 从 n 个元素找出 $p=\frac{n}{\log n}$ 个 select，每 log n 个设一个 select

• A_Select(i) = A(1+(i-1)log$n$) for $1 \le i \le p$
• B_Select(i) = B(1+(i-1)log$n$) for $1 \le i \le p$

↑ 等分段.

并计算它们的 RANK

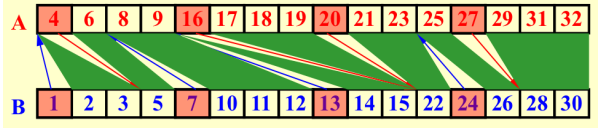


每两个箭头间分为一个 part

$T_1(n)=O(\log n)$

$W_1(n) = O(p \log n) = O(n)$

p个 binary search 并行



共有 2p 个 part，part 间已有序

② 2p 个 part 每个用 serial ranking

$T_2(n) = O(\text{part内元素数}) = O(\log n)$

$W_2(n) = 2p * O(\text{part内元素数}) = O(p \log n) = O(n)$

综合 1,2 . $T = O(\log n)$

$W = O(n)$

跑得快，吃得一样，好.

5. Example: Maximum Finding

法一：

将数组求和中的 + 改为 max 即可.

PRAM model

```
for Pi, 1 ≤ i ≤ n pardo          n个Pi并行 ↓                1
  B(0, i) := A(i)                Pi把A[i]存入B[0,i]         1
  for h = 1 to log n do          对第h个步骤
    if i ≤ n/2^h                 对一部分Pi                 } log n
      B(h, i) := B(h-1, 2i-1) + B(h-1, 2i)
    else stay idle               对另一部分Pi不动
  for i = 1: output B(log n, 1); for i > 1: stay idle       1
                  ↳ P1输出B[log n, 1]. 其它Pi不动
```

$T = O(\log n)$ $W = O(n)$

Work-Depth (WD) Presentation

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

	$T(n)$	$W(n)$
	1	n
	$\log n$	$\frac{n}{2} + \frac{n}{4} + \cdots + 1$
	1	1

$T = O(\log n)$ $W = O(n)$

法二：

```
for Pi, 1 ≤ i ≤ n pardo                    处理同值元素
  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 Pi, 1 ≤ i ≤ n pardo
  if B(i) == 0
    A(i) is a maximum in A
```

i 和 j ∈ [1,n] 需要 n^2 个 processor

$\frac{n(n-1)}{2}$

Pi 要存在比它大的，就将其的 B 置 1

并行时 Pi 涉及读操作，可使用 CRCW 的 arbitrary rule

$T(n) = O(1)$

$W(n) = O(n^2)$

大功率跑车算法.

法三：Doubly-logarithmic Paradigm

设 $h = \log\log n$

(i) 将 n 个元素划分为 $\sqrt{n}$ 个长 $\sqrt{n}$ 的段

$A_1[\,] = A[1], \cdots, A[\sqrt{n}]$ 用 Doubly-log 找最大 → $T(\sqrt{n}), W(\sqrt{n})$

$A_2[\,] = A[\sqrt{n}+1], \cdots, A[2\sqrt{n}]$ 用 Doubly-log 找最大 → $T(\sqrt{n}), W(\sqrt{n})$

$A_3[\,] = A[2\sqrt{n}+1], \cdots, A[3\sqrt{n}]$ 用 Doubly-log 找最大 → $T(\sqrt{n}), W(\sqrt{n})$

⋮

$A_{\sqrt{n}}[\,] = A[n-\sqrt{n}+1], \cdots, A[n]$ 用 Doubly-log 找最大 → $T(\sqrt{n}), W(\sqrt{n})$

pardo 共用 $T_1 = T(\sqrt{n})$, $W_1 = \sqrt{n}\, W(\sqrt{n})$

找到 max1, max2, max3, ⋮, max$\sqrt{n}$

之后从 max1, max2, … max$\sqrt{n}$ 中找最大 用大功率跑车找最大 → $T_2 = O(1)$, $W_2 = O((\sqrt{n})^2) = O(n)$

综合 1,2

$T(n) \le T(\sqrt{n}) + O(1)$

$W(n) \le \sqrt{n}\, W(\sqrt{n}) + O(n)$

解得 $\begin{cases} T = O(\log\log n) \\ W = O(n \log\log n) \end{cases}$

这里非并行

写 O(h) 即 T=W=O(h)

(ii) 将 n 个元素划分为 $\frac{n}{h}$ 个长 h 的段

$A_1[\,] = A[1], \cdots, A[h]$ 遍历找最大 → $O(h)$

$A_2[\,] = A[h+1], \cdots, A[2h]$ 遍历找最大 → $O(h)$

max1

$A_2[\,] = A[h+1], \cdots, A[2h]$ → 遍历找最大 → $O(h)$

$A_3[\,] = A[2h+1], \cdots, A[3h]$ → 遍历找最大 → $O(h)$

$\vdots$ $\quad$ $\vdots$ $\quad$ $\vdots$

$A_{n/h}[\,] = A[n-h+1], \cdots, A[n]$ → 遍历找最大 → $O(h)$

pardo 共用 $T_1 = O(h)$, $W_1 = \frac{n}{h} O(h)$

找到 $\max 1, \max 2, \cdots \max \frac{n}{h}$

$\max 1$

$\max 2$

$\max 3$

$\vdots$

$\max\ n/h$

之后从 $\max 1, \max 2, \cdots \max \frac{n}{h}$ 中找最大 → 用被标划分的 Doubly-log 找最大

$T_2 = O(\log\log \frac{n}{h})$

$W_2 = O(\frac{n}{h} \log\log \frac{n}{h})$

综合 1.2

$$T(n) \le O(h) + O(\log\log \tfrac{n}{h}) \Rightarrow T = O(\log\log n)$$

$$W(n) \le \tfrac{n}{h} O(h) + O(\tfrac{n}{h} \log\log \tfrac{n}{h}) \Rightarrow W = O(n)$$

法四：Random Sampling

n 个数

抽取 $n^{7/8}$ 个数 ↓ 每 $n^{1/8}$ 为一 block

pardo 共用 $T = O(1)$, $W = O(n^{7/8})$

$n^{7/8}$ 个数

$n^{6/8}$ 个 block

$n^{1/8}$	$n^{1/8}$	$n^{1/8}$	$n^{1/8}$	...	$n^{1/8}$	$n^{1/8}$	$n^{1/8}$

对每个 block 用大功率跑车，$T = O(1)$, $W = O((n^{1/8})^2)$ } 所有 block pardo 共用 $T = O(1)$, $W = n^{6/8} * O(n^{1/4}) = O(n)$

获得 $n^{6/8}$ 个 max i.

将 $n^{6/8}$ 个数每 $n^{1/4}$ 为一 block

$n^{6/8}$ 个数

$n^{1/2}$ 个 block

$n^{1/4}$	$n^{1/4}$	$n^{1/4}$	$n^{1/4}$	...	$n^{1/4}$	$n^{1/4}$	$n^{1/4}$

对每个 block 用大功率跑车，$T = O(1)$, $W = O((n^{1/4})^2)$ } 所有 block pardo 共用 $T = O(1)$, $W = n^{1/2} * O(n^{1/2}) = O(n)$

获得 $n^{1/2}$ 个 max i.

...

从而解出 $n^{7/8}$ 个数中最大的，但不是 n 个中最大的.

```
while (there is an element larger than M) {
    for (each element larger than M)
        Throw it into a random place in a new B(n^{7/8});
    Compute a new M;
}
```

定理：Random Sampling 以极高的概率可达 $T = O(1)$ 且 $W = O(n)$

↓

达不到该 T, W 的概率仅为 $O(\frac{1}{n^c})$

Merging:

Maximum Finding:

建立在 Ranking 基础上 $T=O(1)$
$W=O(n+m)$

(i) balanced binary tree 改为 max 两种模型 $T=O(\log n)$
$W=O(n)$

(ii) 大功率跑车 $T=O(1)$
$W=O(n^2)$

(iii) Doubly-log
- 按段长 $\sqrt{n}$ 划分 $T=O(\log\log n)$, $W=O(n\log\log n)$
- 按段长 $\log\log n$ 划分 $T=O(\log\log n)$, $W=O(n)$

(iv) Random Sampling $T=O(1)$
有 $\frac{1}{n^c}$ 概率达不到该表现. $W=O(n)$

Ranking:

Binary Search $T=O(\log n)$, $W=O(n\log n)$

Serial Ranking $T=O(n)$, $W=O(n)$

Parallel Ranking $T=O(\log n)$, $W=O(n)$

算 select 的 RANK 用 Binary Tree
算 part 内 RANK 用 serial Ranking

ADS15 External Sorting

不可能所有数据都在内存上，大量会在磁盘上，且 disk access 比 mem access 复杂得多.

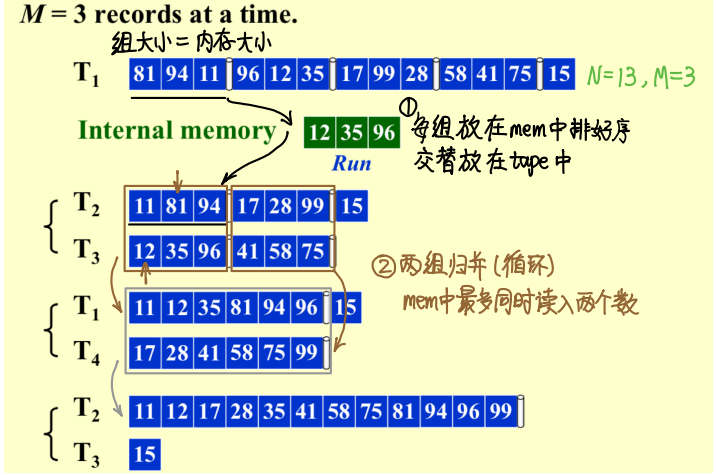
To get a[i] on

- ? internal memory – O(1)
- ? hard disk
  1. find the track; *device-dependent*
  2. find the sector;
  3. find a[i] and transmit.

run：一个已经有序的序列

本节以磁带为例，研究外部排序. tape 只能 sequentially access 且至少涉及3个 tape.

〖Example〗 Suppose that the internal memory can handle $M = 3$ records at a time.



累计对所有数据进行 I/O 次数 = 循环次数 = # pass

$\frac{\text{数据量}}{\text{内存大小}} = \#\,run$

数据：$T_1 \to \begin{matrix}T_2\\T_3\end{matrix} \to \begin{matrix}T_1\\T_4\end{matrix} \to \begin{matrix}T_2\\T_3\end{matrix} \to \begin{matrix}T_1\\T_4\end{matrix} \to \cdots$

① 1 pass

② $\lceil \log_2 \frac{\text{数据量}}{\text{内存大小}} \rceil$ pass

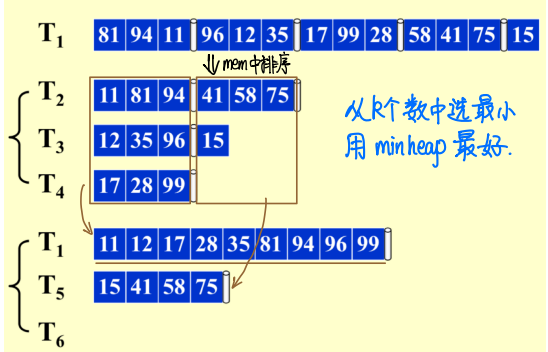
$$\#pass = 1 + \lceil \log_2 \frac{N}{M} \rceil$$

4个方面优化该方法：
- #pass 减少
- merge 加快
- 用 buffer 使能够 parallel
- run 更长

1. #pass 减少

用 k-way merge

Use a *k-way* merge!



k-way 归并. 用 2k 个 tape

数据：$T_1 \to \begin{matrix}T_2\\T_3\\T_4\end{matrix} \to \begin{matrix}T_1\\T_5\\T_6\end{matrix} \to \begin{matrix}T_2\\T_3\\T_4\end{matrix} \to \begin{matrix}T_1\\T_5\\T_6\end{matrix} \to \cdots$

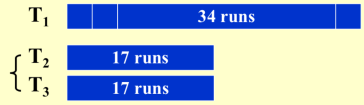
① 1 pass

② $\lceil \log_3 \frac{\text{数据量}}{\text{内存大小}} \rceil$ pass

$$\#pass = 1 + \lceil \log_k \frac{N}{M} \rceil$$

这种当然可以，但要用 2k 个 tape. 能减少吗？

Can we use 3 tapes for a 2-way merge?

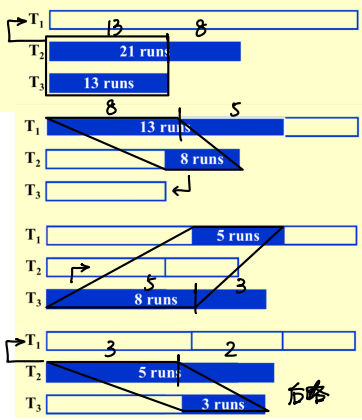


以 2-way merge 用 3 个 tape 为例

之前是均分的，34 → 17, 17

现在 unevenly 更好.

? A smarter way – split *unevenly*



n个短run, n个中run } merge → 更长的 n 个 run

run数为 Fibonacci 数 F_N，则分为 F_{N-1}, F_{N-2} 最好.

对 k-way merge，分法服从 k-way Fibonacci 数.

$F^{(k)}_{0\sim k-2} = 0,\ F^{(k)}_{k-1} = 1,\ F^{(k)}_N = F^{(k)}_{N-1} + \cdots + F^{(k)}_{N-k}$

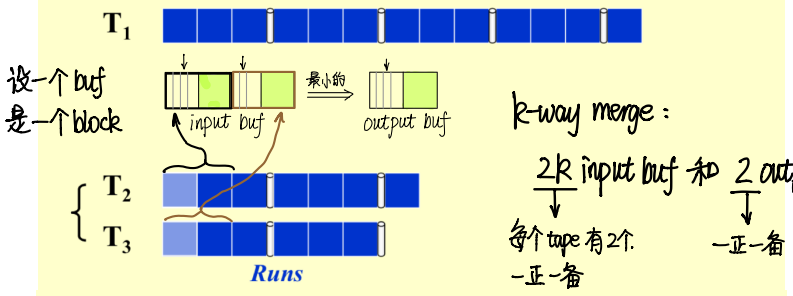
只要 k+1 个 tape

2. handle buffers to parallel

在最初的方法中，我们串行做：内存input，比大小，小的output，内存input，比大小，小的output，……
其间内存I/O开销很大．能否用parallel将I/O隐藏起来？ ⟶ 用buffer控制I/O.

〖Example〗 Sort a file containing 3250 records, using a computer with an internal memory capable of sorting at most 750 records. The input file has a block length of 250 records.

内存I/O基本单位



k-way merge：

2k input buf 和 2 output buf

每个tape有2个，一正一备　　一正一备

备用buf在正buf被merge时就准备好之后的数据．
正buf读完时，立即继续从备buf读．

内存一定，内存划分为多个buf
则每个buf size↓

k ⟶ # of input buffers ⟶ buffer size

seek time ⟵ block size on disk

吃一小口要咬准则更慢　　block size↓ 每次吃一小口

k↑ { pass数 $1+\lceil \log_k \frac{N}{M} \rceil$ ↓ ; buffer handle I/O量 ↑ } k值tradeoff

3. 更长的initial run　replacement selection

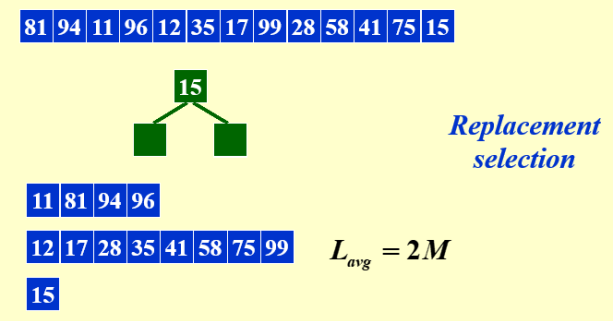
若能让run更长，则run数减少．显然能减少seek time.
之前是将mem大小 这么多个数读入mem，排序后作为一个run．run长度至多是mem大小.

replacement selection

① 将全部mem组织成一个minheap，初始时用tape中数补满它.

② 循环：pop min，放入run，从tape读入数 { 若能放入run，正常插入堆.
若不能放入run，将其放在堆底记为死点，(其他点重新组合保持order property)

〔当堆全为死点，开新run并将堆全复活并重新构成order property〕



这样，平均run长为 2 * mem大小．当原本已nearly sorted会更好.

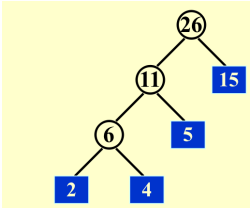
4. minimize merge

〖Example〗 **Suppose we have 4 runs of length 2, 4, 5, and 15, respectively. How can we arrange the merging to obtain minimum merge times?**

merge 开销 = 被 merge 的 run 长之和

怎样 minimize? Huffman Tree

越长的 run 越晚 merge，depth 越浅



$2\times\underline{3}+4\times\underline{3}+5\times\underline{2}+15\times\underline{1}=43$

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

根到叶路径.

$O(\sum$ external path 长 $*$ 叶深$)$

不同 merge 顺序会有不同 merge 开销.