ADS PART2

> 从这里开始，课程就不再讨论经典、传统的算法了。会涉及一些比较前沿的算法研究，以及比较复杂的复杂度分析

NP-COMPLETENESS NP完全性

Undecidable Problem

不是所有问题都可以由计算机求解的。Undecidable Problem指的就是那些无法设计算法来求解的问题

Example: Halting Problem

It is impossible for a compiler to detect all infinite loops. 计算机无法完美检查自己是否陷入死循环

Proof：假设存在下面这样一个死循环检查程序F，我们构建如下程序，让他调用Loop(Loop)

```
void Loop( P )
{
  if( F(P) )
    return;
  else
    while (true); // 死循环
}
```

- 假设F(Loop)认为Loop是死循环，则if条件成立，正常return，不是死循环；
- 假设F(Loop)认为Loop不是死循环，则else成绩，进入死循环

有很多问题都是没有算法进行求解的，Halting Problem只是其中比较有代表性的一个

Poly-Time Reductions 多项式时间化约

Introduction

> 因为在下面介绍NP-Completeness问题时需要用到这个概念，所以先在此介绍

假设我们有一个算法A用于解决问题X。现在给出一个问题Y，如果每个具体的Y问题都可以由多项式数个的步骤 + 多项式数个A的调用，那我们就说Y is polynomial-time reducible to X，写作$Y \leq_P X$

Example: HCP & TSP

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

- **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 $\leq K$?

想要证明TSP是NPC的，需要证明$\text{HCP} \leq_P \text{TCP}$。因此我们需要设计一个多项式时间的方法$f()$实现$G(V,E) \to G'(V',E')$的映射：在一张图中寻找Hamilton环，等效于将该图每一条边都设置为1，并在该图上解决TSP问题

The Class NP

确定性图灵机

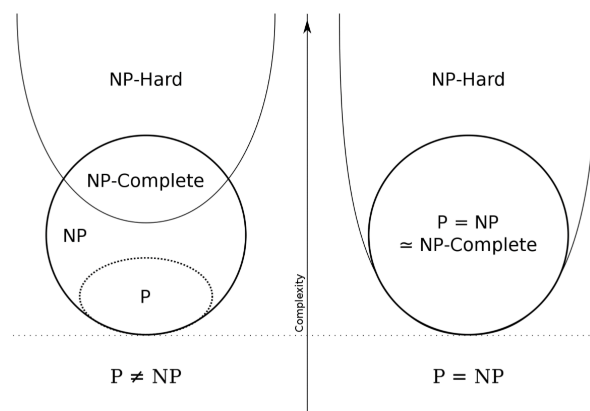
1. **Deterministic Turing Machine**：确定性图灵机。每次执行一条指令，并且这条指令的运行结果指向唯一确定的下一条指令
2. **Nondeterministic Turing Machine**：非确定图灵机。下一步可以在有限集合中自由选择，每个步骤都会带来不同的结果，并且图灵机最终总是会选择正确的那个

NP

1. **P**：Polynomial Time，可以用确定型图灵机在多项式时间内解决
2. **NP**：Nondeterministic Polynomial Time，给出一个问题的可能解，用确定性图灵机可以在多项式时间内验证。等价于可以用非确定图灵机在多项式时间内求解（$\text{P} \in \text{NP}$）
3. **NP-Complete**：NP中最难的决定性问题，满足如下条件
    - 是一个NP问题
    - Any problem in NP can be polynomial reduced to it. 所有NP问题都可以多项式化约为它（因此所有NPC问题难度相同）
4. **NP-Hard**：不要求是NP问题，所有NP问题都可以多项式化约为NPH问题

P = NP吗?

目前尚无法证明P与NP的关系，如果P = NP，则所有NP问题都有多项式算法；如果我们可以找到NPC的多项式解法，就可以证明P = NP。对于上面几个概念，我们有如下关系图：



Formal-Language Framework

Abstract Problem

An abstract problem Q is a binary relation on a set I of problem instances and a set S of problem solutions. 抽象问题：从实例集合到答案集合的二元映射

> **Example**：PPT上的两个例子毫无营养，逻辑降智但表述形式出奇复杂，所以被迫写在这里防止看不懂这种表示形式

> 1. **Shortest-Path Problem**
>
>   输入一个无向图$I = \{< G, u, v >: G = (V, E)\}$，其中起点和终点$u, v \in V$
>
>   输出$S = \{< u, w_1, w_2, \ldots w_k, v >:< u, w_1 >, \cdots < w_k, v > \in E\}$
>
>   对于每一个$i \in I$，最小路径$\mathrm{SP}(i) = s \in S$
>
> 2. **Decision Path Problem**
>
>   给出一个无向图$I = \{< G, u, v, k >: G = (V, E)\}$，其中起点和终点$u, v \in V$，路径长度不超过$k$
>
>   输出$S = 0, 1$，如果有路径满足要求，则返回1，否则给0
>
>   对于每一个$i \in I$，我们都有$\mathrm{PATH}(I) = 1 \text{ or } 0$

Encoding

Map I into a binary string $\{0, 1\}^*$. 把I映射到一个01序列中

Language

> 其实这里的绝大多数定义后面都用不上，我也不知道为什么会出现

- **Alphabet** $\sum$：字母表，一个有限字符的集合
- **Language** L：字母表$\sum$所构造的字符串集合
- **Empty String** ε：空字符串
- **Empty Language** $\emptyset$：空语言。不包含任何字符串的语言
- $\sum^*$：$\sum$中所有可能字符串的集合（这是一个无限的集合，因为字符串可以无限延伸，但每个字符串的个数是有限的）
- **Complement of** L：$\overline{L} = \sum^* - L$。也就是除去$L$以外剩下所有可能的Language
- **Concatenation of** L_1 **and** L_2：$L = \{x_1 x_2 : x_1 \in L_1, x_2 \in L_2\}$
- **Kleene Star of Language** L：$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$。其中$L^k$表示将$L$自身连结$k$次

Accept & Decide

- **接受字符串**：如果$A(x) = 1$，则我们认为算法A接受了字符串$x \in \{0, 1\}^*$（accept）
- **拒绝字符串**：反之如果$A(x) = 0$，我们则认为其拒绝了字符串x（reject）
- **接受语言**：L中的所有字符串都被A接受
- **决定语言**：因为语言L是一个字符串的集合。我们约定，如果L中的每一个字符串都可以被algorithm A接受，并且不在L中的所有字符串都被A拒绝，则称L由A决定，L is decided by A

P：多项式时间内决定的语言

P是一个集合，包含所有能被某算法在多项式时间内decide的语言。或者说，能在多项式时间内decide的语言属于P

$$P = \{L \subseteq \{0, 1\}^* : \text{ there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\} \tag{1}$$

Verify

1. **Verification Algorithm & Certificate**：Verification algorithm是一个2-argument的算法，其中一个为ordinary input string x，另一个为certificate y
2. **验证字符串**：如果输入x满足$A(x, y) = 1$，则称algorithm A verifies the input string x
3. **验证语言**（类似决定）：如果语言L被算法A验证，则其包含所有可以被算法A验证的字符串

$$L=\{x\in\{0,1\}^*: \text{there exists } y\in\{0,1\}^* \text{ such that } A(x,y)=1\} \tag{2}$$

NP：存在certificate可以在多项式时间内验证语言

如果对于语言L中的每一个字符串都可以在多项式的时间内被验证，则该语言属于NP

$$L=\{x\in\{0,1\}^*:\ \text{there exists a certificate } y \text{ with } |y|=O(|x|^c) \text{ such that } A(x,y)=1\} \tag{3}$$

多项式规约

1. **Polynomial-Time Reducible**：如果存在一个多项式时间的函数f，将语言L_1中的所有字符串转化为L_2中的字符串，则认为L_1可以多项式规约到L_2

   A language L_1 is polynomial-time reducible to a language L_2 ($L_1 \le_P L_2$) if there exists a polynomial-time computable function $f:\{0,1\}^* \to \{0,2\}^*$ such that for all $x\{0,1\}^*, x\in L_1 \iff f(x)\in L_2$
2. **Reduction Function**：方程f
3. **Reduction Algorithm**：用于计算方程f的多项式算法

NP-Complete：所有NP语言都可以多项式规约为它

在语言中对于NPC的定义和常规定义是平行的，还是2个要求。我们称语言$L\subseteq\{0,1\}^*$ 为NP-complete，则：

- $L\in \mathrm{NP}$ L本身就属于NP
- $L'\le_P L$ for every $L'\in\mathrm{NP}$ NP语言可以多项式规约为L

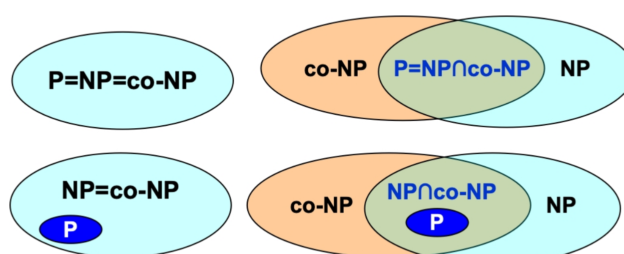
The Class Co-NP

定义

1. **语言定义**：Complexity class co-NP is the set of languages L that $\overline{L}\in\mathrm{NP}$。其中$\overline{L}$是$L$的补语言，即$\overline{L}=\sum^*-L$
2. **问题定义**（PPT上没提，和语言定义平行，作辅助理解用）：Co-NP is the set of decision problems X such that $\overline{X}\in\mathrm{NP}$。其中$\overline{X}(s)=1 \iff X(s)=0$

Co-NP与NP的四种关系

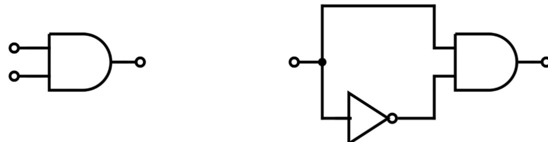
我们并不清楚NP与Co-NP的关系，但有一点是确定的，即$\mathrm{P}\subseteq\mathrm{NP}\cap\text{co-NP}$



Example

SAT

Circuit Satisfiability Problem，是最早被证明为NPC的问题。给定布尔电路，是否有能够使输出为真的输入分配（比如说在下图中左边有，但右边没有）



> 因为所有算法都可以被转化为电路。进一步说，任何一个n bits输入、输出为0或1、运行时间为$T(n)$的算法，都可以被转化为大小为$p(T(n))$的电路。其中$p(\cdot)$为一个多项式函数
>
> 基于这点，我们可以证明任何问题$Y \in \mathrm{NP}$，都可以简化为SAT

3-SAT

3-SAT是SAT的一个特例，对布尔电路（或者说布尔表达式）施加特殊要求（其中的变量允许重复和取非）：

$$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge \cdots \wedge (x_{n-2} \vee x_{n-1} \vee x_n) \tag{4}$$

Clique Problem & Vertex Cover Problem

假设已知clique problem为NP-complete，证明vertex cover problem也NP-complete

- **Clique Problem**：给定一个无向图$G = (V, E)$和一个整数K，求问G是否包含一个有K个节点的完全子图（clique）？

$$\text{CLIQUE} = \{< G, K >: G \text{ is a graph with a clique of size } K\} \tag{5}$$

- **Vertex Cover Problem**：给定一个无向图$G = (V, E)$和一个整数K，找出（不超过）K个节点，这些节点组成的集合K'为K的子集，满足图中的每一条边都经过其中的某个节点（称为vertex cover）

$$\text{VERTEX-COVER} = \{< G, K >: G \text{ has a vertex cover of size } K\} \tag{6}$$

Proof：先证明VC是NP问题，然后再利用多项式规约证明VC是NPC问题

1. $\text{VERTEX-COVER} \in \text{NP}$：找到一个多项式时间的算法验证VC问题

   给定任何一个$x =< G, K >$，我们将$V' \subseteq V$作为certificate y，则验证算法为：

   - 检查节点数是否为K，check if $|V'| = K$
   - 检查是否每条边都包含V'中的节点，Check if for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$

   复杂度为$O(N^3)$

2. $\text{CLIQUE} \leq_P \text{VERTEX-COVER}$：将CLIQUE规约到VERTEX-COVER问题上

   证明：G has clique of size $K \iff \overline{G}$ has a vertex cover of size $|V| - K$

   - 充分性：易证
   - 必要性：所有边都连着至少一个vertex cover中的点，也就是说，没有边是两头都连着vertex cover以外的点的，因此补图就是一个完全图

   映射的复杂度为$O(N^2)$，因为要将G转化为$\overline{G}$需要遍历每一条边（设想一个图的邻接矩阵）

考点

这一章着实比较抽象，所以基本上也都只能考一些概念性质的问题，下面给出一些比较容易错的问题。

值得注意的是，在证明NP-complete时应首先关注其是否属于NP问题，如果不是NP问题就不用证了

1. **多项式复杂度**：If a problem can be solved by dynamic programming, it must be solved in polynomial time. (F)

很多动态规划问题无法在多项式时间内解决，比如背包问题

2. **不可判定问题**：All the languages can be decided by a non-deterministic machine. (F)

   忽略了不可判定问题

APPROXIMATION

Introduction

默认近似算法要求多项式复杂度，因此题目中提到α-approximation时，认为其可以在多项式时间内解决

Approximation Ratio

- **Approximation Ratio**：如果一个算法的approcimation ratio为m，则近似算法最坏不会超过最优解的m倍。如果是最大化算法，则近似算法的答案在最坏不会少于最优解的$1/m$；如果是最小化算法，则近似算法的答案最坏不会超过最优解的m倍
- **α-approximation**：approximation ratio = α

Approximation Scheme

- **Approximation Scheme**：如果有一种近似算法，输入的不仅是问题的实例，还外加一个精度$\varepsilon > 0$，算法可以保证给出的答案是满足$(1+\varepsilon)$-approximation的
- **PTAS**：Polynomial-Time Approximation Scheme，可以在多项式时间内解决。其复杂度允许$1/\varepsilon$不为多项式复杂度，$O(n^{f(1/\varepsilon)})$的复杂度是可以接受的（$f(x)$一般是一个增函数，当ε趋向0时复杂度迅速上升）
- **FPTAS**：Fully Polynomial-Time Approximation Scheme，要求$1/\varepsilon$也是多项式复杂度的，即复杂度为$O((1/\varepsilon)^{m_1}n^{m_2})$

PTAS和FPTAS告诉我们，近似算法可以是任意接近最优解的，不一定被常数限制

Bin Packing Problem

Introduction

给定N个物品，大小分别为$S_1, S_2, \ldots, S_N$，且$0 < S_i \leq 1$。将所有的物品打包到最少的桶里（每个桶的大小为1）

Online Solutions

在线算法就是每次读入一个元素（不能提前看之后的元素），确定其位置，且之后不能反悔。其最好不会好于$5/3M$（M为最优解所需要bin的个数）

1. **Next Fit**

   有一种非常简单的思想，每次取一个物品，如果可以放进当前的桶，就直接放入；如果不行，就直接开新桶

```
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;
  }
}
```

可以证明，这个算法最坏不会使用超过$2M-1$个桶（假设最优解是M）

Proof：反证法，假设使用了$2M$个桶，尝试证明最优解不可能少于$M-1$个

注意到，开新桶的必要条件是之前的桶放不下当前元素，即$S_i > 1 - \text{Size_of}(\text{Current Bin})$，因此相邻2个桶的大小一定大于1。假设$S(B_i)$ 为第i个bin的大小，我们有：

$$
\begin{aligned}
S(B_1) + S(B_2) > 1 \\
S(B_3) + S(B_4) > 1 \\
\cdots \\
S(B_{2M-1}) + S(B_{2M}) > 1 \\
\Rightarrow \sum_{i=1}^{2M} S(B_i) > M
\end{aligned}
\tag{7}
$$

2. **First Fit**

   这种算法的思想也很直观。在上一个算法的基础上，每次拿出一个物品，搜索所有bins，能装进则装进，如果所有都不能则开新bin

```
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;
  }
}
```

   本方法的上界为$1.7M$，并且可以在$O(N\log N)$时间内解决（搜索树或大根堆）

3. **Best Fit**：选择能放入且剩余空间最小的，上界以及时间复杂度与first fit相同

Offline Solution

离线算法现将所有元素读入，在开始确定位置前可以对整个输入进行查看

First Fit Decreasing（Best Fit Decreasing）：在开始前对所有物品进行排序，然后使用first fit。其最坏情况不超过$11M/9+6/9$个桶

Knapsack Problem

Introduction

1. **0-1 Version**：给定一个容量为M的背包，N个物品，物品重量为w_i，价值为p_i。对于这个问题，如果只采用以性价比最高为原则的贪心，最坏结果可以是任意坏的

2. **Fractional Version**：在这个版本中，假定物品是可拆分的，每个物品可取x_i的比例。在这样的条件下，以性价比最高为原则的贪心算法可以找到最优解

Solution to the 0-1 Version

1. **Double Greedy**：做2次贪心，分别采用价值最高（maximum profit）和性价比最高（profit density）为原则，然后取两者间的更优解。在这样的条件下，可以保证approximation ratio = 2（也可写作2-approximation）

> 注意这个double greedy的名字是我自己取的，PPT上没有类似概念

2. **Proof**：假设可以放进的价值最高的物品为$p_{\max}$，最优解为P_{OPT}，分数版本的最优解为P_{frac}

$$\begin{aligned} p_{\max} &\le P_{\text{OPT}} \le P_{\text{frac}} \\ p_{\max} &\le P_{\text{double_greedy}} \\ P_{\text{OPT}} &\le P_{\text{frac}} \le P_{\text{double_greedy}} + p_{\max} \\ \Rightarrow P_{\text{OPT}} &\le 2P_{\text{double_greedy}} \end{aligned} \tag{8}$$

3. **DP Solution**：在之前的动态规划中我们提到了动态规划的一种解，即对于物品i，如果选择，则转化为重量限制为$w - w_i$，物品选择范围为1到$i-1$下的最优解；如果不选择，则问题转化为重量限制为w，物品选择范围为1到$i-1$下的最优解。不难得出复杂度为$(n+1)\times(w+1)$

   这里，我们引入一种新的动态规划思路。$W_{i,p}=$ the minimum weight of a collection from $\{1,\dots,i\}$ with total profit being exactly p. 之前考虑重量限定下的最高价值，现在考虑给定价值下的最低重量。对于每件商品，我们都可以选取或不选取。如果选取，则$W_{i,p} = w_i + W_{i-1,p-p_i}$；如果不，则$W_{i,p} = W_{i-1,p}$；如果无法达到价值$p$，则令$W_{i,p}=\infty$。由此，我们可以得到如下递推式：

$$W_{i,p} = \begin{cases} \infty, & i = 0 \\ W_{i-1,p}, & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\}, & \text{otherwise} \end{cases} \tag{9}$$

   宽泛地说，i可以从1一直到n，而p可以从1一直到$np_{\max}$（这个条件是放松过的，但PPT上就是这么写的额，实际上上界取$\sum p$就足够了），因此复杂度为$O(n^2 p_{\max})$

   当$p_{\max}$较大时，如果还强行计算，本算法的复杂度就相当糟糕了。因此，我们采用一种近似的方法，将所有物品的价值都向上取整（比如仅保留到万位，这样后四位均为0，都可以扔掉。相当于将所有数同时降低了4个数量级，当然，精度也损失了4位）。如此，我们可以得到一个上界：

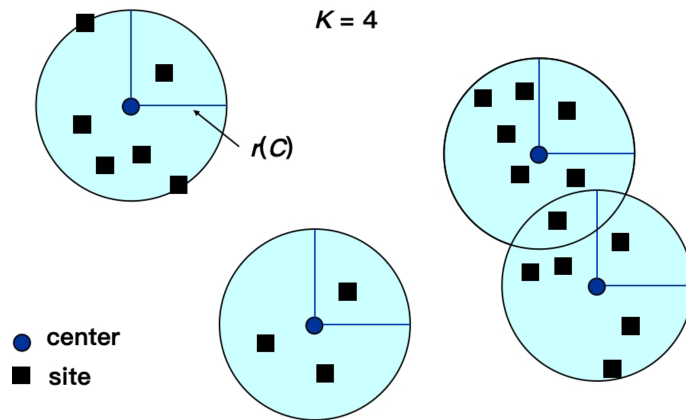
$$(1+\varepsilon)P_{\text{alg}} \le P \text{ for any feasible solution } P \tag{10}$$

   其中ε为precision parameter（精度位）

K-center Problem

Introduction

Input a set of n sites $s_1,\dots s_n$, select K centers C so that the maximum distance from a site to the nearest center is minimized. 输入一组site的位置，算法的任务是找到K个center（定义这K个center的集合为C），保证site到center的最大距离最小



Solution

解决这个问题最重要的理论基础是，两边之和小于第三边（或两点之间线段最短）

$$\mathrm{dist}(x,y) \le \mathrm{dist}(x,z) + \mathrm{dist}(z,y) \tag{11}$$

1. **Traditional Greedy Solution**：有一种贪心算法是，将第一个center放在只能选择一个center情况下，半径最小的地方。然后逐个放点，保证每个增加的点都让covering radius减小值最大

   这样的方法找到的解是可以任意坏的。如下，假定题目给定的是两团点（如下左图），那么第一个插入的点在最中间（如下右图），因为当只插入一个点时要保证半径最小，只能插入在中间。只要两团点间的距离任意大，这个算法的解就任意坏了

2. **给定最优半径，每次以$2r^*$为半径作圆**：我们将问题转化一下，假定我们已经知道最优解为$r(C^*)$（只知道半径，但不知道center到底在哪里），我们可以寻找出一个近似解。该算法可以保证2-approximation，即找到一个最大半径不超过$2r(C^*)$的center的集合

   思路是，随便取一当前未被包含的点作为center，并作半径为$2r$的圆，并将圆内的所有点都视为包含；以此类推

   根据题目，我们并不知道最优解是什么，但可以采用binary search的方法进行猜测。先猜测一个值r_0，如果太小，就猜测$r_1=2r_0$；太大，就猜测$r_1=1/2r_0$。然后进行第二轮猜测，如果介于两者之间，就猜$1/2(r_0+r_1)$，如果还是太小或太大，就猜$1/2r_1$或$2r_1$。以此类推

```
Centers_Greedy_2r( Sites S[], int n, int K, double r ) {
  Sites S'[] = S[]; // S'是当前未被包含的site的集合
  Centers C[] = empty_set;
  while( S'[] != empty_set ) {
    Select any s from S' and add it to C;
    Delete all s' from S' that are at dist( s', s ) <= 2r;
  }
  if( |C| <= K )
    return C;
  else
    ERROR(No set of K centers with covering radius at most r); // 如果找到K个中心后还是有
未包含的点，说明输入的最小半径错误
}
```

证明：该算法一定能找到一个满足$r(C) \le 2r(C^*)$的解。即，如果算法无法用K个center、$r=2r(C^*)$覆盖所有点，则输入的最优解不成立

对于任意一个我们选择的center，其一定被最优解center*包含，即，其一定与某center*的距离不超过r^*。在考虑该center*所能够包含的所有点，与其距离亦不超过r^*。因此，与我们选择的center包含在同一个center*中的点，与center的距离一定不超过$2r^*$

3. **改进：每次选择最远的点为center**。相比选择$2r^*$之外的任意一点作为下一个center，可以选择距离当前center最远的点作为下一个center。这种方法也是2-approximation的，不过不需要维护S'的集合

```
Centers_Greedy_Kcenter( Sites S[], int n, int K ) {
  Centers C[] = empty_set;
  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;
  }
  return C;
}
```

可以证明，除非P = NP，否则这个问题不存在$\rho < 2$的ρ-approximation的近似解

> **证明思路**：如果可以获得一个$(2-\varepsilon)$-approximation的算法，则DOMINATING-SET问题可以在多项式时间内解决（是一个NPC问题）。我们假定一个特殊的输入，所有点的距离均为整数，并且距离要么为1，要么很大。可以证明，DOMINATING-SET Problem有解当且仅当$r(C^*)=1$，那么$2>2-\varepsilon=1$（因为所有距离均为整数，所以近似算法也给应出一个整数解，但又要保证approximation ratio小于2，因此只能为1了），即近似算法必须返回最优解

总结

算法需要考虑三个方面

1. **Optimality**：Quality of a solution
2. **Efficiency**：Cost of computations
3. **All Instances**：对于所有实例都需要有效

Optimality + All Instances：所有实例下都最优，也就是精准的算法

Optimality + Efficiency：在特定输入下，找到高速且准确的解

Efficiency + All Instances：近似算法

注意，即便P = NP，我们仍无法保证同时做到上面三个方面

考点

1. Suppose ALG is an α-approximation algorithm for an optimization problem $\prod$ whose approximation ratio is tight. Then for every $\varepsilon > 0$ there is no $(\alpha-\varepsilon)$-approximation algorithm for $\prod$ unless P = NP. (F)

   注意阅读理解，这个tight指的是可以取到α，即在该算法下这是最严谨的范围

2. **和NP内容结合考察**：As we know there is a 2-approximation algorithm for the Vertex Cover problem. Then we must be able to obtain a 2-approximation algorithm for the Clique problem, since the Clique problem can be polynomially reduced to the Vertex Cover problem. (F)

   虽然多项式规约可以保证复杂度，但无法保证近似精度。因此本问题的关键就是，在VC问题上，2-approximation算法的解，在Clique上是否仍满足2-approximation?

这道题目比较简单，因为前者为最小问题（假设答案为V_1），后者为最大问题（假设答案为V_2）；VC问题的近似算法最坏为$2V_1$，这个结果规约到Clique上就是$V-2V_1$，显然无法保证$< 1/2V_2$

LOCAL SEARCH 局部搜索

Introduction

对于很多问题，我们难以找到一个全局的最优解。但可以随便找一种情况，然后每次都基于这个情况进行改善单步的改善，直到任何单步的改变都无法让当前情况变得更好为止。相当于是通过gradient descent梯度下降找到了一个极值点

一些定义

1. **Local**
   - **Neighborhood**：Define neighborhoods in the feasible set. 在可行集中定义neighborhood
   - **Local Optimal**：The best solution in a neighborhood
   - $S \sim S'$：S' is a neighboring solution of S. S'可以在S的基础上作小改进达到
   - $N(S)$：Neighborhood of S. 可以表示为S'的集合，即$\{S' : S \sim S'\}$
   - FS：Feasible Solution Set
2. **Search**：Start with a feasible solution and search a better one within the neighborhood. A local optimal is achieved if no improvement is possible

Neighbor Relation

由此，Local Search算法可以表示为如下形式：

```
SolutionType Gradient_descent() {
  Start from a feasible solution S in FS; // FS: Feasible Solution Set
  MinCost = cost( S );
  while( true ) {
    S’ = Search( N(S) ); // 找到最佳的S'
    CurrentCost = cost( S’ );
    if ( CurrentCost < MinCost ) {
      MinCost = CurrentCost;
      S = S’;
    }
    else
      break;
  }
  return S;
}
```

Vertex Cover Problem

Introduction

给定一个无向图$G=(V,E)$，找到尽可能少的顶点，保证每条边都与我们选中的点相连。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 in S.

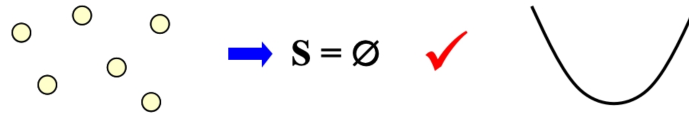
- FS：所有顶点

- $\mathrm{cost}(S)$：$|S|$
- $S \sim S'$：每个顶点集合S最多有$|V|$个neighbor。我们定义从S中删去一个顶点为S'，而S中最多有$|V|$个顶点
- **Search**：从$S = V$开始，每次删去一个顶点，判断其是否依旧保证vertex cover

3 Cases

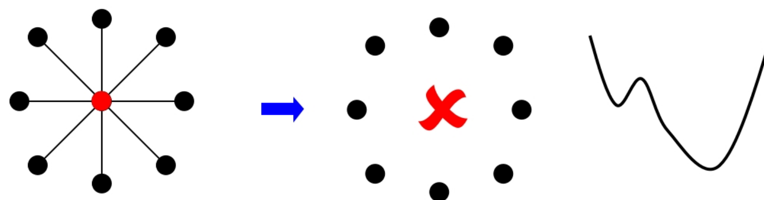
1. **Local Sarch = 最佳解**

   如果$|E| = 0$，则Local Search的解就是最佳解



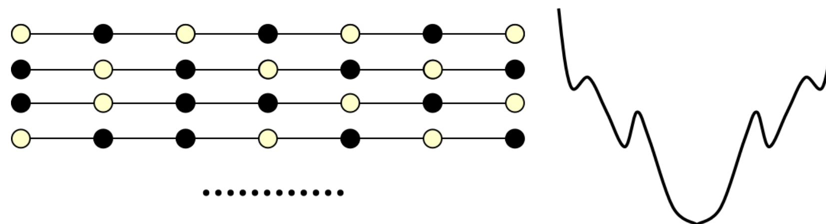
2. **2个极值点**

   对于下面这种实例，如果中间的点被删除，会导致比较坏的情况



3. **多个极值点**

   下面这样的输入存在很多极值点，因此算法很可能会陷入其中任何一个，难以判断最终结果



Improvement: The Metropolis Algorithm

设定一个概率函数，和结果的变坏程度正相关。每次随机选择一个点（不要求查询那个最好的点），如果没有使结果变得更好，但也没有变坏太多，概率函数依旧处于阈值内，继续尝试

```
SolutionType Metropolis() {
  Define constants k and T;
  Start from a feasible solution S in FS ;
  MinCost = cost( S );
  while( true ) {
    S’ = Randomly chosen from N(S); // 不要求找到最佳的S'，随便选
    CurrentCost = cost( S’ );
    if( CurrentCost < MinCost ) { // 如果这个随便选的刚好更好，那就采用
      MinCost = CurrentCost;
      S = S’;
    }
    else { // 如果随便选的并没有更好，但没有变坏很多，则继续；如果变坏很多，认为当前已经是最好的
      With a probability e ^ ( -delta_cost / ( k * T ) ), let S = S’;
      else
        break;
    }
  }
  return S;
```

```
}
```

Simulated Annealing 模拟退火

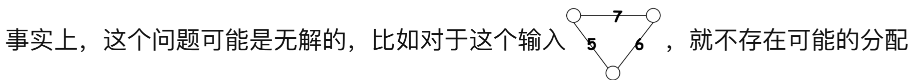
如果我们想要找到一个使目标函数$f(x)$最小化的解。我们假想$f(x)$是一个类似于降温的过程，随着温度下降，其趋势变得更为缓慢，但距离最小值可能还有一定距离。因此想要找到最优解，不能简单地判断Δcost的绝对变化

我们让上面的Metropoli算法中的T随着程序的进行逐渐减小，这样可以抵消cost变化放缓带来的影响。T可以设定为一个数列$T=\{T_1,T_2,...\}$，我们称之为cooling schedule

Hopfield Neural Networks

Introduction

1. **问题介绍**：给定一个图$G=(V,E)$，每个节点都只有± 1两种状态，每条边都有一个权重w_e（$e=(u,v)$，w_e允许为负），其绝对值$|w_e|$代表所连接两点的strength。如果一条边的两个节点相同，则希望该边权重为负；反之则为正。给定图的形状和所有边的权重，输出一个网络的configuration配置S，即每个节点的正负分配，满足同负异正

   事实上，这个问题可能是无解的，比如对于这个输入 

   ，就不存在可能的分配

   因此，我们的目标变为找到一个足够好的解

2. **基本概念**
   - **Good**：如果边$e=(u,v)$满足$w_e s_u s_v<0$，则称其为good
   - **Bad**：$w_e s_u s_v\geq 0$
   - **Satisfied**：节点u所发出的good边的权重大于等于bad边的权重$\sum_{v:e=(u,v)\in E} w_e s_u s_v\leq 0$
   - **Stable**：所有节点都satisfied的configuration。任何一个网络都存在stable的configuration

State-flipping Algorithm

从任意一个点开始，将不satisfied的点取负。可以证明，这种方法可以在有限步内找到答案

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

证明：State-flipping算法最多循环$W=\sum_e|w_e|$次

我们设定一个函数$\Phi(S)=\sum_{e\text{ is good}}|w_e|$。每当我们翻转一个顶点$u$时，所有与其相连的好边都变为坏边，坏边都变为好边。由于翻转前该点不满足satisfied，因此翻转后的好边一定多于坏边

$$\begin{aligned}\Phi(S')=\Phi(S)-&\sum_{e:e=(u,v)\in E,\ e\text{ is bad}}|w_e|+\sum_{e:e=(u,v)\in E,\ e\text{ is good}}|w_e| \\ \geq \Phi(S)+1&\end{aligned} \tag{12}$$

又因为$0\leq\Phi(S)\leq\sum_e|w_e|$，有下界也有上界，所以就算从最小加到最大也不会超过W次循环

Polynomial Algorithm?

显然，这个算法的复杂度是基于W的，不是多项式复杂度。对于构建stable网络的问题，n的多项式复杂度或$\log W$复杂度的算法还未找到

Maximum Cut Problem

Introduction

1. **问题介绍**：给定一个无向图$G=(V,E)$，边的权重w_e为正整数。将图切成两半，使穿过切割面的边权重加和最大（或者说，找到一个节点的分割node partition (A,B)，让A和B相连的边权重总和最大）

$$w(A,B)=\sum_{u\in A,\, v\in B} w_{uv} \tag{13}$$

2. **应用**
   - n个活动，m人，每人安排2项活动，每项活动安排在上午或下午，如何最大限度保证同时享受2项活动的人数
   - 电路布局等

Local Search

- **Feasible Solution Set** FS：任意partition (A,B)
- $S\sim S'$：S'可以由将A中的任意一点挪到B，或B中任意一点挪到A得到

我们发现，其实这就是一个Hopfield Neural Network的特例（+1的节点集视为A，-1的节点集视为B），w_e全为正

State Flipping Algorithm

此处对于MAX-CUT问题的local search算法和Hopfield神经网络的完全一致。显然这种算法无法保证找到最优解，但可以证明它是2-approximation的

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

证明：假设(A^*,B^*)是全局最优解，则$w(A,B)\geq 1/2w(A^*,B^*)$

因为(A,B)是局部最优解，因此，对于任意一点$u\in A$，为保证satisfied，从它发出、连向B（符号相反）中边的权重和 > 从它发出、连向A（符号相同）中边的权重和

$$\forall u\in A, \sum_{v\in A} w_{uv}\leq \sum_{v\in B} w_{uv} \tag{14}$$

将所有$u\in A$加总：

$$
\begin{aligned}
2\sum_{\{u,v\}\subseteq A} w_{uv} &= \sum_{u\in A}\sum_{v\in A} w_{uv} \\
&\le \sum_{u\in A}\sum_{v\in B} w_{uv} = w(A,B) \\
\Rightarrow w(A^*,B^*) &\le \sum_{\{u,v\}\subseteq A} w_{uv} + \sum_{\{u,v\}\subseteq B} w_{uv} + w(A,B) \\
&\le 2w(A,B)
\end{aligned}
\tag{15}
$$

值得注意的是，MAX-CUT问题存在更好的近似算法

Improvement

因为state flipping算法是非多项式时间的，为了提高算法运行效率，当提升不够大时就停止。Only choose a node which, when flipped, increase the cut value by at least

$$
\frac{2\varepsilon}{|V|}w(A,B) \tag{16}
$$

这样可以保证算法是$(2+\varepsilon)$-approximation的，时间复杂度为$O(n/\varepsilon\log W)$（上面那个提升公式记2ε）

局部搜索的优化

1. **Neighborhood要求**
   - **不宜过小**：为保证不陷入较坏的局部最优，neighborhood不宜过小
   - **不宜过大**：如果neighborhood过大，搜索的消耗过大，算法效率过低
2. **改进**
   - **Single-flip $\rightarrow$ k-flip**：每次都选择k个进行翻转，选择其中最优的。搜索neighborhood时间变为$\Theta(n^k)$
   - **K-L Heuristic**：进行n次连续的single-flip，直到$(A_n,B_n)=(B,A)$（第k次single-flip的复杂度为$O(n-k+1)$，因此总的复杂度就是一个等差数列求和）。复杂度为$O(n^2)$，Neighborhood其实就是在n步中选择一个最优的，即$(A,B)=\{(A_1,B_1),\dots(A_{n-1},B_{n-1})\}$

考点

1. **Search Space**：Local search algorithm can be used to solve lots of classic problems, such as SAT and N-Queen problems. Define the configuration of SAT to be X = vector of assignments of N boolean variables, and that of N-Queen to be Y = positions of the N queens in each column. The sizes of the search spaces of SAT and N-Queen are $O(2N)$ and $O(N^N)$, respectively. (F)

   Search Space指的是所有可能，也就是暴力解所需要遍历的情况数
2. **局部最优与全局最优**：Spanning Tree Problem. Given an undirected graph $G=(V,E)$, where $|V|=n$ and $|E|=m$. Let F be the set of all spanning trees of G. Define $d(u)$ to be the degree of a vertex $u\in V$. Define $w(e)$ to be the weight of an edge $e\in E$.
   We have the following three variants of spanning tree problems:
   - Max Leaf Spanning Tree: find a spanning tree $T\in F$ with a maximum number of leaves.
   - Minimum Spanning Tree: find a spanning tree $T\in F$ with a minimum total weight of all the edges in T.
   - Minimum Degree Spanning Tree: find a spanning tree $T\in F$ such that its maximum degree of all the vertices is the smallest.

For a pair of edges (e, e') where $e \in T$ and $e' \in (G - T)$ such that e belongs to the unique cycle of $T \cup e'$, we define `edge-swap( e, e' )` to be $(T - e) \cup e'$.

Here is a local search algorithm:

```
T = any spanning tree in F;
while( there is an edge-swap( e, e' ) which reduces Cost( T ) ) {
  T = T - e + e';
}
return T;
```

Here `Cost(T)` is the number of leaves in `T` in Max Leaf Spanning Tree; or is the total weight of `T` in Minimum Spanning Tree; or else is the minimum degree of `T` in Minimum Degree Spanning Tree.

Which of the following statements is TRUE?

A. The local search always return an optimal solution for Max Leaf Spanning Tree

B. The local search always return an optimal solution for Minimum Spanning Tree

C. The local search always return an optimal solution for Minimum Degree Spanning Tree

D. For neither of the problems that this local search always return an optimal solution

对于局部优化问题的实例，首先需要注意的是$S \sim S'$是如何达到的。在本问题中，S'由S置换一条边达到。A和C找到反例不难，主要问题在于如何快速证明B是正确的。

我们假设，在某种情况下，更换任何一条边都不会让生成树更小，但更换若干条边后生成树最终变得更小。考虑第一个使生成树变得更小的那次替换e''，如果e''替换的是之前步骤未被替换的边e，则在第一步就可以用e''替换e，假设不成立；如果e''替换的是之前步骤替换的边e'，因为e'未使生成树变小，即$e' > e$（假设e为e'替换掉的边），因此在第一步时可以直接用e''替换e，同样矛盾

3. **Max-cut的改进**： Given an undirected graph $G = (V, E)$ with positive integer edge weights w_e, find a node partition (A, B) such that $w(A, B)$, the total weight of edges crossing the cut, is maximized. Let us define $S\prime$ be the neighbor of S such that S' can be obtained from S by moving one node from A to B, or one from B to A. We only choose a node which, when flipped, increases the cut value by at least $w(A, B)/|V|$. Then which of the following is true?

   A. Upon the termination of the algorithm, the algorithm returns a cut (A, B) so that $2.5w(A, B) \geq w(A^*, B^*)$, where (A^*, B^*) is an optimal partition.

   B. The algorithm terminates after at most $O(\log |V| \log W)$ flips, where W is the total weight of edges.

   C. Upon the termination of the algorithm, the algorithm returns a cut (A, B) so that $2w(A, B) \geq w(A^*, B^*)$.

   D. The algorithm terminates after at most $O(|V|^2)$ flips.

   这个问题现场推导比较复杂，关注现成的结论

RANDOMIZED ALGORITHMS 随机算法

Introduction

定义

相比总是产生正确答案的deterministic algorithm，randomized algorithm给出一个结果和它的概率，它无需总是找到最优解，也并非给出一个和最优解接近的近似解，而是返回一个有很大概率为最优解的可能解（其实deterministic算法可以视为randomized算法的一种概率始终为1的特殊情况）

一些概念

1. $\Pr[A]$：事件A发生的概率
2. $\overline{A}$：事件A的complementary，即A不发生的概率

$$\Pr[A] + \Pr[\overline{A}] = 1 \tag{17}$$

3. $E[X]$：随机变量X的期望

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] \tag{18}$$

Hiring Problem

Introduction

需要招聘一个人，一共N位面试者，每次面试一名。面试需要的成本为C_i，雇佣需要的成本为C_h（$C_i \ll C_h$，即，忽略面试成本C_i）。我们需要雇佣最佳的人选，并尽可能降低开销

对于这个问题的算法，我们不研究复杂度，而是研究雇佣的成本，因此下面出现的$O(f(N))$都是指需要花掉的雇佣成本，而给算法运行的时间

Naïve Solution

1. **Introduction**：在面试之前我们无法确定面试者的能力，因此这一定是一个在线算法。最简单的想法就是每个人都面一遍，一旦找到比之前更好的就雇佣

```
int Hiring( Event_Type C[], int N) {
  int Best = 0;
  int Best_Q = the quality of candidate 0;
  for( int i = 1; i <= N; i++) {
    Q_i = interview[i];
    if( Q_i > Best_Q ) {
      Best_Q = Q_i;
      Best = i;
      hire( i );
    }
  }
  return Best;
}
```

2. **Worst Case**：最坏情况就是候选人的能力升序排列，此时每个人都需要雇佣一遍，需要的开销为$O(NC_h)$
3. **Average Cost**：按照正常的思路，我们假设X为雇佣过的总人数。这样的问题是，$\Pr[X]$很难计算。因此，我们作如下定义：

$$\begin{aligned} X_i &= \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is not hired} \end{cases} \\ \Rightarrow X &= \sum_{i=1}^{N} X_i \end{aligned} \tag{19}$$

如此，$\Pr[X_i=1]$就代表了第i位候选人被雇佣的概率 $\Longleftrightarrow$ 第i位候选人在前i人中是最大的。因为前i人中一定有一个是最大的，而其可以在任意位置，因此，$\Pr[X_i=1]=1/i$

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{N} X_i] = \sum_{i=1}^{N}\frac{1}{i} \\ &\in \left(\int_1^{N+1}\frac{1}{i}\mathrm{d}i, \int_0^{N}\frac{1}{i}\mathrm{d}i\right) \\ &= \ln N + O(1) \end{aligned} \tag{20}$$

因此，开销为$O(C_h \ln N + NC_i)$

Randomized Algorithm

随机算法和之前的常规算法在思路上没有任何区别，只是在开始前将候选人随机打乱了。这一步会增加运行时间，但可以避免因为候选人升序排列导致开销过大。换句话说，这个操作不能让平均情况变得更好，也不能让算法运行变快，但可以让最坏情况变得平均

```
int Hiring( Event_Type C[], int N) {
  int Best = 0;
  int Best_Q = the quality of candidate 0;

  randomly permute the list of candidates; // 随机打乱候选人的顺序

  for( int i = 1; i <= N; i++) {
    Q_i = interview[i];
    if( Q_i > Best_Q ) {
      Best_Q = Q_i;
      Best = i;
      hire( i );
    }
  }
  return Best;
}
```

Randomized Permutation Algorithm 随机打乱算法

给每位候选人都随机生成一个随机权重random priority $P[i]$，然后排序

```
void Permute_By_Sorting( Elem_Type A[], int N ) {
  for( int i = 0; i <= N; i++ ) {
    A[i].P = 1 + rand() % ( N ^ 3 );
    Sort A, use P as sort keys;
  }
}
```

Hire Only Once

经典的选对象问题，选择最前面的一部分作为观察样本，不管多好直接刷。然后进行选择，只要遇到比之前观察区更好的就选中并结束

```
int Online_Hiring( Event_Type C[], int N, int k ) {
  int Best = N;
  int Best_Q = -infty;
  for( int i = 1; i <= k; i++ ) {
    Q_i = interview[i];
```

```
    if( Q_i > Best_Q )
      Best_Q = Q;
  }
  for( int i = k + 1; i <= N; i++ ) {
    Q_i = interview[i];
    if( Q_i > Best_Q ) {
      Best = i;
      break;
    }
  }
  return Best;
}
```

现在的问题是，我们应如何确定k，让选中最优候选人的概率最大？假设S_i表示第i位候选人最优，我们有：

$$\Pr[S_i] = \{A := \text{the best one is at position } i\} \cap \{B := \text{no one at positions } [k+1, i-1] \text{ are hired}\} \tag{21}$$

注意到，事件A和B是相互独立的。其中，B等效于从$k+1$到$i-1$中没有比前k个中最大的更大的，也就是前$i-1$中最大的在前k个。前$i-1$个中的最大元素一共有$i-1$个位置可能出现，而我们希望它出现在其中的k个中，因此$\Pr[B] = k/(i-1)$

$$\begin{aligned} \Pr[S_i] &= \Pr[A \cap B] \\ &= \Pr[A] \cdot \Pr[B] \\ &= \sum_{i=k+1}^{N} \frac{1}{N} \cdot \frac{k}{i-1} \\ &= \frac{k}{N} \sum_{i=k}^{N-1} \frac{1}{i} \\ &\in \left(\frac{k}{N} \ln \frac{N}{k}, \frac{k}{N} \ln \frac{N-1}{k-1} \right) \end{aligned} \tag{22}$$

Randomized Quicksort

Introduction

回顾一下快排，每次选一个pivot，将两边的与其比较；递归向下。worst-case为$\Theta(N^2)$，average-case为$\Theta(N \log N)$（假设每种输入序列都是等可能的）

> 黄色PPT上这一部分非常令人迷惑，所以我先按照老师上课讲的版本写，复习的时候如果需要用到黄色PPT的内容再补

随机算法：每次随机选择一个pivot

Analysis

对于快排，时间消耗来自互换（互换基于比较，所以本质上就是比较的次数；而比较只会在pivot和其他元素间发生。换句话说，如果两个元素被pivot分开了，则它们间不会发生比较）

根据每次选择的pivot，可以画出一颗二叉搜索树。我们有一个重要的观察，就是被一个pivot分到两边的两个元素不会发生比较；即在二叉树中，任意一个节点的左右孩子间不会发生比较，仅有祖先和后代会发生比较

> 假设$a_1 < a_2 < \cdots < a_n$，则a_7和a_8发生比较的概率为1（因为它们中间没有额外的元素，递归到最后时，一定会在它俩中间选一个作为pivot），a_1和a_n发生比较的概率为$2/n$（当且仅当一开始选择了a_1或a_n时会发生比较，只要选择了中间任意一个元素，a_1和a_n就会被分开）

由此，a_i和a_j被比较的概率为：

$$\Pr[a_i \text{ and } a_j \text{ are compared}] = \frac{2}{j-i+1} \tag{23}$$

因此，比较次数的预期为：

$$\begin{aligned} E[\text{compare}] &= \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{2}{j-i+1} \\ &= 2\sum_{i=1}^{N}\sum_{j=2}^{N-i+1}\frac{1}{j} \\ &\leq 2N\sum_{j=1}^{N}\frac{1}{j} \\ &\leq 2N(\ln N+1) \\ &= O(N\log N) \end{aligned} \tag{24}$$

因此，随机算法和传统算法的随机时间复杂度是一样的，都是$O(N\log N)$

考点

1. **坑题**： Let $a=(a_1, a_2, \ldots, a_i, \ldots, a_j, \ldots, a_n)$ denote the list of elements we want to sort. In the quicksort algorithm, if the pivot is selected uniformly at random. Then any two elements get compared at most once and the probability of a_i and a_j being compared is $2/(j-i+1)$ for $j>i$, given that a_i or a_j is selected as the pivot. (F)

   没说序列a是有序的，厉害吧

PARALLEL ALGORITHMS 并行计算

Introduction

Machine Parallelism 并行计算机

并行计算机通过并行计算、流水线和超长指令实现，但这是硬件上的实现，不是ADS要讨论的内容

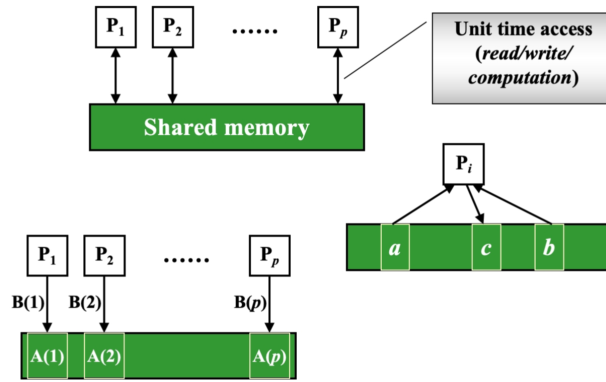
Parallel Algorithms Analysis 并行算法分析

- **PRAM**
- **Work-depth Measurement**

PRAM

Introduction

Parallel Random Access Machine。认为读、写、计算都是一个单位时间，不考虑通讯（读写）和计算的时间差别



Resolving Access Conflicts

严格的方法，缺点是分析过程比较坐牢，因为要一个个数

如果发生同时读或同时写，会发生冲突，有如下三种解决方案：

- **EREW**：Exclusive-Read Exclusive-Write。不能同时读，也不能同时写
- **CREW**：Concurrent-Read Exclusive-Write。允许同时读，不能同时写
- **CRCW**：同时读和同时写均可。同时写时只选择其中一个进行写，其余失效。有如下三种规则：
    1. **Arbitraty Rule**：任意选一个
    2. **Priority Rule**：选择下标最小的进程执行
    3. **Common Rule**：如果所有进程都写同样的内容，其实也就不存在冲突了

Work-depth Measurement

一种近似的方法，仅考虑并行算法树的work和depth，具体过程参考Summation Problem中的分析部分。这是本章节的分析基本均采用这种方法

- **Work**：$W = T_1$，the total amount of unit-time operations required to complete this algorithm. 整个算法线性跑完所需要的时间，也就是树中的节点数
- **Depth**：$D = T_\infty$，the length of the longest chain of sequential dependencies. 并行算法所需要的最短时间（也就是树高，因为父子不能并行，必须按先后顺序跑）

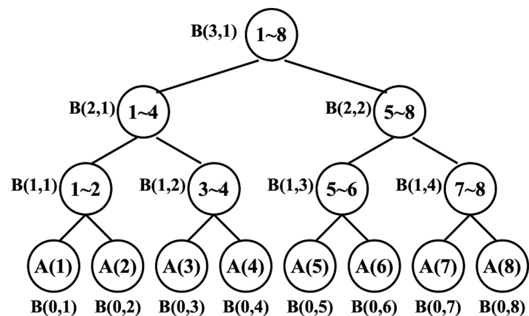
Summation Problem

Introduction

给定一个序列$A(1), A(2), \dots, A(n)$，求$\sum_{i=1}^{n} A(i)$。简单点说，就是将n个数加起来。如果不采用并行的处理，复杂度显然是$\Theta(n)$

Parallel Solution

如下，我们构建一棵二叉加法树，先计算相邻两个数的加和，递归向上



$$B(h,i) = B(h-1, 2i-1) + B(h-1, 2i) \tag{25}$$

```
for( i = 1; i <= n; i++ ) // pardo (parallel-do)
  B[0][i] = A[i];

for( h = 0; h <= log( n ); h++ )
  for( i = 1; i <= n / (2 ^ h); i++ ) // pardo
    B[h][i] = B[h - 1][2 * i - 1] + B[h - 1][2 * i];

return B[log n][1];
```

如果我们有多于4块处理器（$p \geq 4$），则该算法的每一层都只消耗一个单位的时间，此时复杂度为$T_{p\geq 4} = \Theta(\log n)$；但如果只有1块处理器（$p = 1$），还是得将树中的每个节点都一个个跑过来，此时复杂度为$T_1 = \Theta(n)$

但是，如果我们想要获得任意p下的复杂度T_p呢?

Work-depth Measurement

$$\begin{aligned} \frac{W}{p} &\leq T_p && \text{假设没有一个处理器是浪费的} \\ T_p &\leq \frac{W}{p} + D && \text{当处理器多于可并行执行的任务时，需要的时间由}D\text{决定；} \\ & && \text{当处理器不够用时，没有处理器是浪费的，时间由}\frac{W}{p}\text{决定} \end{aligned} \tag{26}$$

在summation problem中，$W = \Theta(n)$，$D = \Theta(\log n)$，因此，我们有：

$$\Theta(\frac{n}{p}) \leq T_p \leq \Theta(\frac{n}{p} + \log n) \tag{27}$$

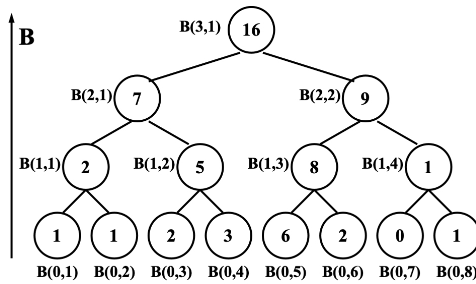
Prefix-Sums

Introduction

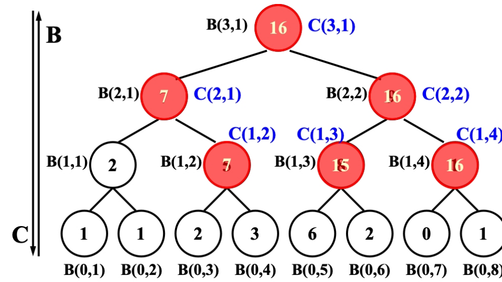
输入一个序列$A(1), A(2), \ldots, A(n)$，输出前k个数的和$\sum_{i=1}^{1} A(i), \sum_{i=1}^{2} A(i), \ldots, \sum_{i=1}^{n} A(i)$

Solution

1. **自底向上计算加和**：执行Summation Problem算法（得到的结果记为B）



2. **自顶向下计算从$B(0,1)$到当前节点所能连到的最右叶子的加和**：结果记为C。$C(h,i)=\sum_{i=1}^{a}A(k)$, where $(0,a)$ is the rightmost decendant leaf of node (h,i).



对于$C(h,i)$，有如下结论：

$$C(h,i)=\begin{cases}B(h,i) & \text{if } i=1 & i\text{为每一层最左边的节点}\\ C(h+1,i/2) & \text{if } i\%2=0 & i\text{是右孩子时，它就等于其父节点}\\ C(h+1,(i-1)/2)+B(h,i) & \text{else} & i\text{是左孩子且不是最左边的那个时，}\\ & & \text{是其父节点的左兄弟与其的加和}\end{cases} \tag{28}$$

发现每一层的C要么基于上一层的C，要么基于该层的B，没有基于同一轮h循环中计算结果的，因此可以并行处理

> 生成C的过程虽然看上去比较复杂，但其实只要搞清楚定义，结合图像还是比较好理解的

```
for( i = 1; i <= n; i++ ) // pardo
  B[0][i] = A[i];
for( h = 0; h <= log( n ); h++ )
  for( i = 1; i <= n / (2 ^ h); h++ ) // pardo
    B[h][i] = B[h - 1][2 * i - 1] + B[h - 1][2 * i];
// 运行Summation Problem

for( h = log( n ); h >= 0; h-- )
  C[h][1] = B[h][1];
  for( i = 2; i <= n / (2 ^ h)) { // pardo
    if( i % 2 == 0 )
      C[h][i] = C[h + 1][i / 2];
    else
      C[h][i] = C[h = 1][(i - 1) / 2] + B[h][i];
  }
for( i = 1; i <= n; i++ )
  cout << C[0][i] << " ";
```

WD分析

$$\begin{aligned}D&=O(\log n)\\ W&=O(n)\end{aligned} \tag{29}$$

Merging

Introduction

合并两个升序排列的序列$A[n]$和$B[m]$，依旧升序。为了简化问题，作如下约定：

1. A和B中的元素两两不同
2. $m = n$
3. $\log n$和$n/\log n$都是整数

Solution

假设我们已经知道了A中每一个数在B中的位置，以及B中每一个树在A中的位置，那就可以算出其在合并后序列C中的位置（注意这里还是默认元素从1开始）

令$\mathrm{RANK}(j, A) = k$表示A中比$B(j)$小的元素个数，也就是说，如果将$B(j)$插入A，应当插入在$A(k)$和$A(k+1)$之间。如此，我们可以计算$A(i)$和$B(j)$在C中的位置：

$$\begin{aligned} C(i + \mathrm{RANK}(i, B)) = A(i) \\ C(j + \mathrm{RANK}(j, A)) = B(j) \end{aligned} \tag{30}$$

因此，我们的任务转化为求A和B中每一个数的RANK。这也不难，只需要把一个序列中的每一个数都在另一个序列中做一次Binary Search就可以。下面给出两个找RANK的方法

1. **Solution1：Binary Search

   ```
   for( i = 1; i <= n; i++ ) {
     RANK_B[i] = BS( A[i], B );
     RANK_A[i] = BS( B[i], A );
   }
   ```

   因为每个元素的搜索搜索互不影响，所以可以并行处理。如果处理器够多，可以同时搜索每个元素

   $$\begin{aligned} D = O(\log n) \\ W = O(n \log n) \end{aligned} \tag{31}$$

2. **Solution2：Serial Ranking**

   和传统算法一致，每次从比较A和B中当前的元素，将更小的那个拿出来

   ```
   i = j = 0;
   while( i <= n || j <= m ) {
     if( A[i + 1] < B[j + 1] )
       RANK_B[++i] = j;
     else
       RANK_A[++j] = i;
   }
   ```

   这个算法无法并行处理，因为i和j事实上是逐个增大的，不能跳过前面的结果并行执行，因此

   $$W = D = O(n + m) \tag{32}$$

   发现当处理器比较少时，这个最笨的算法很可能比并行处理更快

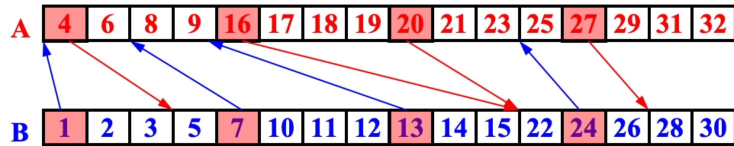
3. **Solution3：Parallel Ranking**

   这个算法在求RANK的时候不是将每个元素都分别进行搜索，而是先找出几个分割点，求分割点的RANK；则在分割点之间的元素的RANK一定也在两个分割点的RANK之间

   - **Stage1：Partitioning**

     每$\log n$选择一个分割点，确定其RANK，则一共选出$p = n/\log n$个点

$$
\begin{aligned}
\text{A_Select}(i) &= A(1+(i-1)\log n) \\
\text{B_Select}(i) &= B(1+(i-1)\log n)
\end{aligned} \tag{33}
$$

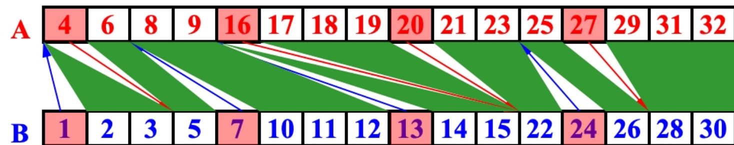


D和之前一样，依旧是单个搜索的时间；W不同，因为此步骤不需要搜索所有节点，只需要p个即可

$$
\begin{aligned}
D &= O(\log n) \\
W &= O(p\log n) = O(n)
\end{aligned} \tag{34}
$$

- **Stage2：Actual Ranking**

  在分割线之间做最笨的merge



  一共$2p$条线，将A、B之间分为$2p$个部分，每个部分不会超过$O(\log n)$（因为每隔$\log n$必然会发出一条线）。因此，D和W和上一步一样

综上，我们得到了一个兼具并行算法和传统算法优势的算法

$$
\begin{aligned}
D &= O(\log n) \\
W &= O(p\log n) = O(n)
\end{aligned} \tag{35}
$$

Maximum Finding

问题很简单，就是找到一个序列中最大的数

Solution1：Summation Algorithm

只需要将之前的求和问题中的"+"改为"max"即可，整个过程就好比打淘汰赛。D和W均不变

$$
\begin{aligned}
D &= O(\log n) \\
W &= O(n)
\end{aligned} \tag{36}
$$

Solution2：比较任意两个

循环赛，将每两个都进行比较，只要输过就说明不是最大的，最后只需要找出没输过的即可。这样虽然会提高W，但因为每个比较都是独立的，所以D是常数级别的

```
for( i = 1; i <= n; i++ ) // pardo
  B[i] = 0;
for( i = 1; i <= n; i++ ) // pardo
  for( j = i + 1; j <= n; j++ ) { // pardo
    if( A[i] <= A[j])
      B[i] = 1;
    else
      B[j] = 1;
  }
for( i = 1; i <= n; i++ )
  if( B[i] = 0 )
    return A[i];
```

$$
\begin{aligned}
D &= O(1) \\
W &= O(n^2)
\end{aligned} \tag{37}
$$

是否有一种分割方式，可以在具有此算法低D值的同时，将W值也降低到可以接受的程度呢?

Solution3：A Doubly-logarithmic Algorithm

> 所谓的doubly-logarithmic，指的是本算法的复杂度为$h = \log\log n$（因此$n = 2^{2^h}$）

1. **Partition by $\sqrt{n}$**：按间隔为$\sqrt{n}$进行分组，$D(n) = O(\log\log n), W(n) = O(n\log\log n)$

$$
\begin{aligned}
M_1 &= \max\{A(1), A(2), \ldots, A(\sqrt{n})\} & \Rightarrow \quad & D(\sqrt{n}), W(\sqrt{n}) \\
M_2 &= \max\{A(1+\sqrt{n}), \ldots, A(2\sqrt{n})\} & \Rightarrow \quad & D(\sqrt{n}), W(\sqrt{n}) \\
\ldots & & \ldots & \\
M_{\sqrt{n}} &= \max\{A(n-\sqrt{n}+1), \ldots, A(n)\} & \Rightarrow \quad & D(\sqrt{n}), W(\sqrt{n})
\end{aligned} \tag{38}
$$

   然后再使用Solution2循环赛的算法，从$\sqrt{n}$个M中找出最大的那个

$$
A_{\max} = \max\{M_1, M_2, \ldots, M_{\sqrt{n}}\} \quad \Rightarrow \quad D = O(1), W = O(\sqrt{n}^2) = O(n) \tag{39}
$$

   因此，递推式如下：

$$
\begin{aligned}
& D(n) \le D(\sqrt{n}) + O(1) \qquad (1) \\
\Rightarrow\ & D(n) = O(\log\log n) \\
& W(n) \le D(\sqrt{n}) + O(1) \qquad (2) \\
\Rightarrow\ & W(n) = O(n\log\log n)
\end{aligned} \tag{40}
$$

2. **Partition by $h = \log\log n$**：将partition的参数换为$\log\log n$，$D(n) = O(\log\log n), W(n) = O(n)$

$$
\begin{aligned}
M_1 &= \max\{A(1), A(2), \ldots, A(h)\} & \Rightarrow \quad & O(h) \\
M_2 &= \max\{A(1+h), \ldots, A(2h)\} & \Rightarrow \quad & O(h) \\
\ldots & & \ldots & \\
M_{n/h} &= \max\{A(n-h+1), \ldots, A(n)\} & \Rightarrow \quad & O(h)
\end{aligned} \tag{41}
$$

   最后一步合并采用partition by $\sqrt{n}$的方法实现

$$
A_{\max} = \max\{M_1, M_2, \ldots, M_{n/h}\} \quad \Rightarrow \quad D = O(\log\log\frac{n}{h}), W = O(\frac{n}{h}\log\log\frac{n}{h}) \tag{42}
$$

   因此W和D为：

$$
\begin{aligned}
D(n) &= O(h + \log\log\frac{n}{h}) = O(\log\log n) \\
W(n) &= O(h\cdot\frac{n}{h} + \frac{n}{h}\log\log\frac{n}{h}) = O(n)
\end{aligned} \tag{43}
$$

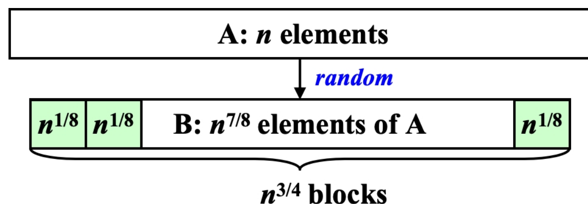
Solution4：Random Sampling

在CRCW PRAM (Current Read, Current Write; Parallel Random Access Machine) 中，这个算法有大概率可以实现$D = O(1)$、$W = O(n)$，但结果一定是正确的

1. **抽样**：在A中随机选择$n^{7/8}$个元素（如果选到了重复的数也正常运行，不影响算法的正确性）

$$
D = O(1), W = O(n^{7/8}) \tag{44}
$$

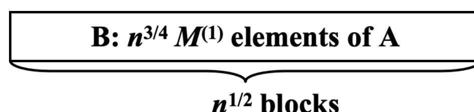
2. **$n^{1/8}$分组**：每组大小为$n^{1/8}$，因此一共有$n^{7/8}/n^{1/8} = n^{3/4}$组



3. **运行循环赛算法**：得到$n^{3/4}$个局部最大值

$$D=O(1),W=O(n^{3/4}\times(n^{1/8})^2)=O(n) \tag{45}$$

4. **$n^{1/4}$分组**：对上一步得到的$n^{3/4}$个元素按照$n^{1/4}$分组，一共$n^{1/2}$组



5. **循环赛**：得到采样的$n^{7/8}$个数中最大的那个

$$D=O(1),W=O(n^{1/2}\times(n^{1/4})^2)=O(n) \tag{46}$$

   到此为止，整个算法都是$D=O(1)$、$W=O(n)$的，但有可能最大的根本就不在一开始选择的那$n^{7/8}$个元素中，因此还需要进行下一步，保证算法得到的结果始终正确

6. **检查**：在原来的序列中查找，如果发现大于之前找出的最大数M的（找更大数的过程D也是1，因为只需要找到更大的就可以，n个数可以同时独立判断是否大于M），就将其放到$n^{7/8}$的序列中（随便替换一个），然后重新运行

   事实上，如果运气不好，这个过程可能会反复执行很多次，导致复杂度上升。但算法无法在一层内完成的概率不到$1/n^c$

EXTERNAL SORTING

Introduction

假设我们想要得到`a[i]`，如果在internal memory中，复杂度是$O(1)$；但是，在disk中，需要经历如下三步（这只是一个笼统的说法，具体情况视不同的硬件而异，即device-dependent）：

1. **Find a Track**：找到轨道
2. **Find the Sector**：找到区块
3. **Find `a[i]` and Transmit**：找到`a[i]`并将磁头移动到该处

为了降低反复访问`a[i]`带来的额外开销，外部排序采用类似merge sort的思路

Merge External Sort

一些定义

- **Run**：一组排好的数字就是一个run（因此也是merge的单位），可以理解为“have already run”
- **Pass**：将$k\cdot c$个run merge成c个run的过程称为一个pass。简单点理解，就是归并树上的一层
- **Tape**：数据存储在tape中。注意初始数据的存储需要一个tape，因此本章节的算法至少需要3个tape才可以实现
- **M Records**：The internal memory can handle M records at a time. 这是题目中常给的一个约定，表示内部存储一次最多处理3个数据
- **k-way Merge**：将全体数据分为k组，merge时将k个run合并为一个

基础版：2-way Merge

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

T_1 81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

1. **Run Construction**：因为内部存储一次最多只能处理3个数据，所以就将原数据3个一组进行分组（将尽可能多的数字分入一组，这样run construction后的结果更有序）

   run construction
   T_2 11 81 94 | 17 28 99 | 15
   T_3 12 35 96 | 41 58 75 |

2. **Merge**：将T_2和T_3的前3个数merge，放入T_1（因为T_1中的信息已经完全进入T_2和T_3中了，所以可以覆盖）；将T_2和T_3的4~6个数merge，放入T_4；然后将还剩下的15放到T_1最后（merge的结果轮流添加到T_1和T_4中）

   T_1 11 12 35 81 94 96 | 15
   T_4 17 28 41 58 75 99 |

   以此类推

   T_2 11 12 17 28 35 41 58 75 81 94 96 99 |
   T_3 15

   Number of passes = 1+3

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

考虑pass的个数，run construction相当于一个pass。接下来考虑需要几步merge，一共N个数，M个一组，因此一共N/M组需要merge，而每次merge都让组数减半，因此一共需要$1 + \lceil \log_2(N/M) \rceil$个pass

$$\text{Number of passes} = 1 + \lceil \log_2(N/M) \rceil \tag{47}$$

k-way Merge

pass的个数减少为$1 + \lceil \log_k(N/M) \rceil$，但需要$2k$个tape

T_1 81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

T_2 11 81 94 | 41 58 75 |
T_3 12 35 96 | 15
T_4 17 28 99 |

T_1 11 12 17 28 35 81 94 96 99 |
T_5 15 41 58 75 |
T_6

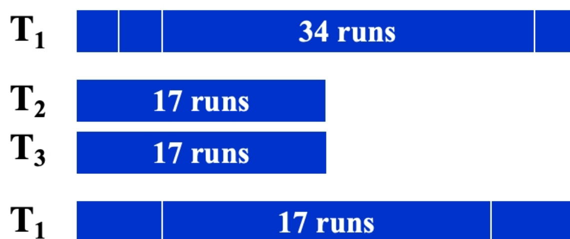
Number of passes $= 1+\lceil \log_k(N/M) \rceil$

Use 3 Tapes for a 2-way Merget

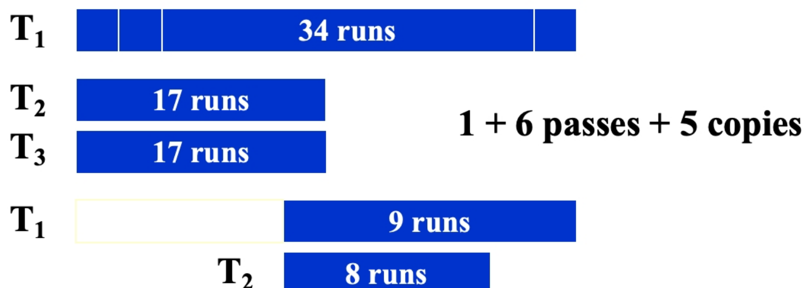
通过复制或者合理规划存储空间，可以减少所需要tape的个数

1. **复制**

先将T_2和T_3全部merge到T_1中



然后将T_1中的一半复制出去



pass和原来一致，还是$1 + \lceil \log_k(N/M) \rceil$，copy需要的个数为：

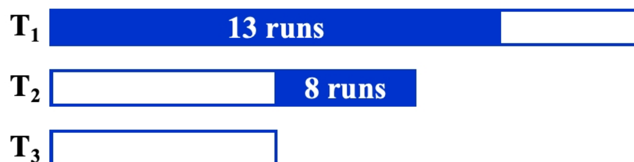
$$\lceil \log_k(N/M) \rceil - 1 \tag{48}$$

即merge的次数减一（因为最后一次直接完成，不需要copy）

2. **一个更聪明的方法：Split Unevenly**

   将T_1按照斐波那契数列分为T_2和T_3（按斐波那契分割可以被证明就是最佳的）

   T_1没用了，将T_2的前13个run与T_3合并，写入T_1。此时T_3无用，T_2还剩8个



   将T_1和T_2合并写入T_3，T_1还剩5个，T_2无用

   将T_1和T_3合并写入T_2，T_3还剩3个，T_1无用

   以此类推

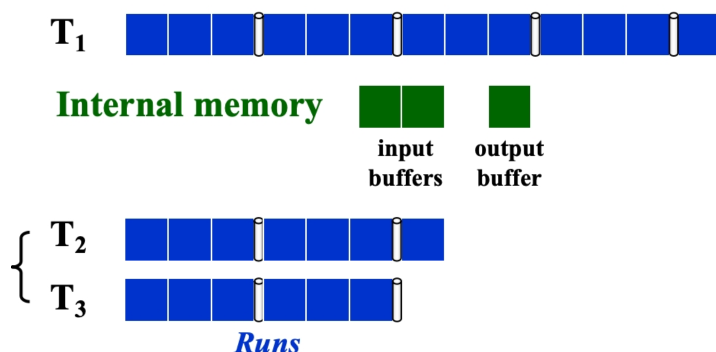
   这个方法会让pass变多，因为相比每次减半，split unevenly是斐波那契向下的，一共是1 + 7个pass，但只需要$k+1$个tape（有点像汉诺塔）

Buffers for Parallel Operation

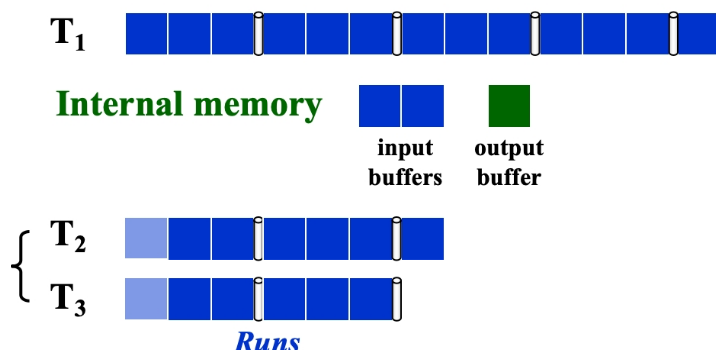
Sort a file containing 3250 records, using a computer with an internal memory capable of sorting at most 750 records. The input file has a block length of 250 records.

- **题干分析**：对3250个元素进行排序，内部处理器最多处理750条。block length是一次搬运record的个数；一共是$3250/250=13$个块，run construction时$750/250=3$个块构成一个run
- **Buffer**：缓存支持并行处理，但不能同时读写，只能并行写出或读入

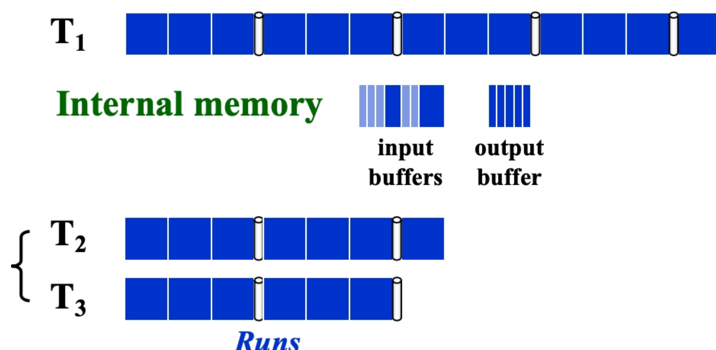
如果我们只有3个block的buffer，则其中2个用作input buffer，1个用作output buffer



先从T_2和T_3各搬运1个block到input buffer中



然后对input buffer中的两个block进行merge，写入output buffer中



此时output buffer写满，需要写出到T_1。但写出时output buffer不能写入，因此merge停止，无法实现并行效果，还是得老老实实等output buffer写完才能继续merge

但是，如果将两个buffer都扩大一倍，就可以让output buffer写满一个block时写出，并且保证还有一个空余的block用于写入。因此，为了保证并行的执行，我们需要$2k$个input buffer（每一行取一个，一共需要k个，然后翻倍保证并行）和2个output buffer（写满一个就读出一个）

注意，k不是越大越好的，因为k变大后IO时间变长，所以需要权衡

Longer Run：Replacement Selection

为了获得更长的run，我们在internal memory中不采用简单的M个一组进行排序的方式（因为这样每次只能获得一个长度固定为M的run，而我们希望其可以更长），而是将其设计为一个小根堆。每次delete min，加到当前run的最后，并读入下一个数；如果min比run的最后一个数小，那就选择堆中比run的最后一个元素大的最小元素，直到所有元素都比run中最后一个元素大为止（听上去有点绕，但看一下下面这个例子就不难理解了）

81 94 11 96 12 35 17 99 28 58 41 75 15

11
94 81

81 94 11 96 12 35 17 99 28 58 41 75 15

81
94 96

11

81 94 11 96 12 35 17 99 28 58 41 75 15

94
96 12

11 81

81 94 11 96 12 35 17 99 28 58 41 75 15

96
35 12

11 81 94

81 94 11 96 12 35 17 99 28 58 41 75 15

17
35 12

11 81 94 96

81 94 11 96 12 35 17 99 28 58 41 75 15

12
35 17

11 81 94 96

（堆中没可以加到第一行尾上的元素了，另起一行）

11 81 94 96

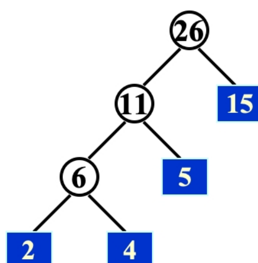
12

当T_1中本来就是有序时，本方法最快，一次即可结束

Minimize the Merge Time

假设run construction得到了4个run，长度分别为2、4、5和15，如何在它们之间merge以最小化时间成本?

构建哈夫曼树（每次将最小的两个出队，合并，然后重新入队）



考点

这一章的考点相对来说比较务实，不会考太多的概念混淆，搞清楚算法运行过程更重要。所以有一些东西在笔记中也略过了。Replacement Selection是考察的重点，老老实实把草稿写清楚就不会错。

1. **pass计算**：Given 100,000,000 records of 256 bytes each, and the size of the internal memory is 128MB. If simple 2-way merges are used, how many passes do we have to do?

   A. 10 **B**. 9 C. 8 D. 7

   首先，我们需要计算run的大小。因为内部存储一共是128MB，每个record占用256B，因此一个run含有 $128\times 2^{20}/256=2^{19}$个record。然后直接套公式（别忘了 + 1）

   $$\text{Pass}=1+\lceil\log_2(10^8/2^{19})\rceil=9 \tag{49}$$

2. **哈夫曼树的构建**：In external sorting, suppose we have 5 runs of lengths 2, 8, 9, 5, and 3, respectively. Which of the following merging orders can obtain the minimum merge time?

   A. merge runs of lengths 2 and 3 to obtain Run#1; merge Run#1 with the one of length 5 to obtain Run#2; merge Run#2 with the one of length 8 to obtain Run#3; merge Run#3 with the one of length 9

   B. merge runs of lengths 2 and 3 to obtain Run#1; merge Run#1 with the one of length 5 to obtain Run#2; merge runs of lengths 8 and 9 to obtain Run#3; merge Run#2 and Run#3

   C. merge runs of lengths 2 and 3 to obtain Run#1; merge runs of lengths 5 and 8 to obtain Run#2; merge Run#1 and Run#2 to obtain Run#3; merge Run#3 with the one of length 9

   D. merge runs of lengths 2 and 3 to obtain Run#1; merge runs of lengths 5 and 8 to obtain Run#2; merge Run#2 with the one of length 9 to obtain Run#3; merge Run#1 and Run#3

   单纯考察哈夫曼树的构建过程，草稿写清楚，别算错了

3. **Replacement Selection**：

4. Suppose we have the internal memory that can handle 12 numbers at a time, and the following two runs on the tapes:

   Run 1: 1, 3, 5, 7, 8, 9, 10, 12

   Run 2: 2, 4, 6, 15, 20, 25, 30, 32

   Use 2-way merge with 4 input buffers and 2 output buffers for parallel operations. Which of the following three operations are NOT done in parallel?

   A. 1 and 2 are written onto the third tape; 3 and 4 are merged into an output buffer; 6 and 15 are read into an input buffer

   B. 3 and 4 are written onto the third tape; 5 and 6 are merged into an output buffer; 8 and 9 are read into an input buffer

   C. 5 and 6 are written onto the third tape; 7 and 8 are merged into an output buffer; 20 and 25 are read into an input buffer

   D. 7 and 8 are written onto the third tape; 9 and 15 are merged into an output buffer; 10 and 12 are read into an input buffer

   请把草稿写清楚！