Last Section

Complete Development of an Algo.

- 1 Statement of the problem
- 2 Development of a Model
- 3 Design of the algorithm
- 4 Correctness of the algorithm
- 5 Implementation
- 6 Analysis and complexity of the algorithm
- 7 Program testing
- 8 Documentation

Methodology

- Algo. A → Algo. B
  - Method
  - Principle
  - Control
- Algo. B → Algo. C
  - Method
  - Control + Data Structure
- Algo. C → Algo. D
  - Data Structure

Divide and Conquer

- MINMAX:

  2n – 2 vs. 3n/2 – 2
- 二分搜索:

  算法BINARYSEARCHREC在n 个元素组成的已排序数组中搜索某个元素所执行的元素比较次数不超过 $\lfloor \log n \rfloor + 1$;

  非递归;
- 合并排序： nlogn
- 寻找第 k 小元素：20cn
- 划分算法与快速排序：n-1;n(n-1)/2

合并排序

- 将待排序数组对半分成两个子数组；
- 分别对两个子数组排序；
- 将两个已排序的子数组合并。

合并排序

MERGESORT

输入：n个元素的数组A[1…n]。

输出：按非降序排列的数组A[1…n]。

- mergesort(A, 1, n)

- Procedure **mergesort** (A, low, high)

1. **if** low < high **then**

2. mid $\leftarrow \lfloor (low + high)/2 \rfloor$

3. mergesort (A, low, mid)

4. mergesort(A, mid + 1, high)

5. MERGE(A, low, mid, high)

6. **end if**

合并排序效率

- 假定n 是2的幂，即$n = 2^k$, k 是大于等于0的整数。
- 如果n = 1, 则算法不执行任何元素的比较
- 如果n > 1, 则执行了步骤2到步骤5，根据函数C 的定义(n元素需比较次数），执行步骤3和步骤4需要的元素比较次数都为C(n/2)。
- 合并两个子数组所需的元素比较次数在n/2与n – 1 之间

Algorithm MERGE

输入：数组A[1..m]，索引1≤p ≤ q <r ≤ m，两个子数组A[p..q]和A[q+1..r]　分别非降序排列

输出：合并A[p..q]和A[q+1..r]的数组A[p..r]

B[p..r]为辅助数组

```
1.      s←p; t←q+1; k←p;
2.      while  s ≤ q  and  t ≤ r
3.              If  A[s] ≤ A[t]  then
4.                      B[k] ← A[s]
5.                      s ← s+1
6.              else
7.                      B[k] ← A[t]
8.                      t ← t+1
9.              end if
10.             k ←k+1
11.     end while
12.     If  s=q+1  then  B[k..r]  ← A[t..r]
13.     else      B[k..r]  ← A[s..q]
14.     end if
15.     A[p..r]  ← B[p..r]
```

Algorithm MERGE

输入：数组A[1..m]，索引1≤p ≤ q <r ≤ m，两个子数组A[p..q]和A[q+1..r]　分别非降序排列

输出：合并A[p..q]和A[q+1..r]的数组A[p..r]

B[p..r]为辅助数组

```
1.      s←p; t←q+1; k←p;
2.      while  s ≤ q  and  t ≤ r
3.              If  A[s] ≤ A[t]  then
4.                      B[k] ← A[s]
5.                      s ← s+1
6.              else
7.                      B[k] ← A[t]
8.                      t ← t+1
9.              end if
10.             k ←k+1
11.     end while
12.     If  s=q+1  then  B[k..r]  ← A[t..r]
13.     else      B[k..r]  ← A[s..q]
14.     end if
15.     A[p..r]  ← B[p..r]
```

* ***n1 ≤n2, 则 n1 ≤ c(n)≤ n-1***

合并排序效率

- 算法所需的最大比较次数由下列递推式给出

$$C(n) = 0 \qquad \text{若} n = 1$$

$$= 2C(n/2) + n - 1 \qquad \text{若} n \geq 2$$

- $C(n)=n\log n - n + 1$

合并排序效率

- 算法所需的最小比较次数由下面的递推式给定

$$C(n) = 0 \quad \text{若} n = 1$$

$$= 2C(n/2) + n/2 \quad \text{若} n \geq 2$$

合并排序效率

(a,c非负整数，b,d,x非负常数，$n=c^k$)：

f(n) =d 若 n=1

=af(n/c)+bn^x 若 n>=2

的解是

$f(n)= bn^x \log_c n + dn^x$ 若 $a=c^x$

$f(n)= \left(d+\frac{bc^x}{a-c^x}\right)n^{\log_c a}-\left(\frac{bc^x}{a-c^x}\right)n^x$ 若 a<>c^x

合并排序效率

最小比较次数

C(n)=(nlogn)/2

因(a,c非负整数，b,d,x非负常数，$n=c^k$)：

f(n) =d 若 n=1

=af(n/c)+bn^x 若 n>=2

的解是

f(n)= $bn^x\log_c n + dn^x$ 若 a=c^x

f(n)= $\left(d+\frac{bc^x}{a-c^x}\right)n^{\log_c a}-\left(\frac{bc^x}{a-c^x}\right)n^x$ 若 a<>c^x

合并排序效率

如果n 是任意的正整数（不必是2的幂），对于由算法MERGESROT执行的元素比较次数C(n)的递推关系式为

$$C(n) = \begin{cases} 0 & \text{若 } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + bn & \text{若 } n \geq 2 \end{cases}$$

$C(n)=\theta(n\log n)$

寻找第 k 小元素

- n 个已排序的A[1…n] 序列的中项是其“中间”元素。
- 如果n 是奇数，则中间元素是序列中第(n + 1)/2 个元素；如果n 是偶数，则存在两个中间元素，所处的位置分别是n/2 和n/2 +1, 在这种情况下，我们将选择第n/2 个最小元素。
- 综合两种情况，中项是第 $\lceil n/2 \rceil$ 最小元素。

- 寻找中项的一个直接的方法是对所有的元素排序并取出中间一个元素，这个方法需要（n log n）时间，因为任何基于比较的排序过程在最坏的情况下必须至少要花费这么多时间。

寻找第 k 小元素

SELECT
输入：n个元素的数组A[1…n]和整数k, 1 ≤ k ≤n。
输出：A中的第k 小元素。

- select(A, 1, n, k)

Procedure **select** (A, low, high, k)

1. p ← high – low + 1
2. **if** p < 44 **then** 将A 排序**return** (A[k])
3. 令q =$\lfloor p/5 \rfloor$。将A分成q组，每组5个元素。如果5不整除p, 则排除剩余的元素。
4. 将q 组中的每一组单独排序，找出中项。所有中项的集合为M.
5. mm ← select(M, 1, q, $\lceil q/2 \rceil$) //mm 为中项集合的中项
6. 将A[low…high] 分成三组
   A1 = {a | a < mm}
   A2 = {a | a = mm}
   A3 = {a | a > mm}
7. **case**
   | A1 | ≥ k: **return** select (A1, 1, |A1|, k)
   | A1| + | A2| ≥ k: **return** mm
   |A1| + |A2| < k: **return** select(A3, 1, |A3|, k-|A1| - |A2|)
8. **end case**

寻找第 k 小元素

- 把n 个元素划分成$\lfloor n/5 \rfloor$组，每组由5个元素组成，如果n不是5的倍数，则排出剩余的元素。
- 每组进行排序并取出它的中项即第三个元素。接着将这些中项序列中的中项元素记为mm，它是通过递归计算得到的。
- 算法的步骤6将数组A 中的元素划分成三个数组：A1, A2, A3, 其中分别包含小于、等于和大于mm 的元素。
- 最后，在第7步，求出第k 小的元素出现在三个数组中的哪一个，并根据测试结果，算法或者返回第k 小的元素，或者在A1 或A3上递归。

寻找第 k 小元素

SELECT
输入：n个元素的数组A[1…n]和整数k, 1 ≤ k ≤n。
输出：A中的第k 小元素。

- select(A, 1, n, k)

Procedure **select** (A, low, high, k)

1. p ← high – low + 1
2. **if** p < 44 **then** 将A 排序**return** (A[k])
3. 令q =$\lfloor p/5 \rfloor$。将A分成q组，每组5个元素。如果5不整除p, 则排除剩余的元素。
4. 将q 组中的每一组单独排序，找出中项。所有中项的集合为M.
5. mm ← select(M, 1, q, $\lceil q/2 \rceil$) //mm 为中项集合的中项
6. 将A[low…high] 分成三组
   A1 = {a | a < mm}
   A2 = {a | a = mm}
   A3 = {a | a > mm}
7. **case**
   | A1 | ≥ k: **return** select (A1, 1, |A1|, k)
   | A1| + | A2| ≥ k: **return** mm
   |A1| + |A2| < k: **return** select(A3, 1, |A3|, k-|A1| - |A2|)
8. **end case**

例

- 8, 33, 17, 51, 57, 49, 35, 11, 25, 37, 14, 3, 2, 13, 52, 12, 6, 29, 32, 54, 5, 16, 22, 23, 23, 7

- 设数组A[1…n] 存储这个序列，k = 13, 即要在数组A 中找到第13小的元素，

例

- 首先把数集划分成5组，每组有5个元素：（8，33，17，51，57），（49，35，11，25，37），（14，3，2，13，52），（12，6，29，32，54），（5，16，22，23，7）。
- 接着以升序对每组排序：（8，17，33，51，57），（11，25，35，37，49），（2，3，13，14，52，（6，12，29，32，54），（5，7，16，22，23）。
- 取每组的中项并形成中项集：M = {33, 35, 13, 29, 16}.

例

- 利用算法递归找出M中的中项元素：mm = 29.
- 将A 划分成三个子序列：
- A1 = {8, 17, 11, 25, 14, 3, 2, 13 12, 6, 5, 16, 22, 23, 7},
- A2 = {29},
- A3 = {32, 51, 57, 49, 35, 37, 52, 32, 54}.
- 因为13<15 = |A1|， A2 和A3中的元素可以被放弃，第13小的元素一定在A1中。

例

- 重复上述过程，因此设A = A1, 把元素划分成三组：（8，17， 11， 25， 14），（3，2， 13， 12，6），（5，16， 22， 23， 7）。
- 每组排序后，找到新的中项集：M = {14, 6, 16}, 这样M 中的中项元素mm 为14。
- 接着将A 划分为三个序列：A1 = {8， 11， 3， 2，13， 12， 6， 5， 7}， A2 = {14}， A3 = {17, 25, 16, 22, 23}.
- 因为13>10 = | A1 | + | A2 |， 设A = A3 ， 在A中找第3小的元素（3= 13-10），算法将返回A[3]= 22.

寻找第 k 小元素

SELECT
输入：n个元素的数组A[1…n]和整数k, 1 ≤ k ≤n。
输出：A中的第k 小元素。

- select(A, 1, n, k)

Procedure **select** (A, low, high, k)

1. p ← high – low + 1
2. **if** p < 44 **then** 将A 排序**return** (A[k])
3. 令q =$\lfloor p/5 \rfloor$。将A分成q组，每组5个元素。如果5不整除p?
4. 将q 组中的每一组单独排序，找出中项。所有中项的集合为M.
5. mm ← select(M, 1, q, $\lceil q/2 \rceil$)　　//mm 为中项集合的中项
6. 将A[low…high] 分成三组
   A1 = {a | a < mm}
   A2 = {a | a = mm}
   A3 = {a | a > mm}
7. **case**
   | A1 | ≥ k: **return** select (A1, 1, |A1|, k)
   | A1| + | A2| ≥ k: **return** mm
   |A1| + |A2| < k: **return** select(A3, 1, |A3|, k-|A1| - |A2|)
8. **end case**

Select 效率

- 1、2均为Θ(1);
- 3 为Θ(n);
- 4为Θ(n);
- 5为T($\lfloor n/5 \rfloor$)
- 6为Θ(n);
- 7为T(0.7n+1.2)

  设0.7n+1.2<=$\lfloor 0.75n \rfloor$, n>=44

  T(n)<=c　　若n<44

  <=T($\lfloor n/5 \rfloor$)+T($\lfloor 3n/4 \rfloor$)+cn　　若n>=44

  T(n)<=20cn (c为一足够大的常数)

寻找第 k 小元素

SELECT
输入：n个元素的数组A[1…n]和整数k, 1 ≤ k ≤n。
输出：A中的第k 小元素。

- select(A, 1, n, k)

Procedure **select** (A, low, high, k)

1. p ← high – low + 1
2. **if** p < 44 **then** 将A 排序**return** (A[k])
3. 令q =$\lfloor p/5 \rfloor$。将A分成q组，每组5个元素。如果5不整除p?
4. 将q 组中的每一组单独排序，找出中项。所有中项的集合为M.
5. mm ← select(M, 1, q, $\lceil q/2 \rceil$)　　//mm 为中项集合的中项
6. 将A[low…high] 分成三组
   A1 = {a | a < mm}
   A2 = {a | a = mm}
   A3 = {a | a > mm}
7. **case**
   | A1 | ≥ k: **return** select (A1, 1, |A1|, k)
   | A1| + | A2| ≥ k: **return** mm
   |A1| + |A2| < k: **return** select(A3, 1, |A3|, k-|A1| - |A2|)
8. **end case**

划分算法

SPLIT

输入：数组A[low…high].

输出：（1）输出A[low] in Position 的重新排列的数组A;

- (2) 划分元素A[low]的新位置w.

- 1. i← low
- 2. x←A[low]
- 3. **for** j← low + 1 t**o** high
- 4. **if** A[j] ≤ x **then**
- 5. i← i + 1
- 6. **if** i ≠j **then** 互换A[i] 和A [j]
- 7. **end if**
- 8. **end for**
- 9. 互换A[low] 和A[i]
- 10. w← i
- 11. **return** A 和w

5	7	1	6	8	3

划分算法

SPLIT

输入：数组A[low…high].

输出：（1）输出A[low] in Position 的重新排列的数组A;

- (2) 划分元素A[low]的新位置w.

- 1. i← low
- 2. x←A[low]
- 3. **for** j← low + 1 t**o** high
- 4. **if** A[j] ≤ x **then**
- 5. i← i + 1
- 6. **if** i ≠j **then** 互换A[i] 和A [j]
- 7. **end if**
- 8. **end for**
- 9. 互换A[low] 和A[i]
- 10. w← i
- 11. **return** A 和w
- n-1次元素比较

5	7	1	6	8	3

快速排序

QUICKSORT

输入：n个元素的数组A[1…n].

输出：按非降序排列的数组A 中的元素。

```
    Quicksort(A, 1, n)

procedure     quicksort (A, low, high)
1.  if low < high then
2.      SPLIT (A[low…high], w)     {w 为A[low]的新位置}
3.      qicksort(A, low, w-1)
4.      quicksort(A, w + 1, high)
5.  end if
```

快速排序

- $T(n) = 2T(n/2) + n-1$

  $T_{average} = \Theta(n\lg n)$

- $T(n) = T(n1) + T(n2) + n-1$

  $= T(0) + T(n-1) + n-1$

  $= 0 + \quad T(0) + T(n-2) + n-2 \quad + \quad n-1$

  $= \ldots\ldots$

  $T_{worst} = \Theta(n^2)$

大整数乘法

- 设u 和v 是两个n 位的整数(二进制)，传统的乘法算法需要(n^2) 数字相乘来计算u 和v 的乘积。
- 使用分治技术，这个上界将显著减小。
- 为简单起见，假定n 是2的幂。
- 把每个整数分为两部分，每部分为$n/2$位，则u 和v 可重写为$u = w2^{n/2} + x$ 和$v = y2^{n/2} + z$

- U

w	x

- v

y	z

大整数乘法

- u 和v 的乘积可以计算为：

  $uv = (w2^{n/2} + x)(y2^{n/2} + z) = wy2^{n} + (wz + xy)\,2^{n/2} + xz$

- 用2^n 做乘法运算相当于简单地左移n 位，它需要θ(n) 时间。这样，在这个公式中，有4次乘法运算和3次加法运算，这蕴含着以下递推式成立

$$T(n) = \begin{cases} d & \text{若 } n = 1 \\ 4T(n/2) + bn & \text{若 } n > 1 \end{cases}$$

  式中b 和d都是大于0的常量。

- 由定理，递推式的解是$T(n) = (n^2)$.

大整数乘法

- 考虑用以下恒等式计算$wz + xy$

  $wz + xy = (w + x)(y + z) - wy - xz$
- 由于wy 和 xz 不需要做二次计算，结合以上二式，仅需3次乘法运算，即

  $uv = wy2^n + ((w + x)(y + z) - wy - xz)\, 2^{n/2} + xz$
- 这样u 和v的乘法运算简化为3次n/2规模整数的乘法运算和6次加法运算，这些加法所花时间是(n)。

大整数乘法

- 此方法产生以下递推式

$$T(n) = \begin{cases} d & \text{若 } n = 1 \\ 3T(n/2) + bn & \text{若 } n > 1 \end{cases}$$

- 上式中的b 和d 是适当选择的某个大于0的常量。由定理得出：

$$T(n) = \Theta(n^{\log 3}) = O(n^{1.59})$$

- 这是对传统方法的一个显著改进。

矩阵乘法

A 和B 是两个n x n 的矩阵，我们希望计算它们的乘积C = AB

传统算法

- *C* 由以下公式计算

$$C(i, j) = \sum_{k=1}^{n} A(i, k)\, B(k, j)$$

- 算法需要n^3次乘法运算和
  $n^3 - n^2$次加法运算，
- 导致其时间复杂性为$\Theta(n^3)$。

分治（递归）方法

- 设 $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ 和 $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

计算 $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

分治（递归）方法

- 设a, m分别表示加法和乘法的耗费

- $T(n) = m$ $\quad n=1$

  $= 8T(n/2)+4(n/2)^2 a$ $\quad n>=2$

  $T(n) = mn^3+an^3-an^2$

  同传统方法

STRASSEN 算法

- 算法的基本思想在于以增加加减法的次数来减少乘法次数：
  - 用了7次n/2 x n/2 矩阵乘法和18次n/2 x n/2 矩阵的加法

- 设 $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ 和 $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
-

- 为了计算矩阵的乘积

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

STRASSEN 算法

首先计算以下的乘积

$$d1 = (a11 + a22)\ (b11 + b22)$$
$$d2 = (a21 + a22)\ b11$$
$$d3 = a11\ (b12 – b22)$$
$$d4 = a22\ (b21 – b11)$$
$$d5 = (a11 + a12)\ b22$$
$$d6 = (a21 – a11)\ (b11 + b12)$$
$$d7 = (a12 – a22)\ (b21 + b22)$$

接着从下面式子计算出C

$$C = \begin{bmatrix} d1 + d4 – d5 + d7 & d3 + d5 \\ d2 + d4 & d1 + d3 – d2 + d6 \end{bmatrix}$$

STRASSEN 效率

- 算法用到的加法是18次，乘法是7次，对于其运行时间产生以下递推式

$$T(n) = m \qquad \text{若} n=1$$

$$= 7T(n/2) + 18(n/2)^2 a \qquad \text{若} n \geq 2$$

$T(n) = mn^{\log 7} + 6an^{\log 7} - 6an^2$

即运行时间为$\Theta(n^{\log 7}) = O(n^{2.81})$。

比较

Θ	乘法	加法	复杂性
传统算法	n^3	$n^3 - n^2$	(n^3)
分治方法	n^3	$n^3 - n^2$	(n^3)
STRASSEN 算法	$n^{\log 7}$	$6\, n^{\log 7} - 6\, n^2$	$(n^{\log 7})$

比较

	n	乘法	加法
传统算法	100	1 000 000	990 000
STRASSEN 算法	100	411 822	2 470 334
传统算法	1000	1 000 000 000	999 000 000
STRASSEN 算法	1000	264 280 285	1 579 681 709
传统算法	10 000	10^{12}	9.99×10^{12}
STRASSEN 算法	10 000	0.169×10^{12}	10^{12}

最近点对

- $T(n)=1$ 若$n=2$
  $=3$ 若$n=3$
  $=2T(n/2)+\Theta(n)$ 若$n>3$

$\Theta(nlgn)$

Code

凸包（快包）

Average: Θ(nlgn)

Worst: Θ(n^2)

Brute Force

根据(x1,y1), (x2,y2)的直线方程

(y2-y1)x+(x1-x2)y=(x1y2-y1x2)

n(n-1)/2, n-2, n^3

Decrease and Conquer

减 治

减 治

- 减治技术利用了一种关系：

  一个问题给定实例的解和同样问题较小实例的解之间的关系。

  一旦建立了这样一种关系，我们既可以递归地，也可以非递归地地来运用减治技术。

- 减治法有 3 种主要的变种：

  减去一个常量 (decrease by a constant)

  减去一个常数因子(decrease by a constant factor)

  减去的规模是可变的(variable size decrease)

Decrease by a constant

- 在减常量变种中，每次算法迭代总是从实例规模中减去一个规模相同的常量。经常地，这个常量等于一。

- 函数$f(n) = a^n$可以用一递归定义来计算

$$f(n) = \begin{cases} f(n-1) * a & \text{如果} n > 1 \\ a & \text{如果} n = 1 \end{cases}$$

*和蛮力法是一样的效率？

得出这个做法的思想过程是不同的

Decrease by a constant factor

- 减常因子技术意味着在算法的每次迭代中，总是从实例的规模中减去一个相同的常数因子。在多数应用中，这样的常数因子等于二。

- 计算a^n的值是规模为n的实例；

  规模减半（常数因子等于二）的实例计算就是$a^{n/2}$ 的值；

  它们之间有着明显的关系： $a^n = (a^{n/2})^2$。

Decrease by a constant factor

$a^n = (a^{n/2})^2$ n是偶数

$= (a^{(n-1)/2})^2 \, a$ n是大于 1 的奇数

$= a$ n = 1

- 上式递归根据所做的乘法次数来度量效率，该算法属于O (log n);

  因为,每次迭代的时候，以不超过两次乘法为代价，问题的规模至少会减小一半。

Decrease by a constant factor

该算法和基于分治思想的算法有所不同:

分治算法对两个规模为n/2的指数问题实例分别求解：

$a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil}$ 如果n>1

$= a$ 如果n = 1

$O(n)$

Variable size decrease

- 在减治法的**减可变规模**变种中，算法在每次迭代时，规模减小的模式都是不同的。

- 欧几里德算法：

  gcd (*m*,*n*) = gcd (*n*, *m* **mod** *n*)

欧几里德算法

Euclid (*m*,*n*)

1. **While** *n*≠0 **do**
2. r← *m* mod *n*
3. *m* ← *n*
4. *n* ← *r*
5. **return** *m*

m 和 n既不是以常数，也不是以常数因子的方式减小

插入排序

用减一技术来对一个数组*A*[0…*n*-1]排序

- 假设对较小数组*A*[0…*n*-2]排序的问题已经解决了，我们得到了一个大小为*n*-1的有序数组：*A*[0]≤…≤*A* [*n*-2]。
- 我们如何把这个较小规模的解和元素*A*[*n*-1]一同考虑， 来得到原问题的解呢？
- 显然，我们所要做的就是在这些有序的元素中为*A*[*n*-1]找到一个合适的位置， 然后把它插入到那里。

插入排序

InsertionSort(*A*[0…*n*-1])
//输入：*n*个可排序元素构成的一个数组*A*[0…*n*-1]
//输出：非降序排序的数组*A*[0…*n*-1]
1. for *i* ← 1 to *n*−1 do
2. *w* ← *A*[*i*]
3. *j* ← *i*−1
4. While *j* ≥ 0 and *A*[*j*] > *w* do
5. *A*[*j*+1] ← *A*[*j*]
6. *j* ← *j*−1
7. *A*[*j*+1] ← *w*

插入排序

- 最坏输入是一个严格递减的数组。 对于这种输入的键值比较次数是

$$C_{worst} = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

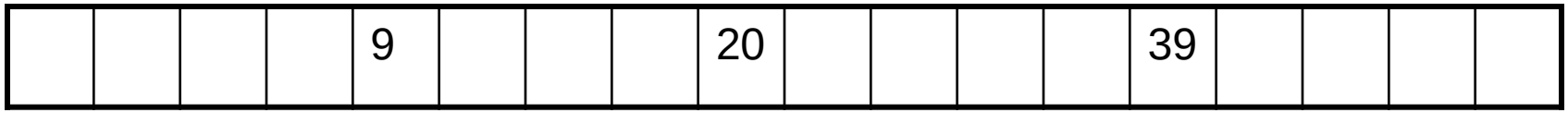
$$C_{best} = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

- 分析表明， 对于随机序列的数组， 插入排序的平均比较次数是降序 数组的一半

$C_{avg} \approx n^2/4$

An advanced App.

- 在用快速排序对数组排序的时候，当子数组的规模变得小于某些预定义的值时（比方说，10个元素），我们可以停止该算法的迭代。
- 那时，整个数组已经基本有序了，我们可以对它应用插入排序来完成接下来的工作。对快速排序做了这种改动之后，一般会减少10%的运行时间。



Decrease by one

- 深度优先查找（DFS）
- 广度优先查找（BFS）

拓扑排序

例：

考虑五门必修课的一个集合{C1, C2, C3, C4, C5},

一个在职的学生必须在某个阶段修完这几门课程。

可以按照任何次序学习这些课程，只要满足下面这些条件：

C1和C2没有任何先决条件；

修完C1 和C2才能修C3；

修完C3 才能修C4；

修完C3、C4才能修C5；

这个学生每个学期只能修一门课程。

这个学生应该按照什么顺序来学习这些课程？

拓扑排序

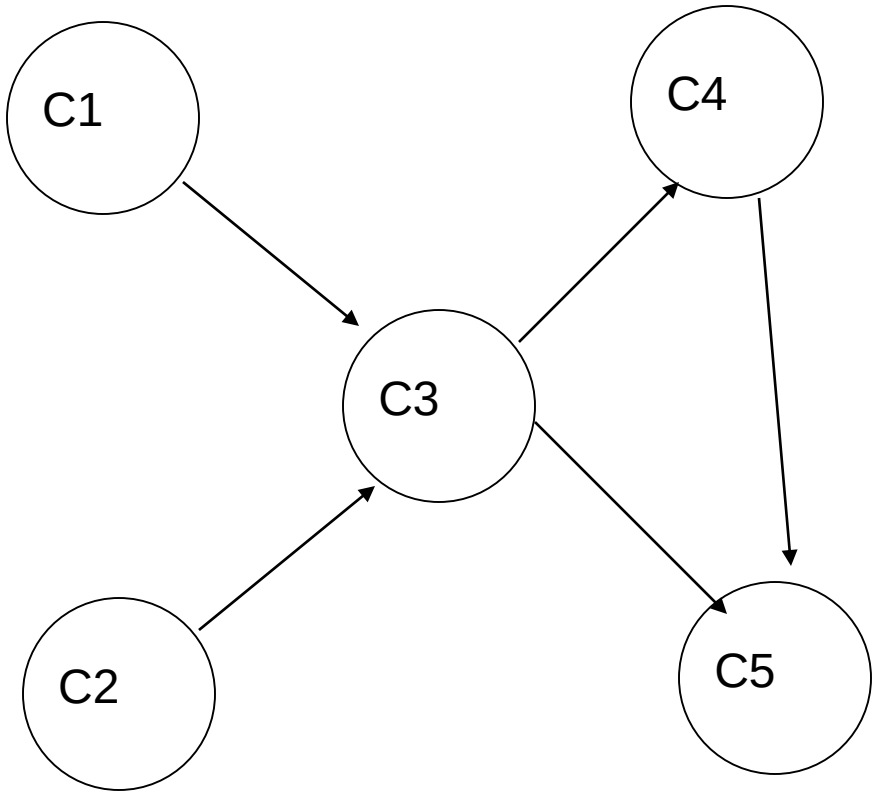
- 若用一个图来建模，它的顶点代表课程，有向边表示先决条件，该问题为：

  是否可以按照这种次序列出它的顶点，使得对于图中每一条边来说，边的起始顶点总是排在边的结束顶点之前。

- 这个问题称为**拓扑排序**。

- 如果有向图具有一个有向的回路，该问题是无解的。因此，为了使得拓扑排序成为可能，问题中的图必须是一个无环有向图。

选课问题



选课问题

- 深度优先查找的一个应用：

  执行一次DFS遍历，并记住顶点变成死端（即退出遍历栈）的顺序；

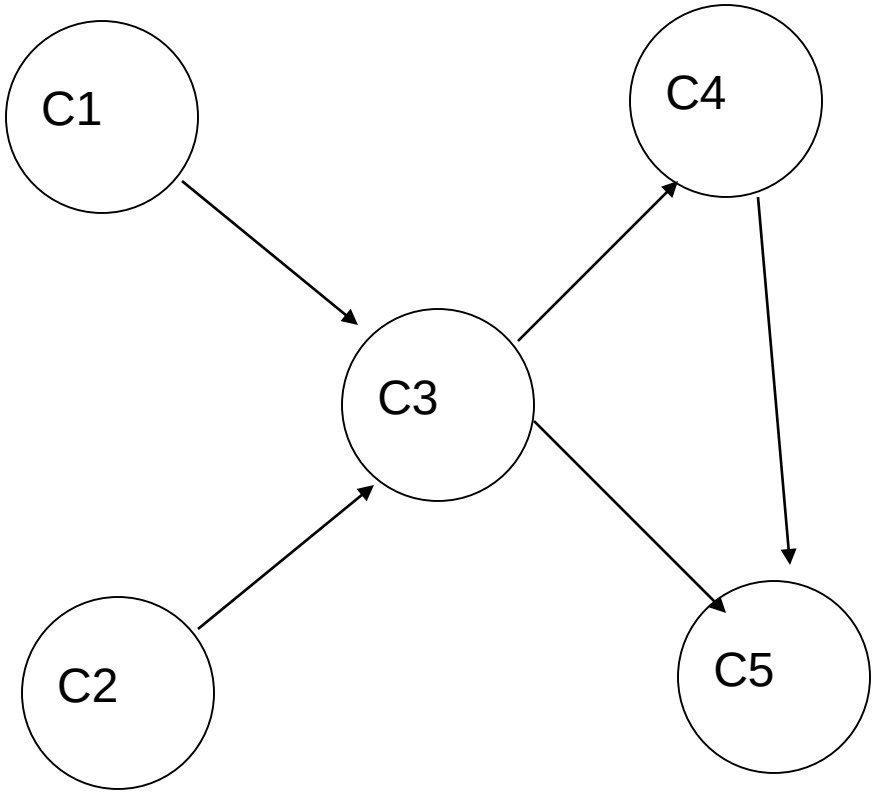
  将该次序反过来就得到了拓扑排序的一个解。

选课问题

- 基于减（减一）治技术的一个直接实现：重复以下过程：

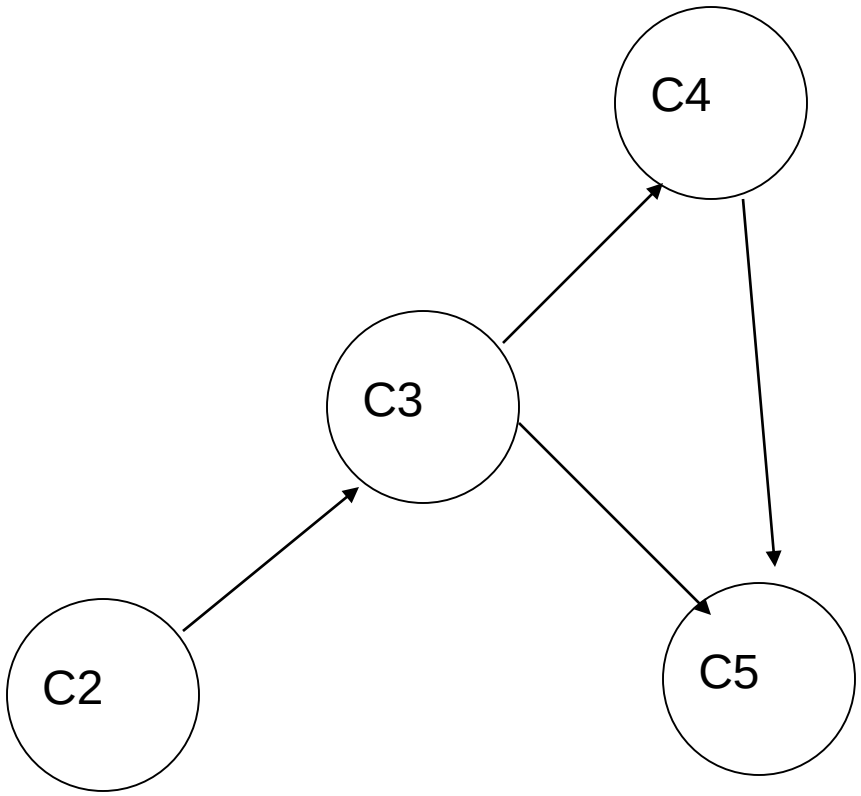
1. 在余下的有向图中求出一个源，它是一个没有输入边的顶点；
2. 然后把该源和所有从它出发的边都删除。（如果有多个这样的源，可以任意选择一个；如果这样的源不存在，算法停止，因为该问题是无解的）
3. 顶点被删除的次序就是拓扑排序的一个解。

拓扑排序



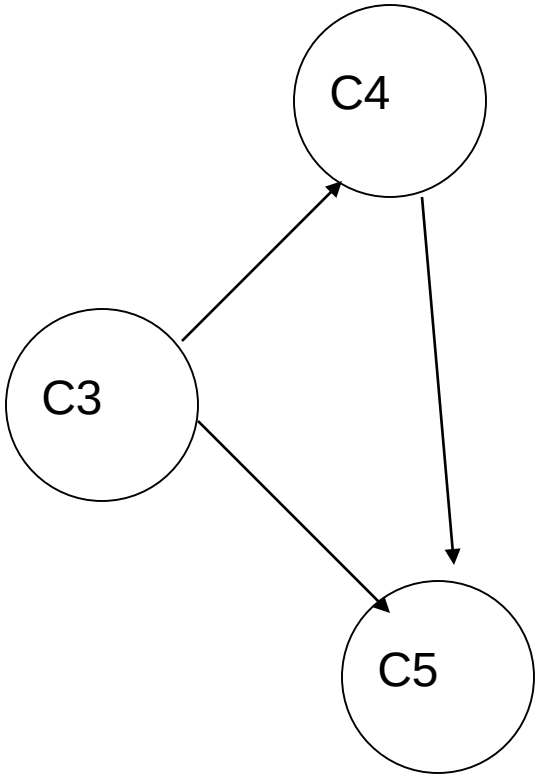
拓扑排序

{C1}



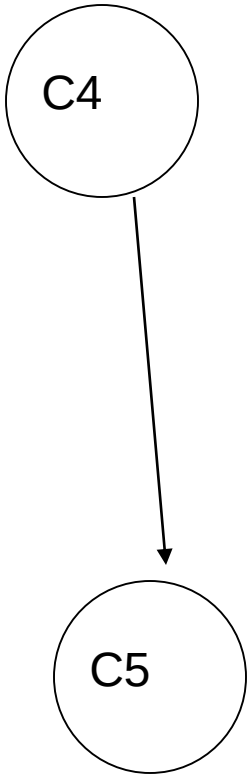
拓扑排序

{C1,C2}



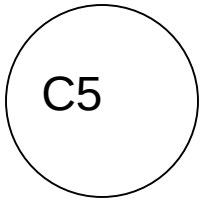
拓扑排序

{C1,C2,C3}



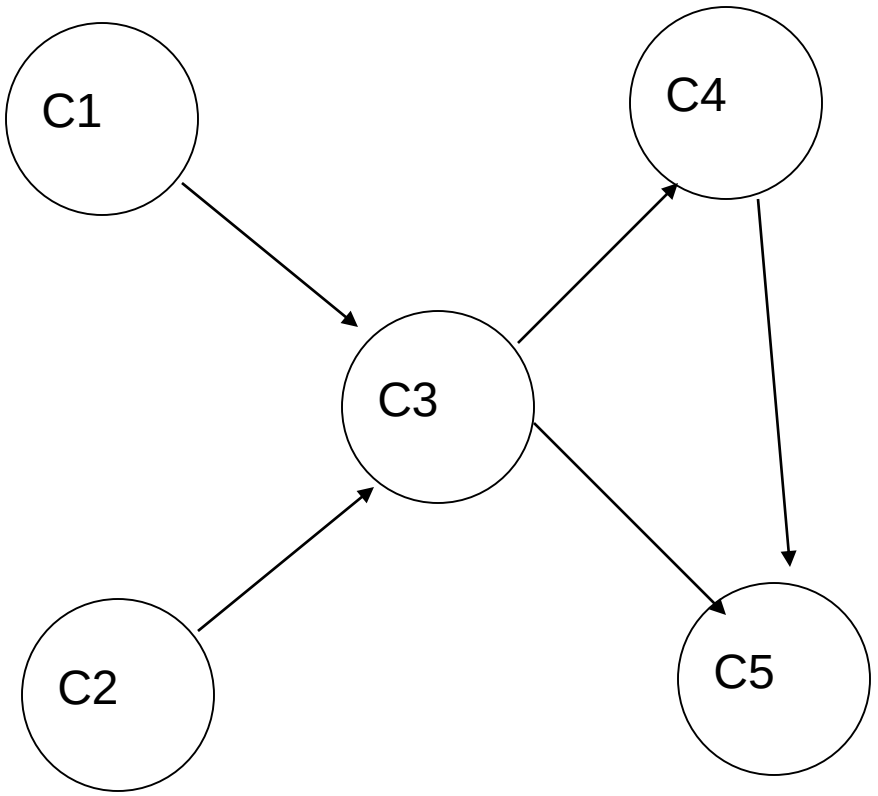
拓扑排序

{C1,C2,C3,c4}



拓扑排序

{C1,C2,C3,c4,C5}



生成排列

- 假设需要对元素进行排列的集合是从1到n 的简单整数集合。
- 对于生成{1,…,n}的所有n!个排列的问题， 减一技术如何应用呢？
- 该问题的规模减一就是要生成 {1…n-1} 的所有(n-1)!个排列。
- 假设这个较小的问题已经解决了，我们可以把n插入到n-1个元素的每一种排列中的n个可能位置中去，来得到较大规模问题的一个解。并且它们的总数量应该是{n(n-1)!=n!}。这样， 我们得到了{1，…,n} 的所有排列。

生成排列

- 对n=3的情况进行处理的例子

开始　　1

插入 2 :　 12　 21

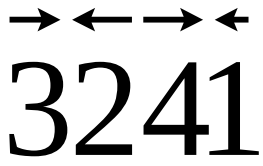
插入 3:　 123　132　312　　321　231　213

- 每处理一个{1…n-1} 的新排列时调换方向：
  右-左，左-右，右-左， ……
- 满足所谓的**最小变化**要求：因为仅仅需要交换直接前趋中的两个元素就能得到任何一个新的排列。

- 提高算法的速度 + 利于使用这些排列的应用(如TSP)

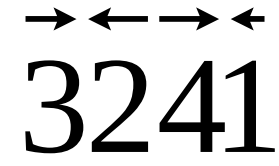
Johnson-Trotter Algorithm

- 给一个排列中的每个分量k 赋予一个方向。例如：



- 如果分量k 的箭头指向一个相邻的较小元素，我们说它在这个以箭头标记的排列中是**可移动**的。

  例如：



  3 和 4 是移动的

Johnson-Trotter Algorithm

JohnsonTrotter(n)

//输入：一个正整数 n

//输出：｛1,…,n｝的所有排列的列表

1. 将第一个排列初始化为 $\overleftarrow{1},\overleftarrow{2},\ldots,\overleftarrow{n}$
2. **While** 存在可移动整数 **do**
3. 求最大的可移动整数k
4. 把k 和它箭头指向的相邻整数互换
5. 调转所有大于k 的整数的方向

Johnson-Trotter Algorithm

- Johnson-Trotter 也是生成排列的最有效的算法之一
- Θ(n!)
- 最优？