INT202W05_矩阵乘法，贪心，动态规划

矩阵

矩阵乘法，普通算法时间复杂度$O(n^3)$

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

子矩阵（Submatrices）法：若矩阵大小为n，对矩阵乘法Z=XY，可将每个矩阵拆分为4个(n/2)x(n/2)大小的矩阵。

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Z=XY

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH \end{aligned}$$

Divide-and-conquer algorithm computes $Z = XY$ by computing I, J, K, and L from the subarrays A through G.

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH \end{aligned}$$

By the above equations, we can compute I, J, K, and L from the eight recursively computed matrix products on $(n/2)\times(n/2)$ subarrays, plus four additions that can be done in $O(n^2)$ time.

Thus, the above set of equations give rise to a divide-and-conquer algorithm whose running time T(n) is characterized by the recurrence

$$T(n) = 8T\left(\frac{n}{2}\right) + bn^2$$

for some constant b > 0.

This equation implies: T(n) = $O(n^3)$ by the master theorem.

复杂度依然是$O(n^3)$

施特拉森算法（Strassen’s algorithm）

减少一个参数，可以只用7个参数计算矩阵乘法

$$S_1 = A(F - H)$$
$$S_2 = (A + B)H$$
$$S_3 = (C + D)E$$
$$S_4 = D(G + E)$$
$$S_5 = (A + D)(E + H)$$
$$S_6 = (D - E)(G + H)$$
$$S_7 = (A - C)(E + F)$$

products, we can compute $I, J, K,$

$$I = S_5 + S_6 + S_4 - S_2 = AE + BG.$$
$$J = S_1 + S_2 = AF + BH.$$
$$K = S_3 + S_4 = CE + DG.$$
$$L = S_1 - S_7 - S_3 + S_5 = CF + DH.$$

Thus, we can compute $Z = XY$ using seven recursive multiplications of matrices of size $(n/2)\times(n/2)$. Thus, we characterize the running time $T(n)$ as

$$T(n) = 7T\left(\frac{n}{2}\right) + bn^2$$

for some constant $b > 0$.

By the master theorem, we can multiply two n x n matrices in $O(n^{\log 7})$ time using Strassen's algorithm.

复杂度是$O(n^{log_2 7}) = O(n^3)$，略优于$O(n^3) = (n^{log_2 8})$

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j]$$

The exponent of matrix multiplication: smallest number ω such that for all ε>0 $O(n^{\omega+\varepsilon})$ operations suffice

- Standard algorithm ω ≤ 3
- Strassen (1969) ω < 2.81
- Pan (1978) ω < 2.79
- Bini et al. (1979) ω < 2.78
- Schönhage (1981) ω < 2.55
- Pan; Romani; Coppersmith + Winograd (1981-1982) ω < 2.50
- Strassen (1987) ω < 2.48
- Coppersmith + Winograd (1987) ω < 2.375
- Stothers (2010) ω < 2.3737
- Williams (2011) ω < 2.3729
- Le Gall (2014) ω < 2.37286

Ranking System

可以通过计数大小顺序相反的数对的个数来比较匹配度。

排列的倒置次数是衡量 “乱序” 程度的指标，可以用来衡量与恒等式排列的 “相似性”。

For example, the permutation

1 2 4 3 ×1

contains one inversion (the 4 and the 3), while the permutation

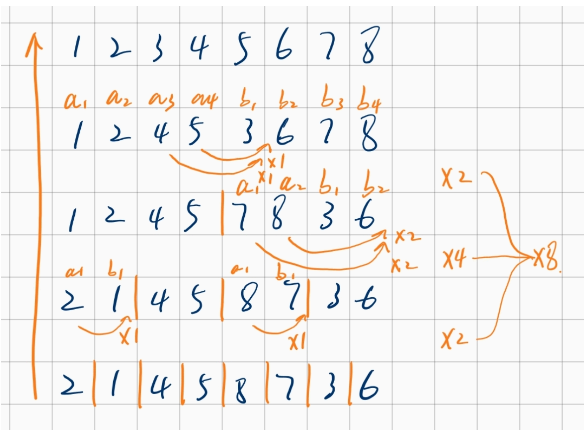
1 4 3 2 ×3

has three (the 3, 4 pair, the 2, 3 and the 2, 4 pair).

由于需要计算所有数对$C(n,2) = \frac{n!}{2!(n-2)} = \frac{n(n-1)}{2}$，故复杂度是$O(n^2)$

可以使用分治法把复杂度降低到$O(nlogn)$

```
COUNTINVERSIONS( L )
    ▷ Input: A list, L, of distinct integers.
    ▷ Output: The number of inversions in L.
if L has one element in it then
        there are no inversions, so Return (0, L)
else
        ▷ Divide the list into two halves
            A contains the first ⌊n/2⌋ elements
            B contains the last ⌈n/2⌉ elements
        (k_A, A) = COUNTINVERSIONS(A)
        (k_B, B) = COUNTINVERSIONS(B)
        (k, L) = MERGEANDCOUNT(A, B)
        Return (k_A + k_B + k, L)
```




```java
public class InversionCount {
    public static long mergeAndCount(int[] arr, int[] temp, int left, int mid, int right) {
        int i = left, j = mid + 1, k = left; // i 是左子数组的起点(left → mid), j 是右子数组的起点(mid + 1 → right); k 指向临时数组temp的当前填充位置
        long invCount = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
                invCount += (mid - i + 1); // 左边剩余的所有元素都比 arr[j] 大, 计数剩余元素个数
            }
        }

        while (i <= mid) temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];

        for (i = left; i <= right; i++) arr[i] = temp[i];   //temp排序好的部分
```

```java
倒给arr

        return invCount;
    }

    public static long mergeSortAndCount(int[] arr, int[] temp, int left, int right) {
        long invCount = 0;
        if (left < right) {
            int mid = (left + right) / 2;
            invCount += mergeSortAndCount(arr, temp, left, mid);  // left part
            invCount += mergeSortAndCount(arr, temp, mid + 1, right);  // right part
            invCount += mergeAndCount(arr, temp, left, mid, right);  // count
        }
        return invCount;
    }

    public static long countInversions(int[] arr) {
        int n = arr.length;
        int[] temp = new int[n];
        return mergeSortAndCount(arr, temp, 0, n - 1);
    }

    public static void main(String[] args) {
        int[] arr = {2,1,4,5,8,7,3,6};
        System.out.println("逆序数对个数: " + countInversions(arr));
    }
}
```

最优化问题-贪心算法

最优化问题有许多解决方案，我们希望可以找到问题的最大或最小值

最优化问题的算法通常是进行一系列步骤，在每步进行一组选择。

贪心算法总是在每步选择当下的最优选择，尽管不一定能达到全局最优。

我们称贪心算法能得到最优解的问题具有贪心选择性质（greedy-choice property）（每一步的贪心决策不会被后续步骤推翻，即通过一系列局部最优选择最终能够得到全局最优解）。

贪心算法可以用于一些困难的问题以生成大致的解。

分数背包问题（Fractional Knapsack Peoblem, FKP）

查找不超过总重量 W 的最大收益子集。在 FKP 中，我们被允许取每个项目的任意分数。

使用基于最大堆的优先级队列来存储 S 的项目，每个项目的键是单位价格$\frac{b_i}{w_i}$。

Object:	1	2	3	4
Benefit:	7	9	9	2
Weight:	3	4	5	2
Value index:	2.33	2.25	1.8	1

FKP满足greedy-choice property，故单位价格越高，越优先加入背包。

若背包未装满，则删除当前根节点，使用新的根节点。每个贪婪选择由于堆的性质需要$O(logn)$时间。

```
FractionalKnapsack(S, W)
    # S是项目的集合，其中项目i重量wi，价格bi
    # 输出每个项目的重量xi，使得总价最大
    for i in range(1, i)
        xi = 0
        vi = bi / wi
        insert (vi, i) to maxHeap H
    w = 0
    while w < weightMax
        remove root of H
        a = min(wi, weightMax - w)
        xi = a
        w = w + a
```

间隔调度（Interval Scheduling）

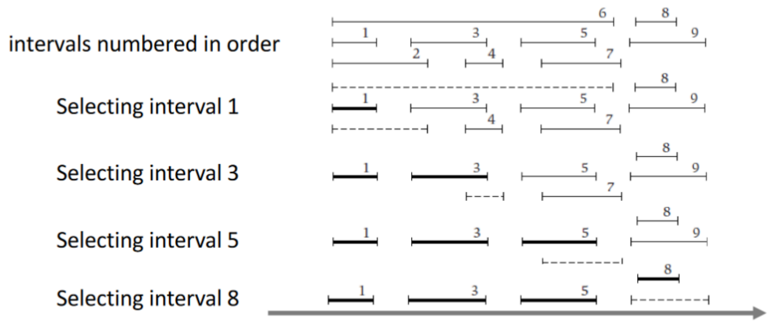
有一个任务集合（如请求使用房间的时间间隔），以及一台可以处理这些任务的机器（如可以举行会议的单个房间）。

目标：我们想要选择任务的子集，以便最大限度地增加我们可以在计算机上计划的任务数。

对任务i和j的起始时间s和结束时间f，$f_i \leq s_j$且$f_j \leq s_i$只有当两个任务不冲突时，才能执行这两个任务。

目标：选择具有最大大小 （任务数） 的非冲突任务的子集。

贪心算法选择最早开始，最短时间都不行。

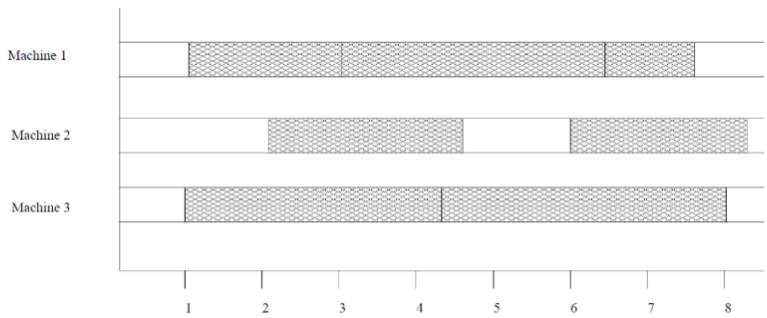


图中被选择的任务加粗，被删除的任务虚线。

选择最先完成的任务是可以的，$O(nlogn)$。根据任务的完成时间对任务进行排序。然后选择最先完成的任务，删除与此任务冲突的所有任务，并重复直到完成。

任务调度（Task Scheduling）

假设我们仍然有一组 T 的 n 个任务，其开始和完成时间与以前一样。现在我们想使用尽可能少的机器来安排所有任务（以一种不冲突的方式）。



贪心算法仍然可以解决这个问题。

按开始时间排序任务，然后对于每个任务 i，如果我们有可以处理任务 i 的机器，它会被调度到该机器上。否则，分配一台新机器，在其上安排任务 i，然后重复贪婪选择过程，直到我们考虑了 T 中的所有任务。

```
TaskSchedule(T)
    # T是任务与开始结束时间的集合，输出不冲突的T时间表
    m = 0
    while !T.isEmpty
        removeMin(T)
        if Exist Machine j no conflicts
            Schedule(i,j)
        else
            m +=1
            Schedule(i,m)
```

动态规划（Dynamic Programming）

用对存储在特殊表中的已计算值的引用来替换（可能）重复的递归调用。动态规划主要用于优化问题。它通常应用于无法通过暴力搜索最优值的情况。但只有当问题具有一定量的可利用结构时，动态规划才是有效的。

标志：

- 最优子结构（optimal substructure）：问题的最优解由子问题的最优解组成
- 重叠的子问题（overlapping subproblems）：总共几个子问题，每个子问题有很多重复的实例（递归算法反复访问同一问题）

基本思想：自下而上解决，构建一个已解决的子问题表，用于解决较大的子问题

01背包问题（{0 – 1} Knapsack Problem）

一般解需要$O(2^n)$时间，应使用动态规划

设S_k是有k个项目的集合，S_0=∅。B[k, w]是获得最大收益的S_k的子集，最大重量w。

$$B[k,w]=\begin{cases} B[k-1,w]\ ,if\ w<w_k \\ max\{B[k-1,w],bk+B[k-1,w-w_k]\}\ ,otherwise \end{cases}$$

W=10,

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w<w_k \\ \max\{B[k-1,w],\ b_k+B[k-1,w-w_k]\} & \text{otherwise} \end{cases}$$

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	25	25	25	25
2	0	0	15	15	15	15	15	25	25	40	40
3	0	0	15	20	20	35	35	35	35	40	45
4	0	0	15	20	20	35	36	36	51	56	56

The solution is to pick up items 3 and 4 for a maximum benefit of 56. 25