Module04-03
C++ 标准库：常用算法简介

- 数据结构简介
- 标准容器
- → 常用算法简介
- 标准算法与函数对象
- 迭代器
- 字符串
- I/O 流
- 数值

标准库 - 常用算法介绍

- 这部分课程将介绍常用的几种排序算法：
  - 排序算法
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Shell Sort
    - Quick Sort
    - Heap Sort

常用算法介绍 - 参考数目

- 本单元参考书目：
  - Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition
    作者： Robert Sedgewick

常用算法介绍 - Bubble Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

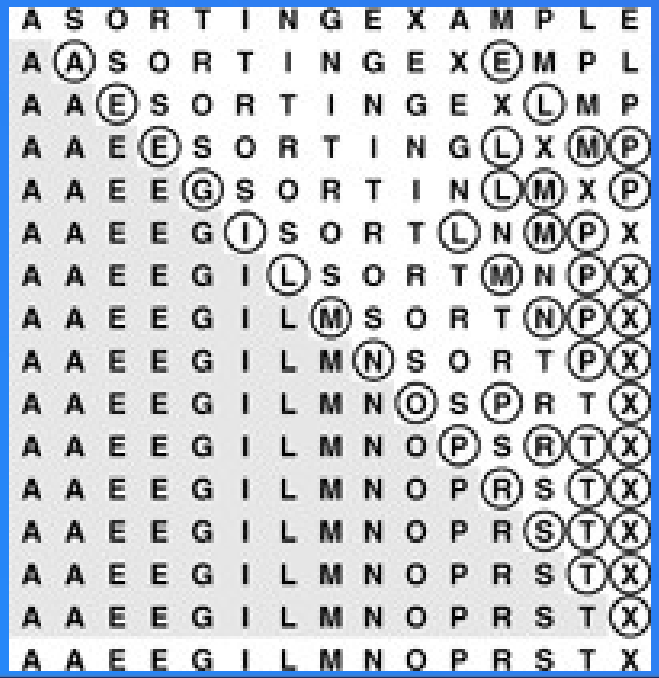
常用算法介绍 - Bubble Sort

- 关于 Bubble Sort
  - Bubble Sort 是一种低效，但容易实现的排序算法，通常用作算法的入门讲解
  - 算法基本思路：
    - 假设从右至左迭代序列中的元素，在第一轮中碰到最小的元素，不断将其与其左边的元素交换，直至其到达最左端
    - 第二轮同第一轮，只是将第二小的元素放到左边第二位
    - 以此类推，直到只剩最右边一个元素结束
  - 算法复杂度：
    - Bubble Sort 平均和最坏情况下大约进行 $N^2/2$ 次比较以及 $N^2/2$ 次交换

常用算法介绍 - Bubble Sort

- Bubble Sort 代码和图示

```
template<class Item>
void bubbleSort(Item a[], int left, int right) {
    int x = 0;
    for (int i = left; i < right; ++i)
        for (int j = right; j > i; --j)
            if (a[j] < a[j - 1])
                swap(a[j], a[j - 1]);
}
```



常用算法介绍 - Selection Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

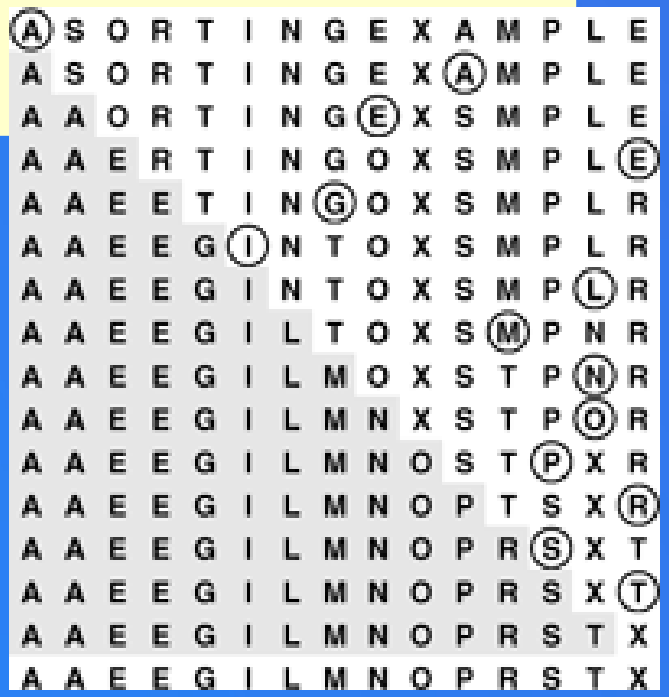
常用算法介绍 - Selection Sort

- 关于 Selection Sort
  - Selection Sort 也是简单的、容易实现的排序算法，但比 Bubble Sort 高效。
  - 算法基本思路：
    - 第一轮从序列中找出最小的元素，与序列的第一个位置上的元素交换
    - 第二轮从序列中找出第二小的元素，与序列的第二个位置上的元素交换
    - 以此类推
  - 算法复杂度：
    - Selection Sort 大约进行 $N^2/2$ 次比较以及 N 次交换

常用算法介绍 - Selection Sort

- Selection Sort 代码和图示

```
template<class Item>
void selectionSort(Item a[], int left, int right) {
    for (int i = left; i < right; ++i) {
        int min = i;
        for (int j = i + 1; j <= right; ++j)
            if (a[j] < a[min])
                min = j;
        swap(a[i], a[min]);
    }
}
```



常用算法介绍 - Insertion Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

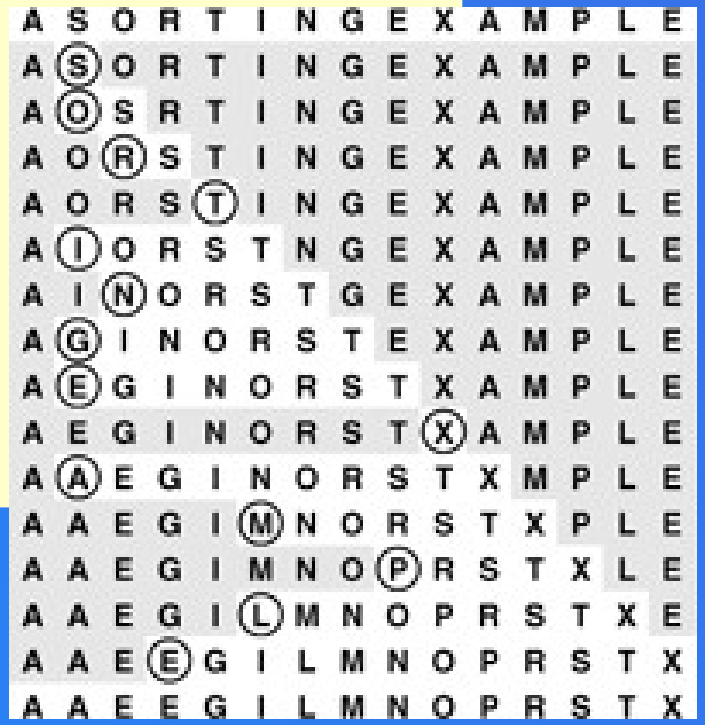
常用算法介绍 - Insertion Sort

- 关于 Insertion Sort
  - 玩桥牌的过程中，人们整理手中的扑克牌，使用的是 Insertion Sort
  - 算法基本思路：
    - 从左至右迭代序列，每次迭代都保证当前 index 到序列最左边的子序列总是已序的，但不一定是最终位置
    - 当当前 index 所在的元素比最左边元素更小的元素时，将左边已序的序列全部右移一个位置，且将当前 index 的元素插入到最左边
    - 以此类推，直到迭代至序列结束
  - 算法复杂度：
    - Insertion Sort 平均情况下大约进行 $N^2/4$ 次比较以及 $N^2/4$ 次半交换（赋值），最坏情况下则是 2 倍于此

常用算法介绍 - Insertion Sort

- Insertion Sort 代码和图示

```
template<class Item>
void insertionSort(Item a[], int left, int right) {
    int i = right;
    for (; i > left; --i)
        if (a[i] < a[i - 1])
            swap(a[i - 1], a[i]);
    for (i = left + 2; i <= right; ++i) {
        int j = i;
        Item v = a[i];
        while (v < a[j - 1]) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = v;
    }
}
```



常用算法介绍 - Shell Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

常用算法介绍 - Shell Sort

- 关于 Shell Sort
  - Shell Sort 是一种改良的 Insertion Sort，由 Donald Shell 于 1959 年提出，也称增量递减 (diminishing increment) 算法
  - 算法基本思路：
    - 0，假设有序列 a 中有 n 个待排序的元素
    - 1，按增量序列中的确定的 gap 将 a 分成 n/gap 个子序列，将每个子序列用 Insertion Sort 算法排成有序序列
    - 2，将 gap 递减到增量序列中的下一个 gap，重复第 2 步的操作

常用算法介绍 - Shell Sort

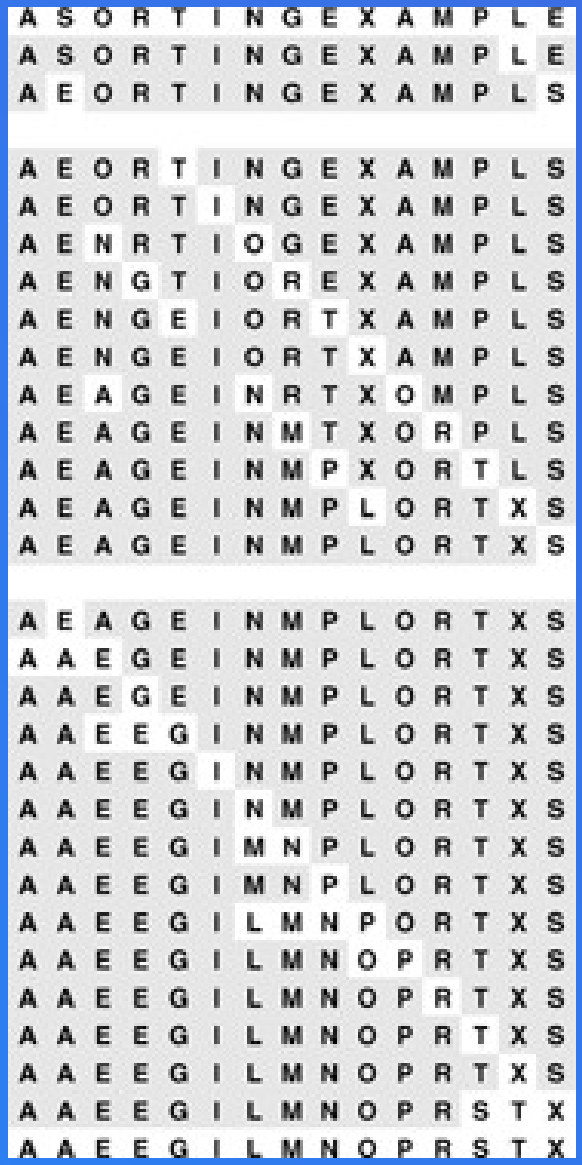
- 关于 Shell Sort（续）
  - 关于增量序列
    - Shell Sort 的性能很大程度上取决于增量序列 (gaps) 的选择
    - 有效的增量序列如：
      - 1, 4, 13, 40, 121, ... ($d_i = d_{i-1} * 3 + 1, i >= 1$)
      - 1, 8, 23, 77, 281, ... ($d_i = d_{i-1} * i + 7, i >= 1$)
      - ...

常用算法介绍 - Shell Sort

- Shell Sort 示例图示

右图所示的排序过程选用
1， 4， 13， 40， ... 增量序列

右图上部示意选用 gap 为 13
右图中部示意选用 gap 为 4
有图底部示意选用 gap 为 1



常用算法介绍 - Shell Sort

- Shell Sort 代码

```
// 以下代码选用的是: 1,4,13,40,...增量序列

template<class Item>
void shellSort(Item a[], int left, int right) {
    int h = 1;
    for (; h <= (right - left) / 9; h = 3 * h + 1)
        ;
    for (; h > 0; h /= 3)
        for (int i = left + h; i <= right; ++i) {
            int j = i;
            Item v = a[i];
            while (j >= left + h && v < a[j - h]) {
                a[j] = a[j - h];
                j -= h;
            }
            a[j] = v;
        }
}
```

常用算法介绍 - Shell Sort

- Shell Sort 性能
  - 根据所采用的增量序列不同， Shell Sort 所呈现的性能也不同：
    - 采用 1,4,13,40,121,... 序列，比较次数小于 $O(N^{3/2})$
    - 采用 1,8,23,77,281,... 序列，比较次数小于 $O(N^{4/3})$

常用算法介绍 - Shell Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

常用算法介绍 - Quick Sort

- 关于 Quick Sort
  - Quick Sort 是目前已知排序算法中可能提供最好性能的算法
  - Quick Sort 采用的是一种分治法，不断将序列一分为二，将划分出的 2 个子序列各自独立排序
  - Quick Sort 依赖于划分操作
  - 每次划分操作之后：
    - `a[i]` 在最终的正确的位置
    - `a[left] .. a[i-1]` 区间所有元素均不大于 `a[i]`
    - `a[i+1] .. a[right]` 区间所有元素均不小于 `a[i]`

常用算法介绍 - Quick Sort

- Quick Sort 基本算法代码
  - partition

```
template<class Item>
int partition(Item a[], int left, int right) {
    int i = left - 1, j = right;
    Item v = a[right];
    for (;;) {
        while (a[++i] < v)
            ;
        while (v < a[--j])
            if (j == left)
                break;
        if (i >= j)
            break;
        swap(a[i], a[j]);
    }
    swap(a[i], a[right]);
    return i;
}
```

常用算法介绍 - Quick Sort

- Quick Sort 基本算法代码
  - quick sort

```
template<class Item>
void quickSort(Item a[], int left, int right) {
    if (right <= left)
        return;
    int i = partition(a, left, right);
    quickSort(a, left, i - 1);
    quickSort(a, i + 1, right);
}
```

常用算法介绍 - Quick Sort

- Quick Sort 性能
  - 最坏情况下， Quick Sort 进行 $N^2/2$ 次比较
  - 平均情况下， Quick Sort 进行 $2N \ln N$ 次比较
- 影响 Quick Sort 性能的几种情况：
  - 对一个已序或近似已序（逆序）的序列排序，这时将导致 Quick Sort 的性能退化（最坏情况）
  - 对于小子序列 (small subfile) 的排序，多次划分后将出现很多小的子序列
  - 序列中重复（相等）的 key 出现的比率

常用算法介绍 - Quick Sort

- 针对小子序列的改善代码
  - Median-of-Three Partitioning

```
static const int M = 10;
template<class Item>
void quickSort(Item a[], int left, int right) {
    if (right - left <= M)
        return;
    swap(a[(left + right) / 2], a[right - 1]);
    if (a[right - 1] < a[left]) swap(a[left], a[right - 1]);
    if (a[right] < a[left]) swap(a[left], a[right]);
    if (a[right] < a[right – 1]) swap(a[right - 1], a[right]);
    int i = partition(a, left + 1, right - 1);
    quickSort(a, left, i - 1);
    quickSort(a, i + 1, right);
}
template<class Item>
void hybridSort(Item a[], int left, int right) {
    quickSort(a, left, right); // 划分到子序列中的元素个数小于M时停止
    insertionSort(a, left, right); // 子序列由插入排序处理
}
```

常用算法介绍 - Quick Sort

- 针对重复 key 的改善代码
  - Quicksort with three-way partitioning

```
template<class Item>
void quickSort(Item a[], int left, int right) {
    int k;
    Item v = a[right];
    if (right <= left)
        return;
    int i = left - 1, j = right, p = left - 1, q = right;
    for (;;) {
        while (a[++i] < v) ;
        while (v < a[--j])
            if (j == left) break;
        if (i >= j) break;
        swap(a[i], a[j]);
        if (a[i] == v) {
            ++p;
            swap(a[p], a[i]);
        }
```

常用算法介绍 - Quick Sort

- 针对重复 key 的改善代码（续）
  - Quicksort with three-way partitioning

```
        if (v == a[j]) {
            --q;
            swap(a[q], a[j]);
        }
    }
    swap(a[i], a[right]);
    j = i - 1;
    i = i + 1;
    for (k = left; k <= p; ++k, --j)
        swap(a[k], a[j]);
    for (k = right - 1; k >= q; --k, ++i)
        swap(a[k], a[i]);
    quickSort(a, left, j);
    quickSort(a, i, right);
}
```

常用算法介绍 - Quick Sort

- Selection
  - 在很多情况下，需将第 k 小的元素 E 放到序列中正确的位置，且保证 E 以前的元素不大于 E，且 E 以后的元素不小于 E，但不要求排序，如将序列 a 中第 3 小的元素放到第 3 位，且保证第 1、 2 位的元素不大于第 3 位的元素，但不要求这三者经过排序
  - 该操作使用 Quick Sort 的 partition 操作十分合适

  注： C++ STL 算法中的 nth_element() 即为该操作 (Selection) 的实现

常用算法介绍 - Quick Sort

- Selection 代码

```
// 递归方式
template<class Item>
void select(Item a[], int left, int right, int k) {
    if (right <= left)
        return;
    int i = partition(a, left, right);
    if (i > k) select(a, left, i - 1, k);
    if (i < k) select(a, i + 1, right, k);
}
```

```
// 迭代方式
template<class Item>
void select(Item a[], int left, int right, int k) {
    while (right > left) {
        int i = partition(a, left, right);
        if (i >= k) right = i - 1;
        if (i <= k) left = i + 1;
    }
}
```

常用算法介绍 - Quick Sort

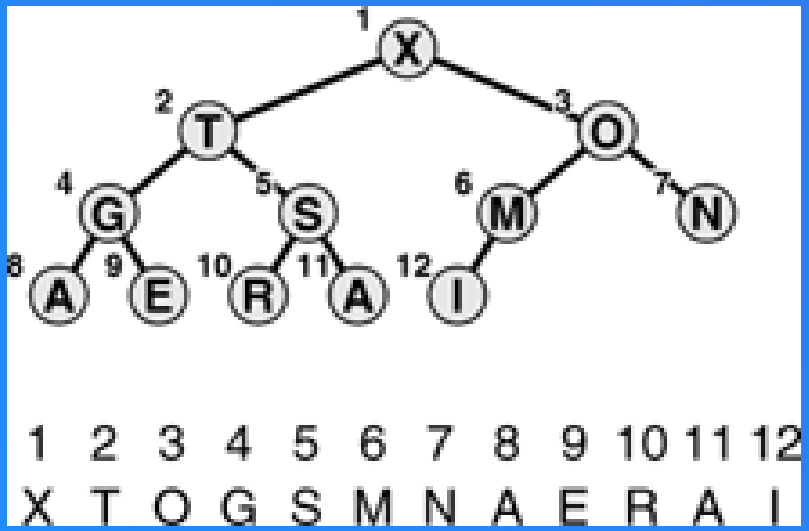
- 基于 Quick Sort 的 selection 算法的性能
  - 平均情况为： O(N)，即线性

常用算法介绍 - Heap Sort

- 常用算法介绍
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Heap Sort

常用算法介绍 - Heap Sort

- Heap 数据结构的性质：
  - 在一个经过堆化 (heapify) 的二叉树中，任意子节点都不大于根节点
- Heap 数据结构的表示：
  - 用完全二叉树（什么是完全二叉树？）来描述、用数组来表示 Heap 数据结构



父节点与子节点的 index（假设 index 从 1 开始）关系：

p — 父节点的 index

l — 左子节点的 index

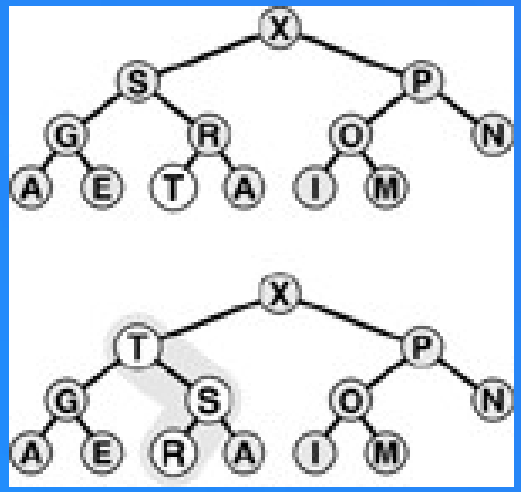
r — 右子节点的 index，则

```
l = 2 * p
r = 2 * p + 1
```

常用算法介绍 - Heap Sort

- Heap 数据结构相关算法 - 自底向上堆化

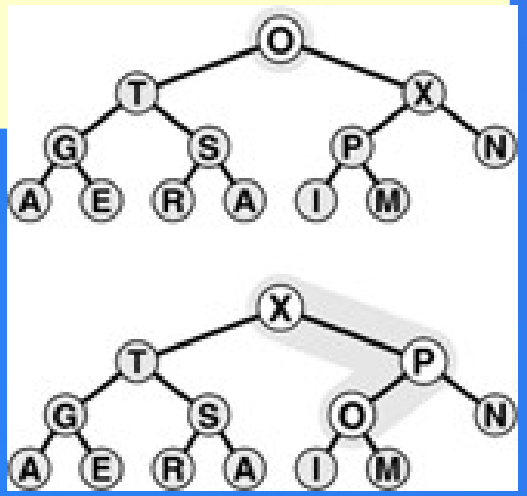
```
template<class Item>
void fixUp(Item a[], int k) {
    while (k > 1 && a[k / 2] < a[k]) {
        swap(a[k], a[k / 2]);
        k = k / 2;
    }
}
```



常用算法介绍 - Heap Sort

- Heap 数据结构相关算法 - 自顶向下堆化

```
template<class Item>
void fixDown(Item a[], int k, int N) {
    while (2 * k <= N) {
        int j = 2 * k;
        if (j < N && a[j] < a[j + 1])
            ++j;
        if (!(a[k] < a[j]))
            break;
        swap(a[k], a[j]);
        k = j;
    }
}
```



常用算法介绍 - Heap Sort

- Priority Queue - 优先队列
  - 优先队列通常基于堆数据结构实现

常用算法介绍 - Heap Sort

- 关于 Heap Sort
  - Heap Sort 算法基本思路
    - 1，序列堆化 (heapify)
    - 2，将堆顶元素 (tree root) 与堆底最后一个元素交换
    - 3，将堆顶到堆底倒手第 2 个元素之间的序列堆化
    - 4，将堆顶元素 (tree root) 与堆底倒手第 2 个元素交换
    - 重复 3， 4 步骤，不断减小堆化的范围
  - 使用自底向上的方式堆化序列的性能
    - 自底向上方式构建堆数据结构，所需线性时间 (O(N))

常用算法介绍 - Heap Sort

- Heap Sort 代码

```
template<class Item>
void heapSort(Item a[], int left, int right) {
    int k, N = right - left + 1;
    Item *pq = a + left - 1;
    for (k = N / 2; k >= 1; --k) // 堆化输入序列
        fixDown(pq, k, N);
    while (N > 1) {     // 排序开始
        swap(pq[1], pq[N]); // 交换root和堆最后的元素
        fixDown(pq, 1, --N); // 重新堆化剩余的元素
    }
}
```

常用算法介绍 - Heap Sort

- Heap Sort 算法的性能
  - 对有 N 个元素的序列进行排序，堆排序进行少于 2N lg N 次比较