

第2章

程序性能分析

程序性能

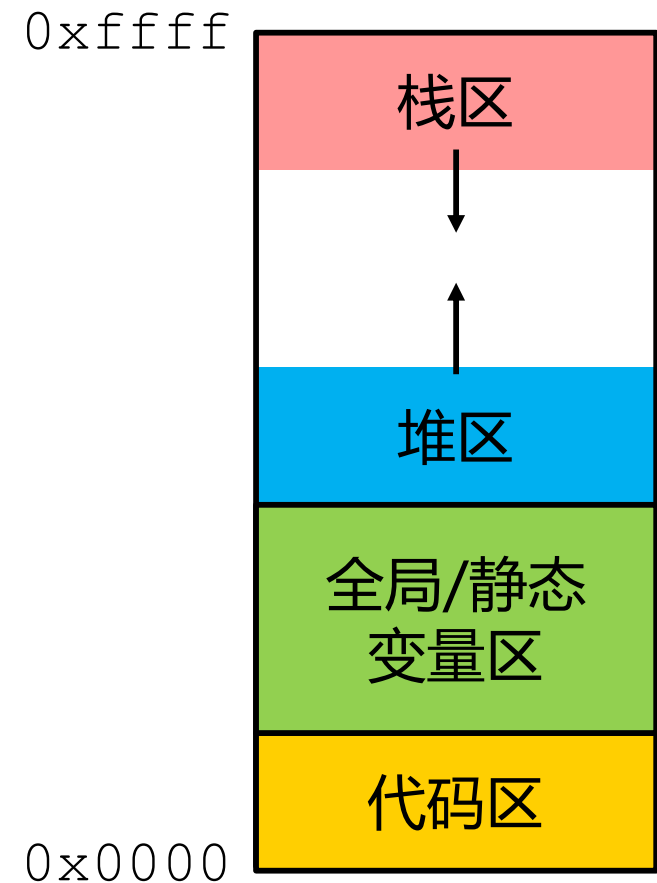
- **程序性能**是指运行一个程序所需要的**内存**和**时间**的多少
 - 空间复杂度
 - 时间复杂度
- 确定一个程序的性能有两种方法
 - **性能分析**
 - 性能测量

空间复杂度

- **空间复杂度(space complexity)** 是指运行完一个程序所需要的内存大小

- 程序所需要的空间构成

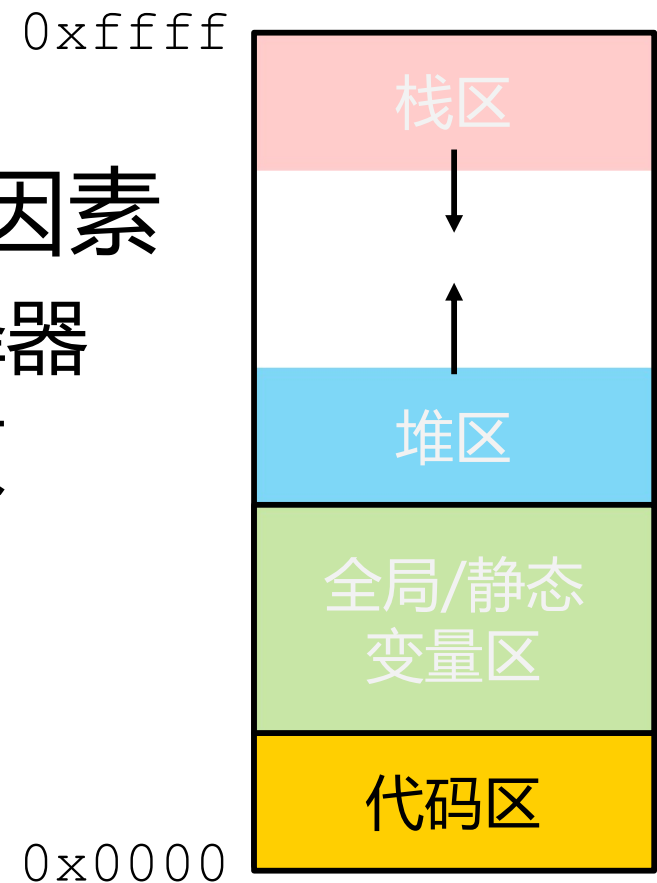
- 指令空间
- 数据空间
- 环境栈空间



指令空间

- **指令空间**是用来存储经过编译之后的程序指令所需的空间

- 指令空间的大小取决于如下因素
 - 把程序编译成机器代码的编译器
 - 编译时实际采用的编译器选项
 - 目标计算机



指令空间实例

■ 计算表达式 $a+b+b*c+(a+b-c)/(a+b)+4$

```
int calc(int a, int b, int c)
{
    return a + b + b * c + (a + b - c) / (a + b) + 4;
}
```

g++ -S

```
_Z4calciiii:
.LFB0:
.cfi_startproc
endbr64
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
movl    %edi, -4(%rbp)
movl    %esi, -8(%rbp)
movl    %edx, -12(%rbp)
movl    -4(%rbp), %edx
movl    -8(%rbp), %eax
addl    %eax, %edx
movl    -8(%rbp), %eax
imull   -12(%rbp), %eax
leal    (%rdx,%rax), %esi
movl    -4(%rbp), %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
subl    -12(%rbp), %eax
movl    -4(%rbp), %ecx
movl    -8(%rbp), %edx
leal    (%rdx,%rcx), %edi
cld
idivl   %edi
addl    %esi, %eax
addl    $4, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
```

g++ -O3 -S

```
.type _Z4calciiii, @function
_Z4calciiii:
.LFB0:
.cfi_startproc
endbr64
addl    %esi, %edi
imull   %edx, %esi
movl    %edi, %eax
subl    %edx, %eax
cld
addl    %edi, %esi
idivl   %edi
leal    4(%rsi,%rax), %eax
ret
.cfi_endproc
.LFE0:
```

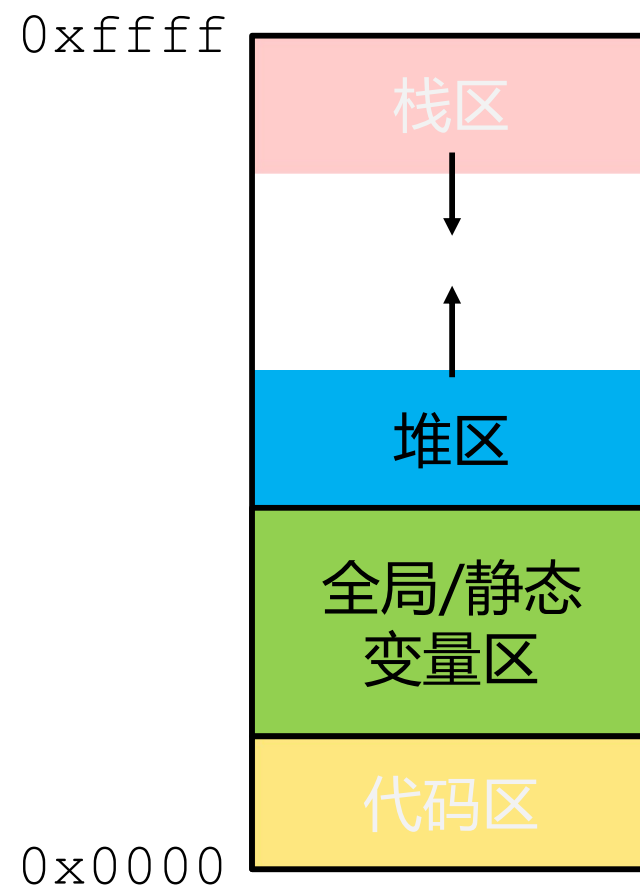
clang -S

```
_Z4calciiii:
.cfi_startproc
# %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
movl    %edi, -4(%rbp)
movl    %esi, -8(%rbp)
movl    %edx, -12(%rbp)
movl    -4(%rbp), %eax
addl    -8(%rbp), %eax
movl    -8(%rbp), %ecx
imull   -12(%rbp), %ecx
addl    %ecx, %eax
movl    -4(%rbp), %ecx
addl    -8(%rbp), %ecx
subl    -12(%rbp), %ecx
movl    -4(%rbp), %edx
addl    -8(%rbp), %edx
movl    %eax, -16(%rbp) # 4-byte Spill
movl    %ecx, %eax
movl    %edx, -20(%rbp) # 4-byte Spill
cld
movl    -20(%rbp), %ecx # 4-byte Reload
idivl   %ecx
movl    -16(%rbp), %esi # 4-byte Reload
addl    %eax, %esi
addl    $4, %esi
movl    %esi, %eax
popq    %rbp
.cfi_def_cfa %rsp, 8
retq
.Lfunc_end0:
```

数据空间

■ 数据空间用来存储常量和变量（非局部变量）所需的空间

- 简单变量和常量
- 结构体变量
- 数组
- 动态分配的内存



变量大小

■ 各种变量在一个典型64位机器上的大小

```
int main(int argc, char *argv[])
{
```

```
pr xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make varsize && ./varsize
```

```
pr g++ varsize.cpp -o varsize
```

```
pr The sizes of data types:
```

```
pr         bool: 1 bytes
```

```
pr         char: 1 bytes
```

```
pr     unsigned char: 1 bytes
```

```
pr         short: 2 bytes
```

```
pr     unsigned short: 2 bytes
```

```
pr         int: 4 bytes
```

```
pr     unsigned int: 4 bytes
```

```
pr         long: 8 bytes
```

```
pr     unsigned long: 8 bytes
```

```
pr         long long: 8 bytes
```

```
pr     unsigned long long: 8 bytes
```

```
pr         float: 4 bytes
```

```
pr         double: 8 bytes
```

```
pr     long double: 16 bytes
```

```
pr         void *: 8 bytes
```

```
pr         int *: 8 bytes
```

```
pr         double *: 8 bytes
```

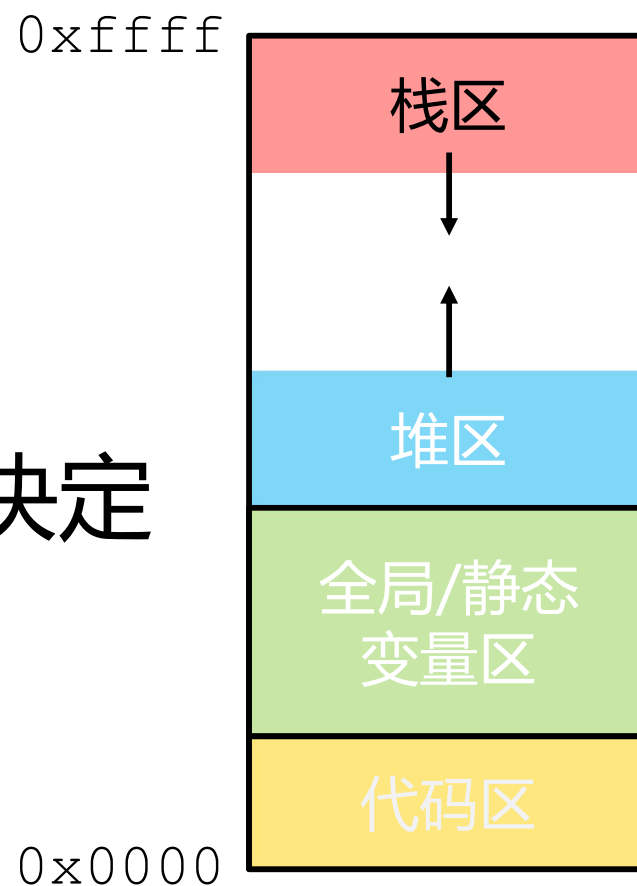
```
}
```

环境栈

- **环境栈**用来保存函数调用相关的信息

- 形式参数
- 局部变量
- 返回地址

- 环境栈具体的使用由编译器决定



结论

- 一个程序所需要的空间**难以精确分析**

- 通过**实例特征**确定**部分空间需求**

- 决定问题规模的因素
- 对 **n** 个数排序，实例特征为 **n**
- 两个 **$m \times n$** 矩阵相加，实例特征为 **m** 、 **n**

- 空间复杂度的度量

$$\begin{array}{ccc} & c + S_p & \\ \nearrow & & \nwarrow \\ \text{固定部分} & & \text{可变部分 (依赖实例特征)} \end{array}$$

结论 (续)

■ 固定部分: 独立于实例特征

- 指令空间、简单变量及常量所占用空间等
- 环境栈 (非递归)

■ 可变部分: 依赖实例特征

- 动态分配的空间
- 环境栈 (递归)

递归求和函数:

$$n+1 \left\{ \begin{array}{l} rSum(a, n) \\ rSum(a, n-1) \\ \dots \\ rSum(a, 1) \\ rSum(a, 0) \end{array} \right.$$

通过实例特征估算可变部分 s_p

顺序搜索

- 在长度为n的数组中搜索对应元素
 - 实例特征： n

```
#include <iostream>
using namespace std;

template <typename T>
int sequentialSearch(T a[], const T& x, int n)
{
    int i;
    for(i = 0; i < n && a[i] != x; i++);

    if(i == n)
        return -1;
    else
        return i;
}

int main(int argc, char *argv[])
{
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int res = sequentialSearch(arr, 3, 10);

    cout << "The search result: " << res << endl;

    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make sequentialSearch
g++ sequentialSearch.cpp -o sequentialSearch
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./sequentialSearch
The search result: 3
```

sequentialSearch空间复杂度:

- 形参： $a[]$ (8B), x (T), n (4B)
 - $a[]$ 的实际数据在main中
- 局部变量： i (4B)
- 返回地址： 8B
- $S_{\text{sequentialSearch}} = 0$

迭代求和

- 求长度为n的数组中所有元素的和（迭代）
 - 实例特征： **n**

```
template<typename T>
T sum(T a[], int n)
{
    T sum = 0;
    for(int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

sum空间复杂度：

- 形参： **a[] (8B)** , **n (4B)**
 - **a[]**的实际数据在**main**中
- 局部变量： **i (4B)** , **sum (T)**
- 返回地址： **8B**
- $S_{\text{sum}} = 0$

递归求和

- 求长度为n的数组中所有元素的和（递归）
 - 实例特征：n

```
template<typename T>
T sumR(T a[], int n)
{
    if(n > 0)
        return sumR(a, n-1) + a[n-1];
    return 0;
}
```

需要展开分析

递归求和函数:

n+1 { $rSum(a, n)$
 $rSum(a, n-1)$
.....
 $rSum(a, 1)$
 $rSum(a, 0)$

sumR空间复杂度:

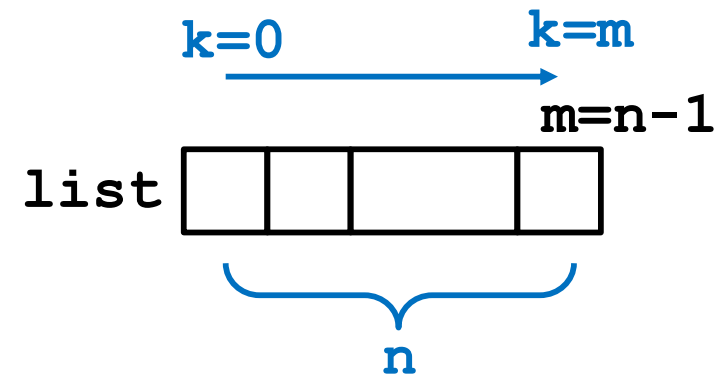
- 形参（单次）：a[] (8B), n (4B)
- 返回地址（单次）：8B
- $S_{sumR} = (n+1) * 20$
- 32位机器： $S_{sumR} = (n+1) * 12$
 - 地址大小为4B

全排列

■ 求长度为n的数组中所有元素的全排列

■ 实例特征： **n**

```
template<typename T>
void permutations(T list[], int k, int m)
{
    if(k == m) {
        copy(list, list+m+1,
             ostream_iterator<T>(cout, " "));
        cout << endl;
    }else{
        for(int i = k; i <= m; i++){
            swap(list[k], list[i]);
            permutations(list, k+1, m);
            swap(list[k], list[i]);
        }
    }
}
```



permutations空间复杂度:

- 形参 (单次) : `list[]` (8B) ,
`k` (4B) , `m` (4B)
- 局部变量: `i` (4B)
- 返回地址 (单次) : 8B
- $S_{\text{permutations}} = n * 28$ (64-bit)
- $S_{\text{permutations}} = n * 20$ (32-bit)

时间复杂度

- **时间复杂度**(time complexity) 是指运行完一个程序所需要的**计算机时间**
- 绝对运行时间**难以**用于比较不同程序（相同功能）
 - 不同机器的性能不同
 - 不同编译器的性能不同

估算时间复杂度

- 根据**实例特征**估算一个程序运行的**计算机时间**（**时间复杂度**）
 - **操作计数**(operation counts): 找出一个或多个**关键操作**，确定这些关键操作的次数；
 - **步数**(step counts): 确定程序总的步数
- **关键操作**: 对时间复杂度影响最大的操作

最大元素

- 求长度为n的数组中最大的元素
 - 实例特征: **n**

```
template<typename T>
int indexOfMax(T a[], int n)
{
    if(n <= 0)
        throw illegalParameterValue("n must be > 0");

    int indexOfMax = 0;
    for(int i = 1; i < n; i++)
        if(a[indexOfMax] < a[i])
            indexOfMax = i;
    return indexOfMax;
}

int main(int argc, char *argv[])
{
    int arr[] =
        {10, 100, 1000, 10000, 9999,
         999, 99, 9, 40, 30};
    int maxIdx = indexOfMax(arr, 10);

    cout << "The max element is arr["
          << maxIdx << "]: "
          << arr[maxIdx] << endl;
    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make max
g++ max.cpp -o max
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./max
The max element is arr[3]: 10000
```

indexOfMax操作计数:

- 关键操作: 比较
 - $n \leq 0$ 时, 0次
 - $n > 0$ 时, $n-1$ 次
- 比较次数: $\max(0, n-1)$

多项式求值

■ 求最高阶为n的任意多项式的值

$$P(x) = \sum_{i=0}^n c_i x^i$$

■ 实例特征: **n**

数组长度n+1

```
template<typename T>
T polyEval(T coeff[], int n, const T& x)
{
    T y=1, value = coeff[0];
    for(int i = 1; i <= n; i++) {
        y *= x;
        value += y * coeff[i];
    }
    return value;
}
```

```
int main(int argc, char *argv[])
{
    int arr[] = {3, 4, 5};
    int res = polyEval(arr, 2, 2);

    cout << "if x = 2, 3 + 4 * x + 5 * x^2 = "
          << res << endl;
    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make poly1
g++ poly1.cpp -o poly1
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./poly1
if x = 2, 3 + 4 * x + 5 * x^2 = 31
```

{3, 4, 5}表示 $3*x^0+4*x^1+5*x^2$ **polyEval操作计数:**

- 关键操作: **乘法、加法**
- 乘法次数: **2n**
- 加法次数: **n**

多项式求值

■ 求最高阶为n的任意多项式的值

Horner法则:

$$P(x) = (\dots (c_n \times x + c_{n-1}) \times x + c_{n-2}) \times x + c_{n-3}) \times x + \dots) \times x + c_0$$

```
template<typename T>
T horner(T coeff[], int n, const T& x)
{
    T value = coeff[n];
    for(int i = 1; i <= n; i++)
        value = value*x + coeff[n-i];
    return value;
}

int main(int argc, char *argv[])
{
    int arr[] = {3, 4, 5};
    int res = horner(arr, 2, 2);

    cout << "if x = 2, 3 + 4 * x + 5 * x^2 = "
          << res << endl;
    return 0;
}
```

从最高阶开始计算

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make poly2
g++ poly2.cpp -o poly2
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./poly2
if x = 2, 3 + 4 * x + 5 * x^2 = 31
```

horner操作计数:

- 关键操作: 乘法、加法
- 乘法次数: n
- 加法次数: n

名次计算

- 计算大小为 n 数组中每个元素的名次
 - 最小元素 $\rightarrow 0$ 、最大元素 $\rightarrow n-1$

```
template<typename T>
void rank1(T a[], int n, int r[])
{
    for(int i = 0; i < n; i++)
        r[i] = 0;

    for(int i = 1; i < n; i++)
        for(int j = 0; j < i; j++)
            if(a[j] <= a[i])
                r[i]++;
            else
                r[j]++;
}

int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};
    int ranks[5];

    rank1(arr, 5, ranks);

    cout << " arr[]: "
          << toString(arr, 5) << endl;
    cout << "ranks[]: "
          << toString(ranks, 5) << endl;
    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make rank1
g++ rank1.cpp -o rank1
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./rank1
arr[]: [4, 3, 9, 3, 7]
ranks[]: [2, 0, 4, 1, 3]
```

$j \rightarrow i$

$a: [4, 3, 9, 3, \dots, 7]$

$r: [1, 0, 2, 0, \dots, 0]$

$[2, 0, 3, 1, \dots, 0]$

rank操作计数:

- 关键操作: **比较**
- 比较次数: $1+2+\dots+(n-1)=n*(n-1)/2$

名次排序

- 利用元素名次为大小为n数组排序
 - $a[]$ 与 $r[]$ 已知

```
template<typename T>
void rearrange(T a[], int n, int r[])
{
    T *u = new T[n];

    for(int i = 0; i < n; i++)
        u[r[i]] = a[i];

    for(int i = 0; i < n; i++)
        a[i] = u[i];

    delete [] u;
}
```

```
int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};
    int ranks[5];

    rank1(arr, 5, ranks);

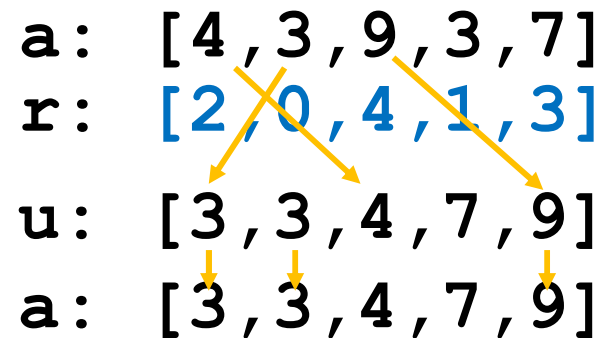
    cout << "  arr[]: " << toString(arr, 5) << endl;
    cout << "ranks[]: " << toString(ranks, 5) << endl;

    rearrange(arr, 5, ranks);

    cout << "After rearranging the array:" << endl;
    cout << "  arr[]: " << toString(arr, 5) << endl;
    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make rearrange
g++ rearrange.cpp -o rearrange
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./rearrange
arr[]: [4, 3, 9, 3, 7]
ranks[]: [2, 0, 4, 1, 3]
After rearranging the array:
arr[]: [3, 3, 4, 7, 9]
```

a: [4, 3, 9, 3, 7]
r: [2, 0, 4, 1, 3]
u: [3, 3, 4, 7, 9]
a: [3, 3, 4, 7, 9]



rearrange操作计数:

- 关键操作: 移动/拷贝
- 移动/拷贝次数: $2n$

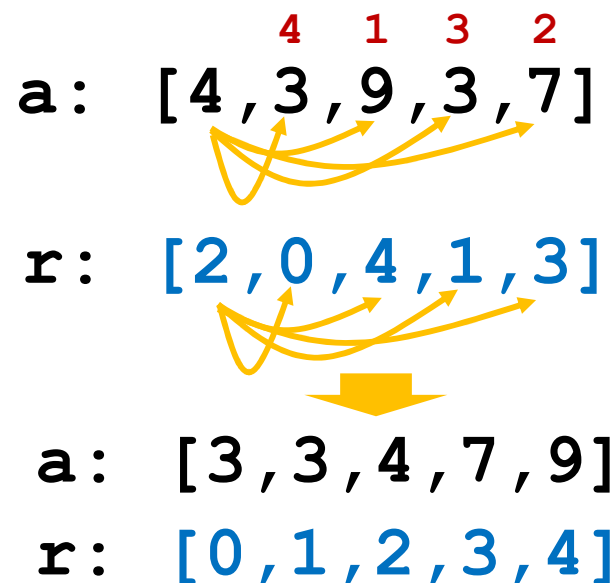
名次原地排序

- 利用元素名次为大小为n数组排序
 - 移除临时数组

```
template<typename T>
void rearrange(T a[], int n, int r[])
{
    for(int i = 0; i < n; i++)
        while(r[i] != i){
            int t = r[i];
            swap(a[i], a[t]);
            swap(r[i], r[t]);
        }
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make rearrange2
g++ rearrange2.cpp -o rearrange2
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./rearrange2
arr[]: [4, 3, 9, 3, 7]
ranks[]: [2, 0, 4, 1, 3]
After rearranging the array:
arr[]: [3, 3, 4, 7, 9]
```

i=0时的交换顺序:



rearrange2操作计数:

- 关键操作: 移动/拷贝
- 移动/拷贝次数: $3 \times 2 \times (n-1)$
 - 每次交换有一个数据 (以及序列) 回到正确的位置
 - 每次交换有3次数据移动/拷贝

最好、最坏和平均操作计数

- 程序的操作计数不但取决于实例特征，还取决于具体数据
- 最好/最坏/平均操作计数
 - 许多程序平均操作计数不好确定，可以给出最好/最坏操作计数
 - 名次原地重排
 - 交换（最好到最坏）：0到 $2 * (n-1)$
 - 移动（最好到最坏）：0到 $3 * 2 * (n-1)$

顺序搜索

■ 在长度为n的数组中搜索对应元素

```
template<typename T>
int sequentialSearch(T a[], const T& x, int n)
{
    int i;
    for(i = 0; i < n && a[i] != x; i++);

    if(i == n)
        return -1;
    else
        return i;
}
```

sequentialSearch操作计数:

- 关键操作: 比较
- 不成功搜索: **n次**比较
- 成功搜索
 - 最少比较次数: **1次**
 - 最多比较次数: **n次**
 - 平均比较次数: **假设每个元素不同且被查找的概率相同**

$$\frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

- 计算成功搜索与不成功搜索的期望
(50%不成功)

$$\frac{n+1}{2} \times 0.5 + n \times 0.5$$

选择排序

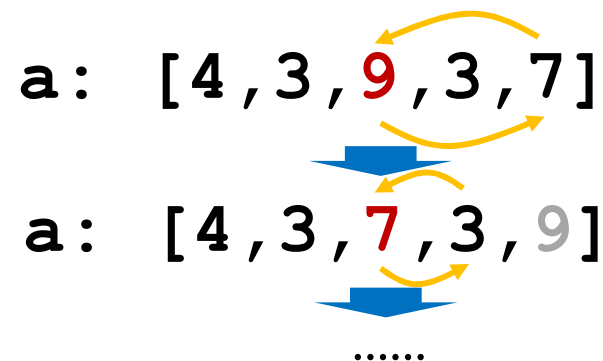
■ 为长度为n的数组排序

```
template<typename T>
void selectionSort(T a[], int n)
{
    for(int size = n; size > 1; size--) {
        int j = indexOfMax(a, size);
        swap(a[j], a[size-1]);
    }
}

int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};

    cout << "before: arr[5]: "
          << toString(arr, 5) << endl;
    selectionSort(arr, 5);
    cout << " after: arr[5]: "
          << toString(arr, 5) << endl;

    return 0;
}
```



```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make selectionSort
g++ selectionSort.cpp -o selectionSort
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./selectionSort
before: arr[5]: [4, 3, 9, 3, 7]
after: arr[5]: [3, 3, 4, 7, 9]
```

selectionSort操作计数:

- 关键操作: 比较、移动
- 比较次数: $1 + \dots + (n-1) = n * (n-1) / 2$
 - indexOfMax每次比较size-1次
- 移动次数: $3 * (n-1)$

选择排序（及时终止的）

- 为长度为n的数组排序
 - 当前子序列为有序时，直接退出

```
template<typename T>
void selectionSort(T a[], int n)
{
    bool sorted = false;
    for(int size = n; !sorted && (size > 1); size--){
        int indexOfMax = 0;
        sorted = true;
        for(int i = 0; i < size; i++){
            if(a[indexOfMax] <= a[i])
                indexOfMax = i;
            else
                sorted = false;
        }
        swap(a[indexOfMax], a[size-1]);
    }
}

int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};

    cout << "before: arr[5]: "
          << toString(arr, 5) << endl;
    selectionSort(arr, 5);
    cout << " after: arr[5]: "
          << toString(arr, 5) << endl;

    return 0;
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make selectionSort2
g++ selectionSort2.cpp -o selectionSort2
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./selectionSort2
before: arr[5]: [4, 3, 9, 3, 7]
after: arr[5]: [3, 3, 4, 7, 9]
```

selectionSort操作计数:

- 关键操作：比较、移动
- 比较次数
 - 最好情况： $n-1$
 - 最坏情况： $n*(n-1)/2$
- 移动次数
 - 最好情况：3
 - 最坏情况： $3*(n-1)$

冒泡排序

- 为长度为n的数组排序
 - 通过“冒泡”将最大数置于数组最后

```
template <typename T>
void bubble(T a[], int n)
{
    for(int i = 0; i < n-1; i++)
        if(a[i] > a[i+1])
            swap(a[i], a[i+1]);
}

template <typename T>
void bubbleSort(T a[], int n)
{
    for(int i = n; i > 1; i--)
        bubble(a, i);
}

int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};

    cout << "before: arr[5]: "
          << toString(arr, 5) << endl;
    bubbleSort(arr, 5);
    cout << " after: arr[5]: "
          << toString(arr, 5) << endl;

    return 0;
}
```

[4, 3, 9, 3, 7]

[3, 4, 7, 3, 9]

[3, 4, 7, 3, 9]

[3, 4, 3, 7, 9]

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make bubbleSort1
g++ bubbleSort1.cpp -o bubbleSort1
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./bubbleSort1
before: arr[5]: [4, 3, 9, 3, 7]
after: arr[5]: [3, 3, 4, 7, 9]
```

bubbleSort操作计数:

- 关键操作: 比较、移动
- 比较次数: $n * (n-1) / 2$
- 移动次数
 - 最好情况: 0
 - 最坏情况: $3 * n * (n-1) / 2$

冒泡排序（及时终止的）

- 为长度为n的数组排序
 - “冒泡”过程无交换时退出

```
template <typename T>
bool bubble(T a[], int n)
{
    bool swapped = false;
    for(int i = 0; i < n-1; i++)
        if(a[i] > a[i+1]){
            swap(a[i], a[i+1]);
            swapped = true;
        }
    return swapped;
}

template <typename T>
void bubbleSort(T a[], int n)
{
    for(int i = n;
        i > 1 && bubble(a, i);
        i--);
}
```

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make bubbleSort2
g++ bubbleSort2.cpp -o bubbleSort2
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./bubbleSort2
before: arr[5]: [4, 3, 9, 3, 7]
after: arr[5]: [3, 3, 4, 7, 9]
```

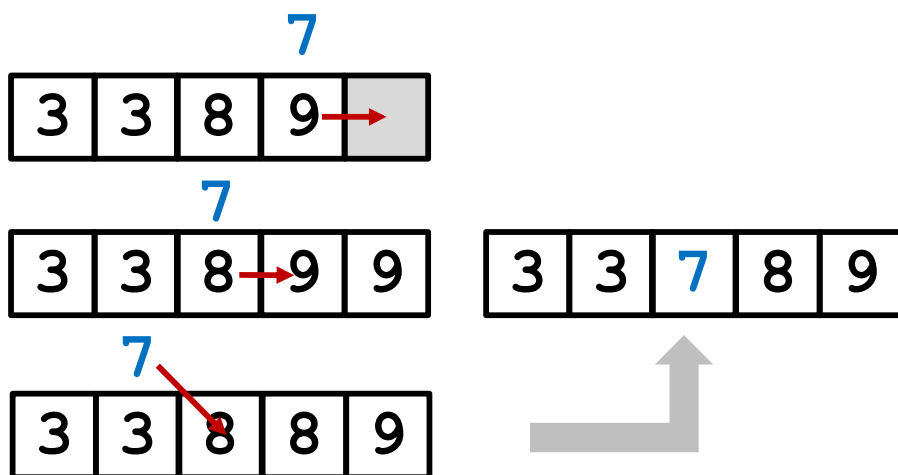
bubbleSort操作计数:

- 关键操作：比较、移动
- 比较次数
 - 最好情况： $n-1$
 - 最坏情况： $n*(n-1)/2$
- 移动次数
 - 最好情况：0
 - 最坏情况： $3*n*(n-1)/2$

插入元素

■ 为长度为n的有序数组中按序插入1个元素

```
template <typename T>
void insert(T a[], int n, const T& x)
{
    int i;
    for(i = n-1; i >= 0 && x < a[i]; i--)
        a[i+1] = a[i];
    a[i+1] = x;
}
```



insert操作计数:

- 关键操作: **比较**
- 比较次数
 - 最好情况: **1**
 - 最坏情况: **n**
 - 平均情况: 插入到*i*+1个位置的概率相等

$$\frac{1}{n+1} \left(\sum_{i=0}^{n-1} (n-i) + n \right)$$

插入数组最左边

插入排序

- 为长度为n的数组排序
 - 将数组元素依次插入

```
template<typename T>
void insertSort(T a[], int n)
{
    for(int i = 1; i < n; i++){
        T t = a[i];
        insert(a, i, t);
    }
}

int main(int argc, char *argv[])
{
    int arr[] = {4, 3, 9, 3, 7};

    cout << "before: arr[5]: "
          << toString(arr, 5) << endl;
    insertSort(arr, 5);
    cout << " after: arr[5]: "
          << toString(arr, 5) << endl;

    return 0;
}
```

i=1: [4, 3, 9, 3, 7]

i=2: [3, 4, 9, 3, 7]

i=3: [3, 4, 9, 3, 7]

i=4: [3, 3, 4, 9, 7]

[3, 3, 4, 7, 9]

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make insertionSort
g++ insertionSort.cpp -o insertionSort
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./insertionSort
before: arr[5]: [4, 3, 9, 3, 7]
after: arr[5]: [3, 3, 4, 7, 9]
```

insertSort操作计数:

- 关键操作: 比较
- 比较次数
 - 最好情况: $n-1$
 - 最坏情况: $n*(n-1)/2$

插入排序2

- 为长度为n的数组排序
 - 移动操作

```
template<typename T>
void insertSort(T a[], int n)
{
    for(int i = 1; i < n; i++){
        T t = a[i];
        int j;
        for(j = i-1; j >= 0 && t < a[j]; j--){
            a[j+1] = a[j];
            a[j+1] = t;
        }
    }
}
```

insertSort操作计数:

- 关键操作: 移动
- 移动次数
 - 最好情况: $2*(n-1)$
 - 最坏情况: $2*(n-1) + n*(n-1)/2$

排序算法时间复杂度比较

算法	最好	最坏
计数排序		
比较	$n(n-1)/2 + n$	$n(n-1)/2 + n$
移动	0	$6(n-1)$
选择排序		
比较	$n-1$	$n(n-1)/2$
移动	3	$3(n-1)$
冒泡排序		
比较	$n-1$	$n(n-1)/2$
移动	0	$3*n(n-1)/2$
插入排序		
比较	$n-1$	$n(n-1)/2$
移动	$2(n-1)$	$2(n-1) + n(n-1)/2$

步数

- **操作计数**忽略了所选择操作之外其他操作的开销
- **步数 (step counting)** 统计程序/函数中所有操作部分的时间开销
- **程序步 (program step)** 定义为一个语法意义上的程序片段，该片段的执行时间**独立于实例特征**
 - 100次加法, 100次减法, 1000次乘法**可被视为一步**
 - `return a+b+b*c+(a+b-c)/(a+b)+4`**可被视为一步**
 - `x=y`**可被视为一步**
 - n 次加法**不能被视为一程序步**

确定步数方法1-全局变量

- 创建一个全局变量stepCount，每当一个语句被执行，做累加操作

```
int stepCount = 0;

template<typename T>
T sum(T a[], int n)
{
    T res = 0;
    stepCount++;
    for(int i = 0; i < n; i++){
        stepCount++;
        res += a[i];
        stepCount++;
    }
    stepCount++; // 判断循环不满足循环条件
    stepCount++;
    return res;
}

int main(int argc, char *argv[])
{
    int arr1[] = {1, 2, 3, 4, 5};
    int sum1 = sum(arr1, 5);
    cout << "steps of adding 5 elements: "
         << stepCount << endl;

    stepCount = 0;
    int arr2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum2 = sum(arr2, 10);
    cout << "steps of adding 10 elements: "
         << stepCount << endl;

    return 0;
}
```

2n+3

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make stepCount
g++ stepCount.cpp -o stepCount
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./stepCount
steps of adding 5 elements: 13
steps of adding 10 elements: 23
```

确定步数方法1-递归函数

- 分析递归函数的步数时，可以得到一个基于步数的递归公式


```
int stepCount = 0;

template<typename T>
T rSum(T a[], int n)
{
    stepCount++;
    if(n > 0) {
        stepCount++;
        return rSum(a, n-1) + a[n-1];
    }
    stepCount++;
    return 0;
}

int main(int argc, char *argv[])
{
    int arr1[] = {1, 2, 3, 4, 5};
    int sum1 = rSum(arr1, 5);
    cout << "steps of adding 5 elements: "
          << stepCount << endl;

    stepCount = 0;
    int arr2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum2 = rSum(arr2, 10);
    cout << "steps of adding 10 elements: "
          << stepCount << endl;

    return 0;
}
```

$$\begin{aligned} t_{\text{rSum}}(n) &= 2 + t_{\text{rSum}}(n-1), \quad n > 0 \\ t_{\text{rSum}}(0) &= 2 \end{aligned}$$

$$t_{\text{rSum}}(n) = 2(n+1)$$

```
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ make stepCount2
g++ stepCount2.cpp -o stepCount2
xiaomb@LAPTOP-IUK2M5JJ:~/code/chapter02-04$ ./stepCount2
steps of adding 5 elements: 12
steps of adding 10 elements: 22
```

确定步数方法1-全局变量

- 不能断定程序 `sum` ($2n+3$ 步) 比程序 `rSum` 慢 ($2n+2$ 步)
 - 程序步不代表精确的时间单位。
 - `rSum` 中的一步可能花更多的时间
- **步数可用来了解程序的执行时间是如何随着实例特征的变化而变化的**
- `Sum/rSum` 的运行时间 (时间复杂度) 随着 n 的增加线性增长

确定步数方法2-步数表


语句	s/e	频率	总步数
	0	0	0
<pre>T sum(T a[], int n) { T res = 0; for(int i = 0; i < n; i++) res += a[i]; return res; }</pre>	0	0	0
	1	1	1
	1	n+1	n+1
	1	n	n
	1	1	1
	0	0	0
总计			2n+3

s/e: 语句每次执行所需要的步数, 即执行该语句所产生的 stepCount 值的变化量。

步数表—s/e是变化的

■ 对大小为n的数组的前置元素求和

a: [3, 3, 4, 7, 9]
b: [3, 6, 10, 17, 26]



```
void inef(T a[], T b[], int n)
{
    for(int j = 0; j < n; j++)
        b[j] = sum(a, j+1);
}
```

语句

s/e

频率

总步数

0

0

0

0

0

0

1

n+1

n+1

$2j+6$

n

$n(n+5)$

0

0

0

总计

n^2+6n+1

步数表—最好、最坏、平均步数

- 步数表也可以按最好、最坏、平均步数来统计
- 顺序搜索 (最好情况步数)

语句	s/e	频率	总步数
	0	0	0
	0	0	0
int sequentialSearch(T a[], T& x, int n)	1	1	1
{	1	1	1
int i;	1	1	1
for(int i = 0; i < n && a[i] != x; i++)	1	1	1
if(i == n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总计			4

步数表—最好、最坏、平均步数

■ 顺序搜索（最坏情况步数）

语句	s/e	频率	总步数
<pre>int sequentialSearch(T a[], T& x, int n) { int i; for(int i = 0; i < n && a[i] != x; i++) if(i == n) return -1; return i; }</pre>	0	0	0
	0	0	0
	1	1	1
	1	n+1	n+1
	1	1	1
	1	0	0
	0	0	0
总计			n+3

步数表—最好、最坏、平均步数

- 顺序搜索（**一般情况步数**），在第**j**个位置找到

语句	s/e	频率	总步数
	0	0	0
	0	0	0
	1	1	1
	1	j+1	j+1
	1	1	1
	1	1	1
	0	0	0
总计			j+4

```
int sequentialSearch(T a[], T& x, int n)
{
    int i;
    for(int i = 0; i < n && a[i] != x; i++)
        if(i == n) return -1;
    return i;
}
```

步数表—最好、最坏、平均步数

■ 顺序搜索（平均步数）

- 找到元素的情况下，每个位置被找到的概率一样

$$t_{\text{SequentialSearch}}^{\text{AVG}}(n) = \frac{1}{n} \sum_{j=0}^{n-1} (j + 4) = (n + 7)/2$$

- 没找到的概率为20%

$$0.8 * (n+7) / 2 + 0.2 * (n+3)$$

$$0.6n + 3.4$$

-
- 作业
 - P59 10 11 12 13