第 10 课
算法分析

薛浩

xuehao0618@outlook.com

阅读

- Programming Abstraction in C++ *Chapter 10*

今日话题

- 算法分析动机
- 大 O 表示法
- 大 O 分析法

计算机、数据中心和网络消耗了全球 10% 的电力

https://en.wikipedia.org/wiki/IT_energy_management

CAN RUST SAVE THE PLANET?

https://www.theregister.com/2021/11/30/aws_reinvent_

算法分析动机

https://zh.wikipedia.org/wiki/国际象棋盘与麦粒问题

运行时分析

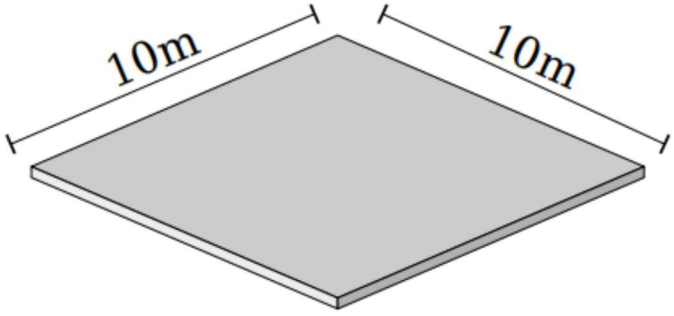
作业 1 完全数算法分析

运行时分析的局限性

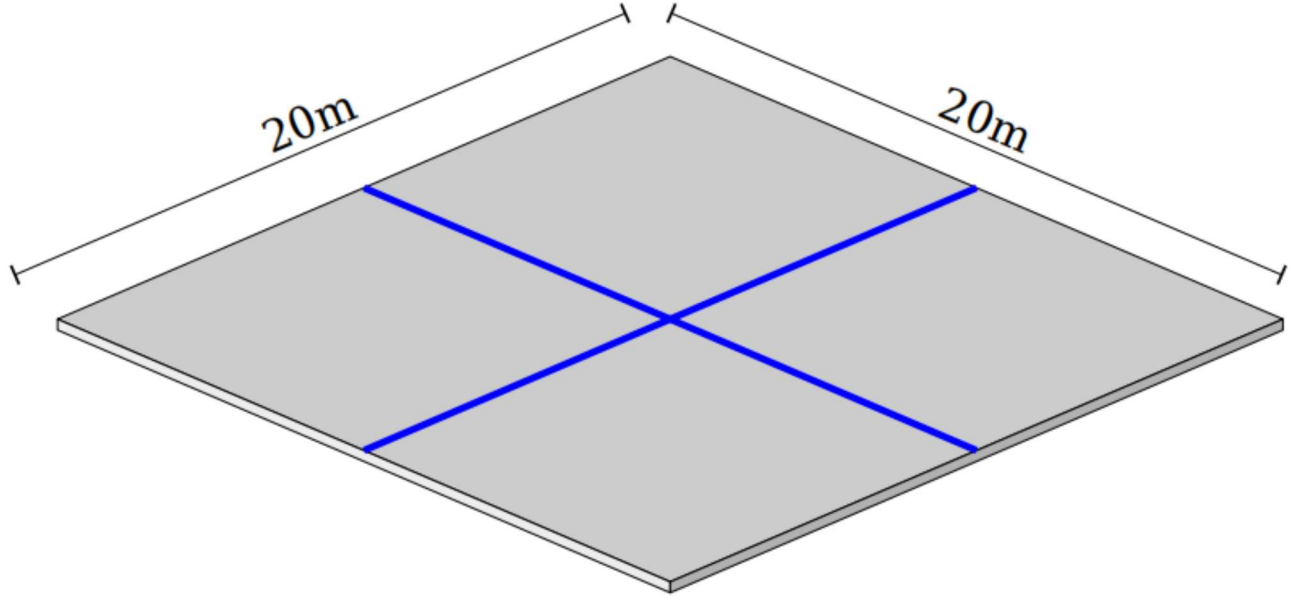
- 必须写好代码
- 依赖计算机性能
- 测试数据影响结果
- 测试范围有限

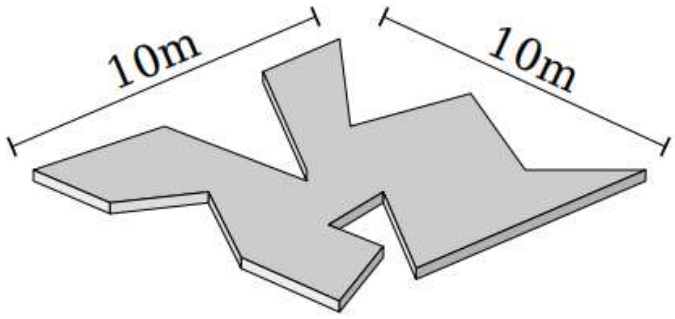
算法复杂度

不依赖具体的测试程序和数据，而是根据统计方法对算法执行效率进行分析

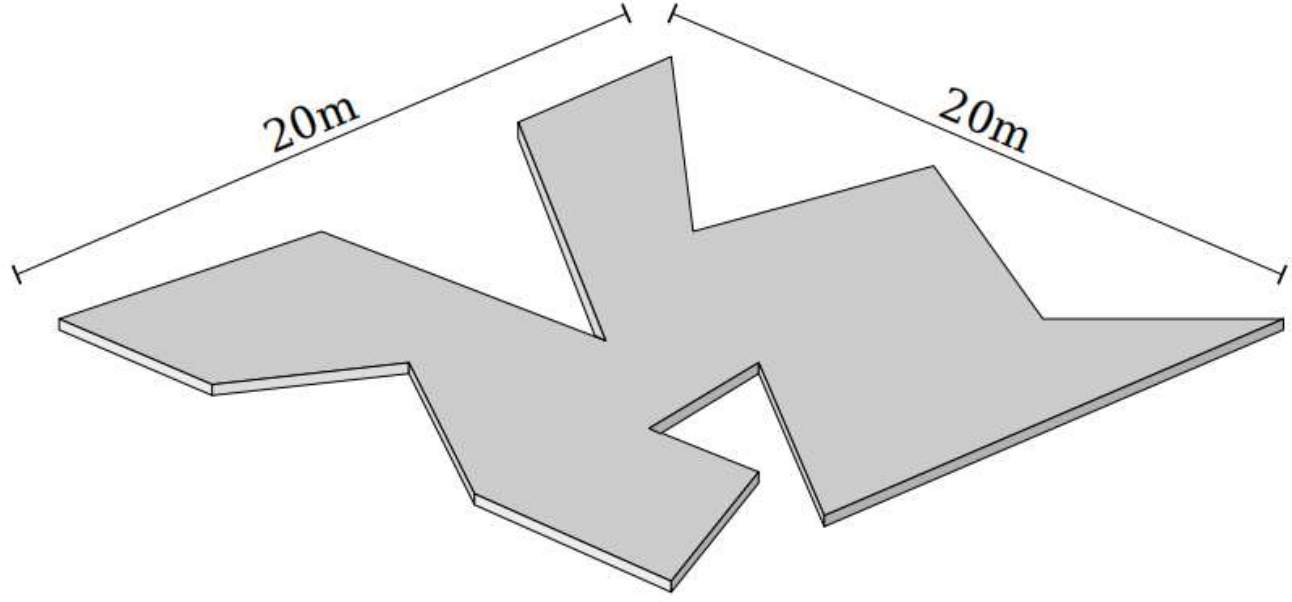


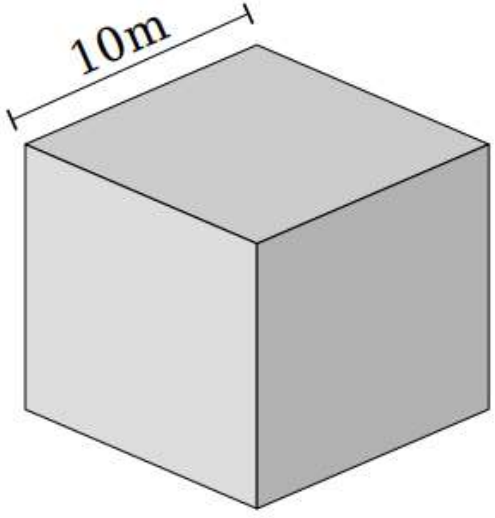
Mass: 100kg



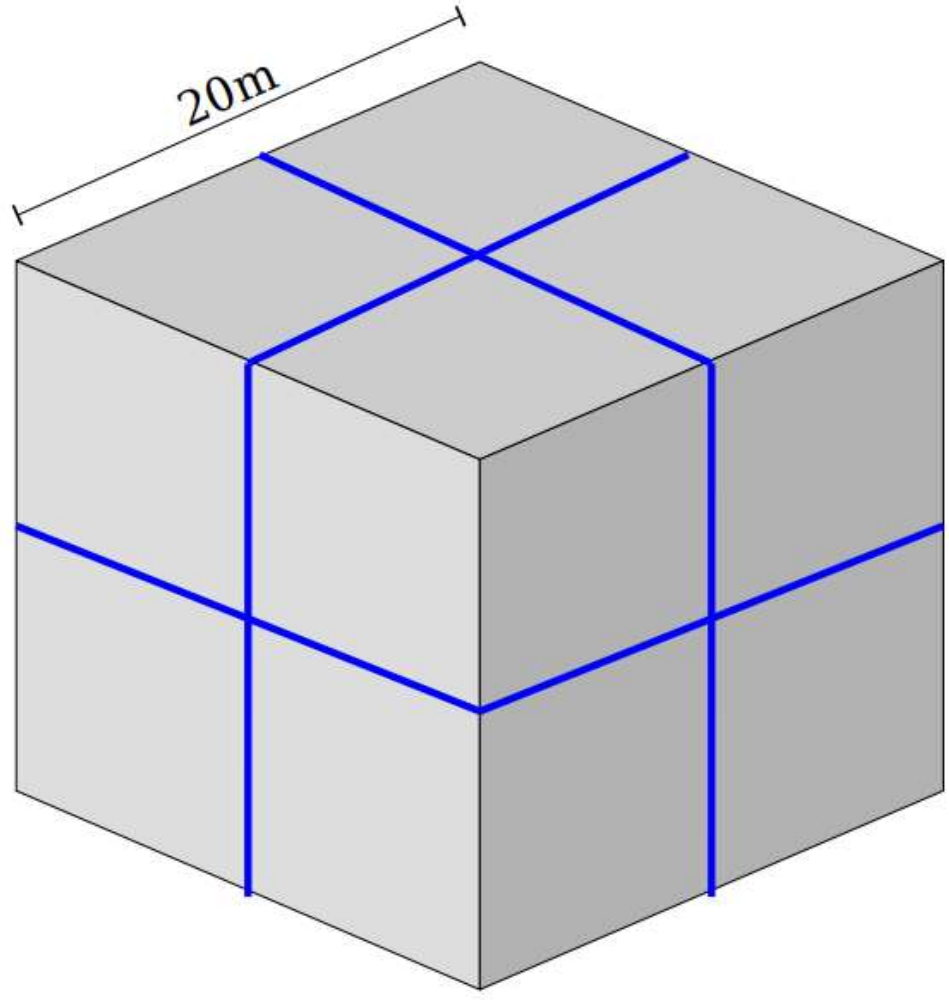


Mass: 60kg





Mass: 100kg



Mass: 1,000kg

✨ 算法复杂度分析

不需要精确的公式，只通过数量级变化时的速率，

就可以预测程序的执行效率

今日话题

- ~~算法分析动机~~
- 大 O 表示法
- 大 O 分析法

大 O 表示法

大 O 符号（Big O notation）

又称为渐进符号，

是用于描述函数渐近行为的数学符号

大 O 表示法使用代数项来描述代码的复杂性

大 O 表示法使用代数项来描述代码的复杂性

举例

```
double averageOf(const Vector<int> &vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```

举例

```
double averageOf(const Vector<int> &vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```

$$3n + 4$$

大 O 表示法只关注重要项，无需精确

$$4n + 4 = O(n)$$

$$137n + 271 = O(n)$$

$$n^2 + 3n + 4 = O(n^2)$$

$$2^n + n^3 = O(2^n)$$

今日话题

- ~~算法分析动机~~
- ~~大 O 表示法~~
- 大 O 分析法

大 O 分析法

案例 1

```
double average(const Vector<int> &vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```

案例 1

```
double average(const Vector<int> &vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```

$$O(n)$$

小试牛刀

```
void omnesia(int n) {
    int total = 0;
    for (int i = 0; i < n * n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                total++;
            }
        }
    }
    return total;
}
```

小试牛刀

```
void omnesia(int n) {
    int total = 0;
    for (int i = 0; i < n * n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                total++;
            }
        }
    }
    return total;
}
```

$$O(n^4)$$

✨ 原则 1：从内向外分析

案例 2：字母测试

```
bool containsAlpha(const string &s) {
    for (int i = 0; i < s.length(); i++) {
        if (isalpha(s[i])) {
            return true;
        }
    }
    return false;
}
```

案例 2：字母测试

```
bool containsAlpha(const string &s) {
    for (int i = 0; i < s.length(); i++) {
        if (isalpha(s[i])) {
            return true;
        }
    }
    return false;
}
```

$$O(n)$$

✨ 原则 2：关注最坏的结果

案例 3：选择排序

案例 3：选择排序

```
void sort(Vector<int> & vec) {
    int n = vec.size();
    for (int i = 0; i < n; i++) {
        int m = i;
        for (int j = i + 1; j < n; j++) {
            if (vec[j] < vec[m]) m = j;
        }
        int tmp = vec[i];
        vec[i] = vec[m];
        vec[m] = tmp;
    }
}
```

案例 3：选择排序

```
void sort(Vector<int> & vec) {
    int n = vec.size();
    for (int i = 0; i < n; i++) {
        int m = i;
        for (int j = i + 1; j < n; j++) {
            if (vec[j] < vec[m]) m = j;
        }
        int tmp = vec[i];
        vec[i] = vec[m];
        vec[m] = tmp;
    }
}
```

$$O(n^2)$$

小试牛刀

```
int countTriples(const Vector<int>& values, int target) {
    int result = 0;
    for (int i = 0; i < values.size(); i++) {
        for (int j = i + 1; j < values.size(); j++) {
            for (int k = j + i; k < values.size(); k++) {
                if (values[i] + values[j] + values[k] == targe
                    result++;
            }
        }
    }
    return result;
}
```

小试牛刀

```
int countTriples(const Vector<int>& values, int target) {
    int result = 0;
    for (int i = 0; i < values.size(); i++) {
        for (int j = i + 1; j < values.size(); j++) {
            for (int k = j + i; k < values.size(); k++) {
                if (values[i] + values[j] + values[k] == targe
                    result++;
            }
        }
    }
    return result;
}
```

$$O(n^3)$$

✨ 原则 3：不能乘的时候就加

案例 4：终端打印

```
void printStar(int n) {
    for (int row = 0; row < n; row++) {
        if (row == 0 || row == n - 1) {
            for (int col = 0; col < n; col++) {
                cout << '*';
            }
            cout << endl;
        } else {
            cout << '*' << endl;
        }
    }
}
```

案例 4：终端打印

```
void printStar(int n) {
    for (int row = 0; row < n; row++) {
        if (row == 0 || row == n - 1) {
            for (int col = 0; col < n; col++) {
                cout << '*';
            }
            cout << endl;
        } else {
            cout << '*' << endl;
        }
    }
}
```

$$O(n)$$

小试牛刀

```
void avareed(int n) {
    for (int i = 0; i < n; i++) {
        if (i >= n / 4 && i < 3 * n / 4) {
            for (int j = 0; j < n; j++) {
                cout << '*';
            }
        } else {
            cout << '?';
        }
    }
}
```

小试牛刀

```
void avareed(int n) {
    for (int i = 0; i < n; i++) {
        if (i >= n / 4 && i < 3 * n / 4) {
            for (int j = 0; j < n; j++) {
                cout << '*';
            }
        } else {
            cout << '?';
        }
    }
}
```

$$O(n)$$

✨ 原则 4：不是所有的循环都是 O(n)

案例 5：查找最大值

```
int maxOf(Vector<int> elems) {
    if (elems.size() == 1)
        return elems[0];

    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    int maxOfRest = maxOf(rest);

    return first > maxOfRest ? first : maxOfRest;
}
```

✨ 原则 5：别忘记函数/接口本身的复杂度

案例 6：二分查找最大值

```
int maxOfDevide(Vector<int> elems) {
    if (elems.size() == 1)
        return elems[0];

    int mid = elems.size() / 2;
    Vector<int> firstPart = elems.subList(0, mid);
    Vector<int> secondPart = elems.subList(mid);

    int maxOfFirst = maxOfDevide(firstPart);
    int maxOfSecond = maxOfDevide(secondPart);

    return maxOfFirst > maxOfSecond ? maxOfFirst : maxOfSecond
}
```

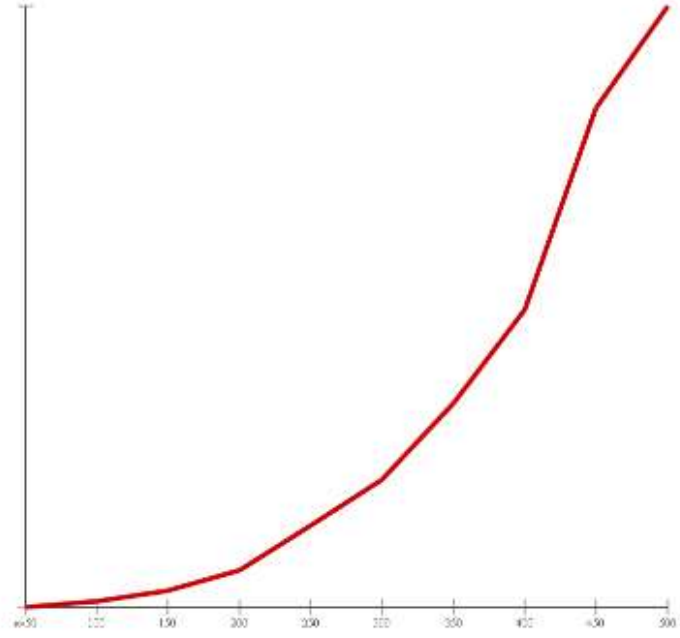
小试牛刀

```
int maxOfRef(const Vector<int> &elems, int start, int end) {
    if (end - start == 1)
        return elems[start];

    int mid = start + (end - start) / 2;
    int maxOfFirst = maxOfRef(elems, start, mid);
    int maxOfSecond = maxOfRef(elems, mid, end);

    return maxOfFirst > maxOfSecond ? maxOfFirst : maxOfSecond
}
```

✨ 原则 6：从函数调用栈角度分析递归复杂度

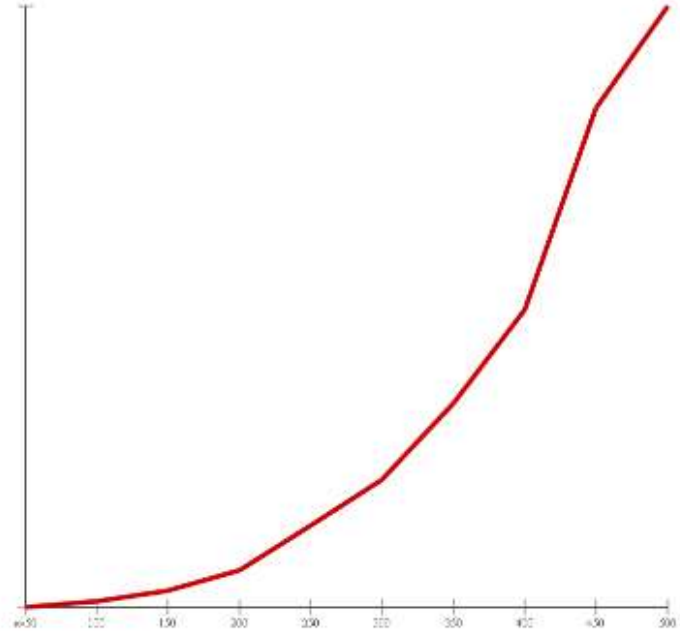
案例 7：未知函数

数据量	执行时间
50	19.0
100	154.1
150	392.8
200	883.5
250	1946.4
300	3017.4
350	4833.9
400	7058.8
450	11809.9
500	14237.2



案例 7：未知函数

数据量	执行时间
50	19.0
100	154.1
150	392.8
200	883.5
250	1946.4
300	3017.4
350	4833.9
400	7058.8
450	11809.9
500	14237.2



$$O(n^2)$$

数据量	翻倍数据量	增长率
50	100	8.1
100	200	5.7
150	300	7.7
200	400	8.0
250	500	7.3

数据量	翻倍数据量	增长率
50	100	8.1
100	200	5.7
150	300	7.7
200	400	8.0
250	500	7.3

$$O(n^3)$$

✨ 原则 7：秒表依然是好帮手

大 O 分析法

- 原则 1：从内向外分析
- 原则 2：关注最坏的结果
- 原则 3：不能乘的时候就加
- 原则 4：不是所有的循环都是 O(n)
- 原则 5：别忘记函数/接口本身的复杂度
- 原则 6：从函数调用栈角度分析递归复杂度
- 原则 7：秒表依然是好帮手

今日话题

- ~~算法分析动机~~
- ~~大 O 表示法~~
- ~~大 O 分析法~~

下一次课

- OOP

THE END