# Информационные технологии и программирование

## Лекция 6. Основы языка C#

**Содержание лекции:**

- Ограничения массивов
- Знакомство с List<T>
- Оценка сложности алгоритмов
- Правила записи Big O
- Проблема поиска
- Словарь (Ключ — Значение)
- Множество (HashSet)

Преподаватель курса: Клюкин Даниил Анатольевич, ст. преподаватель каф. ПМиИТ

# Ограничения массивов

Массив (array) имеет строго фиксированный размер, который задается при создании.

**Проблема:** Что делать, если мы заранее не знаем количество данных? (Например, список покупок, список врагов в игре, сообщения в чате).

**Ручное решение:**

Создать новый массив большего размера.

Скопировать туда все элементы старого массива.

Добавить новый элемент.

*Это неудобно и легко ошибиться.*

```
// 1. У нас есть массив на 2 элемента
string[] playlist = { "Song 1", "Song 2" };
// 2. Хотим добавить третью песню. Но места нет!
// Создаем новый массив побольше (черновик)
string[] biggerPlaylist = new string[3];

// 3. Копируем данные из старого массива в новый
for (int i = 0; i < playlist.Length; i++)
{
    biggerPlaylist[i] = playlist[i];
}

// 4. Добавляем новый элемент в свободную ячейку
biggerPlaylist[2] = "Song 3";
// 5. Заменяем старый массив новым
playlist = biggerPlaylist;
Console.WriteLine($"Теперь песен: {playlist.Length}");
```

# Знакомство с List<T>

**Коллекция** – совокупность объектов (значений переменных). Отличается от массивов дополнительными функциями и гибкостью.

**Виды коллекций**: List, LinkedList, Stack, Queue, HashSet, Dictionary

**List** – список однотипных объектов. Основное отличие от массивов – автоматическое увеличение емкости (в 2 раза) при переполнении.

```
// Создание пустого списка целых чисел
List<int> numbers = new List<int>();

// Создание списка строк с начальными значениями
var names = new List<string> { "Tom", "Alice", "Bob" };
```

<T> — это «тип-заполнитель». Вместо него мы ставим любой нужный нам тип (int, string, double и т.д.).

**Основные методы:**

Add(x) — добавить в конец.

```
// Создание пустого списка для целых чисел
List<int> numbers = new List<int>();

// Добавление элементов
numbers.Add(10);
numbers.Add(20);

// Создание списка строк с начальными значениями
var names = new List<string> { "Tom", "Alice", "Bob" };

// Доступ к элементам (точно так же, как в массиве!)
Console.WriteLine(names[0]); // Выведет "Tom"
```

**Удаление:**

list.Remove(значение); — удаляет первое найденное совпадение.

list.RemoveAt(индекс); — удаляет элемент по его позиции.

```
List<int> scores = new List<int>();
scores.Add(90);       // [90]
scores.Add(100);      // [90, 100]
scores.RemoveAt(0);   // [100]
Console.WriteLine(scores[0]); // Выведет 100
```

**Count**

Это **свойство**, а не метод (пишется без скобок). Оно возвращает количество элементов, которые вы **реально добавили**. Запомните: у массива — .Length, у списка — .Count.

```
var achievements = new List<string>();

// 1. .Add(x) - Добавление в конец списка
achievements.Add("Прийти на пару");
achievements.Add("Не уснуть");
achievements.Add("Написать лекцию");

// 2. .Count - Текущее количество элементов
Console.WriteLine($"У игрока {achievements.Count}
достижений."); // Вывод: 3
```

**Clear()**

Этот метод не уничтожает сам список как объект, он просто делает его пустым. Внутренний массив зануляется, и Count становится равным 0. Это удобнее, чем создавать новый список с нуля.

```
var achievements = new List<string>();

// 1. .Add(x) – Добавление в конец списка
achievements.Add("Прийти на пару");
achievements.Add("Не уснуть");
achievements.Add("Написать лекцию");

// 2. .Clear() – Полная очистка списка
achievements.Clear();
Console.WriteLine($"После очистки: {achievements.Count}");
// Вывод: 0
```

**Capacity**

Внутри List всегда лежит **обычный массив**.

**Count** — сколько элементов *реально* добавлено в список.

**Capacity (Емкость)** — размер внутреннего массива (сколько места забронировано в памяти).

**Алгоритм роста:**

Когда Count становится равен Capacity, список создает новый внутренний массив **в 2 раза больше** предыдущего.

Старые данные копируются в новый массив.

Старый массив удаляется.

```
List<int> test = new List<int>();
for (int i = 0; i < 10; i++)
{
    test.Add(i);
    Console.WriteLine($"Count: {test.Count},
                        Capacity: {test.Capacity}");
}
```

```
Count: 1, Capacity: 4
Count: 2, Capacity: 4
Count: 3, Capacity: 4
Count: 4, Capacity: 4
Count: 5, Capacity: 8
Count: 6, Capacity: 8
Count: 7, Capacity: 8
Count: 8, Capacity: 8
Count: 9, Capacity: 16
Count: 10, Capacity: 16
```

# Оценка сложности алгоритмов

Сам термин "алгоритм" произошел от имени персидского математика Аль-Хорезми, труды которого сыграли важную роль на становление математики как науки.

Алгоритм может иметь входные данные, над которыми производятся вычисления, а также может иметь выходной результат - одно значение или набор значений. Задача алгоритма состоит в преобразовании входных значений в выходные.

Сложность алгоритмов оценивается по 2 критериям:

- требуемому **времени** на выполнение,
- требуемой **памяти** на выполнение.

## Зачем нужна оценка сложности?

Один и тот же результат можно получить разными способами (разными алгоритмами).

**Проблема:** Как понять, какой алгоритм лучше, не запуская его на суперкомпьютере?

Скорость работы зависит от процессора, памяти и языка программирования.

**Решение:** Нам нужна универсальная мера, которая зависит только от количества входных данных (**n**).

**Вычислительная временная сложность (time complexity)** — это максимальное возможное количество выполненных алгоритмом элементарных операций, как функция от размера входных данных.

**Вычислительная ёмкостная сложность (space complexity)** — это максимальный возможный размер занятой алгоритмом дополнительной памяти, как функция от размера входных данных.

*Мы чаще фокусируемся на времени, так как память сейчас дешевая, а время пользователя — нет.*

```
for (int i = 0; i < n; i++)
    count++;
```

**Посчитаем количество элементарных операций:**

1 для int i = 0

n+1 для i < n

2n для i++ (что эквивалентно i = i + 1, а это две операции: присваивание и сложение)

2n для count++

Получаем, что временная сложность алгоритма C(n)=2+5n.

И ещё один пример посложнее, перед тем как перейти к долгожданной О-нотации.

```
int i = 0;
while (i < n)
{
    for(int j = 0; j < n; j++)
        count++;
    i++;
}
```

Посчитаем функцию сложности:

- i = 0 выполнится лишь однажды
- i<n внутри while выполнится n+1 раз
- i+=1 выполнится n раз (+= это две элементарные операции)

цикл for начнет выполняться n раз и каждый раз:

- j=0 выполнится один раз
- j<n выполнится n+1 раз
- j++ выполнится n раз (++ это две элементарные операции)
- count++ выполнится n раз (++ это две элементарные операции)

Итого, сложность $C(n)=1+(n+1)+2n+n(1+(n+1)+2n+2n)=2+5n+5n^2$

Точный расчет времени выполнения невозможен, потому что:

**Операции не равны:** Сложение выполняется мгновенно, а вычисление sin(x) или sqrt(x) может занять в десятки раз больше времени.

**Железо разное:** На одном процессоре операция идет быстро, на другом — медленнее из-за разницы в архитектуре и кэшировании.

**Скрытые расходы C#:** Выделение памяти, работа «сборщика мусора».

**Внешние факторы:** Процессор может быть занят фоновым обновлением Windows или антивирусом.

**Вывод:** Вместо точного времени инженеры оценивают **асимптотическую сложность** — то, как быстро растет время работы программы при росте объема данных.

Для оценки сложности инженеры обычно пользуются так называемой O-нотацией. Например, говорят «сложность этого алгоритма $O(n^2)$ (произносится: о от эн квадрат), а этого $\Theta(n \log(n))$ (произносится: тэта от эн лог эн).

Если очень кратко — $O(n^2)$ означает, что самое быстро растущее слагаемое в функции сложности — это n в степени не более 2, а $\Theta(n^2)$ — что степень в точности 2.

**Определение 1.** Говорят «$f(n) = O(g(n))$» тогда и только тогда, когда $\exists C>0$, $\exists n_0 : \forall n > n_0 \rightarrow f(n) < Cg(n)$

**Определение 2.** Говорят «$f(n) = \Theta(g(n))$» тогда и только тогда, когда $f(n) = O(g(n))$ и одновременно $g(n) = O(f(n))$

Пример.

```
int i = 1;
while (i < n)
{
    for(int j=0; j<n-2; j++)
        count++;
    i++;
}
```

Попробуем сходу найти самую «горячую» операцию — ту, которая выполняется больше всего раз. Чаще всего такая операция находится внутри самого вложенного цикла. В нашем случае это count++. Ну или j++ внутри for — не важно, можно взять любую из них.

Заметим, что count++ выполняется чуть меньше $n^2$ раз. Не точно $n^2$, потому что i начинается с 1, а не с нуля, да и внутренний цикл до n-2. Однако опытный инженер даже не будет пытаться учитывать эти константы — он и так знает, что в итоге будет квадратичная оценка.

Оценка сложности алгоритма $O(n^2)$

# Правила записи Big O

**Отбрасываем константы:**

O(2n), O(50n), O(n/2) – это все O(n)

**Оставляем только старшую степень:**

$O(n^2 + 100n + 500)$ – это $O(n^2)$

**Худший случай:** Мы всегда считаем сложность для самого плохого варианта (например, ищем число, которое стоит в самом конце массива).

## Основные уровни сложности:

| Сложность | Название | Пример |
|---|---|---|
| $O(1)$ | Константная | Доступ к элементу массива по индексу |
| $O(\log n)$ | Логарифмическая | Бинарный поиск (в следующей лекции!) |
| $O(n)$ | Линейная | Один цикл (поиск в обычном массиве) |
| $O(n \cdot \log n)$ | Линейно-логарифмическая | Быстрая сортировка (Array.Sort) |
| $O(n^2)$ | Квадратичная | Два вложенных цикла (Сортировка пузырьком) |

Какая оценка сложности данного алгоритма?

```
var n = int.Parse(Console.ReadLine());
var sum = 0.0;

for (var i = 0; i < n; i++)
{
   sum += n;
}

Console.WriteLine(sum);
```

Какая оценка сложности данного алгоритма?

```
var n = int.Parse(Console.ReadLine());
var sum = 0.0;

for (var i = 0; i < n; i++)
{
   for (var j = 0; j < n; j++)
    {
      sum += n;
   }
}
```

Какая оценка сложности данного алгоритма (добавление нового элемента)?

```
var list = new List<int> { 1, 2, 3, 4 };

// В памяти это выглядит так: [ 1 | 2 | 3 | 4 ]

list.Add(5);
```

**Визуализация:**

**Шаг 1 (Места нет):**

[ 1 ][ 2 ][ 3 ][ 4 ] — *Full!*

**Шаг 2 (Создание нового):**

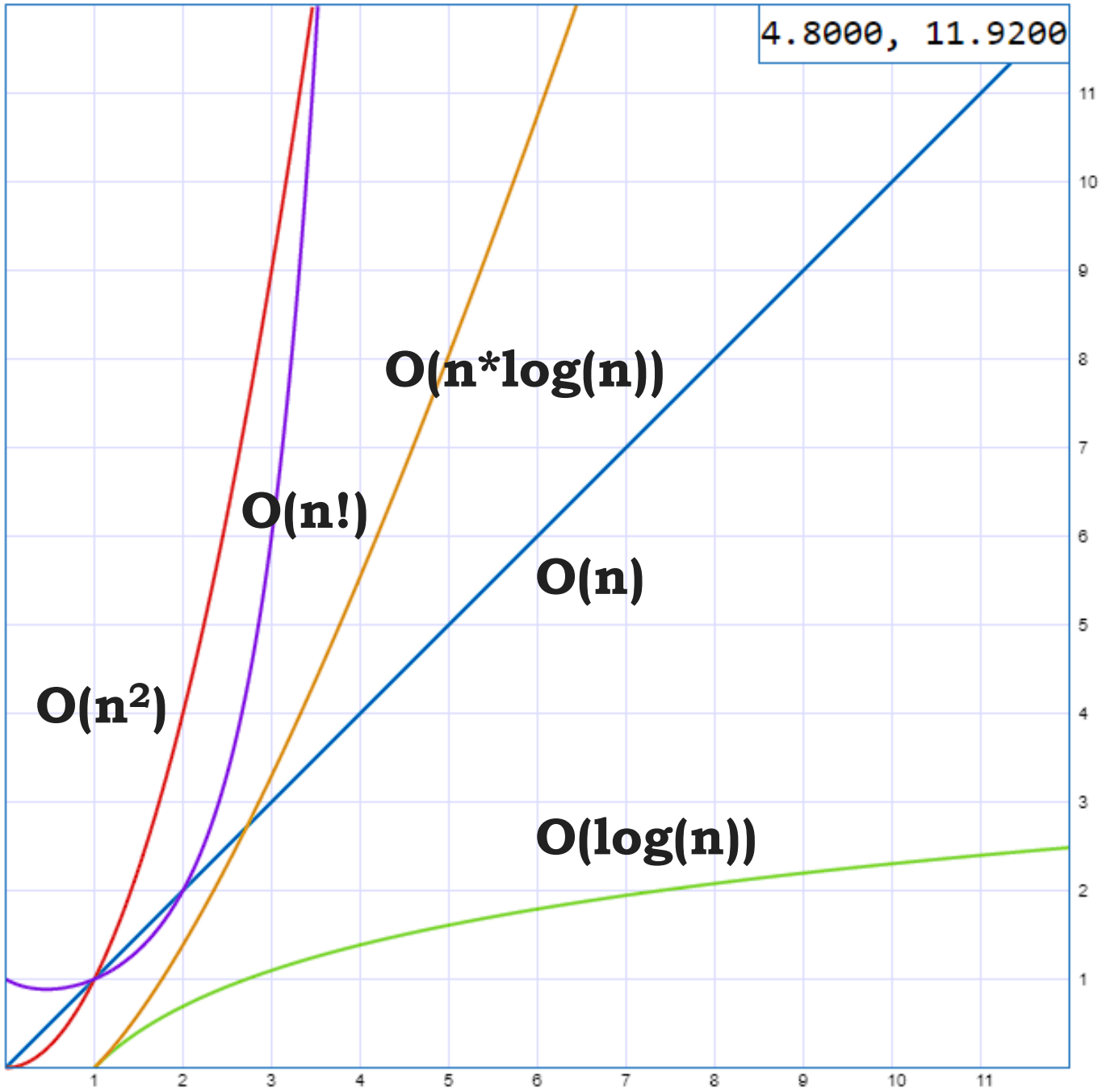
[ ][ ][ ][ ][ ][ ][ ][ ]

**Шаг 3 (Копирование - вот тут прячется O(n))**

[ 1 ][ 2 ][ 3 ][ 4 ][ ][ ][ ][ ]

**Шаг 4 (Наконец-то добавили):**

[ 1 ][ 2 ][ 3 ][ 4 ][ 5 ][ ][ ][ ] — *Готово!*




**Скорость роста** наиболее популярных функций

Важно внимательно подходить к выбору алгоритма, особенно если критично время выполнения расчета.

# Проблема поиска

Чтобы найти элемент в List<T>, компьютер перебирает его с начала до конца.

Если в списке 1 000 000 элементов, поиск может занять 1 000 000 операций.

```
var numbers = new List<int> { 10, 25, 3, 44, 12, 5 };
int target = 44; // Что мы ищем
int foundIndex = -1;

// Мы идем по списку шаг за шагом
for (int i = 0; i < numbers.Count; i++)
{
    // Каждая итерация цикла — это 1 операция сравнения
    if (numbers[i] == target)
    {
        foundIndex = i;
        break; // Нашли! Выходим из цикла
    }
}
Console.WriteLine($"Элемент найден на позиции: {foundIndex}");
```

Сложность поиска в списке: O(n).

**Вопрос:** Можно ли найти элемент за O(1), независимо от размера коллекции?

**Ответ:** Да, если использовать Хэш-таблицы.

Представьте гардероб. Чтобы найти куртку в куче (как в List), нужно перебрать все куртки.

Но в гардеробе есть **номерки**. Вы даете номерок (Ключ) и сразу получаете свою куртку (Значение).

**Хэш-функция** — это «магия», которая превращает любой объект (строку, число) в уникальный номер ячейки памяти.

Благодаря этому мы не ищем элемент, а **сразу прыгаем в нужную ячейку**.

# Словарь (Ключ — Значение)

**Хранит данные парами:** уникальный Ключ и привязанное к нему Значение.

```
// Ключ - ФИО (string), Значение - возраст (int)
var ages = new Dictionary<string, int>();

ages.Add("Иван", 20);
ages["Петр"] = 21; // Удобная запись через скобки
```

Особенности:

Поиск по ключу работает мгновенно: O(1).

Ключи не могут повторяться.

В отличие от списка, в словаре мы работаем не с индексами (0, 1, 2...), а с уникальными ключами.

Если попытаться достать значение по ключу, которого нет в словаре, программа «упадет» с ошибкой. Чтобы этого избежать, используем метод ContainsKey.

```
if (ages.ContainsKey("Иван"))
{
    Console.WriteLine($"Возраст Ивана: {ages["Иван"]}");
}
else
{
    Console.WriteLine("Такого ключа нет!");
}
```

Удаление в словаре происходит мгновенно по ключу.

```
ages.Remove("Иван"); // Удаляет и ключ, и значение
```

При переборе словаря мы получаем объект «Пара» (KeyValuePair), у которого есть свойства Key и Value.

```
foreach (var pair in ages)
{
    // pair.Key – это имя, pair.Value – это возраст
    Console.WriteLine($"{pair.Key} – {pair.Value} лет");
}
```

В современных версиях C# не обязательно писать pair.Key и pair.Value.

Можно «распаковать» пару прямо в заголовке цикла foreach.

```
// Сейчас (распаковка в переменные):
foreach (var (name, age) in ages)
{
    // Мы сразу получили доступ к имени и возрасту!
    Console.WriteLine($"{name} - {age}");
}
```

# Множество (HashSet)

Это коллекция, которая содержит **только уникальные элементы**.

Аналог List, но в нем не может быть двух одинаковых чисел или строк.

```
var uniqueNumbers = new HashSet<int>();
uniqueNumbers.Add(5);
uniqueNumbers.Add(5); // Ничего не произойдет, 5 уже есть
Console.WriteLine(uniqueNumbers.Count); // Выведет 1
```

**Зачем это нужно?**

Мгновенная проверка "есть ли такой элемент в списке" (O(1)).

Быстрое удаление дубликатов.

Самый простой способ очистить список от повторов:

```
var dirtyList = new List<int> { 1, 2, 2, 3, 3, 3 };

var cleanSet = new HashSet<int>(dirtyList);

// Внутри останутся только 1, 2, 3
```

Проверка **Contains** работает мгновенно, независимо от того, сколько в нем элементов (миллион или десять).

```
var bannedUsers = new HashSet<string>
{ "admin", "spammer777", "bot_123" };

Console.Write("Введите ваш логин: ");
string userName = Console.ReadLine();

// Мгновенная проверка
if (bannedUsers.Contains(userName))
{
    Console.WriteLine("Ошибка: Доступ заблокирован!");
}
else
{
    Console.WriteLine("Добро пожаловать в систему!");
}
```