



# Сложность алгоритмов

Елена Щелкунова

Системный программист



# Обо мне

- Работаю программистом с 2010 года
- Работала в 7 компаниях за это время
- Full-stack разработчик (C# / JavaScript – JQuery, ExtJs, React, Angular)
- Много работала с legacy-кодом в крупных проектах
- Много делала оптимизаций производительности с хорошими результатами



# О компании

- Работали всегда, но по документам с 1988 года
- Занимаемся полным циклом разработки предметно-ориентированных платформ и прикладных систем
- Лидер по количеству проектов внедрения ECM-систем
- Просто хорошие ребята 😊



Интеллектуальная система управления цифровыми процессами и документами



Система управления кадровыми процессами, документами и сервисами



Интеллектуальная система для обработки любой текстовой информации



Система для управления проектами и командами

# Зачем это нужно

1

Большое количество повторяющихся операций

2

Клиенты недовольны

3

Клиенты теряют деньги

4

1с немного  $1с * 1000000$  – очень много

# План на сегодня

1. Определения
2. Сводная таблица временной сложности
3. Примеры графиков
4. Примеры алгоритмов
5. Какая сложность у алгоритма?
6. Примеры кода
7. Варианты оптимизаций
8. Какие бывают коллекции и при чем тут сложность

# Оговорки

1. Оцениваем только алгоритмическую сложность  
(без учета ОС, железа и т.д.)
2. Когда нужно оптимизировать? - Когда есть проблема

# Определения

## Сложность алгоритма

это количественная характеристика сколько времени или какой объём памяти потребуется для выполнения алгоритма

## Сложность зависит от размеров входных данных:

массив из 100 элементов обрабатывается быстрее, чем из 1000

## При этом точное время мало кого интересует:

важна асимптотическая сложность (в теории)

## Сложность алгоритмов измеряют в элементарных шагах:

Big O (или O-нотации) от немецкого «Ordnung» - порядок



# Сводная таблица временной сложности

	Название	Формула	Примеры алгоритмов
простые	Константная	$O(1)$	Длина массива
	Логарифмическая	$O(\log(N))$	Бинарный поиск
	Линейная	$O(N)$	Поиск методом перебора
	Линейно-логарифмическая	$O(N \cdot \log(N))$	Быстрая сортировка
	Полиномиальная	$O(N^2)$	Сортировка выбором, пузырьковая сортировка
сложные	Экспоненциальная	$O(2^N)$	Список подмассивов
	Факториальная	$O(N!)$	Список перестановок

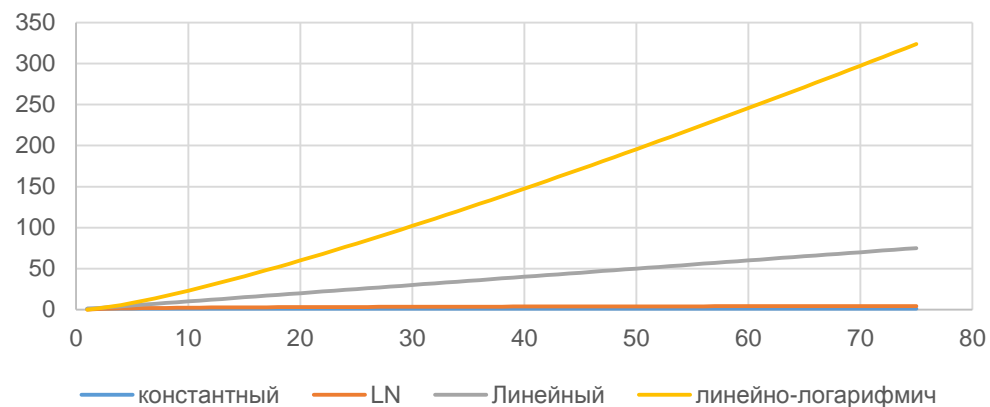
# Время выполнения в секундах

При скорости  $10^6$  операций в секунду:

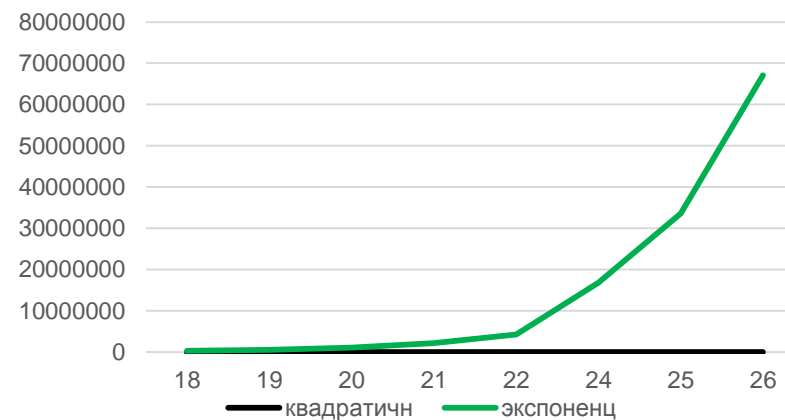
Размер сложности	10	20	30	40	50	60
N	0,00001 с	0,00002 с	0,00003 с	0,00004 с	0,00005 с	0,00006 с
N <sup>2</sup>	0,0001 с	0,0004 с	0,0009 с	0,0016 с	0,0025 с	0,0036 с
N <sup>3</sup>	0,001 с	0,008с	0,027 с	0,064 с	0,125 с	0,216 с
N <sup>5</sup>	0,1 с	3,2с	24,3 с	1,7 мин	5,2 мин	13 мин
2 <sup>N</sup>	0,0001 с	1с	17,9 мин	12,7 дней	35,7 веков	366 веков
3 <sup>N</sup>	0,059 с	58 мин	6,5 лет	3855 веков	2*10 <sup>8</sup> веков	1,3*10 <sup>13</sup> веков

# Графики функций

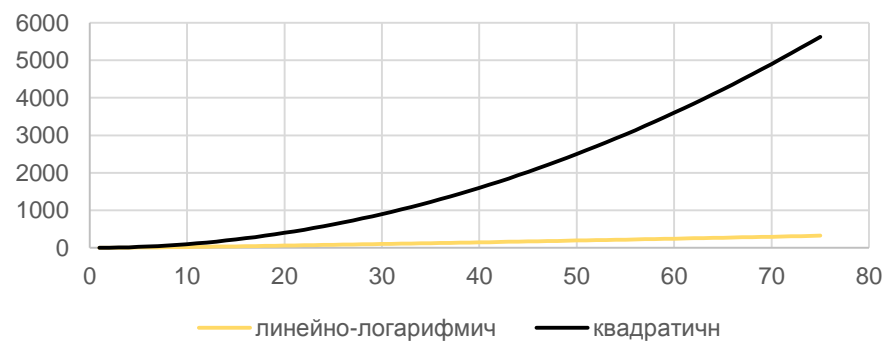
Константная, логарифмическая, линейная, линейно-логарифмическая



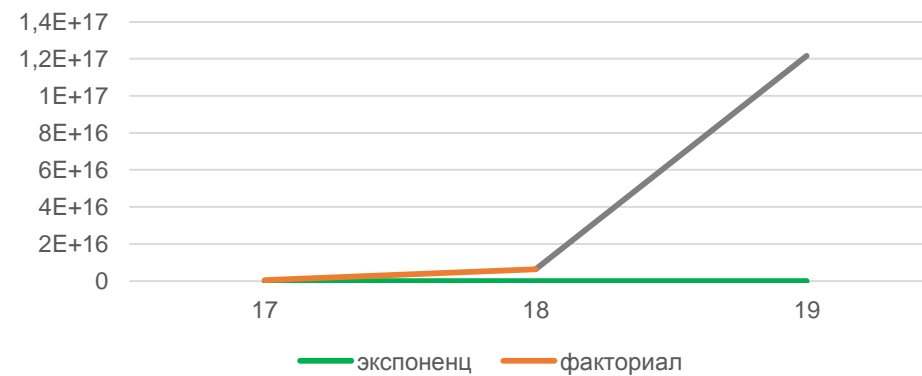
Квадратичная, экспоненциальная



Линейно-логарифмическая, квадратичная



Экспоненциальная, факториальная



# Пример №1

- $N = 10$
- `For(int l = 6; l < N; i++)`
- `For(int j = 9; j < N; j++)`
- `{`
- `DoSomething();`
- `}` // сколько будет итераций цикла?

$$6 \dots 9 * 9 \dots 9 = 4 * 1 = 4$$

- $N = 10$
- `For(int l = 0; l < N; i++)`
- `{`
- `DoSomething();`
- `}` // сколько будет итераций цикла?

$$0 \dots 9 = 10$$

# Пример №2

- $N = 50$
- `For(int l = 6; l < N; i++)`
- `For(int j = 9; j < N; j++)`
- `{`
- `DoSomething();`
- `} // сколько будет итераций цикла?`

$$6...49 * 9...49 = 44 * 41 = 1804$$

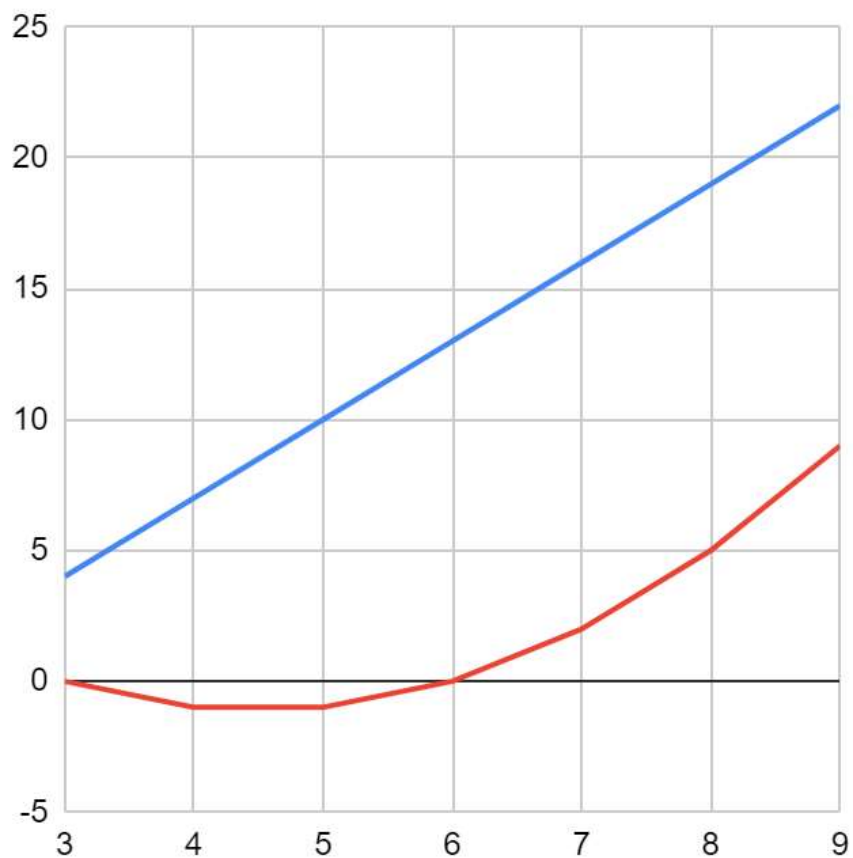
- $N = 50$
- `For(int l = 0; l < N; i++)`
- `{`
- `DoSomething();`
- `} // сколько будет итераций цикла?`

$$0...49 = 50$$

# Примеры графиков

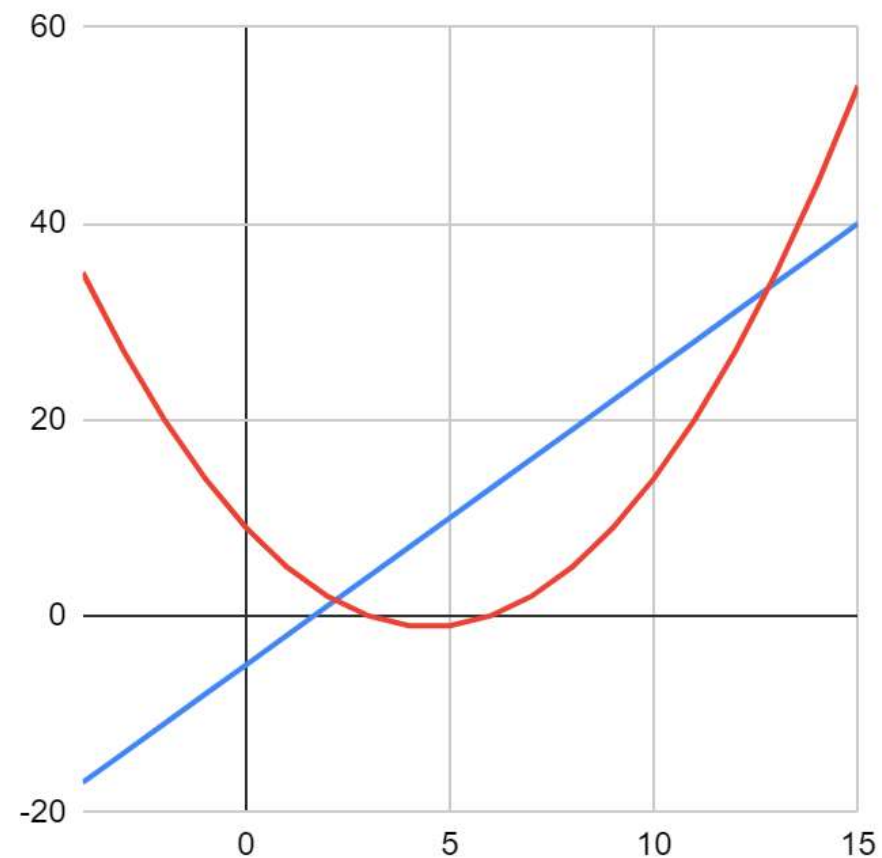
В небольшом масштабе

— —



В большом масштабе

— —



# Пример №3

- $N = 50$
- `For(int I = 4; I < N/10; i++)`
- `For(int j = 2; j < N/25; j++)`
- {
- `DoSomething();`
- } // сколько будет итераций цикла?

$$4 \dots 4 * 3 \dots 3 = 1$$

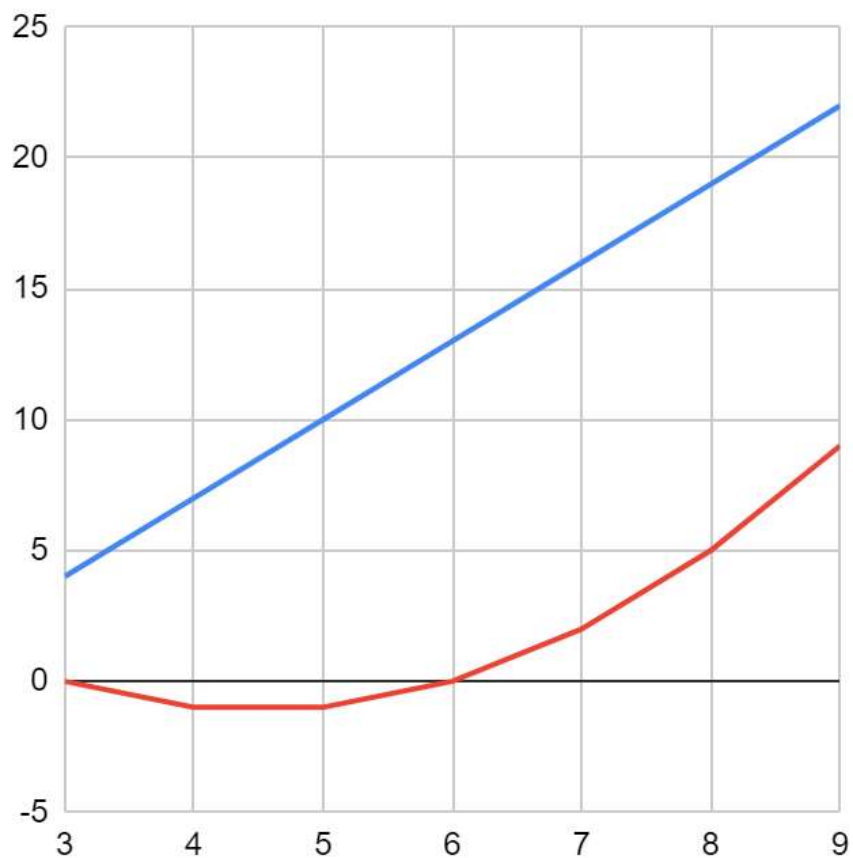
- $N = 50$
- `For(int I = 0; I < N*10; i++)`
- {
- `DoSomething();`
- } // сколько будет итераций цикла?

$$0 \dots 499 = 500$$

# Примеры графиков

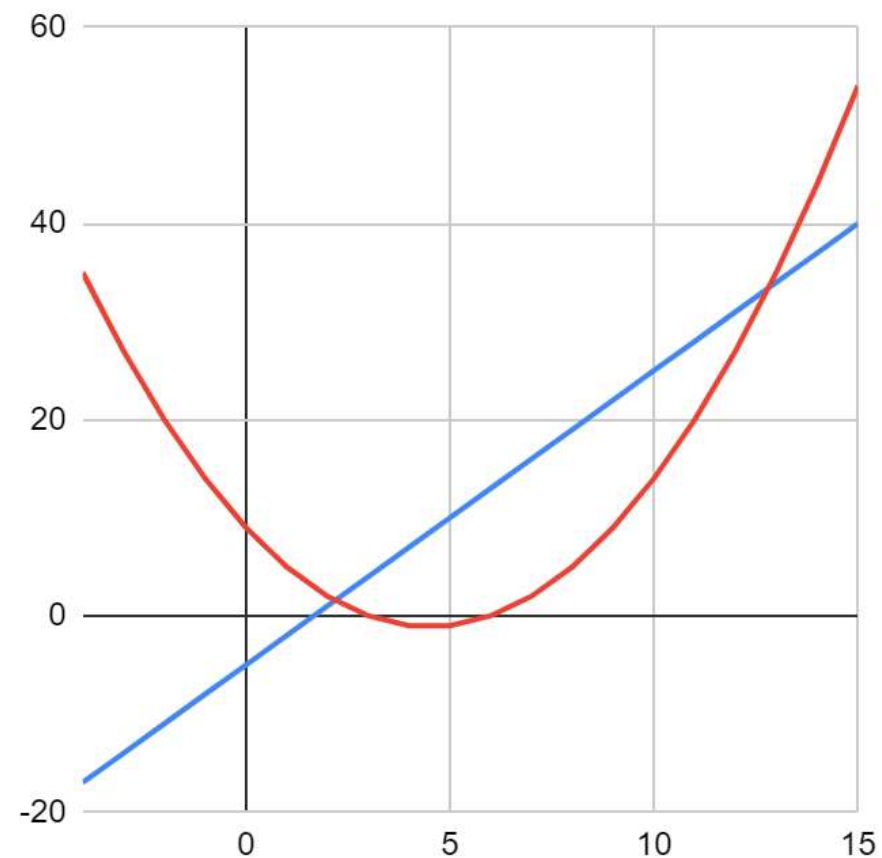
В небольшом масштабе

— —



В большом масштабе

— —





# Примеры алгоритмов

$O(N)$

```
int N = 1000;  
int sum = 0;  
for (int i = 0; i < N; i++)  
{  
    sum += i;  
}
```

# $O(N^2)$

```
int N = 3500;  
long sum = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i; j < N; j++)  
    {  
        sum += i;  
    }
```

# $O(\log N)$

```
int N = 64;
int[] sorted = new int[N];
for(int i = 0; i < N; i++)
    sorted[i] = i;

int j = N / 2, index = 0, c = 63;
while (sorted[j] != c) {
    if (sorted[j] > c)
        j = index + (j - index) / 2;
    else {
        var d = j - index;
        index = j;
        j = index + d / 2;
    }
}
```

# Как в реальности?



# Какая сложность у алгоритма?

- a)  $O(1)$
- b)  $O(N^2)$
- c)  $O(N)$
- d)  $O(N \log N)$

```
// Собираем все маршруты из Старта в Финиш и анализируем их.  
List<int[]> mainRoutes = this.Scheme.FindRoutes(this.Scheme.StartBlockIndex,  
this.Scheme.FinishBlockIndex);  
for (var i = 0; i < mainRoutes.Count; i++) {  
    int[] mainRoute = mainRoutes[i];  
    var otherRoutes = mainRoutes.Where(r => !ReferenceEquals(r,  
mainRoute)).ToList();  
    incorrectParallelBlockIndexes.AddRange(this.ValidateForRecursiveParallel(mainRo  
ute, otherRoutes));  
    if (this.validateEmptyBranches)  
        this.ValidateForEmptyBranches(mainRoute, otherRoutes,  
emptyParallelBlockIndexes);  
}  
incorrectParallelBlockIndexes.AddRange(this.ValidateForIncorrectParallelWay(mainRoutes  
);
```

# Какая сложность у алгоритма?

- a)  $O(1)$
- b)  $O(N^2)$
- c)  $O(N)$
- d)  $O(N \log N)$

// Идём с конца и ищем первый общий блок, являющийся стартом параллельности.

```
for (int i = firstRoute.Length - 1; i >= 0; i--)  
{  
    int blockIndex = firstRoute[i];  
    if (secondRoute.Contains(blockIndex))  
        return this.IsParallelBeginningBlock(blockIndex);  
}  
return false;
```

# Какая сложность у алгоритма?

```
private bool IsContainsParallelCycles(int blockIndex, List<int> visited, int[] route){  
    if (visited.Contains(blockIndex))  
        return true;  
  
    if (route.Contains(blockIndex))  
        return false;  
  
    visited.Add(blockIndex);  
    var downlinks = this.Scheme.GetDownlinks(blockIndex);  
    foreach (int downlinkIndex in downlinks) {  
        if (this.IsContainsParallelCycles(downlinkIndex, visited, route))  
            return true;  
    }  
    return false;  
}
```

- a)  $O(2^N)$
- b)  $O(N^2)$
- c)  $O(N)$
- d)  $O(M^{(N/M)})$



# Варианты оптимизаций

1. Избавление от лишних операций, циклов по всей цепочке вызовов
2. Поиск не по всем данным, а по некоторой части данных
3. Кеширование некоторых данных или результатов исполнения
4. Подготовка данных для поиска:
  - сортировка
  - иная реорганизация хранения с целью оптимизации поиска по данным

# Избавление от лишних вызовов

```
private static int[,] array = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 },
    { 0, -1, -3 }
};

public static void Main() {
    Console.WriteLine(GetMax());
    Console.WriteLine(GetMin());
    Console.WriteLine(GetSum());
    for (int i = 0; i < array.GetLength(0); i++) {
        Console.WriteLine(GetAverage(i));
    }
}
```

```
public static int GetMax() {
    int max = int.MinValue;
    for(int i = 0; i < array.GetLength(0); i++)
        for (int j = 0; j < array.GetLength(1); j++) {
            if (max < array[i, j])
                max = array[i, j];
        }
    return max;
}

public static int GetMin() {
    int min = int.MaxValue;
    for (int i = 0; i < array.GetLength(0); i++)
        for (int j = 0; j < array.GetLength(1); j++) {
            if (min > array[i, j])
                min = array[i, j];
        }
    return min;
}
```

# Варианты оптимизаций

1. Избавление от лишних операций, циклов по всей цепочке вызовов
2. **Поиск не по всем данным, а по некоторой части данных**
3. Кеширование некоторых данных или результатов исполнения
4. Подготовка данных для поиска:
  - сортировка
  - иная реорганизация хранения с целью оптимизации поиска по данным

# Поиск не по всем данным

// Задача: найти все слова, у которых после буквы "а" стоит буква, которую ввел пользователь

```
public static void Main() {  
    var matchedWords = words.Where(x => x.Contains('a',  
StringComparison.CurrentCultureIgnoreCase)).ToArray();  
    while (true) {  
        char letter = Console.ReadKey().KeyChar;  
        Console.WriteLine(FindWord(matchedWords, letter));  
    }  
}  
  
private static string FindWord(string[] matchedWords, char letter) {  
    foreach(var word in matchedWords) {  
        bool aMatch = false;  
        foreach(var wordLetter in word.ToLower()) {  
            if (aMatch && wordLetter == letter) {  
                return word;  
            }  
            aMatch = wordLetter == 'a';  
        }  
    }  
    return string.Empty;  
}
```

```
static string[] words = {  
    "Апельсин",  
    "Абрикос",  
    "Ананас",  
    "Банан",  
    "Авокадо",  
    "Манго",  
    "Киви",  
    "Груша",  
    "Яблоко",  
    "Слива"  
};
```

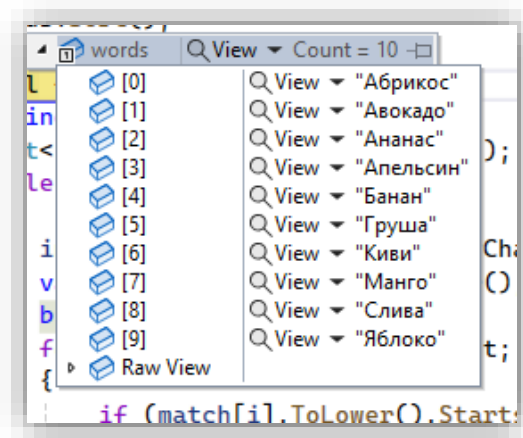
# Варианты оптимизаций

1. Избавление от лишних операций, циклов по всей цепочке вызовов
2. Поиск не по всем данным, а по некоторой части данных
3. Кеширование некоторых данных или результатов исполнения
4. **Подготовка данных для поиска:**
  - **сортировка**
  - иная реорганизация хранения с целью оптимизации поиска по данным

# Сортировка данных

// Задача: выводить пользователю подсказку, когда он начинает печатать слово

```
public static void Main() {  
    words.Sort();  
    bool finish = false;  
    string input = string.Empty;  
    List<string> match = words.ToList();  
    while (!finish) {  
        input += Console.ReadKey().KeyChar;  
        var matches = new List<string>();  
        bool found = false;  
        for (var i = 0; i < match.Count; i++) {  
            if (match[i].ToLower().StartsWith(input)) {  
                matches.Add(match[i]);  
                found = true;  
            } else // если уже нашли, то дальше искать бессмысленно  
                if (found) break;  
        }  
        match = matches;  
        Console.Clear();  
        matches.ForEach(x => Console.WriteLine(x + " "));  
        Console.Write(input);  
    }  
}
```



```
static List<string> words = new List<string> {  
    "Апельсин",  
    "Абрикос",  
    "Ананас",  
    "Банан",  
    "Авокадо",  
    "Манго",  
    "Киви",  
    "Груша",  
    "Яблоко",  
    "Слива"  
};
```

# Варианты оптимизаций

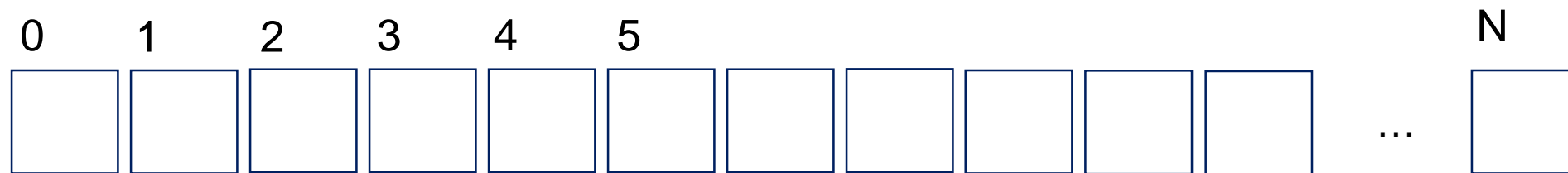
1. Избавление от лишних операций, циклов по всей цепочке вызовов
2. Поиск не по всем данным, а по некоторой части данных
3. Кеширование некоторых данных или результатов исполнения
4. Подготовка данных для поиска:
  - сортировка
  - **иная реорганизация хранения с целью оптимизации поиска по данным**

# Виды коллекций

- Массив
- Список (односвязный, двусвязный)
- Стек
- Хеш-таблица
- Битовый массив

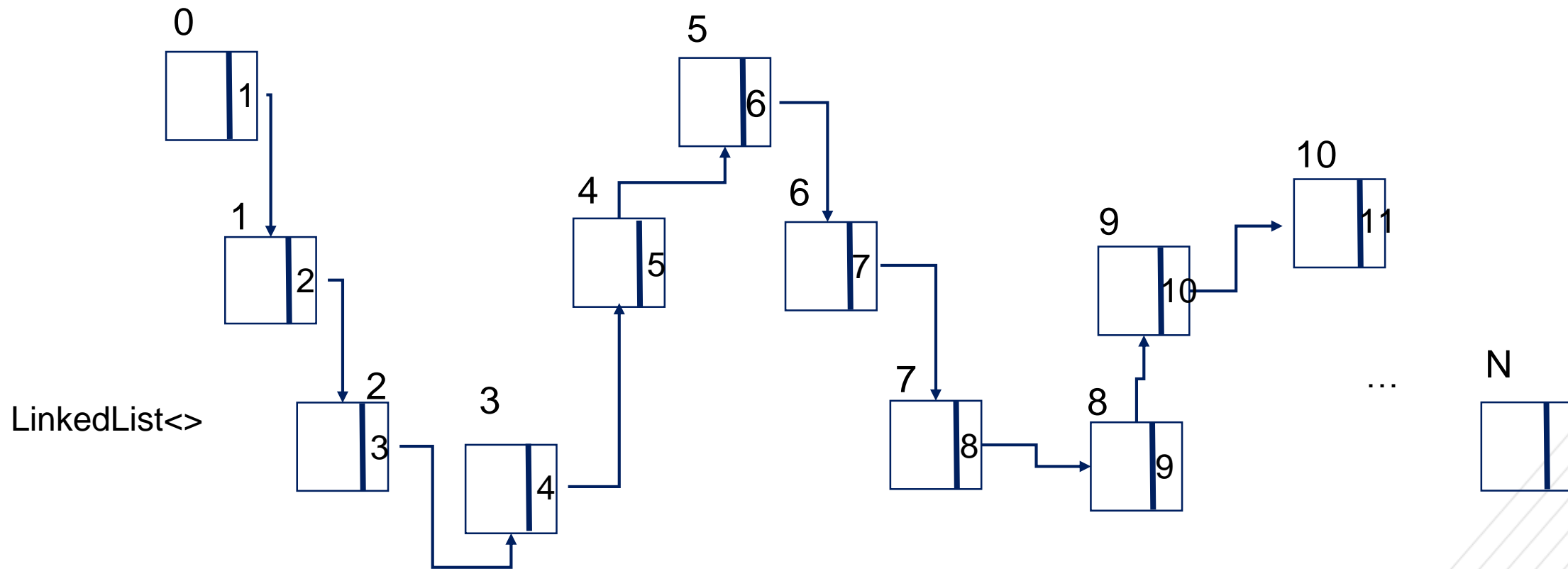


# Массив



Array<>  
ArrayList  
SortedList<,>  
List<>  
SortedDictionary<,>  
Queue<>

# Список



# Хеш-таблица

## Использование

ключ	значение
ключ	значение
ключ	значение
ключ	значение
ключ	значение
ключ	значение

Dictionary<,>  
HashSet<>  
KeyedCollection<,>  
StringDictionary  
Hashtable


...

## Внутреннее устройство



...

# Сравнение видов коллекций

	массив	список	Хеш-таблица
Потребление памяти			
Удаление/вставка	$O(N)$	$O(1)$	$O(1)$
Поиск элемента	$O(N)$	$O(N)$	$O(1)$
Замена элемента	$O(1)$	$O(1)$	$O(1)$
Обращение по индексу	$O(1)$	$O(N)$	-
Копирование элемента	$O(N)$	$O(1)$	-
Размер коллекции	$O(1)$	$O(N)$	$O(1)$

# List vs Hashtable

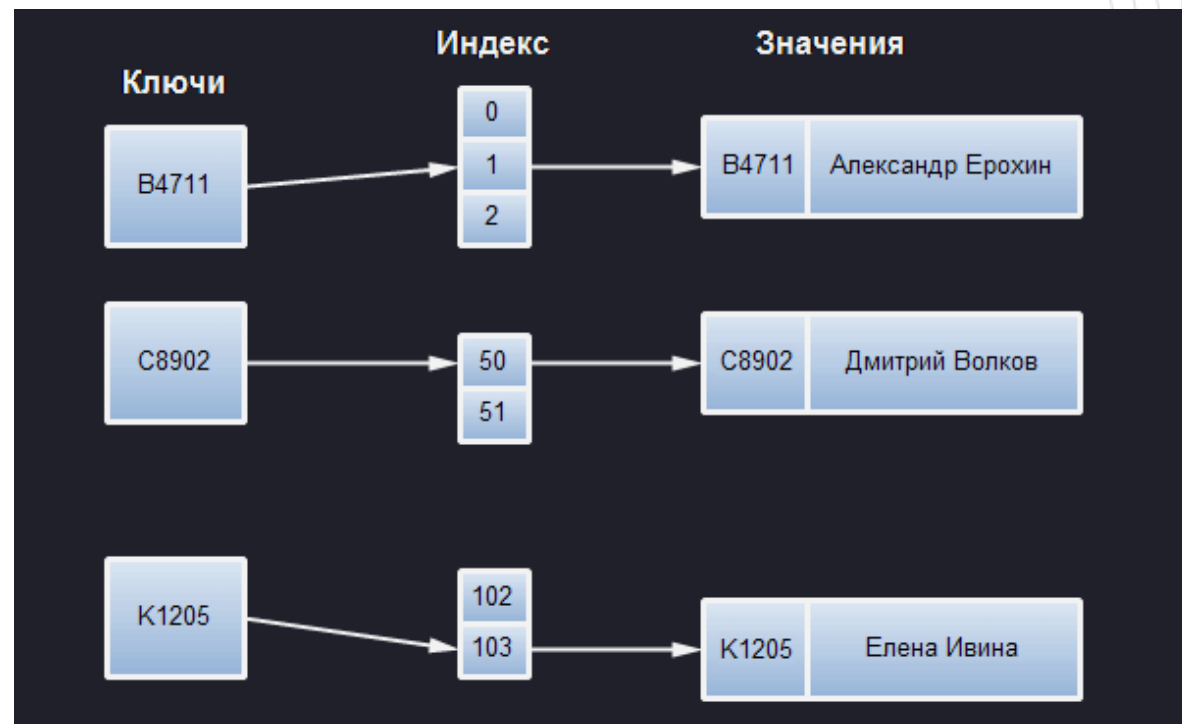
- Упорядоченная коллекция
  - Может содержать дубликаты
  - В основе лежит массив элементов
  - Массив может изменять размер при удалении/добавлении элементов
  - Можно использовать широкий набор Linq-методов (v .NET) как это было раньше во фреймворке?
- Неупорядоченная коллекция
  - Не может содержать дубликатов
  - Динамическая структура
  - Нужно реализовать GetHashCode для элементов
  - Некоторые операции выполняются быстрее (например, Contains, Remove, Add выполняются за  $O(1)$ )

# GetHashCode требования

- Один и тот же объект должен всегда возвращать одно и то же значение.
- Разные объекты могут возвращать одно и то же значение.
- Он должен выполняться насколько возможно быстро, не требуя значительных вычислительных затрат.
- Он не должен генерировать исключений.
- Он должен использовать как минимум одно поле экземпляра.
- Значения хеш-кода должны распределяться равномерно по всему диапазону чисел, которые может хранить `int`.
- Хеш-код не должен изменяться на протяжении времени существования объекта.

# Dictionary

- Элемент хранения – ключ-значение (keyValuePair)
- Доступ к элементам за константное время  $O(1)$



# Реализация поиска Dictionary

```
uint hashCode = (uint) comparer.GetHashCode (key) ;  
int i = GetBucket (hashCode) ;  
Entry[]? entries = _entries ;  
uint collisionCount = 0 ;
```



# Реорганизация данных

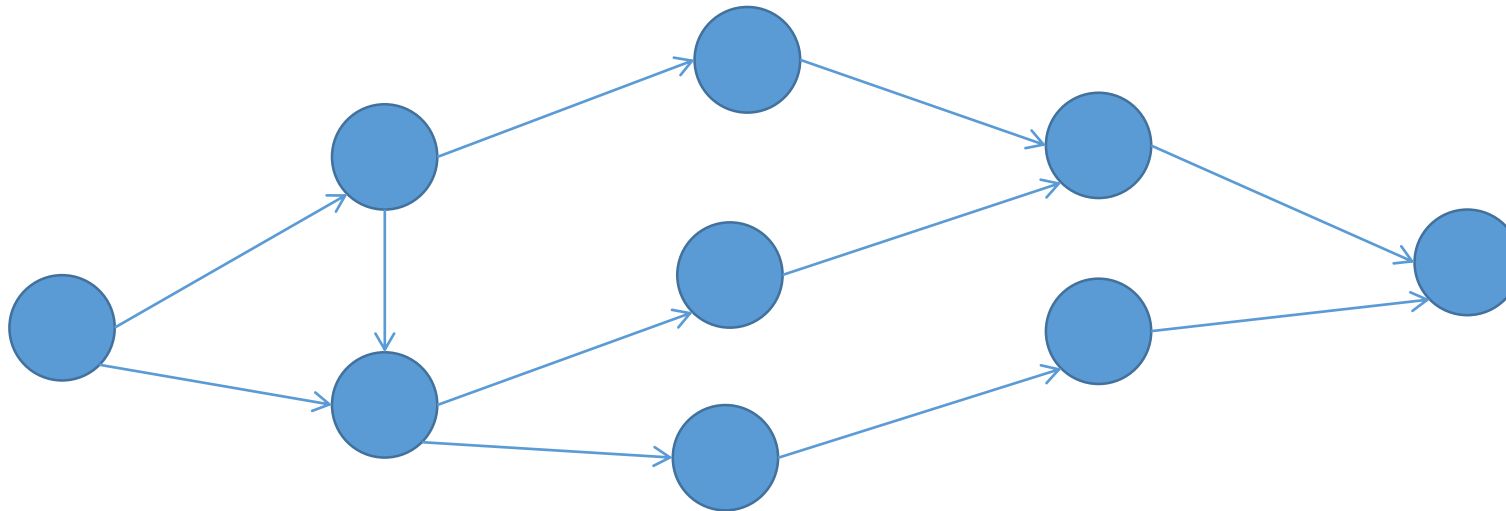
- Неупорядоченная коллекция -> упорядоченная коллекция
- Нефильтрованный список -> фильтрованный группированный список
- Список и функция преобразования -> списки с предвычисленными значениями
- Список -> хеш-таблица

# Варианты оптимизаций

- Избавление от лишних операций, циклов по всей цепочке вызовов
- Поиск не по всем данным, а по некоторой части данных
- Кеширование некоторых данных или результатов исполнения
- Подготовка данных для поиска:
- Сортировка
- Иная реорганизация хранения с целью оптимизации поиска по данным

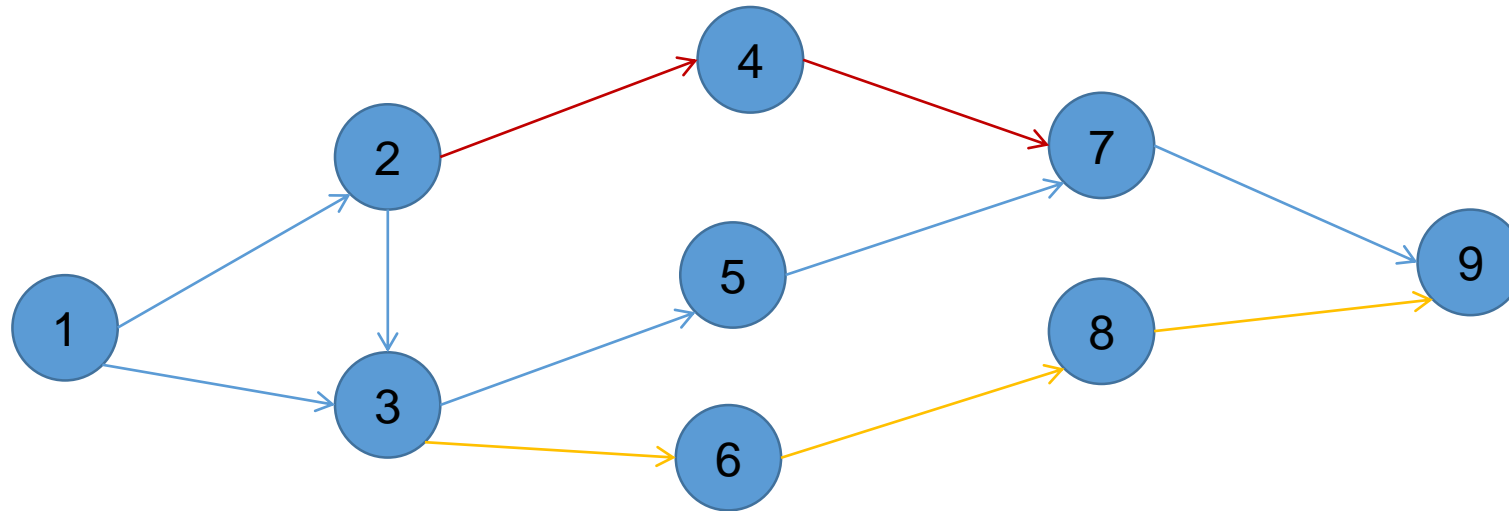
# Задача

Представить граф в виде некоторого объекта, по которому можно быстро определить наличие или отсутствие заданного перехода.



# Задача

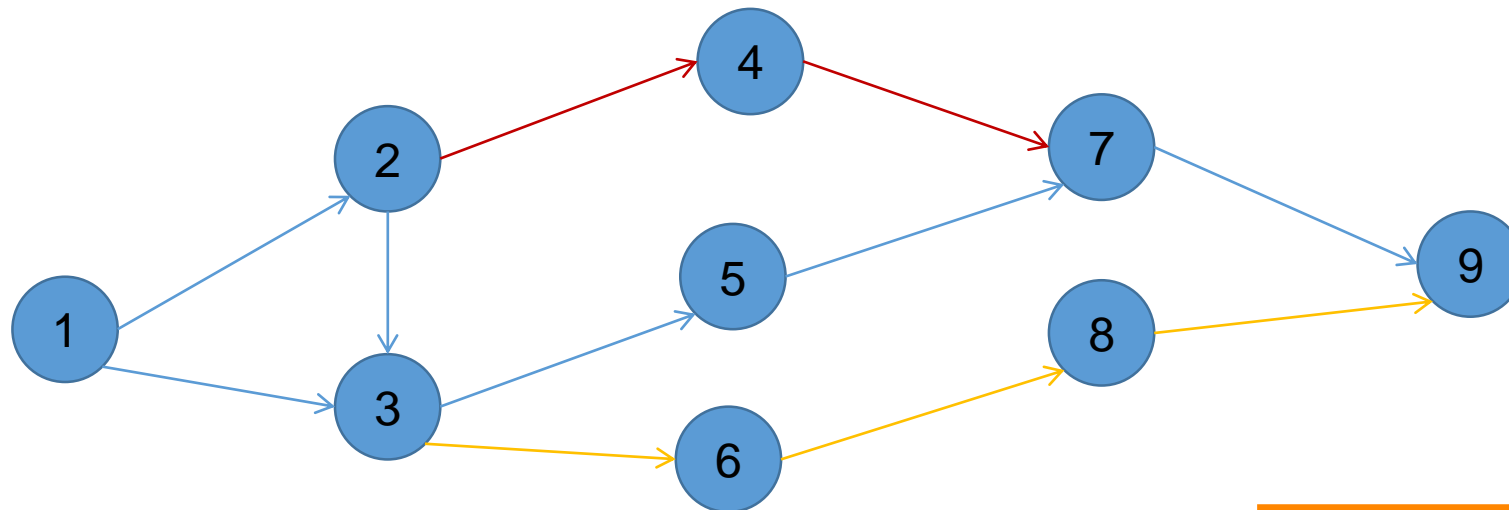
Представить граф в виде некоторого объекта, по которому можно быстро определить наличие или отсутствие заданного перехода.



1-2-4-7-9 1-2-3-5-7-9 1-2-3-6-8-9

1-3-5-7-9 1-3-6-8-9

# Реализация



Dictionary<int, int>

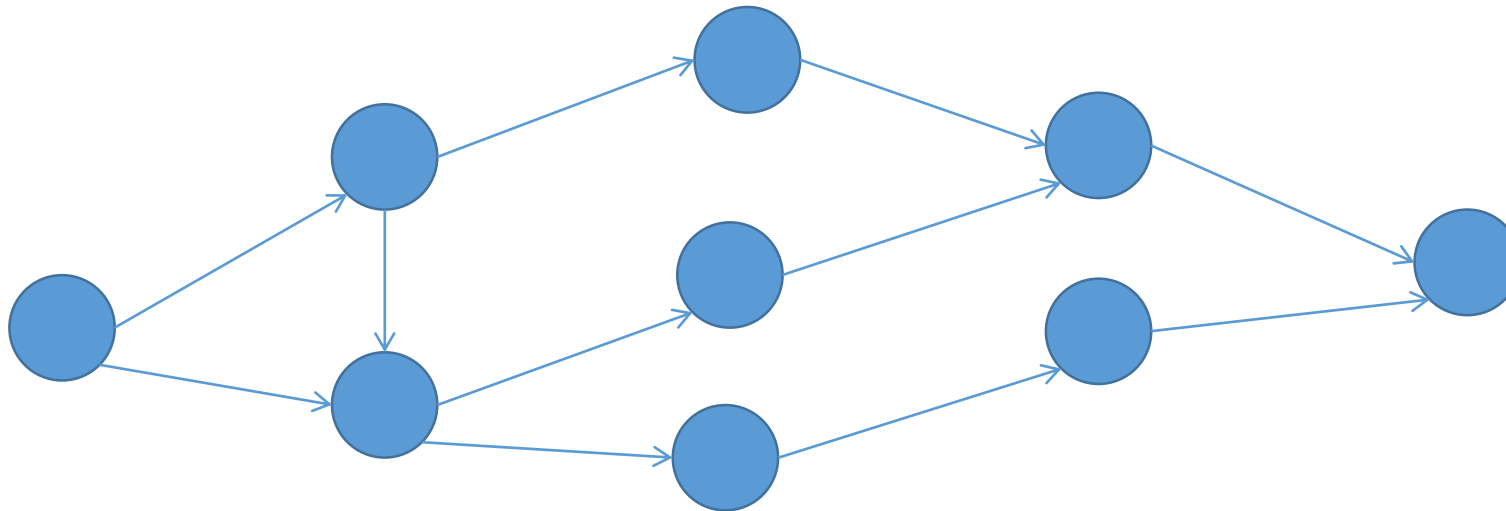
Исходящий узел →      ← Входящий узел

исходящий	входящий
1	2
1	3
2	3
2	4
3	5
3	6

...

# Задача (самостоятельно)

Представить граф в виде некоторого объекта, по которому можно быстро определить, является ли заданный узел таким, из которого выходит 2 или более связи. Какую структуру данных лучше использовать?



# Варианты оптимизаций

- Избавление от лишних операций, циклов по всей цепочке вызовов
- Поиск не по всем данным, а по некоторой части данных
- Кеширование некоторых данных или результатов исполнения
- Подготовка данных для поиска:
- Сортировка
- Иная реорганизация хранения с целью оптимизации поиска по данным

# Литература

- <https://habr.com/ru/articles/782608/>
- [https://ru.hexlet.io/courses/basic-algorithms/lessons/algorithm-complexity/theory\\_unit](https://ru.hexlet.io/courses/basic-algorithms/lessons/algorithm-complexity/theory_unit)
- Исходники .Net  
<https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Collections/Generic/Dictionary.cs>
- С. Голдштейн «Оптимизация приложений на платформе .Net» (2014)
- Также поможет здравый смысл и навыки экономии 😊



# Спасибо!

Ваши вопросы

