Теоретические основы облачных технологий

CAP-теорема

Невозможно построить такую распределенную вычислительную систему, которая бы одновременно обладала следующими тремя свойствами:

- **Consistency** (непротиворечивость) - подразумевает, что данные всегда должны быть непротиворечивыми, в какой бы момент времени мы ни запросили их;
- **Availability** (доступность) - все данные доступны в любой момент времени;
- **Partition Tolerance** (устойчивость к разделению) - предполагает, что система продолжает работу при выходе из строя одного или нескольких распределенных узлов.

Проблема Consistency

Возникает в случае, когда запрос обращается к серверу, на котором отсутствуют запрашиваемые данные.

Решение проблемы – синхронизация данных в между всем узлами распределенной системы.

Проблема Availability

Возникает в случае, когда один или несколько узлов распределенной системы становится недоступным для запросов.

Решение проблемы – «отложенная» или «оффлайновая» синхронизация данных.

Проблема Partition Tolerance

Возникает в случае, когда два или несколько узлов распределенной системы не имеют возможности синхронизации.

Решение проблемы, без увеличения риска потери доступности и возникновения противоречивости **отсутствует**.

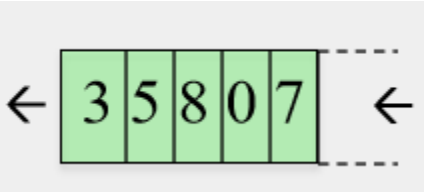
Решение теоремы

Таким образом, решения проблемы CAP фокусируются на различных комбинациях двух из трех фигурирующих в CAP-теореме характеристик.

Облачные технологии также не могут обойти ограничение этой теоремы, но наиболее близки к этому.

Структуры данных FIFO

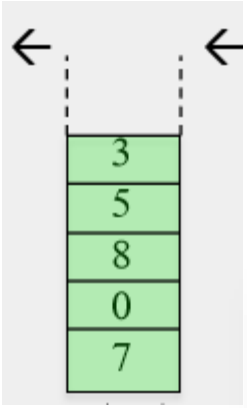
First-In First-Out



Структура данных, где сначала обрабатывается самая старая (первая) запись, или «голова» очереди.
Она аналогична обработке очереди (первым пришел, первым обслужен (ППП0)): где люди покидают очередь в том порядке, в котором они прибывают.

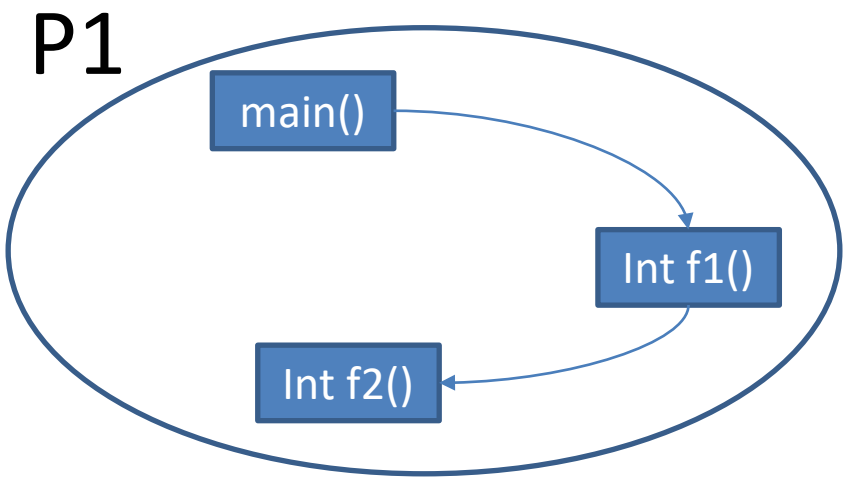
Структуры данных LIFO

Last-In First-Out



Способ организации и манипулирования данными относительно времени и приоритетов. В структурированном линейном списке, организованном по принципу LIFO, элементы могут добавляться и выбираться только с одного конца, называемого «вершиной списка». Структура LIFO может быть проиллюстрирована на примере стопки тарелок: чтобы взять вторую сверху, нужно снять верхнюю, а чтобы снять последнюю, нужно снять все лежащие выше.

Процесс



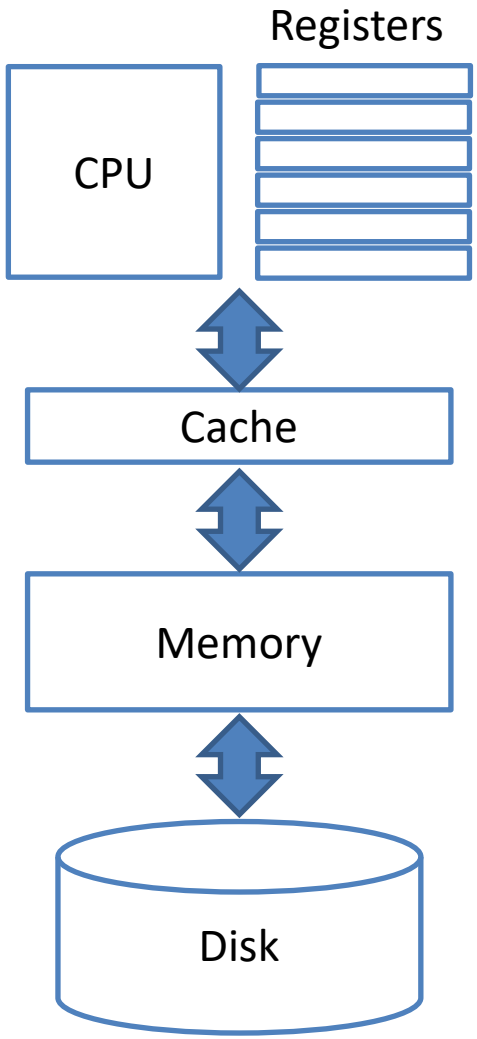

```
void main() {
        …
}
int f1()
{
       int x=1;
        …
}
int f2()
{
        …
}
```

Процесс — программа, которая выполняется в текущий момент. Совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Компьютерная программа сама по себе — это только пассивная последовательность инструкций, в то время как процесс — это непосредственное выполнение этих инструкций.

Также, процессом называют выполняющуюся программу и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и так далее.

Архитектура компьютера



- Программы хранятся на диске (Disk)
- Центральный процессор (CPU) загружает инструкции из оперативной памяти (Memory), кэша (Cashe) и регистров (Registers).

Нотация Big O()

Временная сложность алгоритма

Сложность алгоритма — это то, что основывается на сравнении двух алгоритмов на идеальном уровне, игнорируя низкоуровневые детали вроде реализации языка программирования, «железа», на котором запущена программа, или набора команд в данном CPU.

Примерами чисто вычислительных операций могут послужить:

- операции над числами с плавающей запятой (сложение и умножение);
- поиск заданного значения из находящейся в ОЗУ базы данных;
- запуск шаблона регулярного выражения на соответствие строке.

Нотация Big O()

Поиск максимального элемента массива

Дан входной массив ***A*** размером ***n***:

```
var M = A[ 0 ];

for ( var i = 0; i < n; ++i ) {
    if ( A[ i ] >= M ) {
        M = A[ i ];
    }
}
```

Необходимо подсчитать, сколько здесь вычисляется *фундаментальных инструкций*.

Предположим, что наш процессор способен выполнять как единые инструкции следующие операции:

- Присваивать значение переменной;
- Находить значение конкретного элемента в массиве;
- Сравнивать два значения;
- Инкрементировать значение;
- Основные арифметические операции (например, сложение и умножение).

Нотация Big O()

Мы будем полагать, что ветвление (выбор между if и else частями кода после вычисления if-условия) происходит мгновенно, и не будем учитывать эту инструкцию при подсчёте.

```
var M = A[ 0 ];
```

Для первой строки в коде требуются две инструкции: для поиска ***A[0]*** и для присвоения значения ***M*** (мы предполагаем, что n всегда как минимум 1). Эти две инструкции будут требоваться алгоритму, вне зависимости от величины **n**. Инициализация цикла for тоже будет происходить постоянно, что даёт нам ещё две команды: присвоение и сравнение.

```
i = 0;
i < n;
```

Всё это происходит до первого запуска for. После каждой новой итерации мы будем иметь на две инструкции больше: инкремент i и сравнение для проверки, не пора ли нам останавливать цикл.

```
++i;
i < n;
```

Нотация Big O()

Таким образом, если мы проигнорируем содержимое тела цикла, то количество инструкций у этого алгоритма:

4 + 2n

— четыре на начало цикла for и по две на каждую итерацию, которых мы имеем ***n*** штук.

Теперь мы можем определить математическую функцию ***f(n)*** такую, что зная ***n***, мы будем знать и необходимое алгоритму количество инструкций.

Для цикла for с пустым телом

f(n) = 4 + 2n

Нотация Big O()

Анализ наиболее неблагоприятного случая

В теле цикла мы имеем операции поиска в массиве и сравнения, которые происходят всегда:

```
if ( A[ i ] >= M ) { ...
```

Но тело if может запускаться, а может и нет, в зависимости от актуального значения из массива. Если произойдёт так, что ***A[i] >= M***, то у нас запустятся две дополнительные команды: поиск в массиве и присваивание:

```
M = A[ i ]
```

Мы уже не можем определить ***f(n)*** так легко, потому что теперь количество инструкций зависит не только от ***n***, но и от конкретных входных значений. Таким образом, в наихудшем случае в теле цикла из нашего кода запускается четыре инструкции, и мы имеем.

$$f(n) = 4 + 2n + 4n = 6n + 4$$

Нотация Big O()

Асимптотическое поведение

Наша функция **6n + 4** состоит из двух элементов: **6n** и **4**.

При анализе сложности важность имеет только то, что происходит с функцией подсчёта инструкций при значительном возрастании ***n***.

Очевидно, что 4 останется 4 вне зависимости от значения ***n***, а ***6n*** наоборот будет расти. Поэтому первое, что мы сделаем, — это отбросим 4 и оставим только ***f(n) = 6n***.

Второй вещью, на которую можно не обращать внимания, является множитель перед ***n***.
Так что наша функция превращается в ***f(n) = n***.

Описанные выше фильтры — «отбрось все факторы» и «оставляй только наибольший элемент» — в совокупности дают то, что называется **асимптотическим поведением**

Нотация Big O()

Асимптотическое поведение

Найдём асимптотики для следующих примеров, используя принципы отбрасывания константных факторов и оставления только максимально быстро растущего элемента:

f(n) = 5n + 12 даст f(n) = n
Основания — те же, что были описаны выше

f(n) = 109 даст f(n) = 1
Мы отбрасываем множитель в 109 * 1 , но 1 по-прежнему нужен, чтобы показать, что функция не равна нулю

f(n) = n^2 + 3n + 112 даст f(n) = n^2
Здесь n^2 возрастает быстрее, чем 3n, который, в свою очередь, растёт быстрее 112

f(n) = n^3 + 1999n + 1337 даст f(n) = n^3
Несмотря на большую величину множителя перед n, мы по прежнему полагаем, что можем найти ещё больший n, поэтому f(n) = n^3 всё ещё больше 1999n

Нотация Big O()

Сложность

Фактически, любая программа, не содержащая циклы, имеет ***f(n) = 1***, потому что в этом случае требуется константное число инструкций.

Одиночный цикл от ***1*** до ***n***, даёт асимптотику ***f(n) = n***, поскольку до и после цикла выполняет неизменное число команд, а постоянное же количество инструкций внутри цикла выполняется ***n*** раз.

```php
<?php
    $exists = false;
    for ( $i = 0; $i < n; ++$i ) {
        if ( $A[ $i ] == $value ) {
            $exists = true;
            break;
        }
    }
?>
```

```
v = a[ 0 ] + a[ 1 ]
```

Нотация Big O()

Сложность

```
bool duplicate = false;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        if ( i != j && A[ i ] == A[ j ] ) {
            duplicate = true;
            break;
        }
    }
    if ( duplicate ) {
        break;
    }
}
```

Программа на C++ проверяет, содержит ли вектор (своеобразный массив) A размера n два одинаковых значения

Нотация Big O()

Сложность

Простые программы можно анализировать с помощью подсчёта в них количества вложенных циклов.

Одиночный цикл в n итераций даёт $f(n) = n$

Цикл внутри цикла — $f(n) = n^2$

Цикл внутри цикла внутри цикла — $f(n) = n^3$

И так далее…

Нотация Big O()

Сложность

Если у нас имеется серия из последовательных for-циклов, то асимптотическое поведение программы определяет наиболее медленный из них.

Два вложенных цикла, идущие за одиночным, асимптотически тоже самое, что и вложенные циклы сами по себе. Говорят, что вложенные циклы ***доминируют*** над одиночными.

Нотация Big O()

Когда мы выясняем точную асимптотику f, мы говорим, что наша программа — Θ(f(n)).

Например, в примерах выше программы Θ(1), Θ(n^2) и Θ(n^2), соответственно.

Θ(n) произносится как «тета от n».

Иногда мы говорим, что f(n) есть Θ(что-то). Например, можно сказать, что f(n) = 2n — это функция, являющаяся Θ(n).

Можно так же написать, что 2n ∈ Θ(n), что произносится: «два n принадлежит тета от n».

Такая нотация говорит о том, что если мы посчитаем количество необходимых программе команд, и оно будет равно 2n, то асимптотика этого алгоритма описывается как n (что мы находим, отбрасывая константу).

Нотация Big O()

Имея данную систему обозначений, приведём несколько истинных математических утверждений:

1. $n^6 + 3n \in \Theta(n^6)$
2. $2^n + 12 \in \Theta(2^n)$
3. $3^n + 2^n \in \Theta(3^n)$
4. $n^n + n \in \Theta(n^n)$

Нотация Big O()

Эту функцию (т.е. то, что пишем Θ(здесь)) временной сложностью, или просто сложностью алгоритма.

Таким образом**, алгоритм с Θ(n) имеет сложность n**. Также существуют специальные названия для Θ(1), Θ(n), Θ(n^2) и Θ(log(n)), потому что они встречаются очень часто.

Говорят, что:
Θ(1) — алгоритм с константным временем,
Θ(n) — линейный,
Θ(n^2) — квадратичный,
Θ(log(n)) — логарифмический.

Программы с большим Θ выполняются медленнее, чем с меньшим.

Нотация Big O()

В реальной жизни иногда проблематично выяснить точное поведение алгоритма тем способом, который мы рассматривали выше. Особенно для более сложных примеров.

Однако, мы можем сказать, что поведение нашего алгоритма никогда не пересечёт некой границы. Это делает жизнь проще, так как чёткого указания на то, насколько быстр наш алгоритм, у нас может и не появиться, даже при условии игнорирования констант (как раньше).

Всё, что нам нужно — найти эту границу, а как это сделать — проще объяснить на примере.

Нотация Big O()

```
b = []
n.times do
    m = a[ 0 ]
    mi = 0
    a.each_with_index do |element, i|
        if element < m
            m = element
            mi = i
        end
    end
    a.delete_at( mi )
    b << m
end
```

Нотация Big O()

Если мы внесём **«ухудшающие» изменения**, то наш новый алгоритм будет иметь $\Theta(n^2)$, поскольку получим два вложенных цикла, каждый из которых выполняется ровно n раз.

А если это так, то оригинальный алгоритм имеет $O(n^2)$.

$O(n^2)$ произносится как «большое О от n в квадрате».

Это говорит о том, что наша программа асимптотически не хуже, чем n^2. Она будет работать или лучше, или также.

Любая программа с Θ(a) является O(b) при b худшем, чем a.

Нотация Big O()

1. Алгоритм с Θ(n) имеет O(n)

Истина, поскольку оригинальная программа имеет Θ(n). Следовательно, O(n) может быть достигнуто и без изменения программы

Нотация Big O()

2. Алгоритм с Θ(n) имеет $O(n^2)$

Поскольку n^2 хуже n, то это истина

Нотация Big O()

3. Алгоритм с $\Theta(n^2)$ имеет $O(n^3)$

Поскольку n^3 хуже n^2, то это истина

4. Алгоритм с Θ(n) имеет O(1)

Поскольку 1 не хуже n, то это ложь. Если программа асимптотически использует n инструкций (линейное число), то мы не можем сделать её хуже так, чтобы асимптотически ей требовалась всего 1 (константное число) инструкций.

Нотация Big O()

5. Алгоритм с O(1) имеет Θ(1)

Истина, поскольку обе сложности одинаковые

6. Алгоритм с O(n) имеет Θ(1)

Может как быть, так и не быть истиной, зависит от алгоритма.

В общем случае — это ложь. Если алгоритм является Θ(1), то он, безусловно, и O(n).

Но если он O(n), то может и не быть Θ(1). Например, Θ(n) алгоритм является O(n), а Θ(1) — нет.

Нотация Big O()

Поскольку O-сложность алгоритма представляет собой *верхний предел* его настоящей сложности, которую, в свою очередь, отображает Θ, то иногда мы говорим, что Θ даёт нам *точную оценку*.

Если мы знаем, что найденная нами граница не точна, то можем использовать **строчное o**, чтобы её обозначить.

Например, если алгоритм является Θ(n), то его точная сложность — n. Следовательно, этот алгоритм O(n) и $O(n^2)$ одновременно.

Поскольку алгоритм Θ(n), то O(n) определяет границу более точно. А $O(n^2)$ мы можем записать как $o(n^2)$ (произносится: «маленькое o от n в квадрате»), чтобы показать, что мы знаем о нестрогости границы.

Конечно, лучше, когда мы можем найти точные границы для нашего алгоритма, чтобы иметь больше информации о его поведении, но, к сожалению, это не всегда легко сделать.

Нотация Big O()

Определите строгость границ.

1. Θ(n) алгоритм, для которого мы нашли O(n), как верхнюю границу

В этом случае Θ-сложность и O-сложность одинаковые, поэтому граница строгая

Нотация Big O()

Определите строгость границ.

2. $\Theta(n^2)$ алгоритм, для которого мы нашли $O(n^3)$, как верхнюю границу

Здесь O-сложность большего масштаба, чем Θ, поэтому эта граница нестрогая. В самом деле, строгой границей здесь будет $O(n^2)$. Так что мы можем записать, что алгоритм является $o(n^3)$

Нотация Big O()

Определите строгость границ.

3. Θ(1) алгоритм, для которого мы нашли O(n), как верхнюю границу

Снова, O-сложность большего масштаба, чем Θ, из чего мы заключаем, что граница нестрогая. Строгой будет O(1), а O(n) можно переписать, как o(n)

Нотация Big O()

Определите строгость границ.

4. Θ(n) алгоритм, для которого мы нашли O(1), как верхнюю границу

Мы должны были совершить ошибку при выводе этой границы, потому что она неверна. Θ(n) алгоритм не может иметь верхнюю границу O(1), поскольку n имеет большую сложность, чем 1. Не забывайте, O обозначает верхнюю границу

Нотация Big O()

Определите строгость границ.

5. Θ(n) алгоритм, для которого мы нашли O(2n), как верхнюю границу

Может показаться, что тут нестрогая граница, но, вообще-то, это не так. На самом деле, граница строгая. Асимптотики 2n и n — одинаковые, а O и Θ связаны только с асимптотиками. Так что мы имеем O(2n) = O(n), следовательно, граница строгая, поскольку сложность равна Θ

Нотация Big O()

Выше мы меняли нашу программу, чтобы сделать её хуже (увеличили количество инструкций и, таким образом, время выполнения), чем создали O-нотацию. O говорит нам о том, что наш код никогда не будет работать медленнее определённого предела. Из этого мы получаем основания для оценки: достаточно ли хороша наша программа?

Если же мы поступим противоположным образом, сделав имеющийся код **лучше**, и найдём сложность того, что получится, то мы задействуем Ω-нотацию. Таким образом, Ω даёт нам сложность, лучше которой наша программа быть не может.

Она полезна, если мы хотим доказать, что программа работает медленно или алгоритм является плохим. Так же её можно применить, когда мы утверждаем, что алгоритм слишком медленный для использования в данном конкретном случае.

Например, высказывание, что алгоритм является $\Omega(n^3)$, означает, что алгоритм не лучше, чем n^3. Он может быть $\Theta(n^3)$, или $\Theta(n^4)$, или ещё хуже, но мы будем знать предел его «хорошести».

Таким образом, Ω даёт нам *нижнюю границу* сложности нашего алгоритма. Аналогично o, мы можем писать ω, если знаем, что этот предел нестрогий.

Например, $\Theta(n^3)$ алгоритм является $o(n^4)$ и $\omega(n^2)$. $\Omega(n)$ произносится как «омега большое от n», в то время как $\omega(n)$ произносится «омега маленькое от n».

Сложность

Нотация Big O()

Для следующих Θ-сложностей напишите строгие и нестрогие O-пределы и, по желанию, строгие и нестрогие Ω-пределы (при условии, что они существуют).

1. Θ(1)

Строгие границы будут O(1) и Ω(1). Нестрогой O-границей будет O(n). O даёт нам верхний предел. Поскольку n лежит на шкале выше, чем 1, то это — нестрогий предел, и мы также можем записать его как o(n). А вот найти нестрогий предел для Ω мы не можем, потому что не существует функции ниже, чем 1. Так что придётся иметь дело со строгой границей

Нотация Big O()

Для следующих Θ-сложностей напишите строгие и нестрогие O-пределы и, по желанию, строгие и нестрогие Ω-пределы (при условии, что они существуют).

2. Θ(√n)

Строгие пределы будут теми же, что и Θ-сложность, т.е. O(√n) и Ω(√n) соответственно. Для нестрогих пределов будем иметь O(n), поскольку n больше, чем √n. А поскольку эта граница нестрогая, то можем написать o(n). В качестве нижней нестрогой границы мы попросту используем Ω(1) (или ω(1))

Нотация Big O()

Для следующих Θ-сложностей напишите строгие и нестрогие O-пределы и, по желанию, строгие и нестрогие Ω-пределы (при условии, что они существуют).

3. Θ(n)

Строгие пределы O(n) и Ω(n). Нестрогими могут быть ω(1) и o(n^3). Не самые лучшие границы, поскольку обе достаточно далеки от оригинальной сложности, но они подходят под наше определение

Нотация Big O()

Для следующих Θ-сложностей напишите строгие и нестрогие O-пределы и, по желанию, строгие и нестрогие Ω-пределы (при условии, что они существуют).

4. $\Theta(n^2)$

Строгие границы $O(n^2)$ и $\Omega(n^2)$. В качестве нестрогих границ мы используем $\omega(1)$ и $o(n^3)$, как и в предыдущем примере

Нотация Big O()

Для следующих Θ-сложностей напишите строгие и нестрогие O-пределы и, по желанию, строгие и нестрогие Ω-пределы (при условии, что они существуют).

5. $\Theta(n^3)$

Строгие границы $O(n^3)$ и $\Omega(n^3)$, соответственно. Двумя нестрогими границами могут быть $\omega(\sqrt{n}\, n^2)$ и $o(\sqrt{n}\, n^3)$. Хотя эти пределы по-прежнему нестрогие, но они лучше тех, что мы выводили выше

Нотация Big O()

Оператор сравнения асимптотических оценок	**Оператор сравнения чисел**
Алгоритм является o(что-то)	Число < чего-то
Алгоритм является O(что-то)	Число ≤ чего-то
Алгоритм является Θ(что-то)	Число = чему-то
Алгоритм является Ω(что-то)	Число ≥ чего-то
Алгоритм является ω(что-то)	Число > чего-то

Несмотря на то, что все символы O, o, Ω, ω и Θ необходимы время от времени, O используется чаще всего, поскольку его проще найти, чем Θ, и оно более полезно на практике, чем Ω.

Рекурсивная сложность

Нотация Big O()

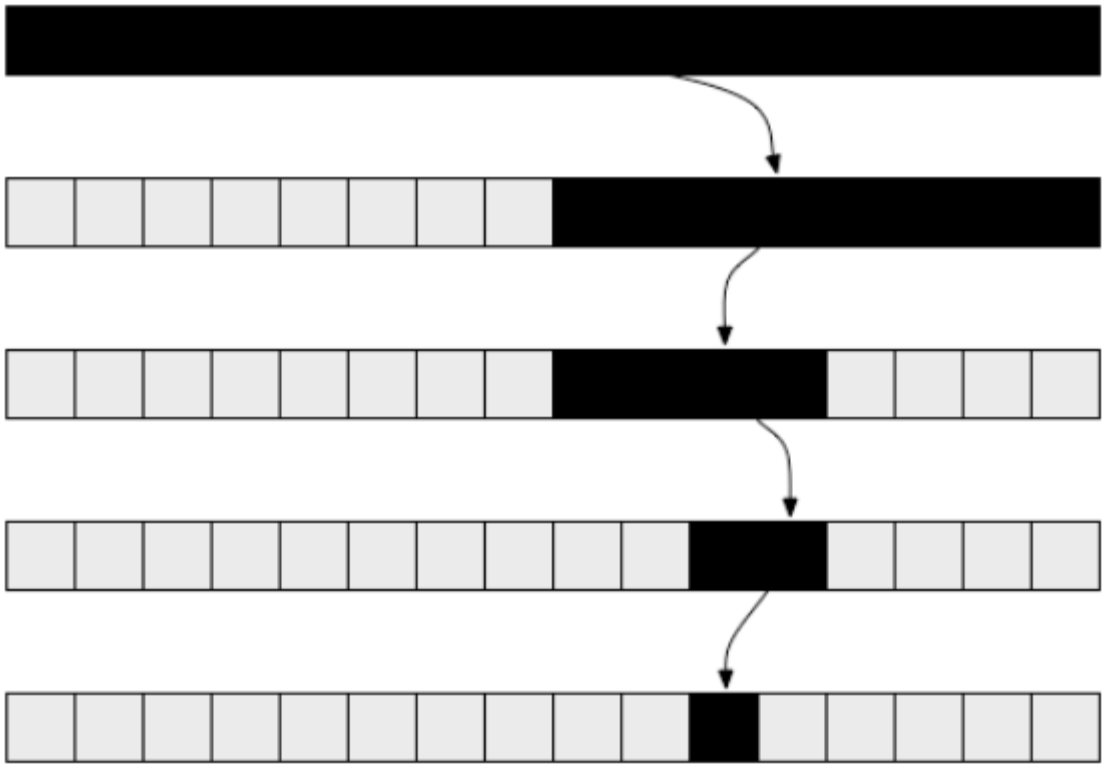
```
def factorial( n ):
    if n == 1:
        return 1
    return n * factorial( n - 1 )
```

Она не содержит циклов, но её сложность всё равно не является константной. Очевидно, что если мы применим эту функцию к некоторому n, то она будет вычисляться n раз. Таким образом, мы видим, что эта функция является Θ(n).

Если вы всё же не уверены в этом, то вы всегда можете найти точную сложность путём подсчёта количества инструкций.

Логарифмическая сложность

Нотация Big O()



Алгоритм бинарного поиска

Логарифмическая сложность

Нотация Big O()

```
def binarySearch( A, n, value ):
    if n = 1:
        if A[ 0 ] = value:
            return true
        else:
            return false
    if value < A[ n / 2 ]:
        return binarySearch( A[ 0...( n / 2 - 1 ) ], n / 2 - 1, value )
    else if value > A[ n / 2 ]:
        return binarySearch( A[ ( n / 2 + 1 )...n ], n / 2 - 1, value )
    else:
        return true
```

Алгоритм бинарного поиска

Нотация Big O()

Количество элементов в массиве на каждом вызове:

0-я итерация: n
1-я итерация: n / 2
2-я итерация: n / 4
3-я итерация: n / 8
…
i-я итерация: n / 2^i
…
последняя итерация: 1

Процесс будет продолжаться, и из каждого большого i мы будем получать меньшее количество элементов до тех пор, пока не достигнем единицы. Если мы захотим узнать, на какой итерации это произошло, нам нужно будет просто решить следующее уравнение:

$1 = n / 2^i$

Логарифмическая сложность

Равенство будет истинным только тогда, когда мы достигнем конечного вызова функции binarySearch(), так что узнав из него i, мы будем знать номер последней рекурсивной итерации. Умножив обе части на 2^i, получим:

$$2^i = n$$

Решив его, мы получим:

$$i = \log(n)$$

Этот ответ говорит нам, что количество итераций, необходимых для бинарного поиска, равняется log(n), где n — размер оригинального массива. Таким образом, сложность бинарного поиска равна Θ(log(n)).

Последний результат позволяет сравнивать бинарный поиск с линейным. Несомненно, log(n) намного меньше n, из чего правомерно заключить, что первый намного быстрее второго. Так что имеет смысл хранить массивы в отсортированном виде, если планируется часто искать в них значения.

Улучшение асимптотического времени выполнения программы часто чрезвычайно повышает её производительность. Намного сильнее, чем небольшая «техническая» оптимизация в виде использования более быстрого языка программирования

Вероятности

Set (набор) – коллекция элементов:

- S = “Количество людей, находящихся на Земле”

Subset (выборка) – коллекция элементов, являющаяся частью набора:

- S2 = “Количество людей, находящихся на территории России”

- S2 является выборкой из S

- S3 = “Количество космонавтов на МКС”

- S3 не является выборкой из S

Вероятности

Каждое событие имеет вероятность возникновения

Если вы просыпаетесь в случайное время в сутках, какова вероятность, что вы проснетесь в промежуток между 10 и 11 утра?

1) В сутках 24 часа

   - Набор содержит 24 элемента = “1, 2 , 3 , 4 … 24”

2) Вы просыпаетесь в любой (случайный) час из набора, т.е. делаете выборку

3) Вероятность попадания 10-го часа в выборку равна 1/24

Умножение вероятностей

Если E1 и E2 – независимые друг от друга события, то вероятность их одновременного возникновения равна произведению вероятностей возникновения каждого из них.

1) У вас есть три майки: синяя, зеленая, красная. Вы просыпаетесь в случайное время суток и «вслепую» одеваете одну из маек.
2) Вероятность проснуться между 10 и 11 часами и одеть зеленую майку равна (1/24)*(1/3) = 1/72
3) **Если события зависимые, то умножать вероятность нельзя**

Вероятности

Сложение вероятностей

E1 и E2 – события, с вероятностями соответственно Prob(E1) и Prob(E2).

1) Если события совместные:
   Prob(E1 OR E2) = Prob(E1) + Prob(E2) – Prob(E1 AND E2)
2) Если события несовместные:
   Prob(E1 OR E2) = Prob(E1) + Prob(E2) – Prob(E1 AND E2)
3) Если невозможно подсчитать Prob(E1 AND E2), то:
   Prob(E1 OR E2) ≤ Prob(E1) + Prob(E2)

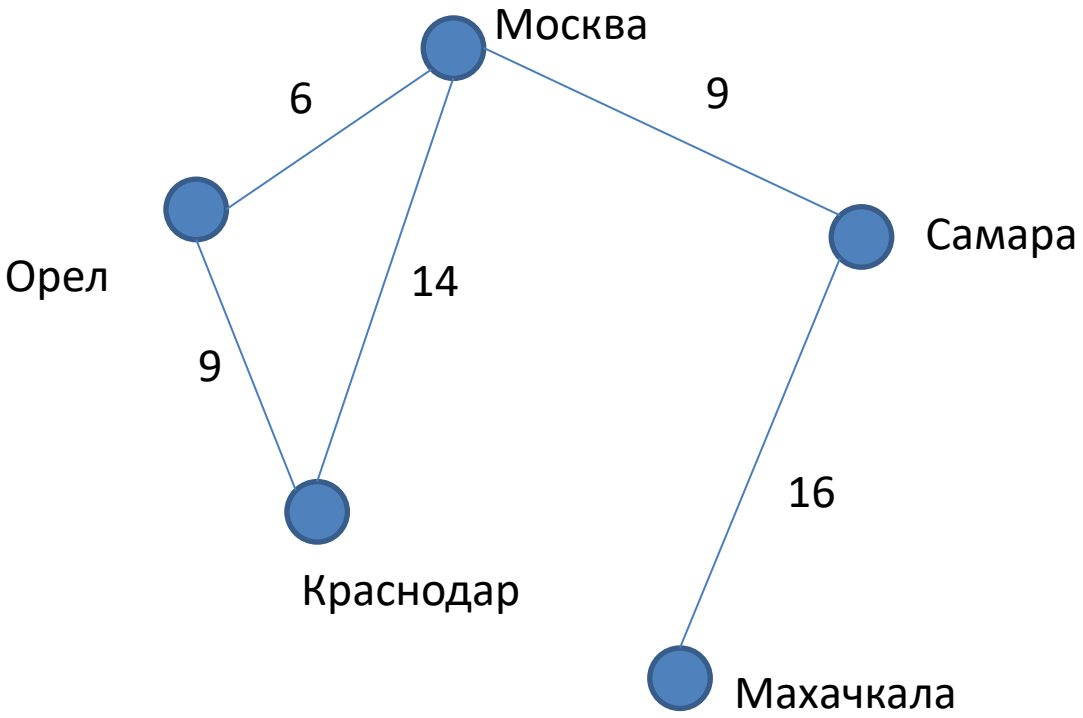
DNS

DNS - Domain Name System (система доменных имен) – коллекция серверов по всему миру.

- На входе – URL (человеко-понятный адрес сервера)
- На выходе – IP-адрес сервера с запрашиваемым содержимым.

IP-адрес может ссылаться на конкретный сервер, или на «зеркало», или на кластер серверов

Графы



- Вершины графа;
- Ребра графа;
- Длина ребра