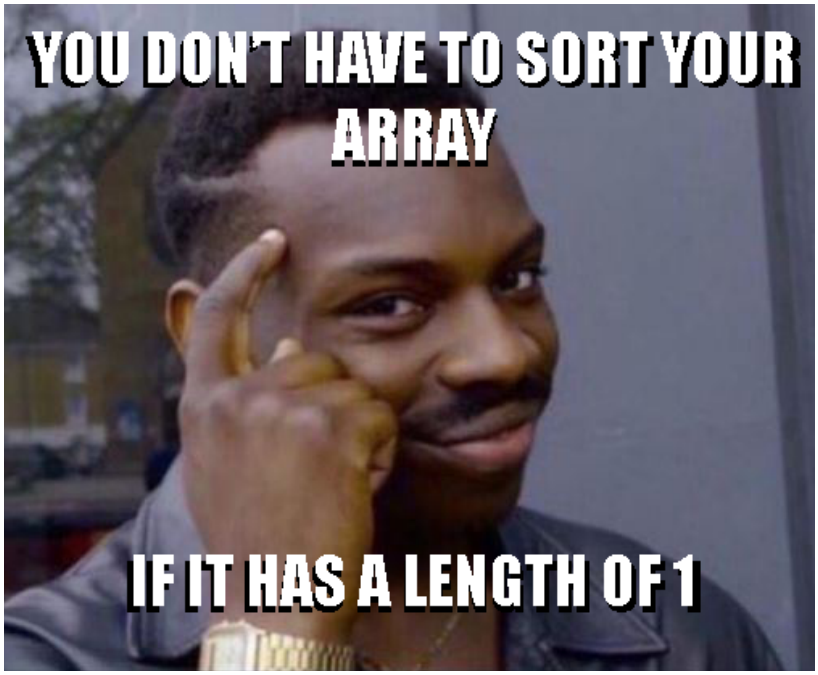
BIG O NOTATION & SORTING ALGORITHMS

DATECS LTD.

GERI ILIEVA, 2017

INTRO



Основно умение за всеки програмист е да може да анализира определен алгоритъм, за да може да оцени поведението му при различни ситуации. Защо е необходимо това? Защото едно нещо може да се изпълни по различни начини. Но средата, в която трябва да работи кода ни, условията, спецификите на системата или ситуацията, могат да дадът предимство на едно решение пред другите. Примерно, ако ще пишем код за ембедед устройство с малко памет, за нас ще е важно да ползваме алгоритми, които ползват минимално количество памет. Ако пък трябва да пишем код, който трябва да се изпълни в кратък период от време, ще трябва да изберем алгоритми, които имат възможно най-доброто време за изпълнение на задачата. Разбира се, когато имаме сложен код не е толкова лесно да измерим точното време, за което ще се изпълни. Поради тази причина, ние програмистите сме си създали начин, с който да опишем приблизителното поведение на една функция, с нарастването на броя данни, които й се подават.

Да разгледаме следната функция:

```
def search (arr, target)
   arr.each_with_index do |x, i|
      if(x == target)
        return i
     end
   end
   return -1
end
```

Горната функция търси `target` в масива `arr`. Ако target не присъства в масива, то горният цикъл ще се изпълни толкова пъти, колкото е голям масивът или `arr.size` пъти. Ако пък `target` присъства в масива, може да бъде във всяка една позиция с еднаква вероятност, което

значи, че средно статистически, цикълът ще се ипълнява `arr.size/2` пъти. Виждате ли, че дали елементът ще присъства или не, има зависимост между големината на масива и броят пъти, в които ще се изпълни цикълът?
Ако удвоим `arr.size` какво ще стане? _____

Какво може да кажем за search функцията?_____

Да разгледаме един друг пример, който ще се базира на горната функция:

```
def areDifferent(arr1, arr2)

     arr1.each do | target |
         if (search(arr2, target) != -1 )
              return false
         end
     end

     return true
end
```

Тази функция сравнява дали `arr1` и `arr2` нямат общи елементи. Ако я анализираме по горния начин ще видим, че цикълът ще се изпълни `arr1.size`. Но всеки път, в който цикълът се изпълни, ще се вика функцията `search`, която пък от горе видяхме зависи от `arr2.size`. Значи при всеки елемент в `arr1`, ще викаме `search`, който вътрешно пък ще се завърти `arr2.size` пъти. Или горната фукнция ще е пропорционална на _____

Както виждате, горният метод на анализиране на код ни позволява да разберем и опишем поведението на алгоритъма при увеличаване на броя входящи данни по начин, по който можем да сравним различни алгоритми. С други думи, вместо да мерим милисекунди, ние разглеждаме как ще реагира кода при увеличаване на входящите данни (как ще се увеличи времето за изпълнение и как ще се промени паметта, която ползва алгоритъмът) и може да го опишем като функция от броя подадени елементи. В първия пример по-горе видяхме, че кодът е пропорционален на `arr1.size`. Ако удвоим размера, ще удвоим и времето за изпълнение. Значи, `search` нараства линейно (linear rate). Във втория случай, ако удвоим броя на входящите елементи, времето ще нарастне четворно. Това е квадратна зависимост.

За по-голямо улеснение компютърните специализсти използват нотацията **O(f(n))**, за да опишат горното поведение. `search` ще има **O(n)**, а `areDifferent` - **O(n^2)**. Това се нарича **“Big-O notation”.** Когато имате цикли най-лесно е да запомните, че един цикъл ще е O(n), цикъл в цикъл - O(n^2), цикъл в цикъл в цикъл - O(n^3) и така нататък. Разбира се, това е валидно, ако цикълът минава през всички елементи подадени й. Например в този случай:

```
for (i=1; i<x.size; i*=2)...
```

кодът няма да е линеен, а всъщност ще е логаритмичен, защото ще се изпълни само за стойности 1, 2, 4, 8, 16, … или **$\log_2$(x.size)**.

FORMAL DEFINITION

Нека да разгледаме следната програма:

```
for i in 0..n
   for j in 0..n
       #simple statement
   end
end

for k in 0..n
   #simple statement 1
   #simple statement 2
   #simple statement 3
   #simple statement 4
   #simple statement 5
end

#simple statement 6
#simple statement 7
. . .
#simple statement 30
```

Да приемем, че всеки “simple statement” отнема единица време за изпълнение. Тогава nested-for циклите ще се изпълнят n2, след това ще имаме n*5, и още 25 отделни инструкции. Или времето за изпълнение ще е:

$T(n) = n^2 + 5n + 25$

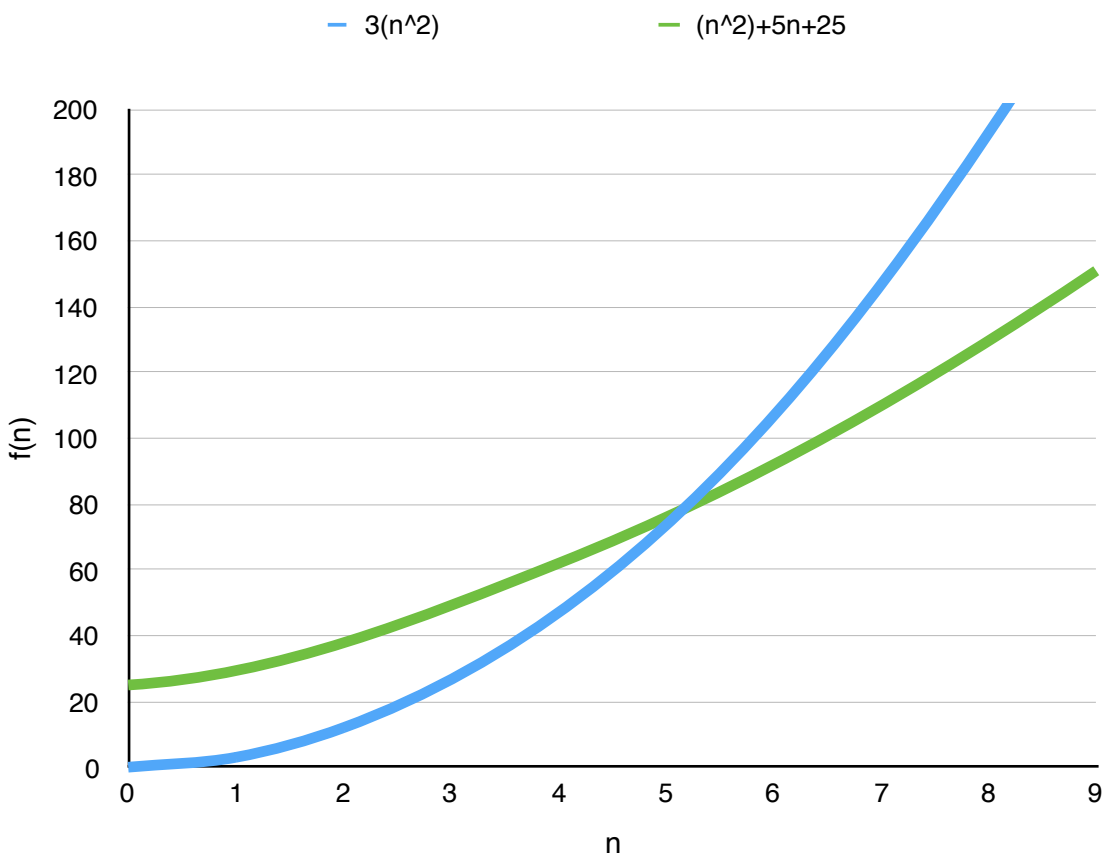
Ясно е, че с увеличение на n, n^2 ще доминира в уравнението. Спрямо T(n), формалното описание на big-O нотацията е:

`T(n) = O(f(n))`, когато съществуват две позитивни и по-големи от 0 константи n_0 и `c` и функция `f(n)`, такава че за всяко $n > n_0$, $cf(n) = T(n)$.

С други думи, когато `n` стане достатъчно голямо има константа `c`, за която времето за изпълнение на алгортиъма ще е винаги по-малка или равна на `cf(n)`. С други думи, `cf(n)` е горната граница на времето за изпълнение на функцията. Времето няма да никога по-голямо (по-лошо) от `cf(n)`, а може да е по-добро.

В горния пример е очевидно, че n^2 ще доминира и ще има $O(n^2)$. И реално, при $n_0 = 5$ и $c = 3$:

$3n^2 > n^2 + 5n + 25$ за всички $n > 5$.



Най-често срещаните O(f(n)), са обобщени по-долу:

Big-O	Име
O(1)	константно (constant)
O(logn)	логаритмично (logarithmic)
O(n)	линейно (linear)
O(n logn)	логаритмично-линейно (log-linear)
$O(n^2)$	квадратно (quadratic)
$O(n^3)$	кубично (cubic)
$O(2^n)$	експоненциално (exponential)
O(n!)	факториел (factorial)

O(1) описва поведение, което ще е константно, независимо от големината на подадените данни. Използваните по-горе simple statement представляват O(1). Всеки краен брой такива стъпки, не зависимо колко, се счита за O(1).

Може би се досещате, че програми с поведение описано с $O(2^n)$ и O(n!) са невъзможни за изпълнение при голям брой входни данни дори от модерните и бързи компютри. Например, ако имаме експоненциален алгоритъм, който ще върви 1 часа за 100 входни елемента, добавянето на само още един елемнет ще добави още един час към изпълнението. Ако добавим 5 - времето ще се удължи с 32 часа, а при добавени 14 нови елемента - 16 384 часа или почти 2 години!

Може би си задавате въпроса защо някой би използвал такива алгоритми. Както казах в началото, много зависи от ситуацията и спецификата на това, което кода ни трябва да изпълнява. В криптографията, където се правят алгоритми, които да закодират данни, така че да не могат да се прихванати от нежелани страни, такъв вид алгоритми са в основата им. Например, някои криптографски алгоритми могат да бъдат пречупени в $O(2^n)$ време, където n е броят на битове в криптографския ключ. Ключ с дължина 40-бита ще се счита за лесен за разбиване от модерните компютри, но 60-битов не е, защото ще отнеме 10 18 пъти повече време, за да бъде разбит ключа.

Задачи

1. Определете колко пъти ще се принтира във всеки от долните фрагменти. Определете дали алгоритъмът е O(n) или $O(n^2)$.

```
A.  for i in 0..n
       for j in 0..n
          puts i.to_s + “  “ + j.to_s
       end
    end

B.  for i in 0..n
       for j in 0..2
          puts i.to_s + “  “ + j.to_s
       end
    end

C.  for i in 1..n
       for j in 0..i
          if(j % i == 0)
              puts i.to_s + “  “ + j.to_s
           end
       end
    end
```

SORTING ALGORITHMS

Сортирането е процес, при който пренареждаме данните/елементите в масив или лист, така че да са в нарастващ или низходящ ред. Тъй като сортирането се ползва изключително често в програмирането, компютърните специалисти са отделили много време в разработването на различни алгоритми, с които да изпълнят това. Въпреки че много от модерните езици като Ruby Java имат вградени функции за сортиране, е наложително за всеки млад програмист, за да разбере как работят и да може да ги пресъздаде, ако му се наложи в бъдеще. Сортиращите алгоритми също са много полезни и в това нагледно да се види big-O анализ и как по него се сравняват алгоритмите.

В следващите страници ще разгледаме следните алгоритми:

1. **Selection Sort**
2. **Bubble Sort**
3. **Insertion Sort**
4. **Shell Sort**
5. **Merge Sort**
6. **Quicksort**

SELECTION SORT

Това е един от най-лесните за разбиране алгоритми, който сортира масив като прави няколко обхождания, селектирайки следващият най-малък елемент в масива и слагайки го на правилното място в масива. Алгоритъмът изглежда така:

1. Обхождаме масива от начало до край, като при всеки цикъл:

    1.1. Задаваме min да е елементът, който разглеждаме в момента

    1.2. Сравняваме го с всеки следващ елемент от масива, докато не намерим минималният

    1.3. Разменяме сегашния елемент с минималния

Да приемем, че имаме следния масив:

45	20	67	13	27	63

Започваме да обхождаме масива от началото. Първият ни елемент е 45. Сравняваме 45 с всички останали елементи в масива, за да намерим най-малкият елемент. Той е 13:

45	20	67	13	27	63

Разменяме ги:

13	20	67	**45**	27	63

Продължаваме със следващото обхождане на масива, но вече от втория елемент, 20:

13	20	67	45	27	63

Обхождайки, виждаме, че 20 е най-малкото число в масива, затова го оставяме на мястото му. и Продължаваме със следващото обхождане от третия елемент

13	20	67	45	27	63

При обхождането, намираме, че 27 е най-малкият елемент, разменяме го със 67 и получаваме:

13	20	**27**	45	**67**	63

И така докато не обходим целия масив. Накрая, всичко ще си е по местата:

13	20	27	45	63	67

Тъй като обхождаме целия масив един път, но за всеки елемент от масива разглеждаме още веднъж част от масива, алгоритъмът е квадратен сортиращ алгоритъм или с $O(n^2)$.

Задачи

1. Покажете какво се случва с масива при всяко завъртане на selection sort алгоритъма:

   40 35 80 75 60 90 70 75 50 22

BUBBLE SORT

Bubble sort сравнява два съседни елемента в масива и ги разменя, ако са в неправилен ред. Идеята е, че така най-малката стойност “bubbles up” до първата позиция на масива, докато най-голямата потъва до края на масива. Алгоритъмът е:

За всяка двйка съседни елементи
    ако стойностите не са под ред
        разменяме стойностите
докато масивът не е сортиран

Да разгледаме стария пример:

45	20	67	13	27	63

Започваме с 45 и 20. Те не са подредени правилно, затова ги разменяме и получаваме:

20	45	67	13	27	63

Продължаваме с 45 и 67, те са наред и ги пропускаме:

20	45	67	13	27	63

След това е ред на 67 и 13. 67 > 13 и затова ги разменяме:

20	45	13	67	27	63

След края на първия обход на масива ще имаме:

20	45	13	27	63	67

Както виждате масивът не е сортиран, затова трябва да обходим масива наново и да разменяме стойности. При второто обхождане 20 и 45 са наред, но 45 и 13 не са. Така ще получим:

20	13	45	27	63	67

Накрая на второто обхождане, масивът ни ще изглежда така:

20	13	27	45	63	67

Налага се още едно обхождане, защото масивът все още не е сортиран. След третото обхождане вече масивът ще е сортиран:

13	20	27	45	63	67

Тъй като броят на сравнения и обхождания в bubble sort зависи много данните в масива, той има много добър performance в някои случай и много слаб при други. Ако масивът е вече сортиран, bubble sort ще има run time O(n), защото ще направи само едно обхождане на масива. Но в най-лошия случай, ще са необходими n на брой обхождания на масива, което ще доведе до $O(n^2)$. В програмирането рядко се случва best case случая, затова bubble sort също се счита за квадратен сортиращ алгоритъм с $O(n^2)$.

Задачи

1. Покажете какво се случва с масива при всяко завъртане на bubble sort алгоритъма:

   40 35 80 75 60 90 70 75 50 22

INSERTION SORT

Insertion sort се базира на начина, по който някои хора си нареждат картите докато играят на карти. Играчът обикновено си държи картите подредени, и когато се появи нова карта я слага на правилното й място. Алгоритъмът е:

За всеки елемент (от втория до последния) в масива:
Сложете елемента на позицията, на която принадлежи

Ето как изглежда това нагледно:

45	20	67	13	27	63

Започваме от втория елемент - 20. 20 е по-малко от 45 и затова ще трябва да го сложим преди 45. Забележете, че за разлика от bubble sort няма да разменяме местата, а ще създадем място, така че да добавим елемента там.

20	45	67	13	27	63

След това имаме 67. Трябва да сравним 67 с 20 и 45 и ще видим, че 67 е след тях. Масивът става:

20	45	67	13	27	63

След това е ред на 13. 13 е най-малкото число, затова трябва да му направим място най-отпред на сортирания ни съб-масив и да го добавим там:

13	20	45	67	27	63

И така нататък докато не сортираме масива:

13	20	27	45	63	67

Тъй като обхождаме масивът веднъж, но за всеки елемент трябва да го сравним с вече подредения съб-масив, не е чудно, че и insertion sort-а е квадратен сортиращ алгоритъм.

Задачи

1. Покажете какво се случва с масива при всяко завъртане на insertion sort алгоритъма:

   40 35 80 75 60 90 70 75 50 22

ALGORITHM	NUM COMPARISONS		NUM EXCHANGES	
	Best	**Worst**	**Best**	**Worst**
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$