Yang Perlu Anda Ketahui Tentang Notasi Big O [Contoh Python]

Notasi Big O adalah metode untuk menentukan seberapa cepat suatu algoritma. Dengan menggunakan notasi Big O, kita dapat mengetahui apakah algoritme kita cepat atau lambat. Pengetahuan ini memungkinkan kami merancang algoritme yang lebih baik.

Bagaimana Kita Mengukur Berapa Lama Algoritma Berjalan?

```
$ python3 -m timeit '[print(x) for x in range(100)]'
100 loops, best of 3: 11.1 msec per loop
$ python3 -m timeit '[print(x) for x in range(10)]'
1000 loops, best of 3: 1.09 msec per loop
# We can see that the time per loop changes depending on the input!
```

Bagaimana jika 100000000 perulangan? Wah pasti akan membutuhkan waktu lama.

Apa ukuran input "sempurna" untuk mendapatkan ukuran "sempurna" dari berapa lama waktu yang dibutuhkan algoritme?

Hal-hal lain yang perlu kita pertimbangkan:

- Ada kecepatan prosesor yang berbeda.
- Bahasa penting. Perakitan lebih cepat dari Awal; bagaimana kita menganggap ini?

Untuk alasan ini, kami menggunakan notasi Big O (diucapkan Big Oh).

Apa itu Notasi Big O?
Big O adalah notasi formal yang menggambarkan perilaku suatu fungsi ketika argumen cenderung ke input maksimum.

Big O mudah dibaca setelah kita mempelajari tabel ini:

The Big O Notation's Order Of Growth				
Constant	**Logarithm**	**Linear**	**Polynomial**	**Exponential**
$O(1)$	$O(log\ n)$	$O(n)$	$O(n^2)$, $O(n^3)$, $O(n^x)$	$O(2^n)$

Di mana semakin jauh mereka berada, semakin lama waktu yang dibutuhkan. nadalah ukuran masukan. Notasi Big O menggunakan fungsi-fungsi ini untuk menggambarkan efisiensi algoritma.

Notasi asimtotik (pengukuran waktu) lainnya adalah:

Asymptotic Notation		
Big Omega (lower bound)	**Big Theta (average bound)**	**Big O (max bound)**
$\omega(n)$	$\theta(n)$	$O(n)$

Secara informal ini adalah:

- Omega Besar (kasus terbaik)
- Big Theta (kasing rata-rata)
- Big O (kasus terburuk)

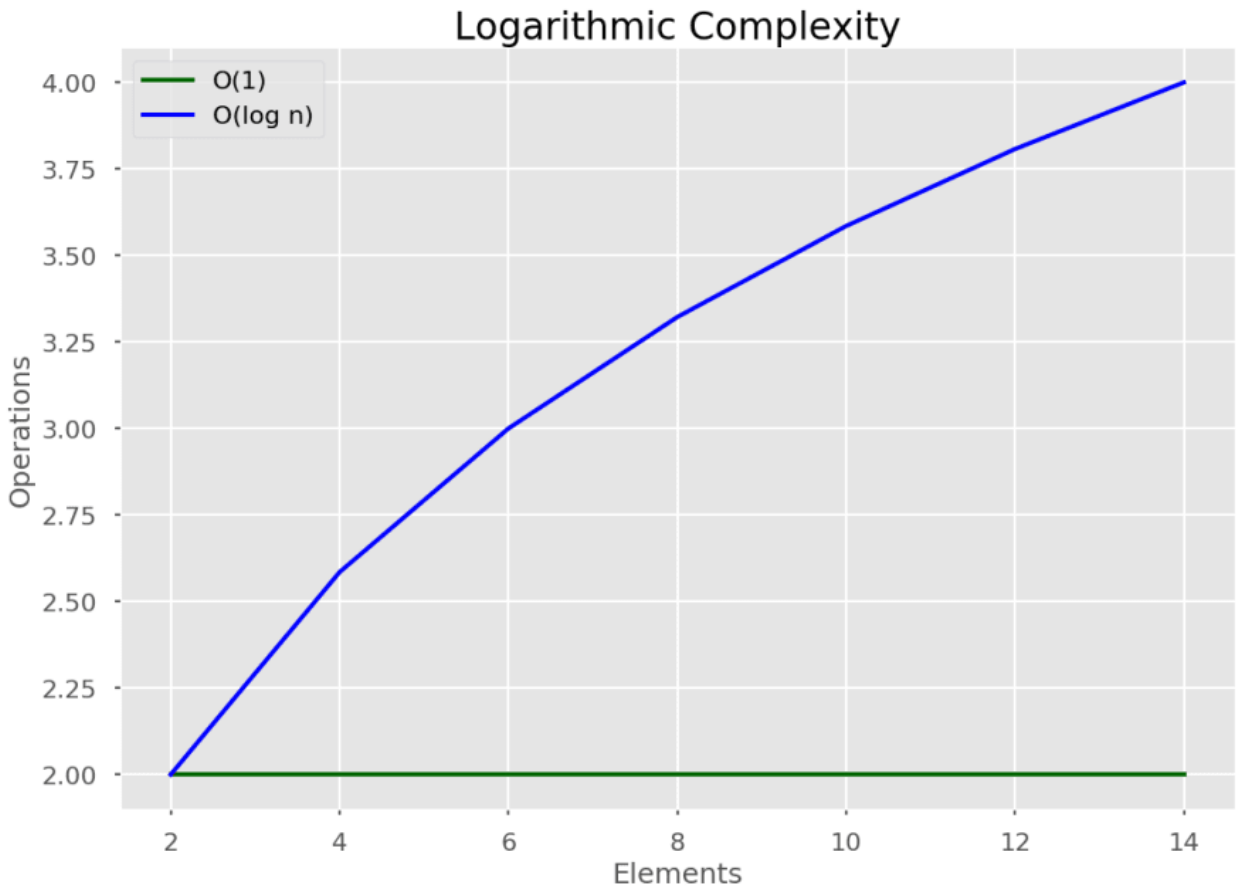
Waktu konstan adalah yang tercepat dari semua kompleksitas waktu Big O. Definisi formal waktu konstan adalah:

Contohnya adalah:

```
def OddOrEven(n):
    return "Even" if n % 2 else "Odd"
```

Koder tersebut memiliki arti return “Even” jika 2 habis dibagi 2 kalo tidak return “Odd”.

Waktu Logaritmik



Log kurang dari O(1) dengan 1 elemen, tetapi di Big O kami tidak peduli dengan ukuran elemen

Inilah penjelasan singkat tentang apa itu logaritma.

Log_{3} {9}

Apa yang ditanyakan di sini adalah "3 pangkat apa yang memberi kita 9?" Ini adalah 3 pangkat 2 memberi kita 9, jadi seluruh ekspresinya terlihat seperti:

Log_{3} {9} = 2

Algoritme logaritmik **membagi dua** daftar setiap kali dijalankan.

Mari kita lihat pencarian biner. Diberikan daftar diurutkan di bawah ini:

```
a = [1, 2, 3, 4, 5, 6 , 7, 8, 9, 10]
```

Kami ingin menemukan angka "2".

Kami menerapkan Pencarian Biner sebagai:

```
def binarySearch(alist, item):
   first = 0
   last = len(alist)-1
   found = False

   while first <= last and not found:
      midpoint = (first + last)//2
      if alist[midpoint] == item:
         found = True
      else:
         if item < alist[midpoint]:
         last = midpoint-1
         else:
            first = midpoint+1

   return found
```

ket:

- Pergi ke tengah daftar
- Periksa untuk melihat apakah elemen itu adalah jawabannya
- Jika tidak, periksa apakah elemen tersebut lebih dari item yang ingin kita temukan
- Jika ya, abaikan sisi kanan (semua angka yang lebih tinggi dari titik tengah) daftar dan pilih titik tengah baru.
- Mulai lagi, dengan menemukan titik tengah dalam daftar baru.

Waktu Linier

Algoritme waktu linier berarti bahwa setiap elemen dari input dikunjungi tepat satu kali, O(n) kali. Karena ukuran masukan, N, meningkatkan skala waktu berjalan algoritme kami persis dengan ukuran masukan.

Item terbesar dari array yang tidak disortir

```
a = [2, 16, 7, 9, 8, 23, 12]
```

Bagaimana cara mengetahui nilai terbesar?

```
a = [2, 16, 7, 9, 8, 23, 12]
max_item = a[0]
for item in a:
   if item > max_item:
      max_item = item
```

Program tersebut akan mengecek max_item yang dimulai dari index pertama sampai index terakhir, jika ada nilai yang lebih besar maka max_item akan terganti (terassigment) ke nilai terebut

Waktu Polinomial

Waktu polinomial adalah fungsi polinomial dari input. Fungsi polinomial terlihat seperti n^2 atau n^3 dan seterusnya.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in a:
   for x in a:
      print("x")
```

kode di atas akan menghasilkan a^2 , kenapa? Karena loop dalam loop.

Kita contohkan dengan bubble sort:

```
def bubbleSort(arr):
   n = len(arr)

   # Traverse through all array elements
   for i in range(n):

      # Last i elements are already in place
      for j in range(0, n-i-1):

         # traverse the array from 0 to n-i-1
         # Swap if the element found is greater
         # than the next element
         if arr[j] > arr[j+1] :
            arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)
```

Bubble sort akan mengecek dari index pertama dan index berikutnya jika posisi index sebelumnya lebih besar maka posisi akan ditukar sampai index terakhir pengecekan dan akan diloop kembali sampai tidak ada index yang bisa ditukar (sudah terurut semua).

berapa banyak dari kata-kata itu yang muncul dalam Kamus Bahasa Inggris?

```
dictionary = ["a", "an"] # imagine if this was the dictionary
sentence = "hello uu988j my nadjjrjejas is brandon nanndwifjasj banana".split(" ")

counter = 0
for word in sentence:
   for item in dictionary:
      if word == item:
         counter = counter + 1
```

variable counter akan bertambah jika item(kata/char) yang akan dicari terdapat/muncul dalam sebuah list

Waktu Eksponensial
Waktu eksponensial adalah $2^{n,}$ di mana 2 bergantung pada permutasi yang terlibat.

Algoritma ini adalah yang paling lambat dari semuanya. Anda melihat bagaimana profesor saya bereaksi terhadap algoritma polinomial. Dia melompat-lompat dalam kemurkaan pada algoritma eksponensial!

Contohnya adalah katakanlah kita memiliki kata sandi yang hanya terdiri dari angka (jadi itu 10 angka, 0 hingga 9). kami ingin memecahkan kata sandi yang memiliki panjang n, jadi untuk memaksa melalui setiap kombinasi kami akan memiliki: 10^{n}

Menyederhanakan notasi Big O

Kompleksitas waktu jarang semudah menghitung berapa banyak for loop yang kita miliki. Bagaimana jika algoritme kita terlihat seperti $O(n + n^{2)?}$ Kami dapat menyederhanakan algoritme kami menggunakan aturan sederhana ini:

Jatuhkan konstanta

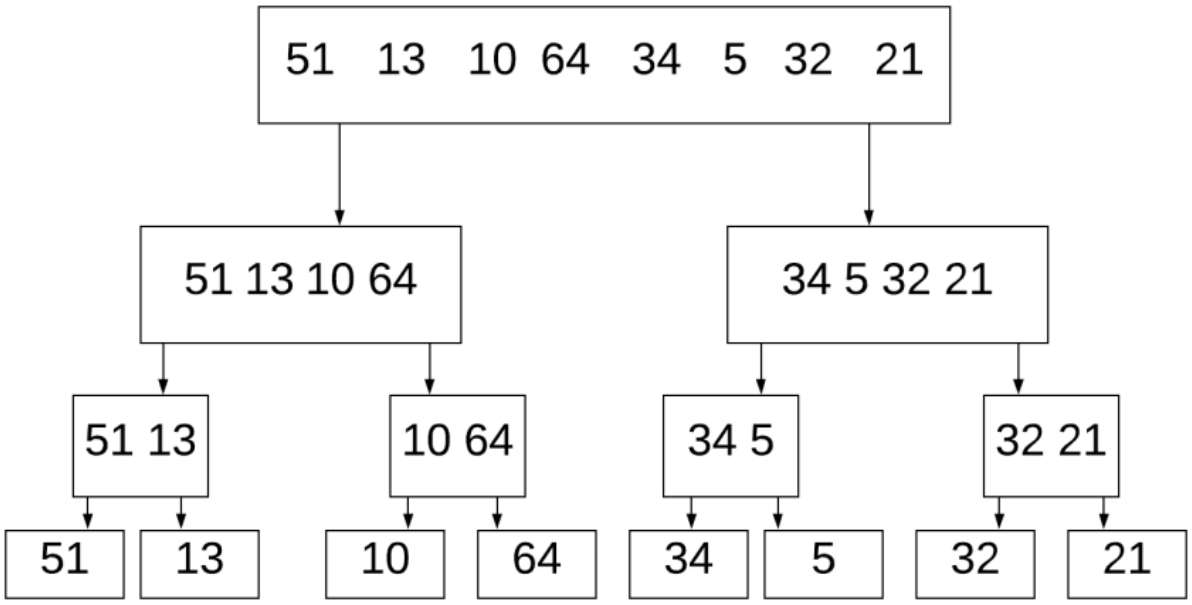
Jika kita memiliki algoritme yang dideskripsikan sebagai O(2n), kita membuang 2 sehingga menjadi O(n).

Jatuhkan istilah non-dominan

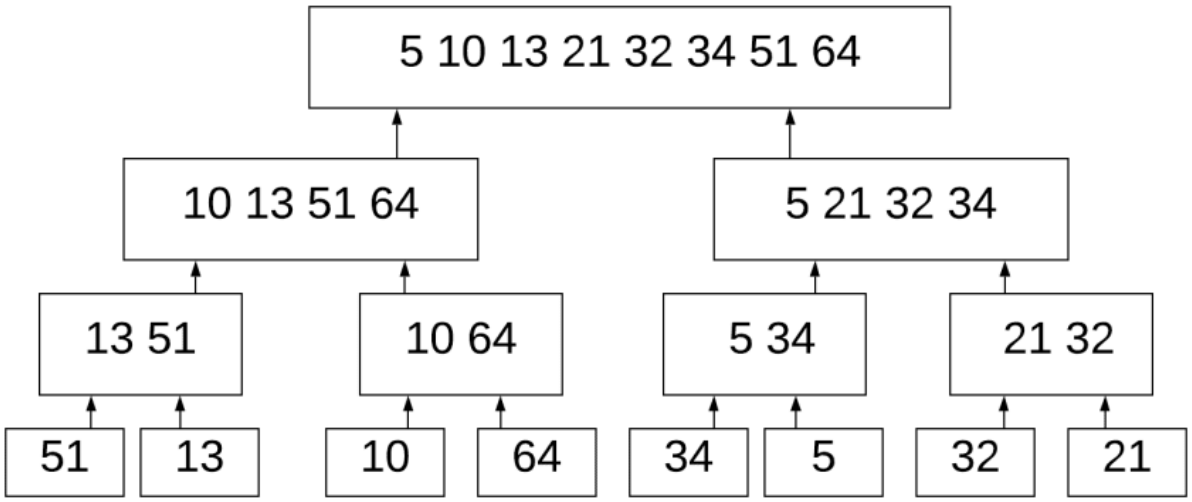
$O(n^2 + n)$ menjadi $O(n^2)$. Hanya simpan yang lebih besar di Big O.

Jika kita memiliki penjumlahan seperti $O(b^2 + a)$, kita juga tidak bisa menghilangkannya tanpa mengetahui apa itu b dan a.

Mergesort bekerja dengan memecah daftar angka menjadi nomor individual:



Dan kemudian menyortir setiap daftar, sebelum menggabungkannya:




```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
```

```
    mergeSort(lefthalf)
    mergeSort(righthalf)

    i=0
    j=0
    k=0
    while i < len(lefthalf) and j < len(righthalf):
      if lefthalf[i] <= righthalf[j]:
        alist[k]=lefthalf[i]
        i=i+1
      else:
        alist[k]=righthalf[j]
        j=j+1
      k=k+1

    while i < len(lefthalf):
      alist[k]=lefthalf[i]
      i=i+1
      k=k+1

    while j < len(righthalf):
      alist[k]=righthalf[j]
      j=j+1
      k=k+1
  print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Oh (n!)

Cara Menghitung Notasi O Besar untuk Algoritma Kita Sendiri dengan Contoh

"Apa input terburuk mutlak untuk program saya?"

Ambil, misalnya, algoritma pencarian berurutan.

```
def search(listInput, toFind):
  for counter, item in enumerate(listInput):
    if toFind == item:
      return (counter, item)
  return "did not find the item!"
```

Masukan terbaik adalah:

```
search(["apples"], "apples")
```

Tapi input terburuknya adalah jika item tersebut berada di akhir daftar panjang.

```
search(["apples", "oranges", "The soundtrack from the 2006 film Nacho Libre", "Shrek"], "Shrek")
```

Skenario terburuknya adalah O(n), karena kita harus melewati setiap item dalam daftar untuk menemukannya.

Bagaimana jika algoritma pencarian kita adalah pencarian biner? Kami mengetahui bahwa pencarian biner membagi daftar menjadi setengah setiap saat. Ini terdengar seperti log n!

Bagaimana jika pencarian biner kita mencari objek, lalu mencari objek serupa lainnya?

```
# here we want to find the film shrek, find its IMDB rating and find other films with that IMDB rating. We are using binary search, then sequential search
toFind = {title: "Shrek", IMDBrating: None}
ret = search(toFind)
ret = search(ret['IMDBrating'])
```

Kita menemukan Shrek dengan skor IMDB 7,8. Tapi kami hanya mengurutkan berdasarkan judul, bukan rating IMDB. Kami harus menggunakan pencarian berurutan untuk menemukan semua film lain dengan peringkat yang sama.

Satu hal lagi, kita tidak selalu berurusan dengan n. Contoh algoritma di bawah ini:

```
x = [1, 2, 3, 4, 5]
y = [2, 6]
y = iter(y)
counter = 0
total = 0.0
while counter != len(x):
   # cycles through the y list.
   # multiplies 2 by 1, then 6 by 2. Then 2 by 3.
   total = total + x[counter] * next(y)
   counter += 1
print(total)
```

Kita memiliki 2 input, x dan y. Notasi kita adalah O(x + y). Terkadang kita tidak bisa membuat notasi kita lebih kecil tanpa mengetahui lebih banyak tentang datanya.

Lembar Curang Notasi O Besar

Saya membuat infografis kecil ini untuk Anda! "Tambahkan +1 untuk setiap perulangan for bersarang" bergantung pada perulangan for, seperti yang kita lihat sebelumnya. Tapi menjelaskannya lagi akan menghabiskan terlalu banyak ruang 😅

Cara Menghitung Notasi Besar O dari Suatu Fungsi (Matematika Diskrit)

Yahhh tidak ada di artikelnyaa….

Big O menunjukkan berapa lama waktu yang dibutuhkan suatu algoritme, tetapi terkadang kami peduli dengan berapa banyak memori (kompleksitas ruang) yang dibutuhkan algoritme juga. Jika Anda pernah mengalami kebuntuan, kembalilah ke halaman ini dan lihat infografisnya!