

## Bucket Sort



Niels Bijl  
HPP

## Inhoud

Tijdscomplexiteit.....	4
BucketSort.....	4
BucketSortRecursive .....	7
Ruimtecomplexiteit.....	10
BucketSort.....	10
BucketSortRecursive .....	10
BucketSort VS BucketSortRecursive.....	11
Tijdscomplexiteit.....	11
Ruimtecomplexiteit.....	11
Tijdsanalyse.....	11
Conclusie .....	12
Extra's.....	12
___ Wat nam ik wel en niet mee in mijn analyse? .....	13
___ Geschikt voor het sorteren van getallen met cijfers achter de komma .....	13
___ Unit tests.....	14
<u>Bronnen</u> .....	<u>15</u>

Time complexity Python functies

Reverse():  $O(n)$

Remove():  $O(n)$

int():  $O(i)$

str():  $O(i)$

max():  $O(n)$

len():  $O(n)$

Variable declaration  $O(1)$

for loop:  $O(n)$

if & else:  $O(1)$

append():  $O(1)$

+/-:  $O(1)$

Zie bronnen

## Tijdscomplexiteit

### BucketSort

BucketSortFunctional time complexity:

```
def bucketSortFunctional(data: list) -> list:
    for i in range(1, int(len(str(max(data))))+1):
        buckets = [[], [], [], [], [], [], [], [], [], [], []]
        for item in data[:]: # distribution pass
            if int(len(str(item))) >= i:
                char = str(item)[-i]
                index = int(char)
                buckets[index].append(item)
                data.remove(item)
        for bucket in buckets: # gathering pass
            for item in bucket:
                data.append(item)
    return data
```

**Totaal:**  
 $2i + n + i(1 + n(4i + n + 7) + n) + 3$

BucketSort time complexity:

```
def bucketSort(data):
    negative = []
    positive = []
    for item in data:
        if item >= 0:
            positive.append(item)
        else:
            negative.append(item)
    if negative:
        negative = [i * -1 for i in negative]
        negative = bucketSortFunctional(negative)
        negative = [i * -1 for i in negative]
        negative = list(reversed(negative))
    if positive:
        positive = bucketSortFunctional(positive)
    return negative + positive
```

**Totaal:**  
 $5n + 12$

De **i** = de lengte van het getal in de lijst met de grootste lengte.

De **n** = de lengte van de lijst.

De zo precies mogelijke tijdscomplexiteit van de bucketSort =

$$2i + n + i(1 + n(4i + n + 7) + n) + 3$$

De Big O waarde van de bucketSort =

$$2i + n + i(1 + n(4i + n + 7) + n) + 3$$

$$2i + n + i(n(4i + n) + n)$$

$$i(n(4i + n) + n)$$

$$i(n(i + n))$$

$$i(ni + n^2)$$

$$n * i^2 + i * n^2$$

$$O(n * i^2) \quad \text{OF} \quad O(i * n^2)$$

**Big O =**

$$O(n * i^2)$$

$$O(i * n^2)$$

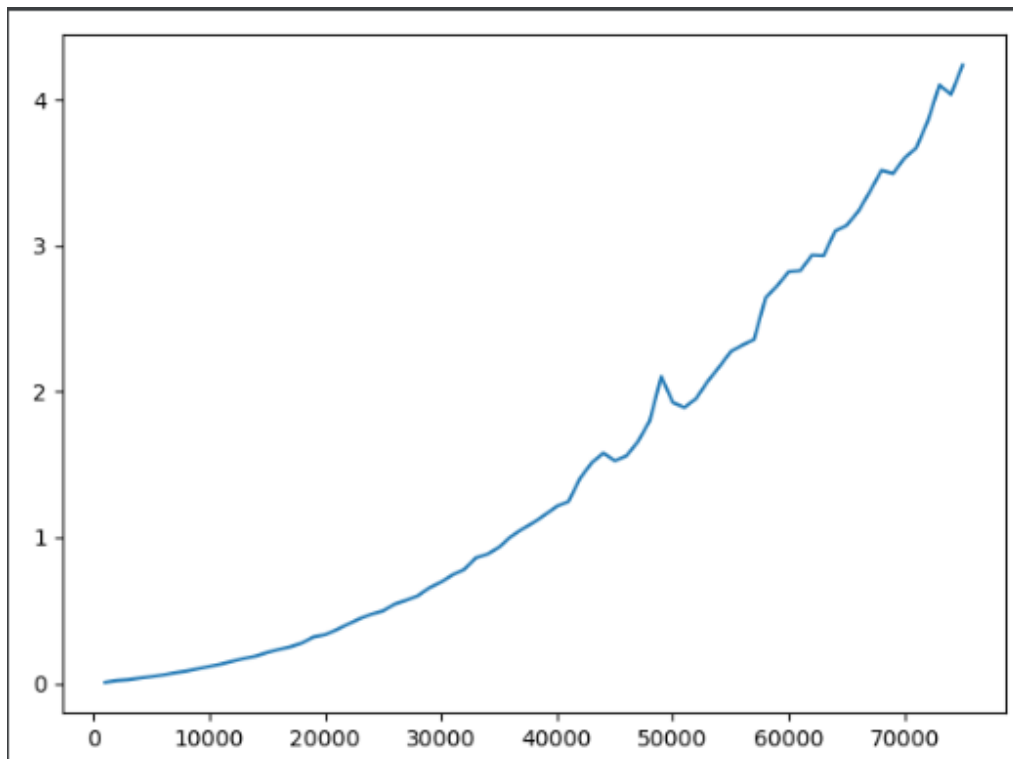
i kan in theorie oneindig zijn, in de praktijk (python) is dit  $2^{32}$  aka 4.294.967.296), dus i is maximaal 10.

Omdat in theorie i en n beide oneindig kunnen zijn maakt het niet uit welke van de 2 formules je kiest ze kunnen beide.

In de praktijk kiezen we voor  $O(n^2)$

## De tijds analyse:

```
Het sorteren van de lijst met random 1.000 items duurde: 0.004981279373168945 seconden.  
Het sorteren van de lijst met random 10.000 items duurde: 0.09374570846557617 seconden.  
Het sorteren van de lijst met random 30.000 items duurde: 0.5814111232757568 seconden.  
Het sorteren van de lijst met gesorteerde 30.000 items duurde: 0.5754613876342773 seconden.  
Het sorteren van de lijst met omgekeerde gesorteerde 30.000 items duurde: 0.6457047462463379 seconden.
```



N vs tijd in secondes. (x-as = n, y-as = tijd)

Dit is met een i van 7

In de grafiek is te zien dat hier duidelijk spraken is van een kwadratische functie.

## BucketSortRecursive

BucketSortFunctionalRecursive time complexity:

```
def bucketSortFunctionalRecursive(data: list, maxDigitPosition: int, digitPosition: int = -1) -> list:
    buckets = [[] for _ in range(10)] 1
    for item in data[:]: # distribution pass n
        if int(len(str(item))) >= abs(digitPosition): 1+i+1+i+1
            index = int(str(item)[digitPosition]) 2 + 2i
            buckets[index].append(item) 2
            data.remove(item) n
    for bucket in buckets: # gathering pass n
        for item in bucket:
            data.append(item)
    if abs(digitPosition) == maxDigitPosition 2
        return data
    return bucketSortFunctionalRecursive(data, maxDigitPosition, digitPosition - 1) 2
```

**Totaal:**  
 $i(1 + n(4i + n + 6)) + n + 4$

BucketSortRecursive time complexity:

```
def bucketSortRecursive(data):
    negative = [] 2
    positive = []
    for item in data: n
        if item >= 0: 2
            positive.append(item)
        else:
            negative.append(item)
    if negative: 1
        negative = [i * -1 for i in negative] n + 1
        negative = bucketSortFunctionalRecursive(negative, int(len(str(max(negative))))) 1+1+i+n+i+n
        negative = [i * -1 for i in negative] n + 1
        negative = list(reversed(negative)) 1 + 1 + n
    if positive: 1
        positive = bucketSortFunctionalRecursive(positive, int(len(str(max(positive))))) 0
    return negative + positive 2
```

**Totaal:**  
 $7n + 2i + 12$

**Worst case heeft geen positieve getallen**

$2i + 5n + 7$

De **i** = de lengte van het getal in de lijst met de grootste lengte.

De **n** = de lengte van de lijst.

De zo precies mogelijke tijdscomplexiteit van de bucketSort =

$$7n + 2i + 12 + i(1 + n(4i + n + 6) + n + 4)$$

De Big O waarde van de bucketSort =

$$7n + 2i + 12 + i(1 + n(4i + n + 6) + n + 4)$$

$$7n + 2i + i(n(4i + n) + n)$$

$$i(n(i + n))$$

$$i(ni + n^2)$$

$$n * i^2 + i * n^2$$

$$O(n * i^2) \quad \text{OF} \quad O(i * n^2)$$

**Big O =**

$$O(n * i^2)$$

$$O(i * n^2)$$

i kan in theorie oneindig zijn, in de praktijk (python) is dit  $2^{32}$  aka 4.294.967.296), dus i is maximaal 10.

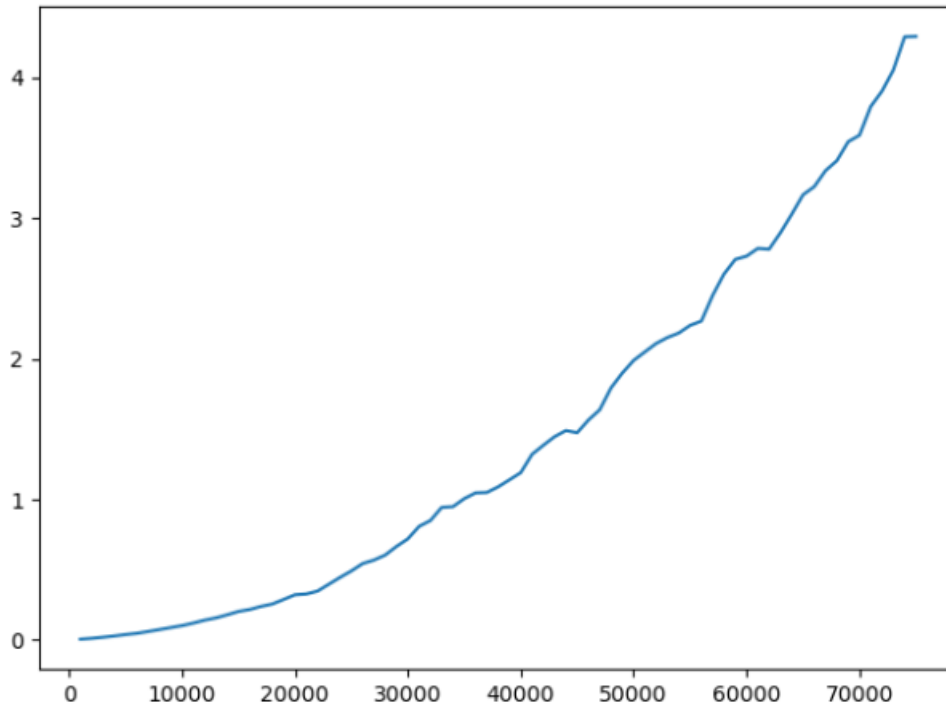
Omdat in theorie i en n beide oneindig kunnen zijn maakt het niet uit welke van de 2 formules je kiest ze kunnen beide.

In de praktijk kiezen we voor  $O(n^2)$



## De tijds analyse:

```
Het recursive sorteren van de lijst met random 1.000 items duurde: 0.005023479461669922 seconden.  
Het recursive sorteren van de lijst met random 10.000 items duurde: 0.09673428535461426 seconden.  
Het recursive sorteren van de lijst met random 30.000 items duurde: 0.6063477993011475 seconden.  
Het recursive sorteren van de lijst met gesorteerde 30.000 items duurde: 0.5972692966461182 seconden.  
Het recursive sorteren van de lijst met omgekeerde gesorteerde 30.000 items duurde: 0.6268672943115234 seconden.
```



N vs tijd in secondes. (x-as = n, y-as = tijd)

Dit is met een i van 7

In de grafiek is te zien dat hier duidelijk spraken is van een kwadratische functie.

## Ruimtecomplexiteit

### *BucketSort*

Hij onthoudt in de worst case:

- Data (n)
- Positieve getallen (0)
- Negative getallen (n)
- Buckets (n)
- De lengte van de langste int (i)

### *BucketSortRecursive*

Bij de recursieve functie onthoudt die hetzelfde. Alleen dan op een recursieve manier.

Hoeveel ruimte is nou N en I?

Dit verschilt per systeem. N is in principe oneindig, maar python kan natuurlijk geen oneindig op slaan. Omdat het met integers werkt is n maximaal: sys.maxsize, de maxsize verschilt per systeem. En I maximaal 10.

De ruimte complexiteit van bucketSort en bucketSortrecursive is  $O(n)$ .

## BucketSort VS BucketSortRecursive

### *Tijdscomplexiteit*

Random 30.000 getallen sorteerden bucketSort in: 0.58 seconden. De recursive versie deed hier: 0.61 seconden over. Het zou kunnen dat mijn systeem minder goed werkt met recursieve functies of dat python in het algemeen hier minder goed mee om kan gaan. De big O was bij beide algoritmes hetzelfde alleen de precies benaderde formule week iets af. Ik denk dat daar het verschil in zit. De niet recursieve had dus een iets minder complexe tijdscomplexiteit.

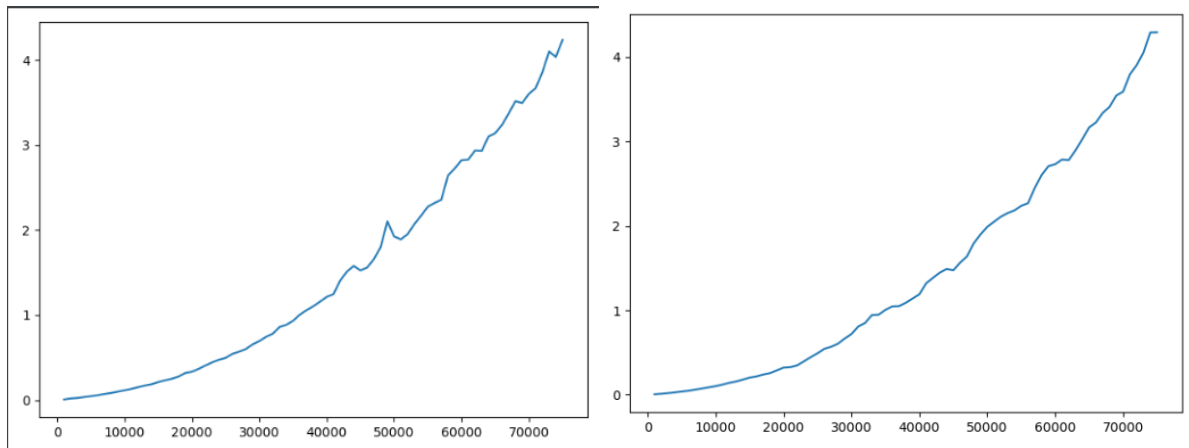
### *Ruimtecomplexiteit*

Bij extreem grote  $n$ 's en  $l$ 's krijg je een error bij de recursieve functie omdat python/mijn systeem dit niet ondersteund. Ik concludeer daaruit dat de ruimtecomplexiteit complexer is bij de recursieve functie.

### *Tijdsanalyse*

Beide functies geven een heel erg vergelijkbare grafiek weer als we kijken naar de tijd die het kost voor het sorteren.

## Conclusie



Bij beide gevallen is de tijdscomplexiteit kwadratisch. Dit is ook terug te zien in de worst case tijdcomplexiteit big O voor beide gevallen:

$$O(n * i^2) \text{ of } O(i * n^2)$$

Hieruit kunnen we concluderen dat deze implementatie van bucketsort niet het meest efficiënte sorteer algoritme is. Insertion Sort heeft bijvoorbeeld als worst case  $O(n^2)$ . Ook is deze implementatie van bucketsort naar mijn mening niet geschikt voor negatieve en/of komma getallen.

Tenslotte heeft het recursieve implementeren geen positief effect en is qua tijdscomplexiteit en ruimtecomplexiteit iets minder effectief.

## Extra's

### *Wat nam ik wel en niet mee in mijn analyse?*

In de afbeeldingen van mijn code die bij de tijdscomplexiteit staan is te zien wat ik allemaal meereken voor stappen. Ik neem dus elke stap van het algoritme mee in mijn tijdscomplexiteits formule. Aan het begin van het document staan hoeveel stappen een python functie kost, deze heb ik gehanteerd bij het tellen van de complexiteit. De bronnen van de python functies staan bij de bronnen in dit document. De getelde stappen blijven een benadering omdat vaak online alleen de big O van de functie is vermeld en niet de exacte aantal stappen. Dit geeft geen significant verschil, maar toch wou ik nogmaals benadrukken dat het een benadering is.

### *Geschikt voor het sorteren van getallen met cijfers achter de komma*

Hij gaat eerst op zoek wat de grootste vermenigvuldiging is. Hij vermenigvuldigd elk item in de lijst met de grootste vermenigvuldiging en cast dit naar een int om de ".0" weg te krijgen. Nu kan de lijst gesorteerd worden met als standaard het bucketSort algoritme. worden alle items weer gedeeld door de grootste vermenigvuldiging en het zijn weer de originele items.

### *Unit tests*

Als laatste extraatje heb ik ook nog unit tests toegevoegd om te laten zien dat het echt werkt en er geen errors zijn. De test cases zijn zo opgesteld dat alle soorten getallen getest worden.

## Bronnen

[https://thecodingbot.com/time-and-space-complexity-analysis-of-pythons-list-reverse-method/#:~:text=Time%20Complexity%3A%20O\(N\),third%20last%2C%20and%20so%20on.](https://thecodingbot.com/time-and-space-complexity-analysis-of-pythons-list-reverse-method/#:~:text=Time%20Complexity%3A%20O(N),third%20last%2C%20and%20so%20on.)

<https://stackoverflow.com/questions/5454030/how-efficient-is-pythons-max-function>

<https://thecodingbot.com/pythons-list-remove-time-and-space-complexity-analysis/>

[https://www.geeksforgeeks.org/internal-working-of-the-len-function-in-python/#:~:text=Hence%2C%20len\(\)%20function%20in,in%20O\(1\)%20complexity.&text=Note%3A%20This%20might%20seem%20very,programming%2C%20especially%20with%20big%20inputs.](https://www.geeksforgeeks.org/internal-working-of-the-len-function-in-python/#:~:text=Hence%2C%20len()%20function%20in,in%20O(1)%20complexity.&text=Note%3A%20This%20might%20seem%20very,programming%2C%20especially%20with%20big%20inputs.)

[https://www.geeksforgeeks.org/append-extend-python/#:~:text=extending%20the%20list.-,The%20length%20of%20the%20list%20increases,of%20elements%20in%20it's%20argument.&text=NOTE%3A%20A%20string%20is%20an,you%20iterate%20over%20the%20string.&text=Time%20Complexity%3A,i.e.%2CO\(1\).](https://www.geeksforgeeks.org/append-extend-python/#:~:text=extending%20the%20list.-,The%20length%20of%20the%20list%20increases,of%20elements%20in%20it's%20argument.&text=NOTE%3A%20A%20string%20is%20an,you%20iterate%20over%20the%20string.&text=Time%20Complexity%3A,i.e.%2CO(1).)