Bucket Sort

Niels Bijl
HPP

Inhoud

Time complexity Python functies

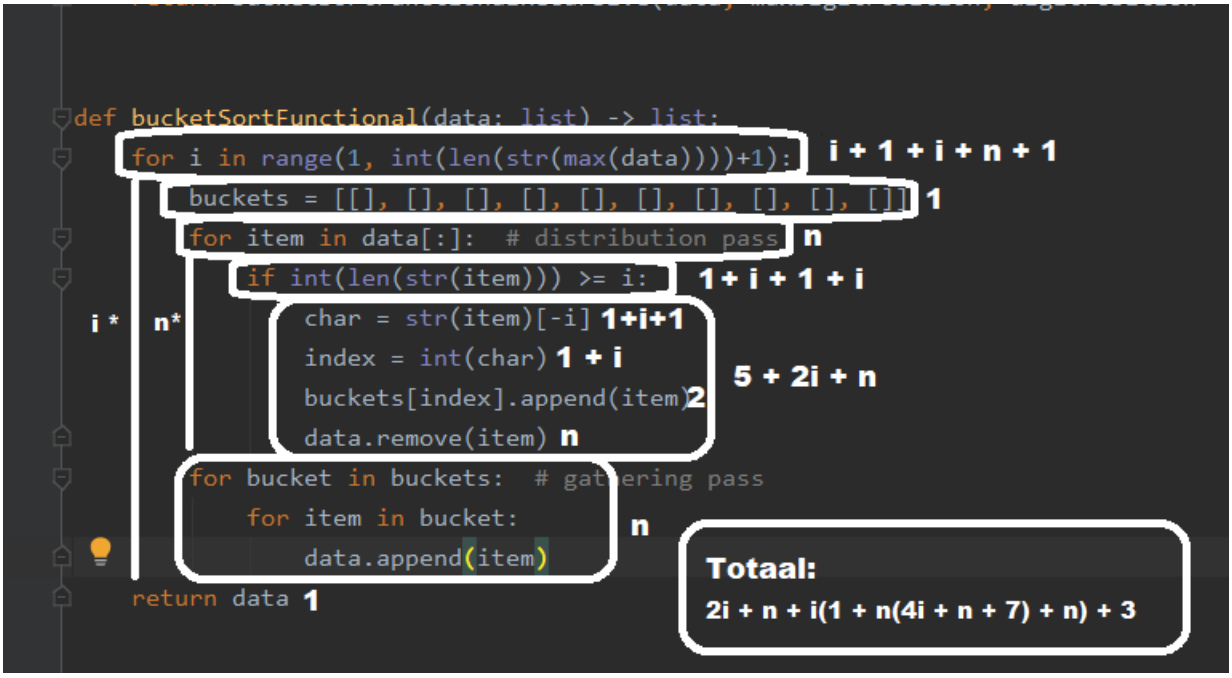
Reverse(): O(n)
Remove(): O(n)
int(): O(i)
str(): O(i)
max(): O(n)
len(): O(n)
Variable declaration O(1)
for loop: O(n)
if & else: O(1)
append(): O(1)
+/-: O(1)

Zie bronnen

Tijdscomplexiteit

BucketSort

BucketSortFunctional time complexity:



BucketSort time complexity:

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 =

$$2i + n + i(1 + n(4i + n + 7) + n) + 3$$
$$2i + n + i(n(4i + n) + n)$$
$$i(n(4i + n) + n)$$
$$i(n(i + n))$$
$$i(ni + n^2)$$
$$n * i^2 + i * n^2$$
$O(n * i^2)$ **OF** $O(i * n^2)$

Big O =

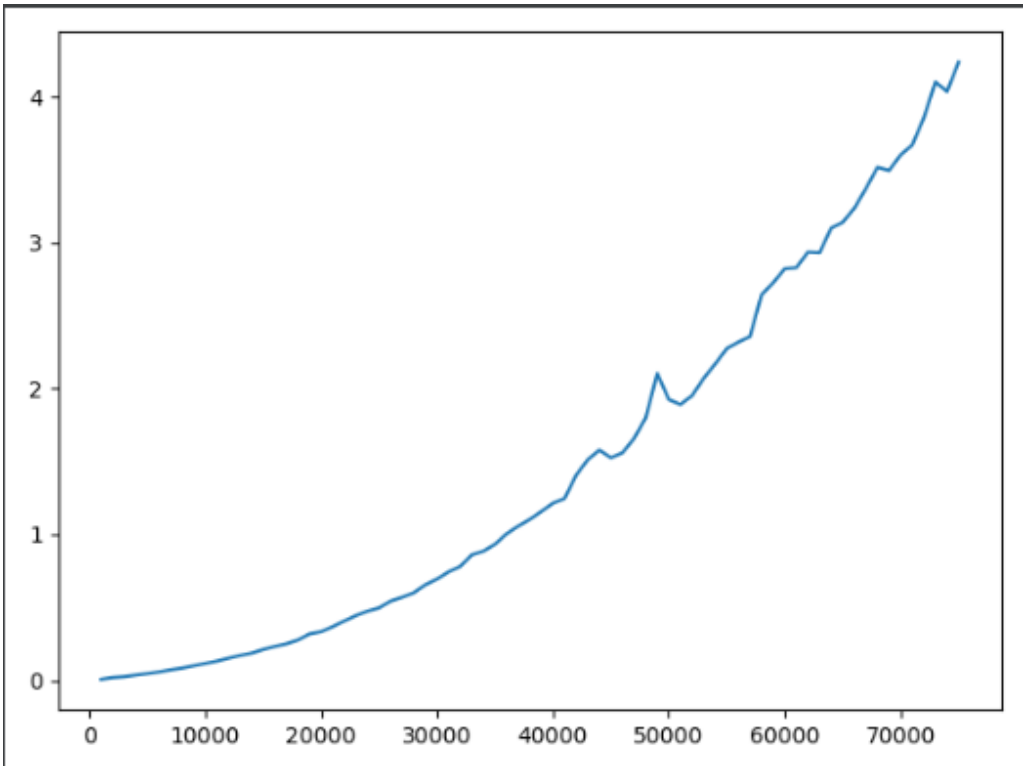
$$O(n * i^2)$$

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

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

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 exponentiele functie.

BucketSortRecursive

BucketSortFunctionalRecursive time complexity:

BucketSortRecursive time complexity:

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 =

$$7n + 2i + 12 + i(1 + n(4i + n + 6) + n + 4)$$
$$7n + 2i + i(n(4i + n) + n)$$
$$i(n(i + n))$$
$$i(ni + n^2)$$
$$n * i^2 + i * n^2$$
$O(n * i^2)$ **OF** $O(i * n^2)$

Big O =

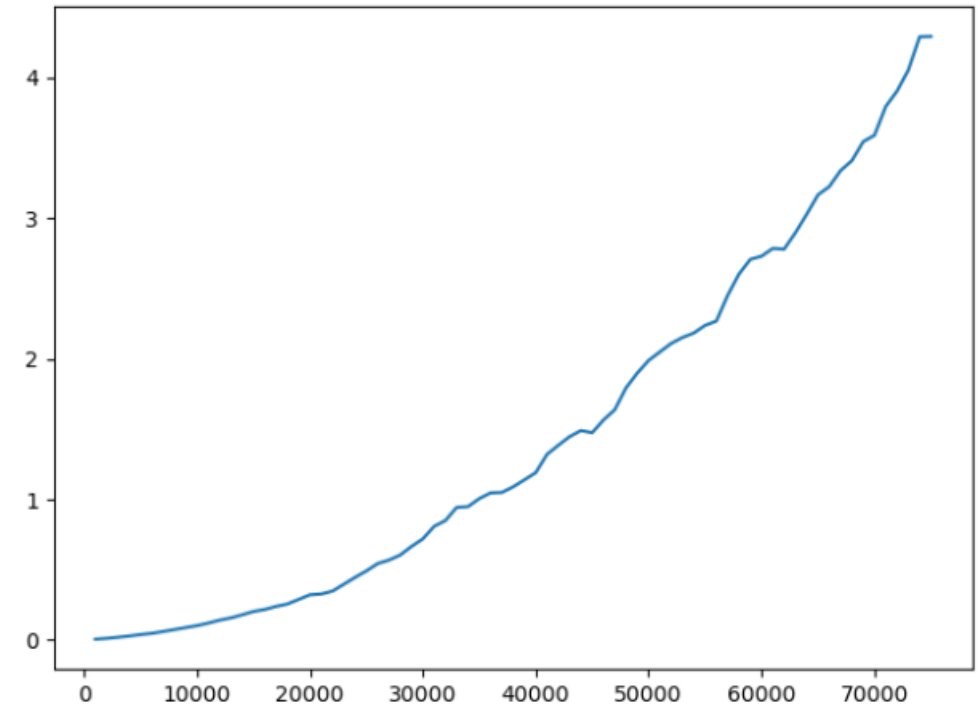
$$O(n * i^2)$$

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

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

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 exponentiele 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 en I maximaal: sys.maxsize, de maxsize verschilt per systeem.

De ruimte complexiteit van bucketSort en bucketSortrecursive is O(n) of O(i), omdat deze beide oneindig kunnen zijn kunnen beide big O's.

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.

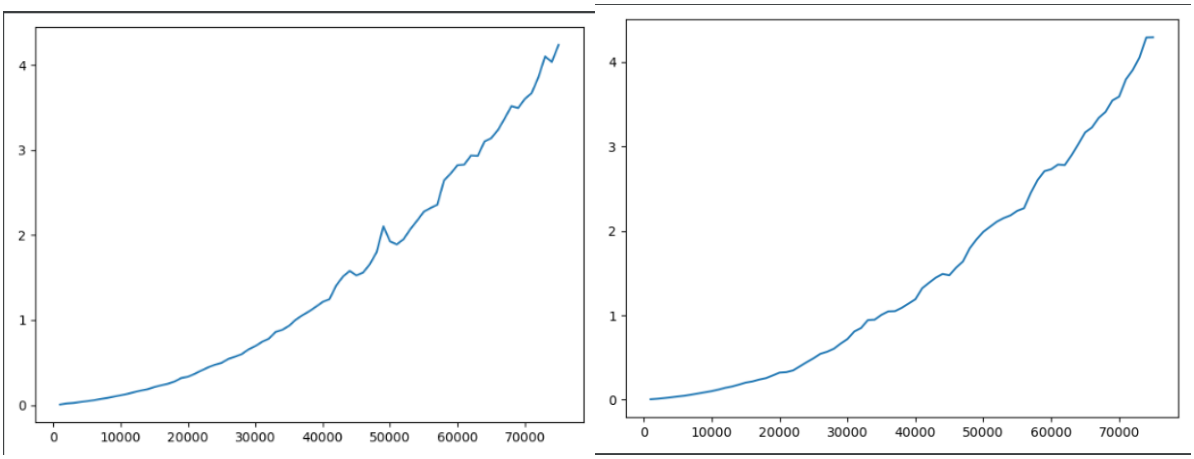
Ruimtecomplexiteit

Bij extreem grote n's en I's krijg je een error bij de recursieve functie omdat python/mijn systeem dit niet ondersteund. Ik concludeer daar uit 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 exponentieel. Dit is ook terug te zien in de worst case tijdcomplexiteit big O voor beide gevallen:

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

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

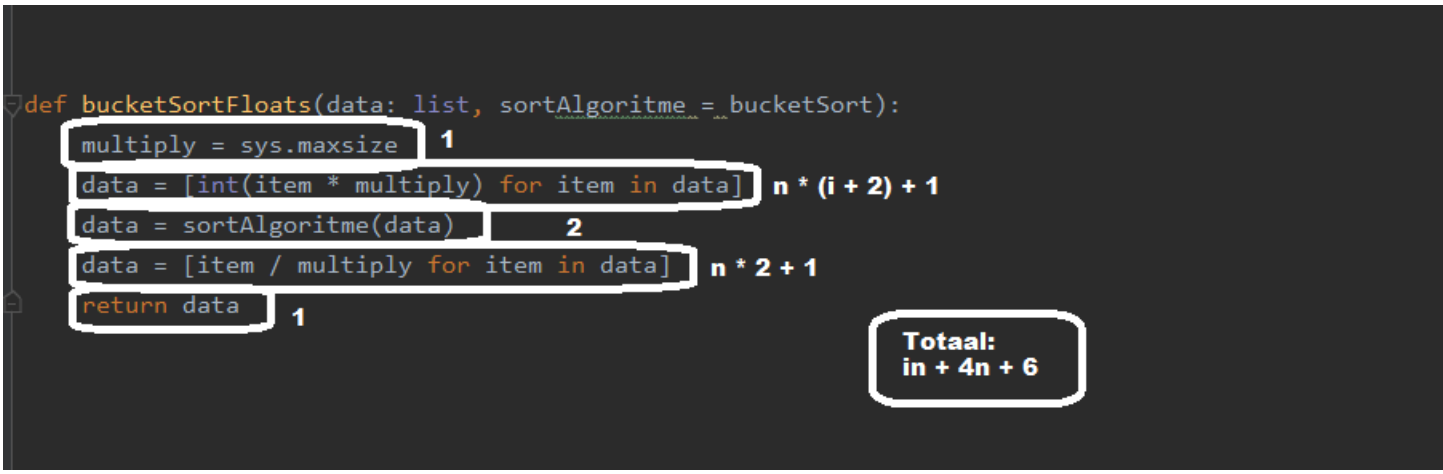
Tenslotte heeft het recursieve implementeren geen positief effect en is qua tijdscomplexiteit en ruimtecomplexiteit iets minder effectief.

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



Ik vermenigvuldig elk getal met het maximale grote getal van python, zodat dit getal geen getallen achter de komma meer heeft. Bijvoorbeeld 1,0001 * 10000 = 10001. Alleen dan doe ik het niet met 10000 maar met de oneindig van python. Ook cast ik het daarna nog naar een int om de .0 weg te krijgen. Nu het integers zijn kan ik mijn bucketSort hier mee gebruiken om ze te sorteren. Nu ze gesorteerd zijn kan ik ze weer delen door oneindig van python en zijn het weer de originele getallen.

De vermenigvuldiging met maxsize heeft jammer genoeg wel grote impact op de de ruimtecomplexiteit, daarom ben ik van mening dat dit niet de meest effectieve manier is. Maar het werkt functioneel gelukkig wel! De tijdscomplexiteit is:
i * n + 4 * n + 6 + (tijdscomplexiteit van het gebruikte sorteer algoritme).

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://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/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).