Sorterings algoritmer

Jens Tinggaard

27. oktober 2019

Indhold

1 Sorterings algoritmer

Sortingsalgoritmer er algoritmer som bruges til at sortere lister, eller andre sorterbare ting. Der findes et utal af dem, og de varierer i hastighed, hukommelseskrav og fordele/ulemper afhængig af "blandbarheden"af listen.

En sorteringsalgoritme er kendetegnet ved at den tager en liste A som input, og returnerer den sorterede liste. Længden af en liste er vist som n, og index kaldes på følgende måde: A_n, hvilket vil være det sidste element i listen.

1.1 Bubble sort

Bubble sort er nok den mest simple sorteringsalgoritme der findes. Den virker ved at sammenligne element A_0 med A_1 og bytter så elementerne om, i tilfælde af at A_1 er større end A_0. Algoritmen kører hele listen igennem, hvorefter man ved at A_n er sorteret. Algoritmen starter så forfra, fra $A_0 \rightarrow A_{n-1}$ hvorefter $A_{n-1} \rightarrow A_n$ er sorteret.
Det vil sige at algortitmen kører listen igennem n gange i alt. Algoritmen har en worst-case på $O(n^2)$ og et gennemsnit på $O(n^2)$. Dog er best-case på $O(n)$ i tilfælde af at listen allerede er sorteret

1.2 Insertion sort

En anden algoritme, som er forholdsvis ligetil, er insertion sort. Den virker på samme måde som de fleste nok ville sortere kort i hånden. Man tager A_1 og sammenligner med A_0 og bytter om på dem hvis nødvendigt. Nu er $A_0 \rightarrow A_1$ sorteret. Derefter tager man og kigger på A_2 og sammenligner med A_1, hvis de bliver byttet, sammenligner man også værdien med A_0. Og ombytter hvis nødvendigt. Nu er $A_0 \rightarrow A_2$ sorteret. Algoritmen har en worst-case på $O(n^2)$ og et gennemsnit på $O(n^2)$ og ligesom Bubble sort, en best-case på $O(n)$.

2 Implementering af bubble sort

Følgende kodeudtræk er min implementering af bubble sort i Python, funktionen tager, 1 obligatorisk argument, hvilket er listen som skal sorteres: `A` Derudover tager den et boolean, `show_progress`, som angiver om listen skal

printes for hver ændring af den, sådan at man kan følge og på den måde debugge funktionen.

2.1 Bubble sort i Python

```
1  def bubblesort(A, show_progress=False):
2      for i in range(len(A) - 1):
3          swapped = False
4
5          for j in range(0, len(A) - i - 1):
6              if A[j] > A[j+1]:
7
8                  if show_progress:
9                      print(A)
10
11                 A[j], A[j+1] = A[j+1], A[j]
12                 swapped = True
13
14         if not swapped:
15             break
16
17     if show_progress:
18         print(A)
19
20     return A
```

Kildekode 1: Implementering af `bubblesort()`

Algoritmen virker som sagt ved at tjekke et givent array A igennem, n antal gange. Eller indtil det er sorteret. Jeg har bygget det op over et såkaldt "nested"for-loop, hvilket i bund og grund er et for-loop inde i et andet. Det yderste for-loop, looper altså over arrayet, $n - 1$ gange. Mens det inderste looper mellem $n - 1$ og 1 gange - altså det antal usorterede elementer der er tilbage. Hvis elementerne ikke er sorteret, bliver de byttet om på, og `swapped` bliver tildelt værdien $True$. Hvis det inderste for-loop når at løbe en hel omgang, uden at bytte om på nogle elementer, vil værdien for `swapped` være $False$ og det yderste loop bliver brudt, da hele listen nu er sorteret.

3 Test af program

For at teste mit program, har jeg lavet en fil kaldet timings.py, som udytter biblioteket *matplotlib*[1], som er et bibliotek brugt til at plotte grafer og andet statistik. Derudover har jeg brugt det indbyggede bibliotek *time*, til at tage tid på de forskellige algoritmer og kunne sammenligne dem.

3.1 Korrekthed af program

Idet jeg har skrevet de forksellige algoritmer, har jeg selvfølgelig testet at de virker korrekt, dette har jeg gjort ved at give funktionerne et valgfrit argument, kaldet `show_progress`, hvilket angiver om listen skal printes for hver gang den ændres. Dette har gjort det super nemt for mig at teste korrektheden af hver enkelt algoritme.

3.2 Fejlfinding af program

Programmet vil formentlig fejle ved:

1. Tomme lister
2. Lister bestående af andet end tal (boolske udtryk, tekst strenge osv.)
3. Andet forkert input i til funktionerne, såsom tal som ikke opfylder $i \in \{\mathbb{N}, 0\}$ for funktionen `compare()`. Samt at $upper \geq lower$.
4. Den givne *key* er ikke gyldig. Altså enten $'avg'$, $'min'$ eller $'max'$

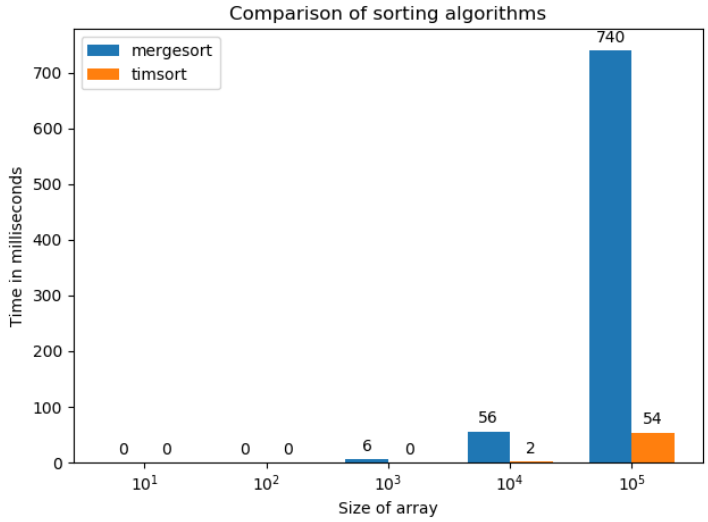
3.3 Eksempel

Inden programmet køres, skal det sættes op i timings.py, hvor man angiver antal prøver for hver liste-længde, samt en oppe og nedre grænse for listelængden, angivet i en potens af 10. Syntaksen er som følger:

```
compare(samples, lower, upper, *algorithms)
```

Kalder man f.eks.

[1] https://matplotlib.org/users/installing.html



Figur 1: Eksempel output

```
samples = 3
lower = 1
upper = 5

compare(samples, lower, upper, 'avg', mergesort, timsort)
```

Vil `mergesort` og `timsort` algoritmerne blive sammenlignet. Med lister i en længde af $10^1, 10^2, 10^3, 10^4$ og 10^5. Med 3 eksempler af hver - listerne er ens for alle de givne funktioner. Resultatet bliver plottet i et nyt vindue - se figur 1
Jeg vil anbefale ikke at gå alt for højt med potenserne, da det hurtigt kommer til at tage meget lang tid på de langsommere algoritmer.

4 Bilag

Jeg har uploadet koden til et repo på GitHub, `https://git.io/JeEmo`, hent det med git:

```
$ git clone https://github.com/Tinggaard/sorting_algorithms.git
```

4.1 sorting.py

```
import random

def random_liste(count):
    """
    Returning a list containing 'count' no. of elements
    with random values between 0 and 100 (both inclusive)
    """

    #Return a list of len 'count', with random numbers from 0..100
    return [random.randint(0, 100) for _ in range(count)]


#Dictionary of random lists
def random_dict(samples, lower_power, upper_power):
    """
    Used for testing operation speeds of different algorithms
    """

    return {10**i : [random_liste(10**i) for _ in range(samples)] for i in
    ↪  range(lower_power,upper_power+1)}


#Check if sorted, used for random_sort
def check_sort(A):
    return all([A[i] <= A[i+1] for i in range(len(A) - 1)])



def randomsort(A):
    """
    This function shuffles the array in place and checs if it's been
↪   sorted,
    otherwise it repeats.
    """

    #While not sorted: shuffle
    while not check_sort(A):
        random.shuffle(A)
    return A


def insertionsort(A, show_progress=False):
    """
```

```
42      Insertion sort is a simple sorting algorithm that works the way we
    ↪   sort playing cards in our hands.
43
44      Sort an arr[] of size n
45      insertionSort(arr, n)
46      Loop from i = 1 to n-1.
47      ......a) Pick element arr[i] and insert it into sorted sequence
    ↪   arr[0...i-1]
48      """
49
50      #Iterate over list from index 1
51      for k in range(1, len(A)):
52          if show_progress:
53              print(A)
54
55          #Initiate variable for index counting
56          i = k
57
58          #While the 2nd item is bigger than the first and the is is > -1
59          while A[i] < A[i-1] and i != 0:
60              #Swap indexes and decrement i
61              A[i], A[i-1] = A[i-1], A[i]
62              i-=1
63
64      if show_progress:
65          print(A)
66
67      return A[:]
68
69
70  def selectionsort(A, show_progress=False):
71      """
72      The selection sort algorithm sorts an array by repeatedly
73      finding the minimum element (considering ascending order)
74      from unsorted part and putting it at the beginning.
75      The algorithm maintains two subarrays in a given array.
76
77      1) The subarray which is already sorted.
78      2) Remaining subarray which is unsorted.
79
80      In every iteration of selection sort, the minimum element
81      (considering ascending order) from the unsorted subarray
82      is picked and moved to the sorted subarray.
83      """
84
```

```
85      #Iterate the array as many times as there are items-1
86      for k in range(len(A)-1):
87          if show_progress:
88              print(A)
89
90          #Reference value for comparison
91          ref = k
92
93          #Find the smallest item of the array and put it in the front
94          for i in range(k+1, len(A)):
95              if A[i] < A[ref]:
96                  ref = i
97
98          A[ref], A[k] = A[k], A[ref]
99
100     if show_progress:
101         print(A)
102
103     return A[:]
104
105
106
107 def mergesort(A, show_progress=False):
108     """
109     Like QuickSort, Merge Sort is a Divide and Conquer algorithm.
110     It divides input array in two halves, calls itself for the
111     two halves and then merges the two sorted halves.
112     """
113
114     #If only 1 element in array, return
115     if len(A) < 2:
116         return A
117     #Devide array on the middle until all arrays of 1 element
118     mid = int(len(A) / 2)
119     l = mergesort(A[:mid], show_progress)
120     if show_progress:
121         print(l)
122     r = mergesort(A[mid:], show_progress)
123     if show_progress:
124         print(r)
125
126     # sort the 2 single elements
127     i = j = 0
128     result = []
129     while i < len(l) and j < len(r):
```

```
130             if l[i] > r[j]:
131                 result.append(r[j])
132                 j += 1
133 
134             else:
135                 result.append(l[i])
136                 i += 1
137 
138         result += l[i:] + r[j:]
139 
140         return result[:]
141 
142 
143     def bubblesort(A, show_progress=False):
144         """
145         Bubble Sort is the simplest sorting algorithm that works by
    ↪    repeatedly
146         swapping the adjacent elements if they are in wrong order.
147         """
148 
149         for i in range(len(A) - 1):
150             swapped = False
151 
152             for j in range(0, len(A) - i - 1):
153                 if A[j] > A[j+1]:
154 
155                     if show_progress:
156                         print(A)
157 
158                     A[j], A[j+1] = A[j+1], A[j]
159                     swapped = True
160 
161             if not swapped:
162                 break
163 
164         if show_progress:
165             print(A)
166 
167         return A[:]
168 
169 
170     def timsort(A):
171         """
172         The default python sorting algorithm
173         """
```

```
174     return sorted(A)
175
176
177
178
179
180 # To do some testing
181 if __name__ == '__main__':
182
183     A = random_liste(10)
184
185     sorteret = mergesort(A, True)
186     print(sorteret)
```

4.2 timings.py

```
from time import time
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
from sorting import *

# matplotlib.rcParams['text.usetex'] = True


def spent(algorithm, liste):
    start = time()
    algorithm(liste)
    return (time() - start) * 1000




def compare(samples, lower, upper, key='avg', *algorithms):
    """
    possible keys: avg, min, max
    """

    tal = random_dict(samples, lower, upper)

    #create dict for results
    result = {a.__name__: {10**i: {} for i in range(lower, upper+1)} for a
    ↪  in algorithms}

    #Iterating algorithms
    for a in algorithms:

        #Creating stats
        for k in tal.keys():
            result[a.__name__][k]['samples'] = [spent(a, l) for l in
            ↪  np.copy(tal[k])]
            #Have to np.copy, to ensure deep copy

            result[a.__name__][k]['min'] =
            ↪  min(result[a.__name__][k]['samples'])
            result[a.__name__][k]['max'] =
            ↪  max(result[a.__name__][k]['samples'])
            result[a.__name__][k]['avg'] =
            ↪  sum(result[a.__name__][k]['samples']) / samples
```

```
39
40          print('Done with {alg}, {nr}/{of}'.format(alg=a.__name__,
            ↪  nr=algorithms.index(a)+1, of=len(algorithms)))
41
42
43      ################################################
44      ################## PLOTTING ####################
45      ################################################
46
47      #Getting each of the labels for bottom of chart
48      labels = [10**i for i in range(lower, upper+1)]
49
50
51      #Putting the 'key'-time into a dict
52      comp = {a: [result[a][s][key] for s in result[a]] for a in result}
53
54
55
56      x = np.arange(len(labels)) #Number of labels
57      no = len(algorithms) #Number of algorithms
58      width = 0.7 / no #Width of each bar
59
60
61      fig, ax = plt.subplots()
62      bars = [ax.bar(x + en*width, c[1], width, label=c[0]) for en, c in
        ↪  enumerate(comp.items())]
63
64
65      labels = [r'$10^{' + str(n) + r'}$' for n in range(lower, upper+1)]
66
67      print(x)
68      ax.set_xlabel('Size of array')
69      ax.set_ylabel('Time in milliseconds')
70      ax.set_title('Comparison of sorting algorithms')
71      ax.set_xticks(x + (width/2)*(no-1))
72      ax.set_xticklabels(labels)
73      ax.legend()
74
75
76      # Plotting the value of the bar on top of it
77      for bar in bars:
78          for rect in bar:
79              height = int(rect.get_height())
80              ax.annotate('{}'.format(height),
81                          xy=(rect.get_x() + rect.get_width() / no, height),
```

```
                        xytext=(0, 3),  # 3 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')

    fig.tight_layout()
    plt.show()




if __name__ == '__main__':
    samples = 3
    lower = 1
    upper = 3

    compare(samples, lower, upper, 'avg', mergesort, timsort, bubblesort,
    ↪  insertionsort)
```