Sorterings algoritmer

Jens Tinggaard

2. november 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
```

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

3.3 Eksempel

Programmet køres fra kommandolinjen med følgende syntaks

```
$ python timings.py [-h] [-s SAMPLES] [-l LOWER] [-u UPPER] [-k KEY]
                    [-q] [-f FUNCTIONS [FUNCTIONS ...]]
```

Er du i tvivl, køres koden blot med flaget `-h`, for at få en hjælpe-guide frem. Alle flagene har en stadard-værdi, som blot bliver passeret til filen, hvis andet ikke er angivet.

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

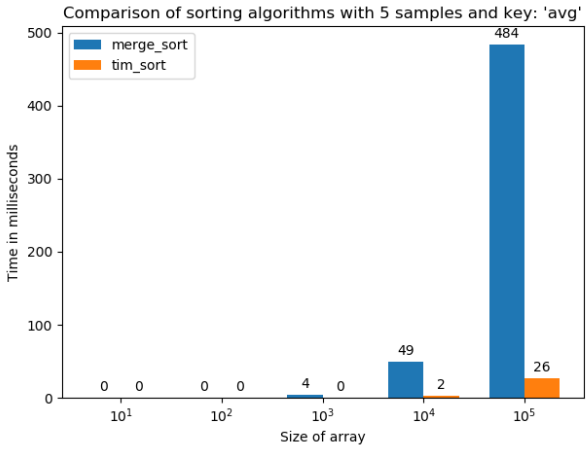
Standardværdier for flagene

- `-s` 3
- `-l` 1
- `-u` 4
- `-k` *avg*
- `-q` *False*
- `-f` *merge_sort*

For faktisk at sammenligne forskellige algoritmer, skal der selvfølgelig passeres flere algoritmer til `-f` flaget, som blot adskilles af mellemrum.
Kalder man f.eks.

```
$ python timings.py -f merge_sort tim_sort
```

Vil `merge_sort` og `tim_sort` algoritmerne blive sammenlignet. Med lister i en længde af $10^1, 10^2, 10^3$ og 10^4. 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.



Figur 1: Eksempel output

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

    #While not sorted: shuffle
```

```
34      while not check_sort(A):
35          random.shuffle(A)
36      return A
37
38
39  def insertion_sort(A, show_progress=False):
40      """
41      Insertion sort is a simple sorting algorithm that works the way we
    ↪   sort playing cards in our hands.
42
43      Sort an arr[] of size n
44      insertionSort(arr, n)
45      Loop from i = 1 to n-1.
46      .......a) Pick element arr[i] and insert it into sorted sequence
    ↪   arr[0...i-1]
47      """
48
49      #Iterate over list from index 1
50      for k in range(1, len(A)):
51          if show_progress:
52              print(A)
53
54          #Initiate variable for index counting
55          i = k
56
57          #While the 2nd item is bigger than the first and the is is > -1
58          while A[i] < A[i-1] and i != 0:
59              #Swap indexes and decrement i
60              A[i], A[i-1] = A[i-1], A[i]
61              i-=1
62
63      if show_progress:
64          print(A)
65
66      return A
67
68
69  def selection_sort(A, show_progress=False):
70      """
71      The selection sort algorithm sorts an array by repeatedly
72      finding the minimum element (considering ascending order)
73      from unsorted part and putting it at the beginning.
74      The algorithm maintains two subarrays in a given array.
75
76      1) The subarray which is already sorted.
```

```
    2) Remaining subarray which is unsorted.

    In every iteration of selection sort, the minimum element
    (considering ascending order) from the unsorted subarray
    is picked and moved to the sorted subarray.
    """

    #Iterate the array as many times as there are items-1
    for k in range(len(A)-1):
        if show_progress:
            print(A)

        #Reference value for comparison
        ref = k

        #Find the smallest item of the array and put it in the front
        for i in range(k+1, len(A)):
            if A[i] < A[ref]:
                ref = i

        A[ref], A[k] = A[k], A[ref]

    if show_progress:
        print(A)

    return A



def merge_sort(A, show_progress=False):
    """
    Like QuickSort, Merge Sort is a Divide and Conquer algorithm.
    It divides input array in two halves, calls itself for the
    two halves and then merges the two sorted halves.
    """

    #If only 1 element in array, return
    if len(A) < 2:
        return A
    #Devide array on the middle until all arrays of 1 element
    mid = int(len(A) / 2)
    l = merge_sort(A[:mid], show_progress)
    if show_progress:
        print(l)
    r = merge_sort(A[mid:], show_progress)
```

```
122     if show_progress:
123         print(r)
124 
125     # sort the 2 single elements
126     i = j = 0
127     result = []
128     while i < len(l) and j < len(r):
129         if l[i] > r[j]:
130             result.append(r[j])
131             j += 1
132 
133         else:
134             result.append(l[i])
135             i += 1
136 
137     result += l[i:] + r[j:]
138 
139     return result
140 
141 
142 def bubble_sort(A, show_progress=False):
143     """
144     Bubble Sort is the simplest sorting algorithm that works by
    ↪ repeatedly
145     swapping the adjacent elements if they are in wrong order.
146     """
147 
148     for i in range(len(A) - 1):
149         swapped = False
150 
151         for j in range(0, len(A) - i - 1):
152             if A[j] > A[j+1]:
153 
154                 if show_progress:
155                     print(A)
156 
157                 A[j], A[j+1] = A[j+1], A[j]
158                 swapped = True
159 
160         if not swapped:
161             break
162 
163     if show_progress:
164         print(A)
165 
```

```
166     return A
167
168
169 def tim_sort(A):
170     """
171     The default python sorting algorithm
172     """
173     return sorted(A)
174
175
176
177
178
179 # To do some testing
180 if __name__ == '__main__':
181
182     # A = random_liste(10)
183     #
184     # sorteret = merge_sort(A, True)
185     # print(sorteret)
186
187     a = []
188     b = [6]
189     print(bubble_sort(a), bubble_sort(b))
190     print(merge_sort(a), merge_sort(b))
191     print(insertion_sort(a), insertion_sort(b))
192     print(selection_sort(a), selection_sort(b))
```

4.2 timings.py

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

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


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




def compare(samples, lower, upper, key='avg', plot=True, *algorithms):

    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

```

```
38          print('Done with {alg}, {nr}/{of}'.format(alg=a.__name__,
            ↪  nr=algorithms.index(a)+1, of=len(algorithms)))
39
40      if not plot:
41          return result
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      ax.set_xlabel('Size of array')
68      ax.set_ylabel('Time in milliseconds')
69      ax.set_title('Comparison of sorting algorithms with {} samples and
        ↪  key: \'{}\''.format(len(labels), key))
70      ax.set_xticks(x + (width/2)*(no-1))
71      ax.set_xticklabels(labels)
72      ax.legend()
73
74
75      # Plotting the value of the bar on top of it
76      for bar in bars:
77          for rect in bar:
78              height = int(rect.get_height())
79              ax.annotate('{}'.format(height),
```

```
80                          xy=(rect.get_x() + rect.get_width() / 2, height),
81                          xytext=(0, 3),  # 3 points vertical offset
82                          textcoords="offset points",
83                          ha='center', va='bottom')
84
85      fig.tight_layout()
86      plt.show()
87
88
89
90
91
92  def main():
93      functions = {'bubble_sort': bubble_sort,
94      'insertion_sort': insertion_sort,
95      'selection_sort': selection_sort,
96      'merge_sort': merge_sort,
97      'tim_sort': tim_sort}
98
99      parser = argparse.ArgumentParser(description='Compare sorting
        ↪  algorithms')
100
101     parser.add_argument('-s', '--samples', default=3, type=int,
        ↪  help='Number of samples')
102
103     parser.add_argument('-l', '--lower', default=1, type=int, help='The
        ↪  minimum size of the array as a power of 10')
104
105     parser.add_argument('-u', '--upper', default=4, type=int, help='The
        ↪  maximum size of the array as a power of 10')
106
107     parser.add_argument('-k', '--key', default='avg', type=str, help='The
        ↪  key to use for the timings, \
108     valid keys are: \'avg\', \'min\', \'max\'')
109
110     parser.add_argument('-q', '--quiet', dest='plot',
        ↪  action='store_false', help='Weather to plot or not')
111     parser.set_defaults(plot=True)
112
113     parser.add_argument('-f', '--functions', default='merge_sort',
        ↪  nargs='+', type=str, help='The sorting algorithms to use')
114
115
116     args = parser.parse_args()
117
```

```
118
119
120     return compare(args.samples, args.lower, args.upper, args.key,
        ↪  args.plot, *[functions[k] for k in args.functions])
121
122
123 if __name__ == '__main__':
124     main()
```