ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი
კომპიუტერულ მეცნიერებათა დეპარტამენტი

ვილიამ სააკიანი

დავითი ბუმბეიშვილი

# სწრაფი დალაგების ალგორითმი და მისი გაუმჯობესების გზები

ხელმძღვანელი: **ალექსანდრე გამყრელიძე**,

სრული პროფესორი

თბილისი
2013

# სარჩევი

# ანოტაცია

ნაშრომის ძირითადი მიზანია სწრაფი დალაგების ალგორითმის მიმოხილვა და მისი მუშაობის სისწრაფის გაუმჯობესების ვარიანტების განხილვა. ასევე შეცვლილი პროგრამული კოდის შემოწმება 1.000.000 ელემენტიან სიმრავლეზე.

# შესავალი

ინფორმატიკაში დახარისხების ალგორითმებს ძალიან მნიშვნელოვანი ადგილი უკავია. არსებობს შედარებაზე დაფუძნებული რამდენიმე ალგორითმი,ისინი მონაცემების მიხედვით განსხვავდებიან შესრულების სისწრაფით. მოცემულ ნაშრომში გვინდა განვიხილოთ სწრაფი დალაგების ალგორითმი, რომლის შესრულების დრო უკეთეს შემთხვევაში არის $n \log n$ ხოლო უარეს შემთხვევაში კი $n^2$ და მისი მოდიფიკაცია.

# სწრაფი დალაგების ალგორითმი: (QuickSort)

ალგორითმს საფუძველი დაედო 1960 წელს (Hoare) და მას შემდეგ იგი შეისწავლა ბევრმა ადამიანმა. სწრაფი დალაგების ალგორითმი განსაკუთრებით პოპულარულია, რადგან ის მარტივად არის რეალიზირებული, არის საკმაოდ კარგი ზოგადი დანიშნულების ალგორითმი, რომელიც გამოსადეგია ბევრ სიტუაციაში, და თან ნაკლებად იყენებს რესურსებს ვიდრე სხვა ალგორითმები.

მთავარი უპირატესობა ამ ალგორითმის არის ის, რომ იგი არის წერტილოვანი (იყენებს მხოლოდ მცირე დამატებით სტეკს), საშუალოდ ის მოითხოვს მხოლოდ N log N ოპერაციას,N ელემენტიანი სიმრავლის დალაგებისას და აქვს ძალიან მოკლე შიდა ციკლი. ალგორითმის ნაკლი არის ის, რომ ის არის რეკურსიული (განხორციელება ძალიან რთულია, როდესაც რეკურსია დაშვებული არ არის), უარეს შემთხვევაში ის მოითხოვს $N^2$ ოპერაციას, გარდა ამისა, ის არის "მყიფე": მცირე ხარვეზებმა რეალიზაციაში, რომლებიც ადვილად შეიძლება დარჩნენ შეუმჩნეველნი, შეიძლება გამოიწვიოს ის, რომ ალგორითმი იმუშავებს ძალიან ცუდად ზოგიერთ ფაილთან.

Quicksort-ს შესრულების პროცესი კარგად არის შესწავლილი. ალგორითმი ექვემდებარება მათემატიკურ ანალიზს, ასე რომ არსებობს ზუსტი მათემატიკური ფორმულები, რომლებიც არიან დაკავშირებულნი მის შესრულების საკითხთან. ანალიზის შედეგები არაერთხელაა შემოწმებული ემპირიულად და შედეგად ალგორითმი გახდა ისეთი სრულყოფილი, რომ ხშირად გამოიყენება დახარისხების დავალებების შესრულების ფართო სპექტრში. ეს ყველაფერი ალგორითმს ხდის ღირსს, რომ დეტალურად იქნეს შესწავლილი მისი უფრო ეფექტურად მუშაობის გზები.მსგავსი მეთოდების განხორციელება ასევე მიზანშეწონილია სხვა ალგორითმებისთვისაც, მაგრამ სწრაფი დახარისხების ალგორითმი გამოყენებისას, ჩვენ შეგვიძლია ვიყოთ დარწმუნებული მის დიდ საიმედოობაში.

სწრაფი დახარისხების ალგორითმის გაუმჯობესება - დიდი ცდუნებაა: სწრაფი დახარისხების ალგორითმი - ერთგვარი ხაფანგია პროგრამისტებისთვის. თითქმის იმ მომენტიდან, როცა პირველად გამოქვეყნდა ალგორითმი, ლიტერატურაში ჩნდებოდა "გაუმჯობესებული" . გამოცდილია და შესწავლილია ბევრი იდეები, მაგრამ მაინც ადვილია შესაძლებელია მოტყუვდე რადგან ალგორითმი ისე კარგადაა დაბალანსებული, რომ ერთ ნაწილში გაუმჯობესებამ შესაძლოა გააუარესოს მეორე ნაწილი. ჩვენ დეტალებში განვიხილავთ ალგორითმის გაუმჯობესების სამ ვერსიას .

ძალიან კარგად დაწერილი სწრაფი დალაგების ალგორითმი სავარაუდოდ ბევრად უფრო სწრაფია, ვიდრე ნებისმიერი სხვა ალგორითმი. თუმცა,ერთხელაც გავიხსენოთ, რომ ალგორითმი ძალიან მყიფეა და ნებისმიერმა ცვლილებამ შეიძლება გამოიწვიოს არასასურველი მოვლენები გარკვეული ტიპის მონაცემებისთვის

# ალგორითმის არსი

ჯერ ირჩევ რომელიმე რიცხვს, გამყოფს, რომლის მიხედვითაც უნდა დაყო სიმრავლე, მაგალითად, პირველ ელემენტს. შემდეგ K სიმრავლეს ჰყოფ 3 ქვესიმრავლედ AB,C , A-ში ჩაიწერება ის ელემენტები, რომლებიც ამ პირველ ელემენტზე ანუ გამყოფზე ნაკლებია,B- თვითონ გამყოფები და C-ში ამ გამყოფზე დიდი ელემენტები. ალგორითმი ბოლოს აბრუნებს

{quicksort(A),B,quicksort(C)}-ს.

# ფსევდოკოდი

Quicksort(A)

- If ( |A|=0 ან |A|=1 ) return A;
- a-ს მივანიჭოთ A სიმრავლის პირველი ელემენტი
- A სიმრავლე დავყოთ სამ ქვესიმრავლედ { $A_1$ , $A_2$ , $A_3$ } სადაც

$$A_1 = \{x \in A | \ x < a\}$$

$$A_2 = \{x \in A | \ x = a\}$$

$$A_3 = \{x \in A | \ x > a\}$$

- Return { Quicksort$(A_1)$, $A_2$,Quicksort ( $A_3$)}

# კოდი პროგრამულ ენაზე

```
14  static int partition(std::vector<int> &arr, int l, int h)
15  {
16      int x = arr[h];
17      int i = (l - 1);
18
19      for (int j = l; j <= h - 1; j++)
20      {
21          if (arr[j] <= x)
22          {
23              ++i;
24              std::swap(arr[i], arr[j]);
25          }
26      }
27      ++i;
28      std::swap(arr[i], arr[h]);
29
30      return (i);
31  }
32
33  void quick_sort_rec_std(std::vector<int> &arr, int l, int h)
34  {
35      if (l < h)
36      {
37          int p = partition(arr, l, h);
38          quick_sort_rec_std(arr, l, p - 1);
39          quick_sort_rec_std(arr, p + 1, h);
40      }
41  }
42
```

სწრაფი დალაგების "შიდა ციკლი" შედგება მიმთითებლის ზრდის და მასივის ელემენტების ფიქსირებული ნომრით შედარებისგან. ზუსტად ეს ხდის ამ ალგორითმს სწრაფს. ძნელი წარმოსადგენია უფრო მარტივი შიდა ციკლი. გასაღებების დადებით ეფექტებს ასევე აქვს თავისი გავლენა, რადგან ამის გარდა კიდევ ერთი შემოწმება უარყოფით ზეგავლენას ახდენს შიდა ციკლში ალგორითმის შესრულებაზე.

ყველაზე საეჭვო თვისება ზემოთ მოყვანილი პროგრამის იმაში მდგომარეობს, რომ მას აქვს პატარა ეფექტი მარტივი ქვეფაილებზე. მაგალითად, თუ ფაილი უკვე დალაგებულია, პროგრამა უბრალოდ N ჯერ გამოიძახებს თავის თავს, ყოველ დროს ერთი ელემენტით უფრო მცირე ქვეფაილისთვის. ეს იმას ნიშნავს, რომ პროგრამის შესრულების დრო დაეცემა დაახლოებით $N^2$–მდე.საბედნიეროდ, არსებობს საკმაოდ მარტივი გზა, რათა "ყველაზე ცუდი" შემთხვევა არ მოხდეს პროგრამის პრაქტიკული გამოყენების დროს.

როდესაც ფაილში მსგავსი გასაღებები არსებობენ, მაშინ არსებობს ორი საეჭვო კითხვაც. პირველი, ორივე მიმთითებელი უნდა გაჩერდნენ გასაღების ტოლ ელემენტზე,მხოლოდ ერთი მათგანი უნდა გაჩერდეს, ხოლო მეორემ ყველა მათგანი უნდა გაიაროს,თუ ორივე მიმთითებლეებმა უნდა გაიაროს მათზე. ფაქტობრივად, ეს საკითხი დეტალურად არის შესწავლილი და შედეგებმა აჩვენა, რომ საუკეთესო ვარიანტია ის - როცა ორივე მიმთითებელი ჩერდებიან.ეს გვაძლევს საშუალებას შევინარჩუნოთ მეტნაკლებად დაბალანსებული სექციები ბევრი იდენტური გასაღებების დროს. ფაქტობრივად, ეს პროგრამა შეიძლება ოდნავ გაუმჯობესდეს შეწყვეტის სკანირების j <i და შემდეგ გამოვიყენოთ quicksort (l, j) პირველი რეკურსიული გამოძახებისთვის.

# ალგორითმის შესრულების მახასიათებლები

საუკეთესო შემთხვევაში ხდება მოცემული სიმრავლის თითქმის თანაბარი რაოდენობის ელემენტიან ქვესიმრავლესთან მუშაობის გაგრძელება, შედეგად ალგორითმი ბიჯების რაოდენობა შეიძლება გამოისახოს შემდეგი რეკურენტული ფორმულით

$$T_N = 2*T_{N/2}+N$$

ვიცით ისიც რომ ამ ფორმულის სავარაუდო გამოხატვა არის O (N log N )მიუხედავად იმისა რომ ასეთ სიტუაციას არც ისე ხშირად ვაწყდებით, ალგორითმის შესრულების საშუალო დრო სწორედ ამ ფორმულით გამოითვლება.

# გაუმჯობესების მეთოდები

1. მცირე რაოდენობის ელემენტიანი ქვესიმრავლეები

როგორც ვიცით quicksort ალგორითმს ძირითადად იყენებენ დიდი რაოდენობის მქონე ელემენტების შემცველი სიმრავლეების დასალაგებლად. ალგორითმზე მცირეოდენი დაკვირვების შედეგად,ადვილად შევნიშნავთ რომ მცირე ელემენტებიანი ქვესიმრავლეებისთვის (5-25 ელემენტიანი) quicksort ალგორითმი მრავალჯერ გამოუძახებს საკუთარ თავს. მაგალითად:
60,79, 82, 58, 39, 9, 54, 92, 44, 32 მონაცემებისთვის ის საკუთარ თავს 15-ჯერ გამოუძახებს.

ამ პრობლემის აღსაკვეთად მცირე ელემენტიანი სიმრავლეებისთვის უნდა გამოვიყენოთ არა შერწყმის,არამედ ჩასმით დალაგების ალგორითმი (insertsort).

ამ ნაწილის გათვალისწინებით პროგრამის შესრულების დრო მცირდება დაახლოებით 20%-ით.

```
69  void insertion_sort(std::vector<int> &arr, int l, int h)
70  {
71      int val = 0, pos = 0;
72      for (int i = (l + 1); i <= h; ++i)
73      {
74          val = arr[i];
75          pos = i;
76          while (pos > 0 && val < arr[pos - 1])
77          {
78              arr[pos] = arr[pos - 1];
79              --pos;
80          }
81          arr[pos] = val;
82      }
83  }
84
```

## 2. სამი ელემენტის მედიანით გაყოფის მეთოდი

ეს მეთოდი არის მცდელობა იმისა რომ სიმრავლის გასაყოფად გამოიყენოს საუკეთესო გამყოფი ელემენტი.ამისთვის რამდენიმე ვარიანტი არსებობს,ყველაზე კარგი იქნება, რომ ავირჩიოთ თვითნებური ელემენტი რომელიც,უარესი შემთხვევის მოხდენის ალბათობას მკვეთრად შეამცირებს.მაშინ  უარესი შემთხვევის ალბათობა ხდება უმნიშვნელო. ეს არის მარტივი "ალბათური" ალგორითმის მაგალითი, რომელიც თითქმის ყოველთვის მუშაობს, შეტანილი მონაცემების მიუხედავად. თვითნებობა კარგი ინსტრუმენტია ალგორითმების შექმნისას, განსაკუთრებით მაშინ, როდესაც შეტანილი მონაცემები შეიძლება იყვნენ საეჭვო.

სასარგებლო გაუმჯობესება იმაში მდგომარეობს, რომ ავიღოთ სამი ელემენტი ფაილიდან, და შემდეგ გამოვიყენოთ სამი ელემენტის საშუალოს - როგორც გამყოფი წევრი. თუ ელემენტები აღებულია დასაწყისში, შუაში და ფაილის ბოლოში, მაშინ შესაძლებელია ავარიდოთ მცველი ელემენტების გამოყენება: ვასორტირებთ სამ ელემენტს, შემდეგ ვცვლით ცენტრალურ ელემენტს a[h-1]-ით, და შემდეგ ვიყენებთ ალგორითმს მასივზე a[l +1 .. h-2]. ამას ეწოდება სამი ელემენტის მედიანით გაყოფის მეთოდი.

სამი ელემენტის მედიანით გაყოფის მეთოდი სასარგებლოა სამი მიზეზის გამო. პირველი, უარესი შემთხვევის ალბათობა  გაცილებით დაბალია. იმისთვის, რომ ამ ალგორითმმა გამოიყენოს $N^2$-ის პროპორციული დრო, ორი ელემენტი სამიდან უნდა იყვნენ ან ყველაზე მცირე ელემენტები, ან უდიდესი ელემენტები, და ეს უნდა განმეორდეს ყოველ ქვესიმრავლეში. მეორე, ეს მეთოდი გამორიცხავს მცველი ელემენტების აუცილებლობას, რადგან მათ როლშია ერთი ელემენტი სამიდან, რომლებიც ავიღეთ ელემენტების გაყოფის წინ. მესამე, ის რეალურად ამცირებს მუშაობის დროს დაახლოებით 5%.

```
43  static int mediana_index(const std::vector<int> &arr, int a, int b, int c)
44  {
45      return (arr[a] < arr[b] ?
46              (arr[b] < arr[c] ? b : arr[a] < arr[c] ? c : a) :
47              (arr[b] > arr[c] ? b : arr[a] > arr[c] ? c : a));
48  }
49
50  static void mediana(std::vector<int> &arr, int l, int h)
51  {
52      int len = h - l + 1;
53
54      if (len > 7)
55      {
56          int a = l, b = l + (len >> 2), c = h;
57          if (len > 40)
58          {
59              int s = len / 8;
60              a = mediana_index(arr, a, a + s, a + s * 2);
61              b = mediana_index(arr, b - s, b, b + s);
62              c = mediana_index(arr, c - s * 2, c - s, c);
63          }
64          b = mediana_index(arr, a, b, c);
65          std::swap(arr[b], arr[h]);
66      }
67  }
68
```

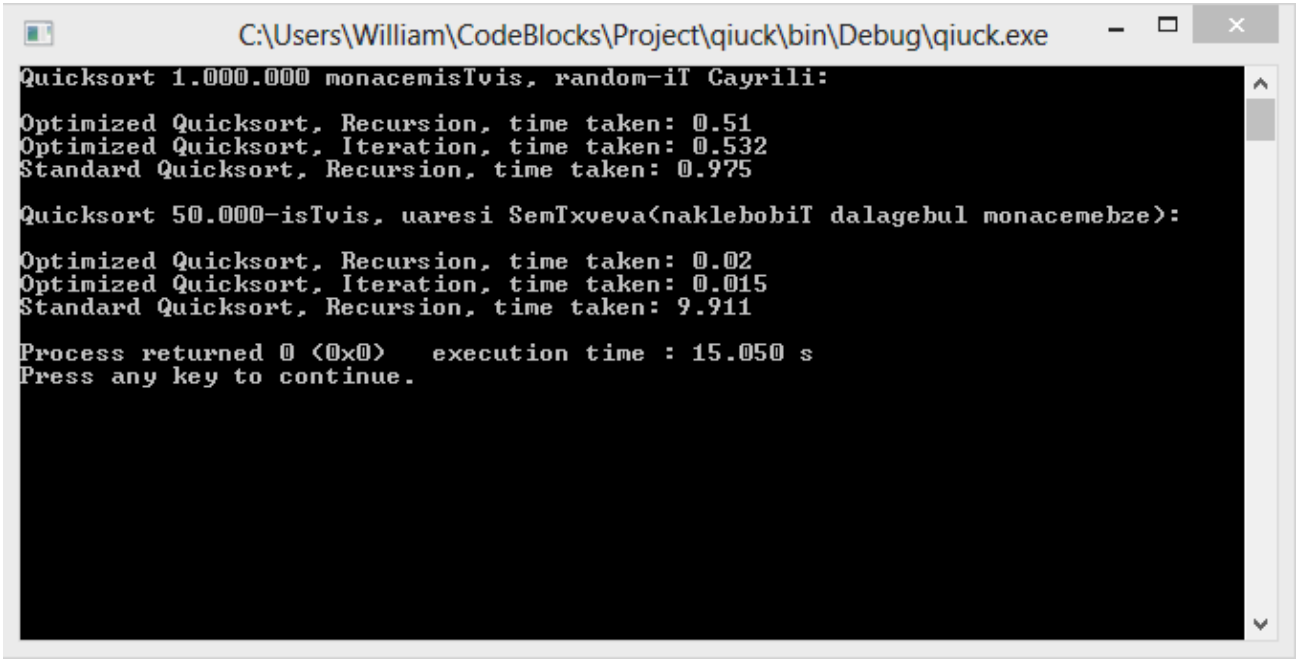
### 3. ალგორითმის იტერაციული რეალიზაცია

რეკურსიული quicksort ალგორითმის ზემოთ მოცემული მეთოდების გათვალისწინებით იტერაციულად გადაწერა მოგვცემს გარკვეულწილად კიდევ ერთ მცირე შეღავათს,ემპირიულად დადასტურებულია რომ ამით დაახლოებით 5 %-ით შეგვიძლია პროგრამის შესრულების დროის დაჩქარება.

```
125   void quick_sort_iter_opt(std::vector<int> &arr, int l, int h)
126   {
127       std::vector<int> stack(h - l + 1);
128       int top = -1;
129
130       stack[++top] = l;
131       stack[++top] = h;
132
133       while (top >= 0)
134       {
135           h = stack[top--];
136           l = stack[top--];
137
138           if ((h - l + 1) > 50)
139           {
140               mediana(arr, l, h);
141
142               int p = partition(arr, l, h);
143
144               if (p - 1 > l)
145               {
146                   stack[++top] = l;
147                   stack[++top] = p - 1;
148               }
149
150               if (p + 1 < h)
151               {
152                   stack[++top] = p + 1;
153                   stack[++top] = h;
154               }
155           } else
156           {
157               insertion_sort(arr, l, h);
158           }
159       }
160   }
161
```

# დასკვნა

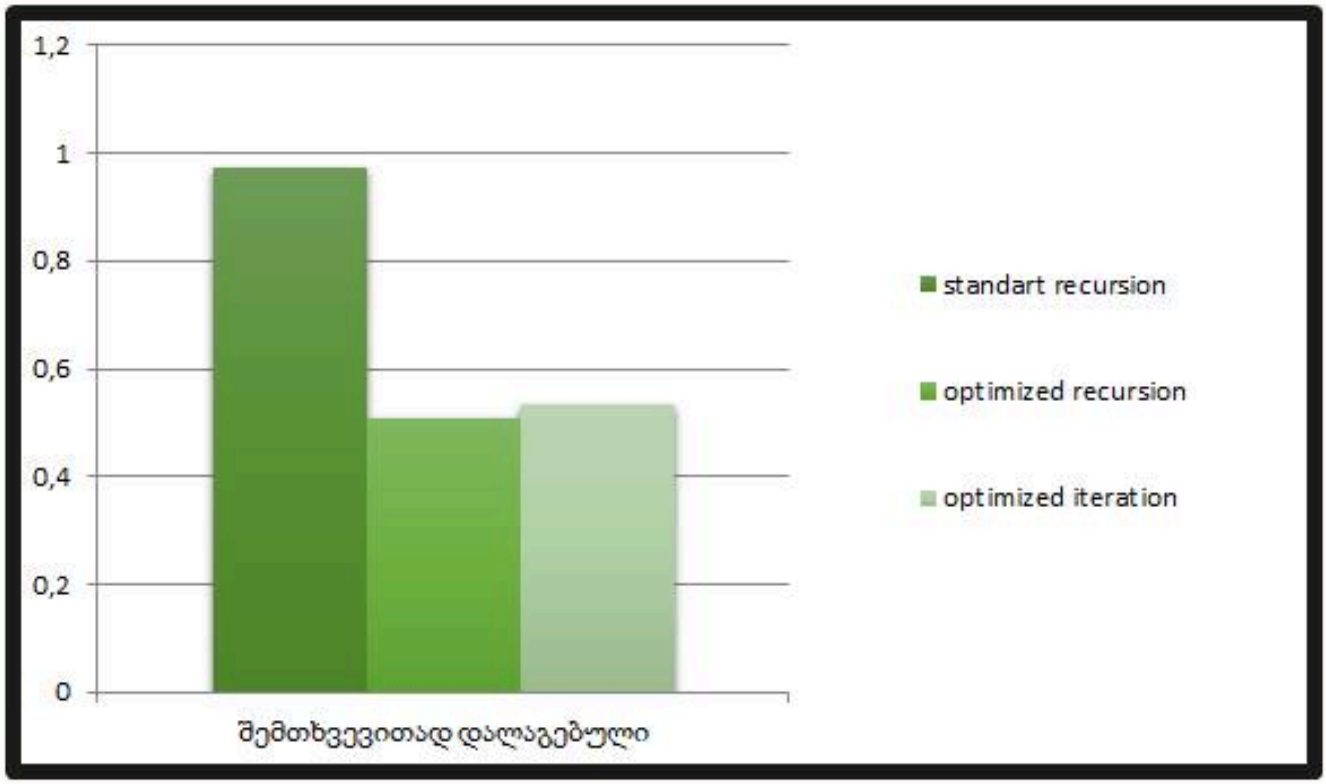
სამივე მეთოდის გამოყენებით ჩვენ შევძელით vector -ში ჩაყრილი 1.000.000 შემთხვევითი ელემენტისთვის დალაგება მოგვეხდინა 25-30 %-ით უფრო სწრაფად,ვიდრე სტანდარტული QuickSort ალგორითმით და ცუდი შემთხვევის დრო გავაუმჯობესეთ 300-1000-ჯერ და რაც უფრო დიდია მონაცემების რაოდენობა, მით უფრო დიდი განსხვავებაა სისწრაფეში.




# გამოყენებული ლიტერატურა

[1].ალექსანდრე გამყრელიძე: ალგორითმები და მონაცემთა სტრუქტურები, კურსის ლექციათა კონსპექტი.
[2].http://ru.wikipedia.org/wiki/Быстрая_сортировка
[3].Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001). Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill.

შესრულების დრო შემთხვევითად დალაგებული 1 000 000 მონაცემისთვის




შესრულების დრო revers-თი დალაგებული მონაცემებისთვის.

ოპტიმიზებული ალგორითმის მუშაობის დრო,სტანდარტულთან შედარებით იმდენად მცირეა (0.02 რეკურსიულის, 0.015 იტერაციულის),რომ დიაგრამაზე საერთოდ არ ჩანს.