Optimizarea consumului de carburant

Tema numarul 16

8 mai 2020

Dinu Ion George
Calculatoare Romana Anul I
Grupa 1.1 B

1 Enuntul problemei.

Optimizarea consumului de carburant.

Sa presupunem ca doriti sa conduceti intre doua orase aflate la o distanta foarte mare, de exemplu de la Craiova la Londra.Rezervorul plin al masinii dumneavoastra va permite o autonomie de m km. Aveti la dispozitie o harta care indica distantele pana la toate statiile de benzina pe care le puteti intalni in drumul dumneavoastra.

Fie aceste distante $d_1 < d_2 < d_3 < ... < d_n$ astfel incat d_i este distanta de la orasul de origine pana la distanta cu numarul i.

Se presupune ca exista n statii de benzina si ca distanta dintre oricare doua statii de benzina succesive este cel mult m.

Obiectivul este de a calatorii de la orasul de origine pana la orasul destinatie, realizand cat mai putine opriri pentru alimentarea cu benzina.

2 Algoritmi

2.1 Descrierea problemei

Din datele problemei stim ca avem o masina care poate sa se deplaseze pe o distanta de cel mult m km fara oprire. Parametrul m reprezinta mai exact numarul de km pe care se poate deplasa masina cu rezervorul plin.

De altfel suntem informati ca exista n benzinarii pe traseu.

Stim ca benzinariile sunt pozitionate la anumite distante fata de punctul de pornire, astfel incat distanta dintre oricare doua statii succesive este de cel mult m km.

Am presupus ca punctul A, notat cu d_0 si initializat cu 0, este punctul de pornire iar punctul B este destinatia, notata cu d_{n+1}. Astfel putem scrie inegalitatea :
$A = d_0 < d_1 < d_2 < d_3 < ... < d_n < d_{n+1} = B$.

De-a lungul rutei soferul va oprii doar la anumite statii, pana cand acesta va ajunge la destinatie.

Cerinta problemei este sa aflam numarul minim de opriri pe care il poate face soferul masinii pentru ajunge la destinatie.

2.2 Abordare

Pentru rezolvarea acestei probleme am folosti metoda algoritmilor "greedy". Aceasta presupune in a face o alegere lacoma (greedy) pentru a reduce problema in probleme mai mici(sub-probleme), repetand acest lucru pana cand aflam rezultatul cautat.

O subproblema este o problema similara, de dimensiune mai mica, a carei solutie contribuie la rezolvarea problemei initiale.

Putem sa facem trei alegeri diferite:

- oprim la cea mai apropriata statie
- oprim la cea mai departata statie la care putem ajunge cu un rezervor plin
- mergem pana ramanem fara combustibil

Alegerea potrivita este cea de a doua, mai precis, in problema de fata soferul masinii va opri doar la cea mai departata statie la care poate ajunge cu rezervorul plin.

Presupunem ca soferul masinii, pleaca din punctul A(d_0) spre destinatia noastra, oprind doar la cea mai departata statie la care poate ajunge,notata d_i pentru $i \in [1, n]$. Astfel reducem problema intr-o problema mai mica, d_i devenind noul nostru punct de plecare. Acum, problema noasta este de a ajunge de la noul nostru punct A(d_i) la B, cu un numar minim de opriri.

Repetand procedeul descris, dupa un numar de pasi vom afla nummarul minim de opriri pe care il va face soferul pentru a ajunge la destinatie.

2.3 Algoritmul numarul 1

```
ALGORITHM_1(n, m, gas_station)
 1  last_station = gas_station[0]
 2  station_number = 0
 3  refill = 0
 4  for iterator = 1 to n + 1
 5       if gas_station[iterator] − gas_station[iterator − 1] > m
 6            return 0
 7  while station_number <= n
 8       distance = 0
 9       while distance < m and station_number <= n
10            if distance >= gas_station[station_number] − last_station and
    m − distance >= gas_station[station_number + 1] − gas_station[station_number]
11                 station_number = station_number + 1
12            distance = distance + 1
13       last_station = gas_station[station_number]
14       if station_number <= n
15            count[refill] = station_number
16            refill = refill + 1
17            station_number = station_number + 1
18  // Returnam numarul de opriri
19  return refill
20  // Returnam indicele fiecarei benzinarii unde opreste soferul
21  return count
```

2.4 Algoritmul numarul 2

```
ALGORITHM_2(n, m, gas_station)
 1  station_number = 0
 2  refill = 0
 3  for iterator = 1 to n + 1
 4       if gas_station[iterator] − gas_station[iterator − 1] > m
 5            return 0
 6  while station_number <= n
 7       last_station = station_nummber
 8       while gas_station[station_number + 1] − gas_station[last_station]) <= m and
    station_number <= n
 9            station_number = station_nummber + 1
10       if station_number <= n
11            count[refill] = station_number
12            refill = refill + 1
13  // Returnam numarul de opriri
14  return refill
15  // Returnam indicele fiecarei benzinarii unde opreste soferul
16  return count
```

2.5 Analiza complexitatii algoritmilor

Analiza complexitatii unui algoritm are ca scop estimarea volumului de resurse de calcul necesare pentru executia algoritmului. Prin resurse intelegem:

- Spatiul de memorie necesar pentru stocarea datelor pe care le prelucreaza algoritmul
- Timpul necesar pentru executia tuturor prelucrarilor specificate in algoritm.

Complexitatea spatiului de memorie.

Pentru a afla complexitatea spatiului de memorie trebuie sa aflam volumul de memorie necesar algoritmului pentru a functiona.

Algoritmul numarul 1

In algoritmul de fata avem variabilele last_station, station_number, refill si distance. Fiind numere intregi, putem spune ca fiecare ocupa un volum de 4 bytes din memorie, fiind o valoare constanta. De altfel n, m, si vectorul gas_station fac parte din spatiul de intrare, fiind argumentele functiei si ocupand de asemenea un volum constant.

Putem observa faptul ca in algoritmul de fata este creat un nou vector temporar numit : count. Acesta face parte din spatiul auxiliar folosit de program in timpul executiei. Vectorul count are un numar de "refill" elemente.

In cazul de fata avem urmaroarea ecuatie, ce reprezinta memoria folosita, unde n reprezinta spatiul auxiliar, iar k_1 reprezinta valorile constante:

$$f(n) = k_1 + n$$

Aceasta functie poate fi scrisa si sub forma : $O(k_1 + n)$, care prin conventie este notat cu $O(n)$.

In concluzie complexitatea spatiului de memorie al algoritmului de fata este $O(n)$.

Algoritmul numarul 2

Asemanator cu primul algoritm, si in acest caz avem variabilele station_number si refill, doua variabile care ocupa un volum constant de memorie. N, m, si vectorul gas_station fac parte din spatiul de intrare, fiind argumentele functiei.

Vectorul count reprezinta spatiul auxiliar folosit de program.

In concluzie complexitatea spatiului de memorie al algoritmului de fata este in mod asemanator $O(k_1 + n)$, adica $O(n)$.

Complexitatea timpului de executie.

Timpul de executie al unui algoritm reprezinta durata de timp de la pornirea programului pana la afisarea rezultatelor.

Algoritmul numarul 1

In cel mai defavorabil caz, avem urmator tabel, ce reprezinta numarul de repetitii al fiecarei linii de cod:

Operatie	*Cost*	*Nr.Repetari*
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	$n+2$
5	c_5	$n+1$
6	c_6	-
7	c_7	$n+2$
8	c_8	$n+1$
9	c_9	$(n+1)*(m+1)$
10	c_{10}	$(n+1)*m$
11	c_{11}	$(n+1)*m$
12	c_{12}	$(n+1)*m$
13	c_{13}	$n+1$
14	c_{14}	$n+1$
15	c_{15}	n
16	c_{16}	n
17	c_{17}	n

Pentru a calcula timpul de executie al algoritmului prezent trebuie sa urmarim mai precis variabilele: station_number, refill si distance.

Consideram constant timpul de executie al liniilor $1 \to 3$.

Variabila iterator din instructiunea **for** de la linia 4 se schimba de cel mult $n+1$ ori.

Instructiunea **while** este o instructiune repetitiva, astfel variabila station_number va lua cel mult n+1 valori. In bucla avem doua variabile, distance si last_station, o a doua instructiune **while** si o instructiune conditionala **if**.

Tot ce este in interiorul primei bucle(i.e linia $7 \to 17$), in afara de bucla interioara (i.e linia $9 \to 11$), se schimba de n ori. Deci variabila refill ia cel mult n valori.

Variabila distance odata ce intra in bucla interiora se schimba de cel mult m ori. In acest mod variabila de fapt se va schimba de $n*m$ ori.

Astfel putem scrie functia:

$f(n) = c_1+c_2+c_3+2*c_4+c_5+2*c_7+c_8+c_9+c_{13}+c_{14}+n[c_4+c_5+c_7+c_8+m(c_9++c_{10}+c_{11}+c_{12})+c_{13}+c_{14}+c_{15}+c_{16}+c_{17}]+m(c_9+c_{10}+c_{11}+c_{12}) \iff$

$f(n) = k_1 + n(k_2 + m * k_3) + k_4 * m$

Functia este echivalenta cu expresia $f(n) = O(k_1+n(k_2+m*k_3)+k_4*m) \iff O(n*m+m) \iff O(n*m)$.

In concluzie in cazul cel mai defavorabil, algoritmul va functiona in $O(n*m)$.

Algoritmul numarul 2

In cel mai defavorabil caz avem urmator tabel, ce reprezinta numarul de repetitii al fiecarei linii de cod:

Operatie	*Cost*	*Nr.Repetari*
1	c_1	1
2	c_2	1
3	c_3	$n+2$
4	c_4	$n+1$
5	c_5	-
6	c_6	$n+2$
7	c_7	$n+1$
8	c_8	$n+1$
9	c_9	$n+1$
10	c_{10}	$n+1$
11	c_{11}	n
12	c_{12}	n

Pentru a afla timpul de executie al acestui algoritm, trebuie sa urmarim de cate ori se schimba variabilele station_number si refill.

In mod asemanator cu primul algoritm, consideram constant timpul de executie al liniilor 1 si 2, iar variabila iterator se va schimba de cel mult $n+1$ ori.

Variabila station_number din prima bucla va lua cel mult $n+1$ valori. In interiorul buclei avem o a doua instructiune **while**, o variabila last_station si o instructiune conditionala **if**. Observam faptul ca variabila station_number creste, in bucla din interior,dupa fiecare pas cu o unitate. De altfel variabila refill creste cu o unitate la fiecare pas, schimbandu-se de cel mult n ori.

Putem scrie functia:

$f(n) = c_1 + c_2 + 2 * c_3 + c_4 + 2 * c_6 + c_7 + c_8 + c_9 + c_{10} + n(c_3 + c_4 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12}) \Longleftrightarrow$

$f(n) = k_1 + n * k_2$

Functia este echivalenta cu expresia $f(n) = O(k_1 + n * k_2)$ adica $O(n)$.

In concluzie in cazul cel mai defavorabil, algoritmul va functiona in $O(n)$.

3 Date experimentale

In limbajul C, entru a genera date random dintr-un interval dat, trebuie folostita functia **rand** impreuna cu functia **srand**.

Functia **rand**,are o limita in legatura cu intervalul de valori pe care le poate genra. Aceasta va genera valori aleatorii doar din intervalul $[0, RAND\_MAX]$, unde $RAND\_MAX$ este o valoare ce variaza in functie de sistemul de operare, si este cel putin egala cu 32767 ($0x7FFF$).

Pentru a mari acest domeniu am realizat urmatorul algoritm:

RAND_RANGE(n)
1 $x = rand()$
2 $x = x << 15$
3 $x = x \oplus rand()$
4 $x = x \bmod n$
5 **return** x

In algoritmul de mai sus, **n** reprezinta limita superioara a inervalului de unde se pot genera valori random.

Algoritmul functioneaza in modul urmator: Functia rand este chemata de mai multe ori. Cum aceasta ne va returna o valoare de cel putin 15 biti, ne deplasam 15 biti la stanga dupa care aplicam XOR, iar valoarea returnata este data de restul impartiri lui x cu limita superioara.

Pentru a genera date de intrare netriviale cu valori mari, am folosit algoritmul de mai sus impreuna cu urmatorul algoritm, care pentru problema propusa genereaza aleator, destinatia unde trebuie sa ajunga soferul, numarul de kilometri pe care ii poate parcurge masina fara a opri si numarul de benzinarii de pe traseu, impreuna cu pozitia acestora fata de origine. Aceste date pot lua valori intre 10^3 si 10^6. Am ales acest interval pentru ca algoritmul sa genereze datele rapid. Cu cat limita superioara este mai mare cu atat timpul de executie va creste.

Algoritmul pentru generarea datelor de intrare este:

```
RANDOM_GENERATOR(no_test)
 1  FUEL = 700
 2  MAX = 1000000
 3  for case_number = 1 to no_test
 4       m = 400 + rand_range(FUEL − 400 + 1)
 5       destination = 600 + m + rand_range(MAX − (m + 600) + 1)
 6       n_must_fill = ⌈destination/m⌉
 7       n = n_must_fill + rand_range(destination/10)
 8       arr[0] = 0
 9       arr[n + 1] = destination
10       is_used[] = {0}
11       is_used[arr[0]] = 1
12       for iterator = 1 to n
13            r_value = arr[iterator − 1] + m − rand_range(100)
14            if r_value < destination and is_used[r_value] = 0
15                 arr[iterator] = r_value
16            else
17                 while True
18                      r_value = 1 + rand_range(destination − 1)
19                      if is_used[r_value] = 0
20                           break
21                 arr[iterator] = r_value
22                 is_used[r_value] = 1
23       Sort(arr)
24       n = n − 1
25       test(n, m, arr)
26       return n
27       return m
28       return arr
```

Observam ca generatorul de teste de mai sus are ca argument variabila *no_test*. Valoarea acesteia reprezinta numarul de teste pe care le va genera algoritmul.

Variabila m declarata in linia 4 reprezinta numarul de kilometri pe care ii poate parcurge soferul fara a opri. Aceast numar este generat aleatoriu si poate lua orice valoare din intervalul $[400, FUEL]$. Limita superiaora a intervalului de unde poate fi ales m este valoarea constantei FUEL.

Variabila destination reprezinta destinatia soferului masinii. Aceasta este situata la un numar de kilometri ales aleatoriu din intervalul $[m + 600, MAX]$, unde MAX reprezinta limita superioara a intervalului de unde poate fi alesa valoarea variabilei. Limita inferioara a intervalului este $m + 600$ deoarece dorim ca destinatia sa fie pozitionata la cel putin 1000 de kilometri fata de origine. Cum m poate avea cel putin valoarea 400, atunci limita inferioara a variabilei destination va fi $400 + 600 = 1000$.

n_must_fill este un numar orientativ de benzinarii la care trebuie sa opreasca soferul. Sa presupunem ca in cel mai bun caz, benzinariile se afla la fix **m** km departate una de cealalta. Astfel, daca stim ca destinatia noasta se afla la **destination** km departare, atunci putem aproxima ca o sa trebuiasca sa oprim la $destination/m$ benzinarii. Vom numi acest numar de benzinarii, benzinariile sigure.

Numarul benzinariilor de pe traseu este un numar cuprins intre n_must_fill si $n\_must\_fill + destination/10$. n_must_fill reprezinta un numar presupus de statii la care trebuie sa opreasca. Pe langa acestea mai exesista si statii peste care soferul o sa treaca mai departe. Vom numi acest numar de statii, statiile optionale. Numarul acestora este ales aleatoriu din intervalul $[0, destination/10]$. Am ales acest interval pentru a evidentia optimizarea consumului de carburant pe care il poate face soferul.

Pozitiile statiilor de alimentare sunt stocate in vectorul **arr**. Astfel in liniile 8 si 9, atribuim primei pozitii din acest vector, valoarea 0,reprezentand locul de plecare / originea, si ultimei pozitii $(n + 1)$ numarul de km pana la destinatie.

Pentru a genera valori unice pentru pozitiile benzinariilor, am folosit un nou vector,initializat cu 0 pe toate pozitiile, numit is_used, ce realizeaza o permutare. Astfel pozitiile statiilor reprezinta indicii acestui vector. Daca o pozitie este folosita atunci aceasta este memorata in vector cu valoarea 1. In linia 11 marcam punctul de plecare (originea) ca fiind folosit.

In liniile $12 \rightarrow 22$ este prezentat algoritmul pentru generarea celor n statii de alimentare. In linia 13, este declarata variabila r_value care memoreaza pozitiile statiilor sigure la care poate sa opreasca soferul, acestea fiind situate la cel mult m km departare una de cealalta. Valorile acestora sunt formate din suma algebrica dintre pozitia statiei anterioare si m. Deoarece am dorit ca statiile sa aiba pozitii aleatorii pe distanta ceruta am scazut rezultatul cu un numar aleatoriu dintre 0 si 100. Atunci cand este completata distanta ceruta cu un numar de benzinarii sigure, pot fi generate si celelalte pozitii ale benzinariilor ramase.

Deoarece pozitiile benzinariilor sunt generate aleatoriu, acestea trebuie sortate in ordine crescatoare.

In linia cu numarul 13, se genereaza in ordine crescatoare pozitiile benzinariilor sigure. Generarea acestora se opreste odata ce ultima pozitie generata trece de valoarea **destination**. Exista posibilitatea ca pozitia ultimei statii sa fie generata astfel incat distanta dintre aceasta si destinatie sa fie prea mare (mai mare ca m) . Pentru a evita cazul in care se intampla acest lucru scadem numarul benzinariilor la $n-1$ iar destinatia se muta la pozitia ultimei benzinarii generate.

Algoritmul de mai sus genereaza aleatoriu date de intrare pentru testarea problemei noastre. Acesta genereaza o lista de n valori ce reprezinta pozitiile statiilor de alimentare de pe traseu, cat si un numar aleatoriu m, ce reprezinta distanta ce o poate parcurge soferul fara sa opreasca.

Pentru ca datele de intrare sa fie relevante pentru problema noasta, acestea trebuie sa indeplineasca 3 conditii:

- distanta dintre doua pozitii consecutive a benzinariilor trebuie sa fie de cel mult m km
- pozitiile acestora trebuie sa fie valori unice(nu pot exista doua benzinarii pe aceeasi pozitie)
- acestea trebuie sa fie ordonate crescator

Cele trei conditii enumerate mai sus sunt testate in algoritmul nostru folosind functia **test**, in linia 25. Functia urmatoare se bazeaza pe functia assert, si este implementata in modul urmator:

```
TEST(n, m, arr)
1  for iterator = 1 to n + 1
2        assert(arr[iterator] − arr[iterator − 1] <= m)
3        assert(arr[iterator]! = arr[iterator − 1])
4        assert(arr[iterator] > arr[iterator − 1])
```

In limbajul Python, pentru ca a functiona intr-un timp de executie cat mai mic, am modificat algoritmul folosind functiile predefinite din modulele limbajului.

In comparatie cu limbajul C, Python este mai lent din punct de vedere a timpului de executie. Astfel este recomandat sa fie folosite functiile deja scrise din modulele limbajului.

Algoritmul pentru generarea pozitiilor statiilor de alimentare arata in limbajul Python arata in modul urmator:

```
1  arr1 = [0]
2  r_value = arr1[0]
3  while r_value < destination
4        r_value = r_value + m − randint(1, 100)
5        arr1.append(r_value)
6  arr2 = list(sample(range(1, destination − 1), n − 1))
7  arr2.append(destination)
8  arr = list(set(arr1)|set(arr2))
9  arr.sort()
10 n = len(arr) − 2
```

Fata de codul precedent in acesta generam benzinariile sigure in linia $3 \rightarrow 5$ si le memoram in lista **arr1**, dupa care generam pozitiile celorlalte benzinarii in lista **arr2** (linia 6). Facem reuniunea celor doua liste si notam noua lista creata cu **arr**, dupa care sortam o sortam in ordine crescatoare. Deoarece cele doua multimi, arr1 si arr2, contin punctul de plecare si destinatia, in linia cu numarul 10, scadem 2 din numarul de elemente al listei. Aceasta valoare este reprezentata de pozitia de plecare (d_0) si destinatia (d_{n+1}).

Functiile din modulul random din Python, pot genera valori mari/netriviale, astfel nu este necesar folosirea functiei *rand_range* de la pagina 8.

Conform celor de mai sus, putem sustine faptul ca datele generate de algoritmul prezentat sunt semnificative pentru testarea problemei noastre.

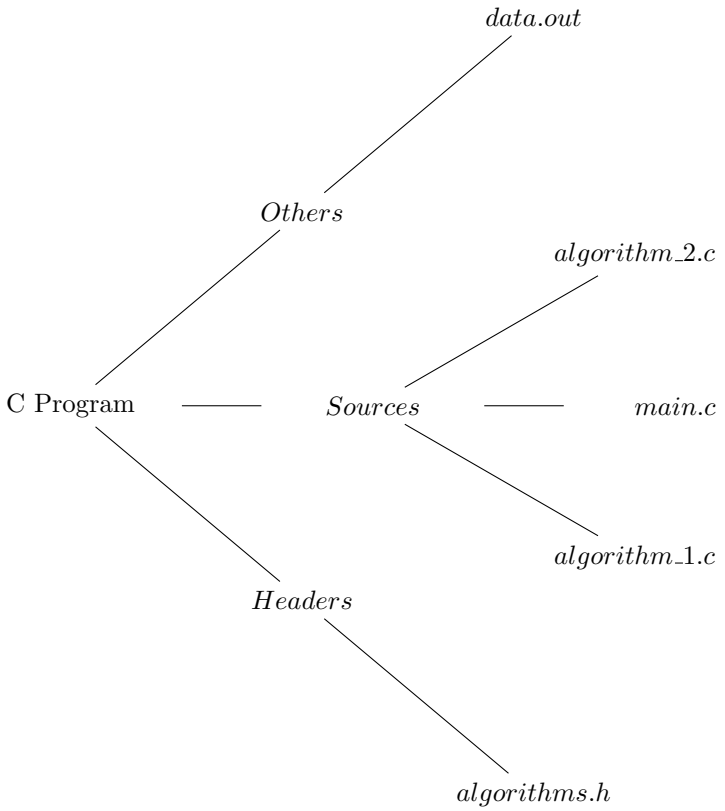
4 Proiectarea aplicatiei experimentale

Structura de nivel inalt a aplicatiei

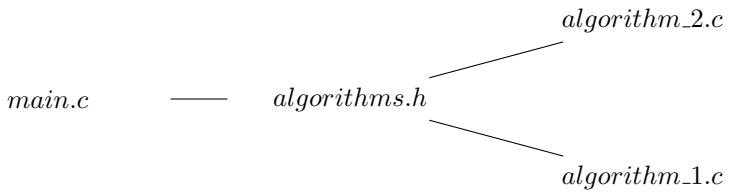
Aplicatia dezvoltata pentru rezolvarea problemei are o structura modulara. Fiecare modul contine functii ce ajuta la rezolvarea problemei.

Astfel pentru rezolvarea problemei in limajull C, programul este divizat in mai multe fisiere.

Structura acestuia arata in felul urmator:



Fiecare antet, contine la randul lui functiile ce sunt scrise in fisierele sursa. Intre fisierele sursa si antete exista urmatoarea legatura:



Toate functiile scrise sunt apelate in fisierul main.c cu ajutorul antetului. main.c reprezinta fisierul principal ce ruleaza programul.

In fisierul Others, este prezent fisierul ce contine datele de iesire ale celor doi algoritmi.

Functiile programului sunt impartite in fisiere in felul urmator:

- *main.c*
- *algorithm_1.c*
  - *algorithm_1*$(n, m, gas\_station, case\_number)$
- *algorithm_2.c*
  - *algorithm_2*$(n, m, gas\_station, case\_number)$

Functiile *algorithm_1* si *algorithm_2* reprezinta algoritmii creati pentru rezolvarea problemei. Argumentele celor doua functii sunt : n - numarul de benzinarii ce apar pe traseu, m - numarul de km pe care ii poate parcurge soferul fara a opri, *gas_station* - vectorul ce contine pozitiile statiilor de benzinarie de pe traseu, iar *case_number* reprezinta numarul testului care este executat.

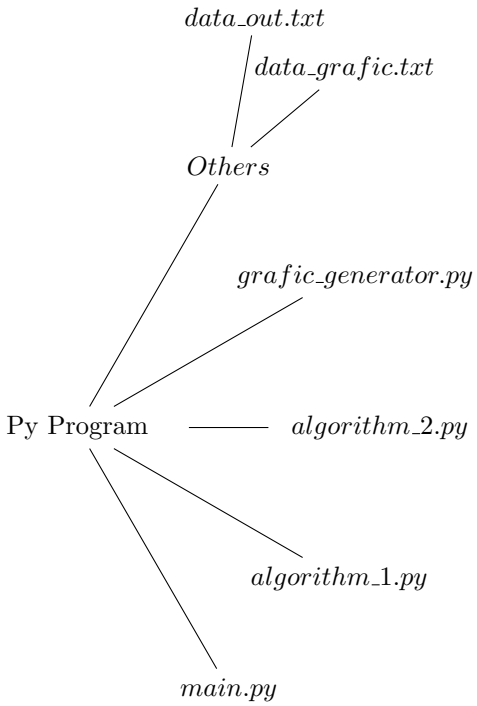
Cele doua functii ce contin algoritmii propusi sunt apelate in fisierul *main.c*.

Datele de intrare sunt citite din fisierul *data.in*, localizat in structura directorului $GENERATOR\_TESTE$.

Dupa ce sunt executati algoritmii datele obtinute sunt scrise in fisierul data.out. De fiecare data cand rulam programul, ambele metode sunt executate pentru fiecare test. Acesta reprezinta un mod prin care ne putem verifica daca datele obtinute sunt relevante. Putem observa cu usurinta daca exista o diferenta intre datele returnate de prima metoda fata de datele returnate de cea de a doua.

In limbajul Python structura aplicatiei este aproximativ identica, singura diferenta este faptul ca nu exista antete, astfel ca putem importa functiile direct din fisirele dorite.

Structura programului arata in felul urmator:



Functiile programului sunt impartite in fisiere in felul urmator:

- *main.py*
- *algorithm_1.py*
  - *algorithm_1*$(n, m, gas\_station, case\_number)$
- *algorithm_2.py*
  - *algorithm_2*$(n, m, gas\_station, case\_number)$
- *grafic_generator.py*
  - *grafic_1*()
  - *grafic_2*()
  - *grafic_3*()

Functiile programului sunt apelate in fisierul *main.py*. Acesta reprezinta fisierul principal ce ruleaza programul.

Pentru a interpreta mai usor datele obtinute prin rularea algoritmilor, putem crea tabele sau grafice. Astfel ca am creat un algoritm care creaza in mod automat 3 grafice. Acest algoritm este descris in fisierul $grafic\_generator.py$. Acesta contine trei functii $grafic\_1$, $grafic\_2$ si $grafic\_3$. Prima functie realizeaza un grafic cu toate pozitiile benzinariilor unde a oprit soferul pe traseul propus, iar a doua si a treia functie realizeaza doua grafice pentru a observa mai usor diferenta dintre numarul de benzinarii la care a oprit soferul si numarul total de benzinarii.

Aceste functii nu sunt apelate in fisierul *main.py*, utilizatorul fiind cel care alege daca vrea sa le execute sau nu. Pentru executarea celor trei functii se va executa fisierul$grafic_generator.py$.

Pentru realizarea graficelor explicate mai sus s-au folostit datele din fisierul $data\_grafic.txt$. Acestea reprezinta datele primului test generat de generatorul de teste.

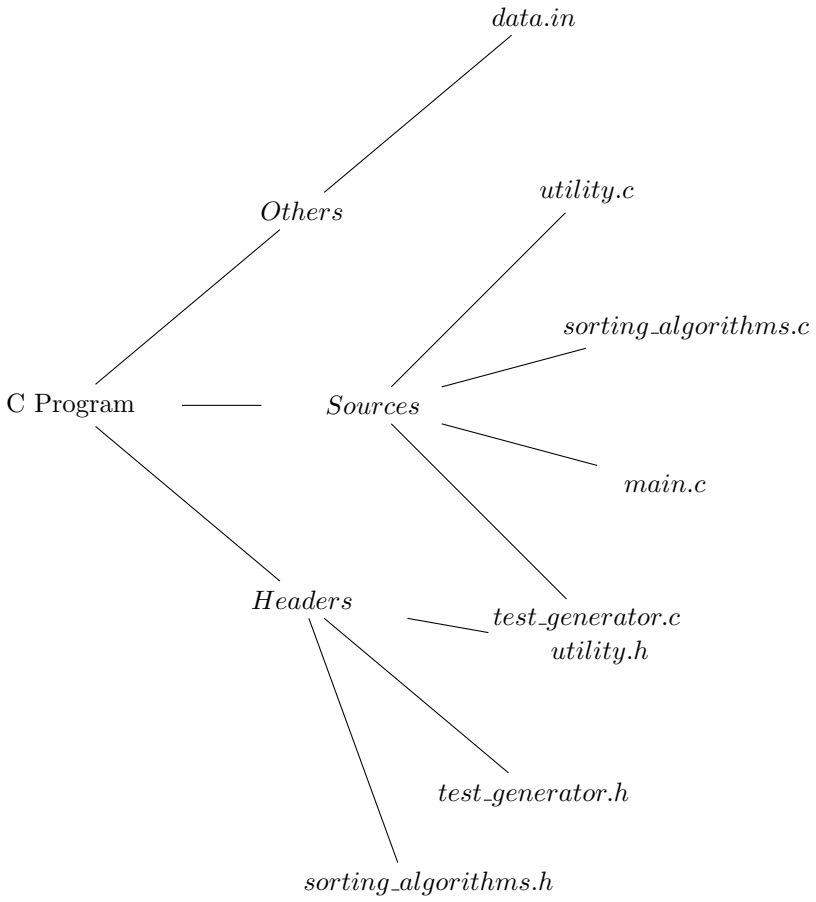
Functiile $algorithm\_1$ si $algorithm\_2$ reprezinta algoritmii creati pentru rezolvarea problemei. Argumentele functiilor sunt aceleasi ca cele a algoritmilor din limbajul C. Asemanator cu proiectul scris in limbajul C, datele de intrare sunt citite din fisierul *data.in* localizat in structura directorului $GENERATOR\_TESTE$.

Rezultatele obtinute in urma executatiei algoritmilor de mai sus sunt memorate in fisierul $data\_out.txt$.

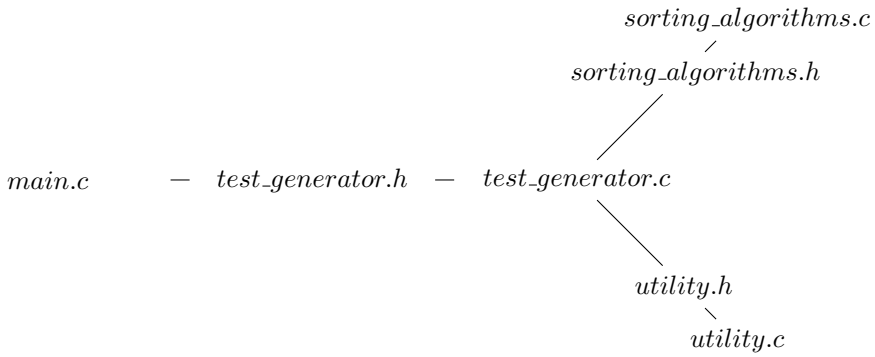
Structura Generatorului de teste

Pentru generarea datelor de intrare pentru testarea celor doi algoritmi implementati a fost creat un generator de teste. Acesta genereaza aleatoriu date de intrare relevante si netriviale pentru testarea algoritmilor.

Structura acestui generator de teste este urmatoarea:



Functiile scrise in interiorul fisierelor programului sunt apelate in fisierul main.c. Relatia dintre fisierele programului este urmatoarea:



Functiile programului sunt impartite in fisiere in felul urmator:

- *main.c*
- *test_generator.c*
  - *random_generator*(*no_test*)
- *sorting_algorithms.c*
  - $mergeSort(arr, left, right)$
  - $merge(arr, left, middle, right)$
- *utility.c*
  - *rand_range*(n)
  - *test*(n, m, arr)

Functia *merge_sort* este apelata in fisierul *test_generator.c* pentru a sorta vectorul ce contine pozitiile statiilor de benzinarie. Argumentul functiei este reprezentat de : arr - vectorul care trebuie sortat, left - limita inferioara, si right - limita superioara. Functia *merge* este o componenta a functiei *merge_sort*. Aceasta functie ajuta in procesul de sortare a datelor.

Functia *random_range* este apelata in fisierul *test_generator.c* si folosita pentru a genera valori netriviale (foarte mari). Aceasta este o componenta pentru functia *random_generator*.

Functia *test* este apelata in fisierul *test_generator.c* si este folosita pentru a testa daca volorile generate anterior sunt relevante pentru problema de fata. Argumentul functiei este reprezentat de variabila n - numarul de statii de alimentare generate, m - numarul de km pe care ii poare parcurge soferul fara a opri si arr - vectorul ce contine pozitiile celor n statii. Aceasta este o componenta a functiei *random_generator*.

Functia *random_generator* este apelata in fisierul *main.c* pentru a genera teste aleatorii pentru algorimii creati. Argumentul functiei este reprezentat de variabila *no_test*, ce reprezinta numarul de teste generate pentru problema noastra.

Generatorul de teste este scris si in limbajul python. Acesta spre deosebire de cel de mai sus are o structura triviala, folosindu-se in special functiile deja scrise din modulele limbajului.

Acesta este localizat in fisierul $GENERATOR\_TESTE\_PY$, datele returnate fiind memorate in fisierul *data.in* din interiorul directorului.

Datele de intrare/iesire

Datele de intrare pentru testarea algoritmilor sunt create cu ajutorul generatorului de teste. Acesta returneaza in mod aleatoriu o serie de date ce reprezinta: numarul de statii de benzinarie pe care le va intampina soferul pe traseu, numarul de km pe care ii poate parcurge fara a opri la o benzinarie, si un vector ce contine pozitia fiecarei statii pe traseul propus.

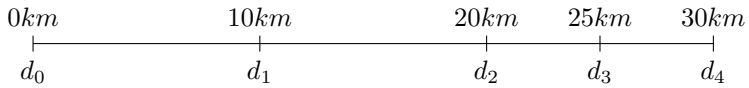
Datele de iesire sunt generate atat de $algorithm\_1$ cat si de $algorithm\_2$ si memorate in fisierul *data.out/data_out.txt*. Cele doua functii reprezinta doua metode prin care poate fi rezolvata problema.

Singura diferenta dintre cei doi algoritmi este timpul de executie si complexitatea spatiului de memorie, $algorithm\_2$ fiind metoda mai eficienta/optima.

Pentru a verifica faptul ca datale de iesire sunt relevante / corecte, am testat cei doi algoritmi creati intr-un numar de cazuri diferite:

Cazul 1

Presupunem ca soferul poate sa mearga $m = 10$ km fara sa opreasca la o statie de benzinarie. Pe traseu exista 3 statii iar destinatia se afla la 30 km departare fata de punctul de plecare.

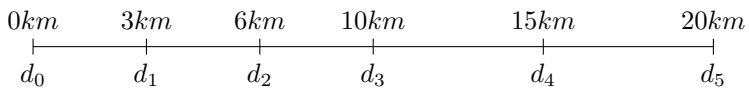


Acesta este cazul de baza, in care statiile de benzinarie sunt situate la distante fixe(d_1, d_2), iar soferul trebuie sa treaca peste benzinaria d_3.

Daca algoritmul o sa functioneze soferul o va opri de doua ori, la benzinaria d_2 respectiv d_3.

Cazul 2

Presupunem ca soferul poate sa mearga $m = 10$ km fara sa opreasca la o statie de benzinarie. Pe traseu exista 4 statii iar destinatia se afla la 20 km departare fata de punctul de plecare.

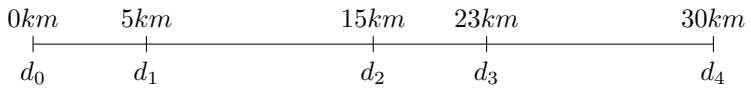


In cazul de fata, benzinaria la care trebuie sa opreasca soferul este la mijlocul traseului. Astfel observam ca soferul are optiunea de a oprii atat inainte de

acesta benzinarie, cat si dupa aceasta. Pentru a isi optimiza traseul acesta, va opri o singura data la benzinaria d_3.

Cazul 3

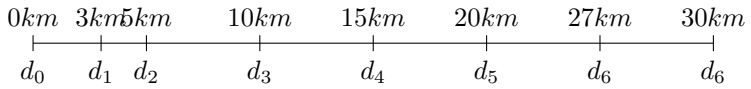
Presupunem ca soferul poate sa mearga $m = 10$ km fara sa opreasca la o statie de benzinarie. Pe traseu exista 3 statii iar destinatia se afla la 30 km departare fata de punctul de plecare.



In cazul de fata, soferul masinii nu poate ajunge la benzinaria d_2 plecand din origine. Acesta desi are o cantitate mare de combustibil trebuie sa opreasca la benzinaria d_1, dupa care poate sa isi parcurga traseul. Soferul masinii va trebui sa opreasca de doua ori in cazul de fata.

Cazul 4

Presupunem ca soferul poate sa mearga $m = 10$ km fara sa opreasca la o statie de benzinarie. Pe traseu exista 6 statii iar destinatia se afla la 30 km departare fata de punctul de plecare.

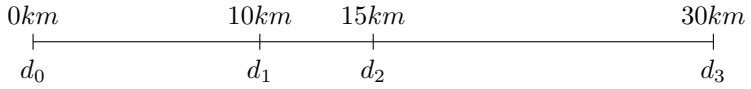


In cazul de fata pozitiile statiilor sunt alese in mod aleatoriu, astfel soferul va stationa doar la statiile ce sunt sitate la distanta maxima pe care o poate parcurge fara sa opreasca. Acesta va opri la statia d_3 si d_5, sitate la 10 km, respectiv 20 km, fata de punctul de plecare.

Cazul 5 (EXCEPTIE)

Desi in problema de fata ne este specificat faptul ca benzinariile trebuie sa aiba o distanta mai mica sau egala cu m, nu trebuie neglijat cazul in care benzinariile sunt la distante prea mari una de cealalta. In acest caz soferul nu poate ajunge la destinatie.

Presupunem ca soferul poate sa mearga $m = 10$ km fara sa opreasca la o statie de benzinarie. Pe traseu exista 2 statii iar destinatia se afla la 30 km departare fata de punctul de plecare.

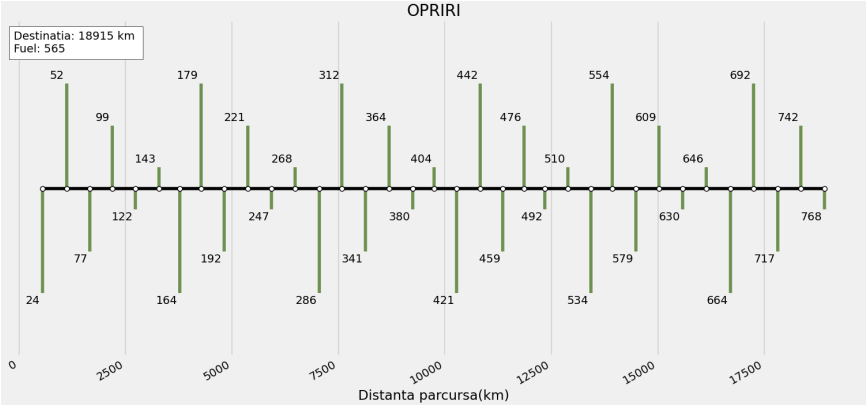


Soferul masini, in cazul de fata va face doua opriri, odata la statia d_1 si odata la statia d_2, dupa care nu va putea ajunge la destinatia, deoarece distanta dintre d_3 si d_2 este prea mare.

Algoritmul descris in paginile precedente, indeplineste conditiile explicate mai sus, deci putem spune ca datele de iesire sunt relevante.

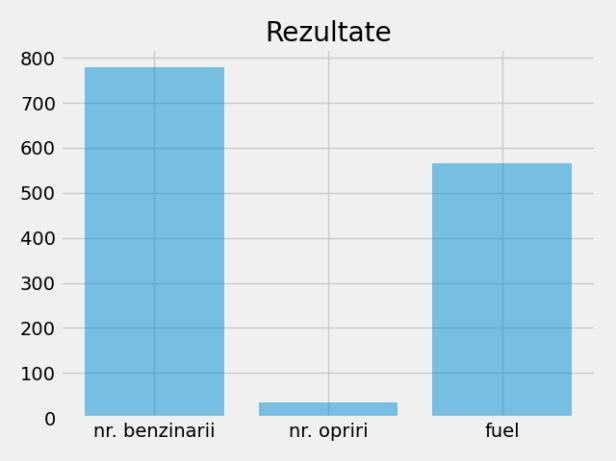
5 Rezultate si concluzii

Pentru interpretarea datelor de iesire se pot efectua tabele sau grafice. Astfel pentru realizarea acestora, am implementat un cod, scris in Python, folosind modulul Matplotlib. Fisierul ce contine codul este atasat proiectului din Python. Algoritmul respectiv realizeaza automat un grafic cu toate statiile de benzinarie unde a oprit soferul soferul masinii si un grafic in care este prezentata diferenta dintre numarul de benzinarii intalnite pe traseu si numarul de opriri pe care le face soferul. Presupunem aleatoriu ca soferul are un traseu de 18915 km, si o autonomie de 565 km. Pentru cazul de fata putem interpreta datele de iesire in felul urmator:



Barile verticale din figura de mai sus reprezinta indicele statiei unde a oprit soferul. Figura prezentata reprezinta traseul pe care la efectuat soferul masinii pe o distanta de 18915 km, avand o autonomie de 565 km.

Putem evidentia faptul ca soferul si-a optimizat traseul oprind doar la anumite statii cu ajutorul graficului urmator, ce prezinta difenta dintre numarul de benzinarii de pe traseu si numarul de benzinarii la care a oprit acesta.



O altfel de reprezentare este urmatoarea:

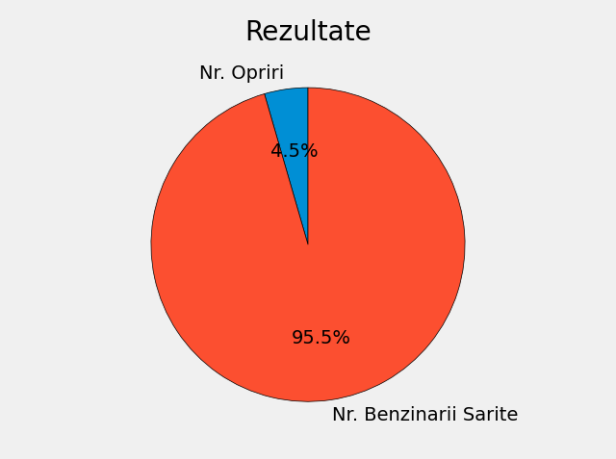


Figura precedenta prezinta cat la suta din numarul total de benzinarii, au reprezentat opririle.

Din punct de vedere a performantei putem spune ca al doilea algoritm este mai optim in comparatie cu primul.

Un prim argument prin care putem dovedi acesta afirmatie, reprezinta diferenta dintre timpul de executie al acestora.

In sectiunea 2.5 am demonstrat faptul ca primul algoritm functioneaza intr-un timp $O(n * m)$, iar cel de al doilea functioneaza in $O(n)$.

Pentru a evidentia diferenta dintre performantele acestor doua metode de rezolvare a problemei putem rula diferite teste cu date de intrare din ce in ce mai mari.

Pentru algoritmii scrisi in limbajul C avem urmatoarele rezultate:

Test	Distanta(km)	Nr. Benzinarii	Timpul CPU (s)
nr.1 *Algorithm_1*	800298	62752	0.010
nr.1 *Algorithm_2*	800298	62752	0.007
nr.2 *Algorithm_1*	278875	6648	0.006
nr.2 *Algorithm_2*	278875	6648	0.005
nr.3 *Algorithm_1*	191459	18886	0.007
nr.3 *Algorithm_2*	191459	18886	0.005
nr.4 *Algorithm_1*	648508	64759	0.008
nr.4 *Algorithm_2*	648508	64759	0.007
nr.5 *Algorithm_1*	516274	14575	0.007
nr.5 *Algorithm_2*	516274	14575	0.006
nr.6 *Algorithm_1*	724555	50956	0.009
nr.6 *Algorithm_2*	724555	50956	0.006

Putem observa faptul ca timpul de executie al programului este direct proportional cu dimensiunea datelor de intrare. Astfel, cum datele de intrare au valori din ce in ce mai mari, timpul de executie al algoritmilor va creste.

De altfel putem observa diferenta dintre timpul de executie al primului algoritm si celui de al doilea.

Media timpurilor de executie a primului algoritm realizata pe 6 teste aleatorii este 0.0078 s, iar media timpului de executie a celui de al doilea algoritm pentru aceleasi teste este 0.006 s.

Conform datelor din tabelul de mai sus putem spune ca cel de al doilea algoritm este mai optim din punct de vedere al timpului de executie.

De altfel putem face diferenta si intre limajele de programare folosite.

In limbajul Python, avem urmatoarele 6 rezultatele pentru aceleasi date de intrare :

Test	Distanta(km)	Nr. Benzinarii	Timpul CPU (s)
nr.1 *Algorithm_1*	800298	62752	0.146
nr.1 *Algorithm_2*	800298	62752	0.023
nr.2 *Algorithm_1*	278875	6648	0.054
nr.2 *Algorithm_2*	278875	6648	0.003
nr.3 *Algorithm_1*	191459	18886	0.039
nr.3 *Algorithm_2*	191459	18886	0.007
nr.4 *Algorithm_1*	648508	64759	0.126
nr.4 *Algorithm_2*	648508	64759	0.023
nr.5 *Algorithm_1*	516274	14575	0.094
nr.5 *Algorithm_2*	516274	14575	0.007
nr.6 *Algorithm_1*	724555	50956	0.137
nr.6 *Algorithm_2*	724555	50956	0.019

In cazul de fata putem observa o diferenta mai mare intre timpul de executie a celor doi algoritmi. Astfel media timpului de executie pentru primul algoritm, pentru testele de mai sus este 0.099 s, iar pentru cel de al doilea algoritm este 0.042 s. De data aceasta diferenta dintre timpul de executie a celor doi algoritmi este mai mare, ceea ce ne confirma faptul ca cel de al doilea algoritm este mai eficient.

Conform datelor din tabelele de mai sus, algoritmii scrisi in libajul C sunt mult mai optimi din punct de vedere a timpului de executie, fata de cei scrisi in limbajul Python.

Aceasta diferenta nu este foarte mare, dar odata ce valorile datelor de intrare cresc, aceasta diferenta devine sesizavila.

De exemplu odata ce algoritmul din Python va primi date, ce depasesc 10^6, acesta va avea un timp de executie prea mare, ceea ce nu este convenabil in cazul in care dorim sa rulam mai multe teste in acelasi timp.

Putem sesiza diferenta dintre cele doua limbaje de programare cel mai bine prin examinarea timpului de executie al generatorului de teste. Deoarece am dorit ca timpul de executie sa nu fie exagerat de mare, am ales ca limita superioara valoarea 10^6. Astfel avem urmatoarele 6 teste, generate aleatoriu in limbajul Python:

Test *Python*	Nr. Benzinarii Generate	Timpul CPU (s)
nr.1	7962	0.011
nr.2	48106	0.064
nr.3	51027	0.076
nr.4	45347	0.068
nr.5	18112	0.023
nr.6	13705	0.019

Atunci cand alegem valori mai mari ca 10^6 timpul de executie creste exponential. De exemplu avem urmatoarele valori :

Test *Python*	Nr. Benzinarii Generate	Timpul CPU (s)
nr.1	341684	0.645
nr.2	248018	0.418
nr.3	5397651	10.353
nr.4	2974736	6.678
nr.5	3200229	6.560
nr.6	3225793	6.065

Media timpului de executie in cazul de fata pentru cele 6 teste precedente este 5.119 secunde.

Generatorul de teste scris in limbajul C, are un timp de executie mai mic fata de cel scris in Python, astfel ca poate genera date cu o limita mai mare de 10^6. De exemplu avem urmatoarele teste:

Test *C*	Nr. Benzinarii Generate	Timpul CPU (s)
nr.1	117438	0.049
nr.2	489330	0.186
nr.3	619679	0.274
nr.4	6651354	2.760
nr.5	3951482	1.599
nr.6	3461465	1.414

Media timpului de executie pe ultimele 6 teste este 1.047 secunde.

Aceste diferente reprezinta un argument care ajuta la demonstrarea faptului ca algoritmii scrisi in limbajul C, sunt mai optimi din punct de vedere al timpului de executie fata de cei scrisi in limbaul Python.

Conform datelor prezentate mai sus, putem afirma faptul ca, exista o diferenta semnificativa intre timpul de executie a algoritmilor scrisi in cele doua limbaje de programare.

Concluzii

Consider ca, *Tema numarul 16*, a reprezentat o provocare pentru mine.

In primul rand in perioada ultimei luni, am incercat sa imi concentrez timpul doar asupra acestui proiect, deoarece au existat multe cerinte care cereau notiuni pe care nu le stapaneam indeajuns.

De-a lungul acestei perioade de timp am avut posibilitatea sa acumulez cunostinete noi, imbunatatindu-mi astfel aptitudinile pentru rezolvarea de probleme.

Din punctul meu de vedere cea mai provocatoare problema pe care am intampinat-o a fost realizarea unui generator de date de intrare pentru problema propusa. A fost nevoie de mult timp pentru a dezvolta un algoritm ce genereaza date relevante pentru a testa problema. Dupa ce am citit un numar de articole, cat si capitole din carti de specialitate despre generarea aleatorie de date, am reusit sa implementez algoritmul descris in paginile anterioare ale acestui raport.

De altfel, o bariera in progresul meu a fost reprezentata si de lipsa mea de experienta. Pana in momentul de fata nu am folosit niciodata formatul latex, dar m-am adaptat si am reusit sa invat comenzile destul de repede.

Realizarea acestei teme a fost un lucru constructiv pentru mine, timpul petrecut asupra acesteia aducandu-mi doar beneficii.

References

[1] GeeksForGeeks - Algorithms section - greedy algorithms
`https://www.geeksforgeeks.org/greedy-algorithms/`

[2] Jon Bentley. *Programming pearls - second edition.* Bell Labs, Lucent Technologies Murray Hill, New Jersey

[3] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein. *Introduction to algorithms - third edition.*