

# PROIECTAREA ALGORITMILOR

---

## Selectarea aspersoarelor

---

**Studenta:** BĂDIȚĂ GEORGIANA -  
ELENA

**Grupa :** CR1.1A

**An :** I

**Specializarea :** Calculatoare română

## 1 Enunțul problemei

**Selectarea aspersoarelor.** Se consideră  $n$  aspersoare instalate să ude o bandă orizontală de iarbă ce are  $L$  metri lungime și  $l$  metri lățime. Fiecare aspersor este centrat vertical pe banda de iarbă respectivă. Pentru fiecare aspersor se cunosc: i) poziția sa ca distanță față de capătul din stânga al liniei care străbate pe orizontală mijlocul fâșiei de iarbă și respectiv ii) raza sa de operare. Să se determine numărul minim de aspersoare care trebuie pornite pentru a putea uda întreaga bandă de iarbă. Implementați doi algoritmi diferiți.

## 2 Algoritmii propuși

Fișierul *functions.c*

**Algoritm I**

**Metoda I**

Pseudo-codul funcției **minAspensNumber**, cea care determină care este numărul minim de aspersoare ce trebuie pornite pentru a putea uda banda de iarbă.

```
[1] MIN-ASPENS-NUMBER()
1.  init minAspens = 1
2.  init startIndex = 0
3.  init index = 1
4.  while index < aspenTotal and aspenPositionArrayindex.start = 0 do
5.      if aspenPositionindex.start = 0 and aspenPositionArraystartIndex.end
        < aspenPositionArrayindex.end
6.          startIndex = index
7.      end if
8.      index = index + 1
9.  end while
10. currentIndex = startIndex
11. while aspenPositionArraycurrentIndex.end < length and currentIndex < aspenTotal do
12.     nextIndex = currentIndex
13.     index = currentIndex + 1
14.     while aspenPositionArrayindex.start < aspenPositionArraycurrentIndex.end
        and index < aspenTotal do
15.         if aspenPositionArrayindex.start < aspenPositionArraycurrentIndex.end
            and aspenPositionArraynextIndex.end < aspenPositionArrayindex.end
16.             nextIndex = index
17.         end if
18.         index = index + 1
19.     end while
```

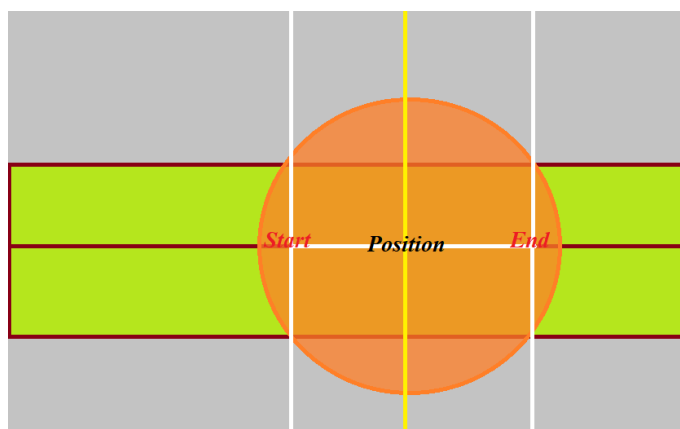
```

20.         if currentIndex  $\neq$  nextIndex
21.             currentIndex = nextIndex
22.             minAspens = minAspens + 1
23.         end if
24.         then
25.             break
26.         end while
27.         if aspenPositionArraycurrentIndex.end < length
28.             minAspens = - 1
29.         end if
30.         return minAspens

```

Înainte de a fi apelată această funcție în **main.c**, a mai fost apelată și funcția **arrayAspenToArrayAspenPosition**, o funcție care convertește fiecare element ce se regăsește în vectorul numit *aspenArray* în tipul *aspenPosition* și memorează în vectorul numit *aspenPositionArray*. Acest vector **aspenPositionArray**, care este unul de tipul *struct aspenPosition*, memorează pentru fiecare aspersor începutul și sfârșitul de unde până unde udă cu exactitate aspersorul (acest lucru este exemplificat în *Figura 2* de mai jos). De asemenea, fiecare aspersor a fost sortat prin metoda *Bubble-sort* (o metodă de sortare ce are ca și timp de complexitate  $O(n^2)$ ), în codul sursă se regăsește ca și funcția intitulată **sortAspenPositionArrayBySort**, în funcție de distanța sa față de latura din stânga a dreptunghiului (benzii de iarbă), fiecare aspersor aflându-se pe linia verticală ce împarte dreptunghiul în două jumătăți egale.

De asemenea, funcția numită **aspenToAspenPosition** convertește datele unui aspersor, distanța față de stânga și raza în distanță stânga a cercului și distanța dreapta a cercului. Această funcție returnează tipul de date **struct de aspenPosition**.



Reprezentare interval start - end.

Figura 2 .Exemplifică intervalul pe care udă cu exactitate aspersorul.

La baza principiului rezolvării acestei probleme stă Teorema lui Pitagora. Cu această teoremă se află cât acoperă cercul (adică aspersorul) curent din suprafața ierbii, sub forma unui dreptunghi. Acest lucru este realizat în funcția de tip struct intitulată **aspenToAspenPosition**, cea menționată și în paragraful anterior. Prin *Teorema lui Pitagora* se află acea distanță față de poziția aspersorului deja știută. Astfel că variabila *halfDistanceStartEnd* care este

$$radius^2 - (width/2)^2$$

Acest lucru este exemplificat în figura de mai jos, unde *halfDistanceStartEnd* este reprezentată ca „X”:

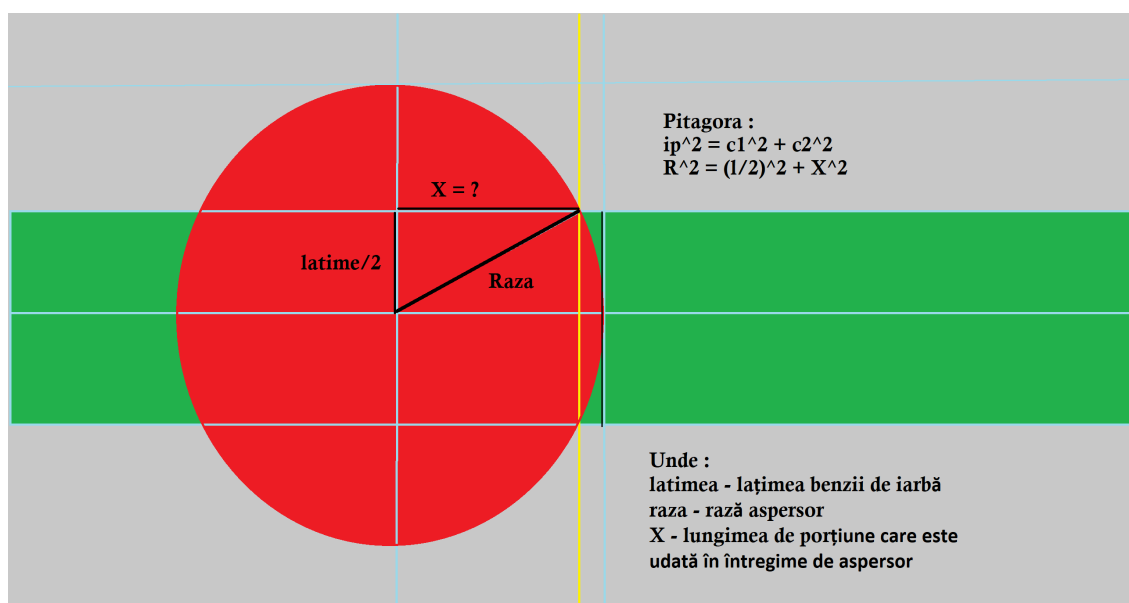


Figura 2.1 .Reprezentare Teorema lui Pitagora pentru rezolvarea acestui tip de problemă.

**Fișierul *modules.c***  
**Algoritm II**  
**Metoda II**

Pseudo-codul funcției intitulată **minAspenGreedy**, cea care determină care este numărul minim de aspersoare ce trebuie pornite pentru a putea uda banda de iarbă.

```
[2] MIN-ASPEN-GREEDY( intervalAspen *coord)
1.   init minAspensors = 0
```

```

2.  if  $coord_{noAspensor-1}.final < length$  or  $coord_0.prim > 0$ 
3.      print "No apensors can be placed..."
4.      return
5.  end if
6.  minAspensors = minAspensors + 1
7.  for index1 = 1 to noAspensors
8.      if  $coord_{index1}.prim > coord_{index1-1}.final$ 
9.          print "No apensors can be placed..."
10.         return
11.     else
12.         index2 = index1 - 1
13.         while  $coord_{index1}.prim \leq coord_{index2}.final$ 
14.             index1 = index1 + 1
15.         end while
16.         minAspensors = minAspensors + 1
17.     end for
18.  print "You will need a min of minAspensors"

```

### Analiză complexitate computațională

În tabelul de mai jos este exemplificat o analiză a best, worst și average case pentru algoritmi propuși, dar și o analiză pentru metoda de sortare **Bubble - sort** ce este folosită în ambele programe.

Analiză Best, Worst și Average Case			
Algoritmi	Best Case	Worst Case	Average Case
Algoritm I Metoda I	$O(1)$	$O(n^2 + n)$	$\Theta(\frac{1}{2}n - 1)$
Algoritm II Metoda II	$O(1)$	$(n)$	-
Bubble-Sort	$O(n)$	$O(n^2)$	$\Theta(n^2)$

Notății asimptotice

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

$BigO$ —upper bound

$\Omega$ —lower bound

$\Theta$ —average bound

Notățiile asimptotice pentru cei doi algoritmi în C :

Notății asimptotice			
Algoritmii	Upper bound	Lower bound	Average bound
Algoritm I Metoda I	$O(n^2)$ – <i>ptratic</i>	$\Omega(n \log n)$	$\Theta(n^2)$
Algoritm II Metoda II	$O(n)$	$\Omega(n)$	$\Theta(n)$

### 3 Proiectarea aplicației experimentale

- Structura de nivel înalt a aplicației

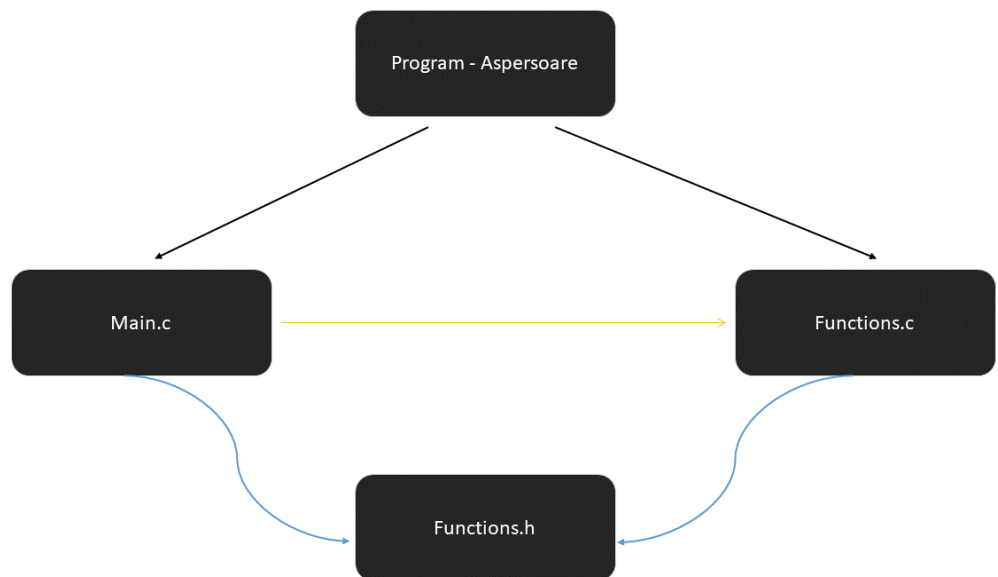


Figura 3.1. Structura pentru Metoda I de program în limbaj C

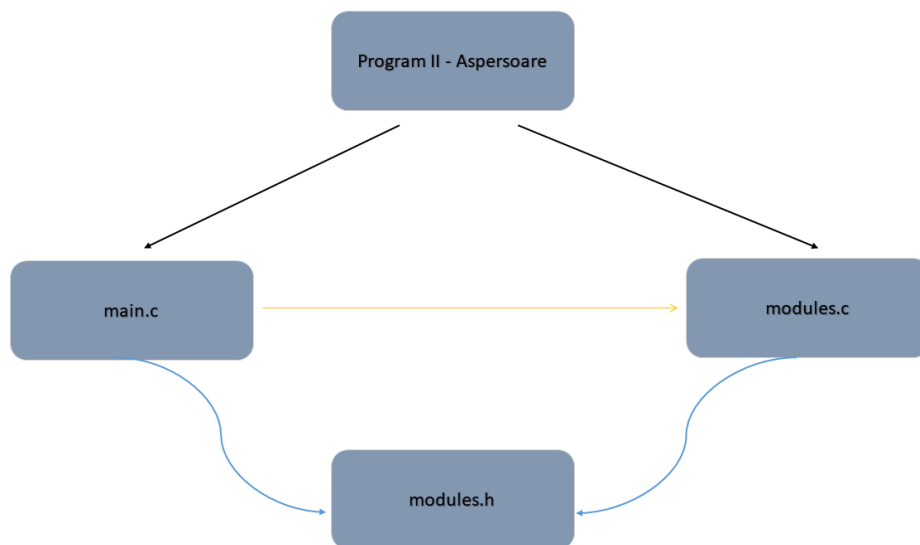


Figura 3.2. Structura pentru Metoda II de program în limbaj C

- Descrierea mulțimii datelor de intrare

Datele de intrare au fost generate cu ajutorul unui generator de date. S-au generat date pentru lungimea, lățimea dreptunghiului, adică a benzii de iarbă, numărul de aspersoare ce sunt instalate pe banda orizontală ce străbate dreptunghiul prin mijlocul acestuia și, de asemenea, raza de operare și poziția (distanța față de capătul din stânga al liniei care străbate pe orizontal mijlocul fișiei de iarbă). Datele generate sunt date netriviiale, acestea sunt declarate ca și **double** - pentru lățime și lungimea, rază, iar **int** pentru numărul de aspersoare și poziție.

Datele generate pentru intrare sunt obținute aleator. Au fost generate 10 seturi de date diferite, la fiecare set de date sunt generate numerele dintr-un anumit interval. Cum ar fi intervalul [1,1000], din aceasta au fost generate doar date ce sunt cuprinse în el, de exemplu datele din Figura 3.3., date ce se regăsesc și într-un fișier de input al programului :

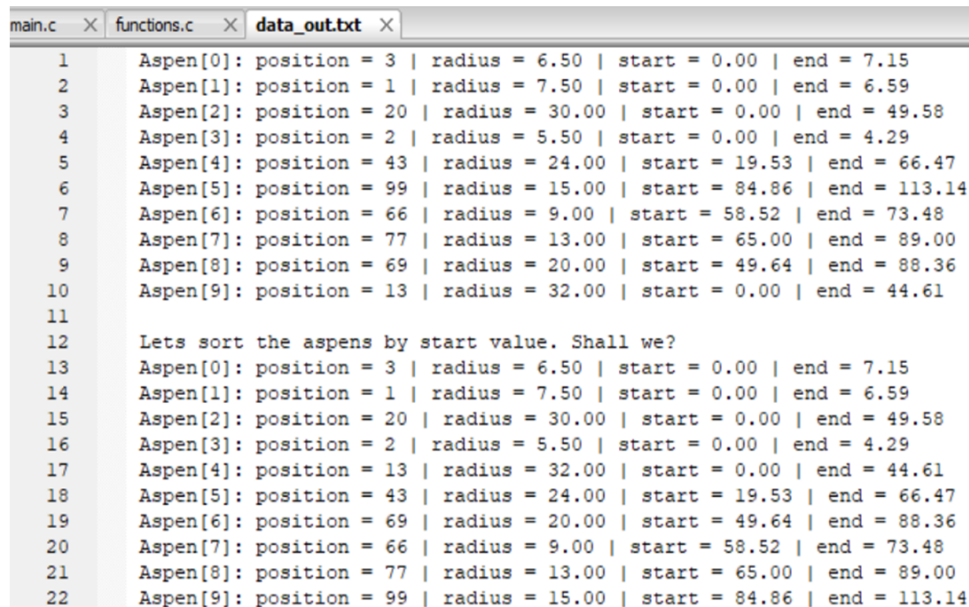
Lățime	10.50									
Lungime	100.50									
NrAspensoare	10									
Raza	6.50	7.50	30.00	5.50	24.00	15.00	9.00	13.00	20.00	32.00
Poziția	3	1	20	2	43	99	66	77	69	13

Figura 3.3. Exemplu date de intrare generate

Mulțimea datelor de intrare au fost generate astfel: un număr oarecare pentru lungimea benzii de iarbă, un număr oarecare pentru numărul de aspersoare, un număr oarecare pentru lățimea benzii, dar mai mic decât lungimea. Iar pentru rază și poziție, numere oarecare, în cazul poziției numere mai mici decât lungimea benzii de iarbă. Raza a fost generată ca și când poate să fie mai mare decât lățimea, nu s-au impus altfel de condiții.

- Descrierea ieșirilor / rezultatelor

Primul program, adică prima metodă implementată afișează numărul minim de aspersoare necesare pentru a putea fi udată în întregime banda de iarbă. Dar dacă dorim să vedem și începutul, sfârșitul intervalului pe care udă aspersorul de-a lungul benzii (suprafața ierbii, care este sub forma unui dreptunghi, acest interval reprezentând linia de simetrie verticală ce trabate cercul, cercul fiind o figură geometrică simetrică la fel ca și dreptunghiul), putem afișa și acest lucru. Datele vor fi afișate sub numele de *start* și *end* pentru fiecare aspersor. Un exemplu de rulaj este prezentat în figura de mai jos, acestea pot fi afișate prin apelarea funcției **printAspenArrayAndAspenPositionArray()** și înainte de a fi sortate, dar și după sortare:



```

main.c × functions.c × data_out.txt ×
1   Aspen[0]: position = 3 | radius = 6.50 | start = 0.00 | end = 7.15
2   Aspen[1]: position = 1 | radius = 7.50 | start = 0.00 | end = 6.59
3   Aspen[2]: position = 20 | radius = 30.00 | start = 0.00 | end = 49.58
4   Aspen[3]: position = 2 | radius = 5.50 | start = 0.00 | end = 4.29
5   Aspen[4]: position = 43 | radius = 24.00 | start = 19.53 | end = 66.47
6   Aspen[5]: position = 99 | radius = 15.00 | start = 84.86 | end = 113.14
7   Aspen[6]: position = 66 | radius = 9.00 | start = 58.52 | end = 73.48
8   Aspen[7]: position = 77 | radius = 13.00 | start = 65.00 | end = 89.00
9   Aspen[8]: position = 69 | radius = 20.00 | start = 49.64 | end = 88.36
10  Aspen[9]: position = 13 | radius = 32.00 | start = 0.00 | end = 44.61
11
12  Lets sort the aspens by start value. Shall we?
13  Aspen[0]: position = 3 | radius = 6.50 | start = 0.00 | end = 7.15
14  Aspen[1]: position = 1 | radius = 7.50 | start = 0.00 | end = 6.59
15  Aspen[2]: position = 20 | radius = 30.00 | start = 0.00 | end = 49.58
16  Aspen[3]: position = 2 | radius = 5.50 | start = 0.00 | end = 4.29
17  Aspen[4]: position = 13 | radius = 32.00 | start = 0.00 | end = 44.61
18  Aspen[5]: position = 43 | radius = 24.00 | start = 19.53 | end = 66.47
19  Aspen[6]: position = 69 | radius = 20.00 | start = 49.64 | end = 88.36
20  Aspen[7]: position = 66 | radius = 9.00 | start = 58.52 | end = 73.48
21  Aspen[8]: position = 77 | radius = 13.00 | start = 65.00 | end = 89.00
22  Aspen[9]: position = 99 | radius = 15.00 | start = 84.86 | end = 113.14

```

Figura 3.4. Dacă se dorește afișarea pentru datele calculate

Se poate observa în Figura 3.4 că unele aspersoare au ca valoare de *start* valoarea 0. Atunci când punctul de *start* calculat are valoare negativă ( $start < 0$ ), calculat cu ajutorul variabilei *halfDistanceStartEnd* de tip *double* prin intermediul Teoremei lui Pitagora (de altfel partea cea mai importantă din re-



zolvarea problemei), această valoare este rotunjită la 0, deoarece ne dorim să fie pozitivă.

De asemenea, se poate afișa în fișierul de output doar numărul minim necesar de aspersoare pentru a putea fi udată fâșia de iarbă, însoțit de un mesaj corespunzător. Iar dacă banda de iarbă nu poate fi acoperită cu datele de intrare generate, se va afișa mesaj corespunzător.

Cel de-al doilea program implementat, poate afișa la fel ca primul valoarea pentru începutul și sfârșitul intervalului pe care udă aspersorul.

O concluzie privind datele de ieșire ale celor două programe, este faptul că la cel de-al doilea program valoarea de început, intitulată **prim** nu mai este rotunjită la 0 ca și valoarea **start** din primul program. Astfel că, acest program calculează numărul minim de aspersoare necesare și cu valorile posibile negative ale variabile *prim*. De exemplu, în figura 3.5, se poate observa această diferență între datele de ieșire cu privire la valorile variabilelor *prim* și *final* din cel de-al doilea program, și faptul că acestea sunt numere fără zecimale, fiind afișate ca numere întregi, nu de tip double.

Rezultate generate	
Valoare <b>prim</b>	Valorea <b>final</b>
-19.00	45.00
-10.00	50.00
-6.50	8.50
-3.50	7.50
-3.50	9.50
19.00	67.00
49.00	67.00
57.00	75.00
64.00	90.00
84.00	114.00

Figura 3.5 Posibile date de ieșire pentru a doua implementare

Datele din Figura 3.4 și 3.5 au avut aceleași date de intrare, respectiv datele exemplificate în Figura 3.3

- Lista tuturor modulelor aplicației și o scurtă descriere a lor  
Pentru implementare primei rezolvări:
  - *main.c* - în acest modul se citesc din multiplele fișiere datele de intrare și sunt apelate funcțiile necesare ce se află în *functions.c*, de asemenea aici se calculează timpul de execuție al algoritmului implementat.
  - *functions.c* - în acest modul se află funcțiile programului, ce ajută la rezolvarea problemei
  - *functions.h* - în acest modul sunt declarate structurile folosite pentru datele despre aspersor, cum ar fi **struct apen** și **struct apenPosition**, lungimea, lățimea și numărul de aspersoare, dar și prototipurile

funcțiilor apelate.

Pentru implementarea celei de-a doua rezolvări:

- *main.c* - în acest modul se citesc din multiplele fișiere datele de intrare și sunt apelate funcțiile necesare ce se află în *modules.c*, de asemenea aici se calculează timpul de execuție al algoritmului implementat.
- *modules.c* - în acest modul se află funcțiile programului, ce ajută la rezolvarea problemei
- *modules.h* - în acest modul sunt declarate structurile folosite pentru datele despre aspersor, cum ar fi **struct aspersor** și **struct intervalAspen**, lungimea, lățimea și numărul de aspersoare, dar și protipurile funcțiilor apelate.

- Lista tuturor procedurilor aplicației, grupate pe module

Prima implementare:

- *main.c* :
  - \* *clock()* - calculează timpul de execuție al algoritmului *minAspensNumber()*
  - \* *void arrayAspenToArrayAspenPosition ()* - convertește fiecare element de tipul *aspenArray* în *aspenPosition* și le memorează în vectorul *aspenPositionArray*. Este o funcție de tip **void**
- *functions.c*:
  - \* *void sortAspenPositionArrayByStart ()* - sortează crescător prin metoda Bubble - Sort aspersoarele în funcție de distanța, calculată cu Teorema lui Pitagora și scăzând poziția știută din datele problemei, față de capătul din stânga al liniei care străbate pe orizontală mijlocul fâșiei de iarbă. Prin această sortare sunt mutate odată cu aspersorul și informațiile despre el, cum ar fi valorile de start, end, raza, poziția.
  - \* *int minAspensNumber ()* - calculează numărul minim de aspersoare necesare prin comparare valorile de start și end. Prima dată se caută aspersoarele care au cea mai mare valoare de end și valoarea de start egală cu zero. Apoi se parcurge în continuare vectorul numir *aspenPositionArray* până când se ajunge la ultimul aspersor care se află la finalul benzii de iarbă și valoarea sa este mai mare sau egală cu lungimea sau nu mai avem aspersoare. Iar în final dacă nu sunt suficiente aspersoare sa acopere fâșia de iarbă se va returna valoarea -1, altfel se va returna prin variabila de return **minAspens** numărul minim aflat.
  - \* *init(double lengthAux, double widthAux, int aspenTotalAux)* - dacă se dorește și afișarea lungimii, lățimii și numărului de aspersoare.
  - \* *void printAspenArrayAndAspenPositionArray ()* - dacă se dorește să se afișeze poziția, raza, start-ul și end-ul pentru fiecare aspersor, așa cum este exemplificat în Figura 3.3

Cea de a doua implementare:

- main.c:
  - \* clock() - pentru calcularea timpului de execuție a algoritmului intitulat minAspenGreedy (coord);
- modules.c
  - \* void starting(struct aspensor \*properties, struct intervalAspen \*coord) - determină intervalul ce are ca și capetele **prim**, **final** pentru a afla distanța pe care udă aspersorul. Parametrii semnifică tipul lui *properties* și *coord*, ce sunt alocați dinamic.
  - \* void eliminate(struct intervalAspen \*coord, double position) - elimină aspersoarele care au aceeași poziție, dar celălalt are raza mai mare, este funcție auxiliară pentru funcția "duplicat". Ca și parametrii avem \*coord , alocat dinamic și poziția.
  - \* void duplicate(struct aspensor \*properties, struct intervalAspen \*coord) - elimină aspersorul care se află în interiorul celui alt aspersor, iar raza acestuia este mai mică decât cea a celui alt. Aspersorii au aceeași poziție, dar raza diferită.
  - \* void sorting (struct intervalAspen \*coord, struct aspensor \*properties) - sortează crescător prin metoda Bubble - Sort aspersoarele în funcție de distanța, calculată cu Teorema lui Pitagora și scăzând poziția știută din datele problemei, față de capătul din stânga al liniei care străbate pe orizontală mijlocul fâșiei de iarbă. Prin această sortare sunt mutate odată cu aspersorul și informațiile despre el, cum ar fi valorile de start, end, raza, poziția.
  - \* void minAspenGreedy (struct intervalAspen \* coord) - calculează numărul minim de aspersoare necesare pentru a acoperi întreaga bandă de iarbă folosind metoda Greedy.

## 4 Date experimentale

- Descrierea algoritmului folosit pentru generarea de mulțimi de date de intrare

Algoritmul folosit pentru generarea de mulțimi de date de intrare generează valori pentru lățimea, lungimea, numărul de aspersoare și raza, poziția pentru fiecare aspersor. Datele sunt de tipul următor:

Tipul datelor generate	
<b>double</b>	<b>int</b>
lungimea(length)	numărul de aspersoare(noAspen)
lățimea(width)	poziția(position)
raza(radius)	-

Pentru a se putea genera valori s-a folosit funcția pentru generare automată de date aleatorii **rand**. Această funcție generează un număr între o valoare maximă impusă și una minimă impusă. Valoarea de maxim poate fi modificată, deoarece a fost declarată cu macro **#define**.

Lungimea(**length**) este generată între 1 și o valoare maximă definită, lățimea(**width**) este generată între 1 și **lungimea/10**, deoarece trebuie să fie o valoare mai mică decât lungimea, iar numărul de aspersoare este o valoare între 1 și o limită impusă, de asemenea, și aceasta definită prin **#define**. Pentru raza și poziția fiecărui aspersor s-au impus astfel valorile pentru limita inferioară și cea superioară: pentru poziție valori între 0 și lungimea, iar pentru rază valorile între 0 și **lungimea/8**, deoarece trebuie să fie valori mai mici decât lungimea, dar și posibil mai mari decât lățimea. Toate acestea sunt afișate în fișier.

Datele sunt corect generate și semnificative pentru teste, deoarece pe un set mic de date au fost testate și verificate, iar acestea sunt non-triviale.

- Pseudo-codul generatorului de date:

Funcția pentru a genera date aflate între anumite limite (limită superioară și inferioară)

**RANDOM-NUMBER(min, max)**

1. **return (rand()\* (max - min)) / RAND-MAX + min**

*n main :*

1. Se declară un fișier pentru output
2. **length = RANDOM-NUMBER(1, maxim)**
3. **print length**
4. **width = RANDOM-NUMBER(1, length/10)**
5. **print width**
6. **noAspen = RANDOM-GENERATOR(1, limit)**
7. **print noAspen**
8. **for iter 0 to noAspen**
10. **print RANDOM-NUMBER(0, length)**
11. **print RANDOM-NUMBER(0, length/8)**

- Am implementat acest cod din C și în Python pentru a putea observa diferențele dintre cele două limbaje. Pentru ambele implementări am ales aceleași limite inferioare și superioare. În plus, față de implementările pentru rezolvare în C, în programul din Python am adăugat și generatorul de date. Astfel că generatorul generează date într-un fișier, iar de acolo sunt luate în mod direct de către programul principal ca mulțimi de date de intrare. S-au generat și în Python, la fel ca pentru programele în C, 10 seturi de date de intrare. Am adăugat acest generator doar pentru unul dintre programele în Python, este atașat în programul care nu conține **class**, am încercat să modific unul dintre programele din C ce conțin amândouă **struct**, în Python într-un program care nu se folosește de **class**, dar care urmărește același principiu de rezolvare a problemei.

## 5 Rezultate și concluzii

- Descrierea datelor de ieșire obținute folosind implementările în C și Python (comparație)
  - Pentru implementările în C datele de ieșire sunt afișate în fișiere, câte un fișier separat pentru fiecare mulțime de date de intrare, iar datele de ieșire procesate de implementările în Python sunt afișate la ecran pentru unul dintre programe(program I), pentru celălalt intitulat Program II afișarea se face în fișier, dar citirea mulțimilor se face din fișier în ambele implementări. În ambele implementări, atât Python cât și C, este specificat atunci când o mulțime de date nu cuprinde date necesare pentru a putea uda banda de iarbă, acest lucru este marcat cu câte un mesaj corespunzător, precum : „There aren’t enough aspersors to cover the entire length/rectangle” sau în implementarea în Python: „None”.
  - O altă precizare importantă mai este faptul că ambele coduri în C au aceleași mulțimi de date de intrare, iar primul program în Python (Program I), cel care conține integrat în program un generator de date, a conținut datele ce se regăsesc în fișierele de intrare doar o singură dată, altfel că o singură dată au fost putute compara toate implementările. De asemenea, pentru acele date de intrare s-a putut observa faptul că rezultatele sunt ușor diferite, deoarece în cazul codurilor în C, Algoritm I este diferit de Algoritm II, astfel că există o marjă de eroare de un aspersor între cele două implementări în C (acest lucru depinde de tipul datelor, adică din ce interval fac parte). Deoarece primul cod rotunjește la valoarea zero , poziția de start a unui aspersor atunci când aceeaș prin calculul efectuat cu Teorema lui Pitagora dă o valoare negativă, iar în cel de-al doilea, aflarea numărului minim de aspersoare se face indiferent dacă acea valoare, numită în acest algoritm prim este negativă sau nu.
  - Din păcate, al doilea cod în Python nu îți afișează numărul minim de aspersoare, aceasta calculează doar valorile lui *prim* și *final*, pe care le afișează în fișier.
- Timpul de execuție al algoritmilor pentru fiecare mulțime de date(acest timp a fost calculat cu ajutorul funcției clock).
- Intervalul pentru fiecare algoritm corespunzător unei anumite mulțimi de date variază între valorile cuprinse în acel interval, după cel puțin 3 teste ale algoritmului.

În tabelul de mai jos este exemplificat timpul de execuție pentru fiecare algoritm implementat pe o anumită mulțime de date.

Timp de execuție		
Algoritm I	Algoritm II	Interval mulțime date
[0.117s, 0.138s]	[0.119s, 0.126s]	[1,200]
[0.167s, 0.208s]	[0.120s, 0.145s]	[1,50]
[0.151s, 0.178s]	[0.141s, 0.191s]	[1,10]
[0.102s, 0.151s]	[0.105s, 0.147s]	[10,600]
[0.111s, 0.196s]	[0.104s, 0.127s]	[50,300]
[0.138s, 0.177s]	[0.096s, 0.106s]	[1,1000]
[0.125s, 0.177s]	[0.100s, 0.136s]	[1,100]
[0.107s, 0.115s]	[0.113s, 0.139s]	[100,10000]
[0.107s, 0.124s]	[0.106s, 0.163s]	[1,5000]
[0.101s, 0.157s]	[0.095s, 0.158s]	[1000,100000]

- Astfel din tabelul prezentat se observă media timpilor de execuție reprezentat și prin diagrama de mai jos.
- Se poate observa cum variază timpul fiecărui algoritm în funcție de mulțimea datelor de intrare.

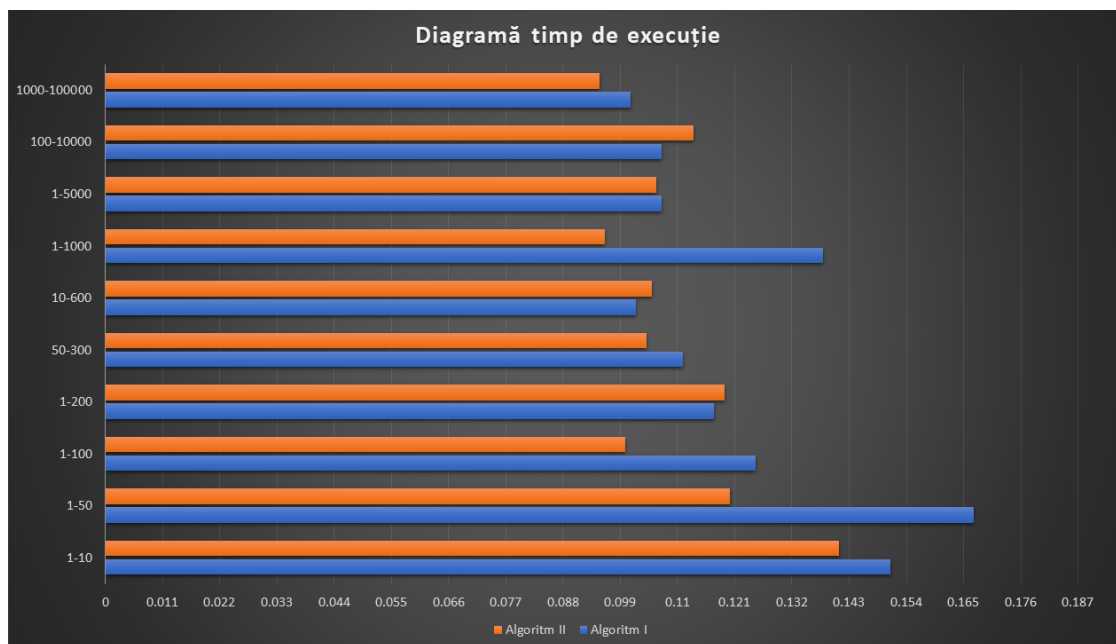


Figura 5.1 Diagramă pentru variația timpului de execuție

- În urma analizei acestei diagrame se poate observa că Algoritmul I are un timp de execuție mai mare decât Algoritmul II, ceea ce era de aștept, deoarece Algoritmul I are o complexitate de  $O(n^2)$ , iar cel de-al doilea complexitatea  $O(n)$ , astfel se efectuează mai multe operații, necesitând timp mai mult.



Figura 5.1 Diagramă comparație timp de execuție

- Timpul de execuție pentru codurile în Python nu au putut fi calculate, deoarece folosind funcția `time()` din Python, aceasta îmi afișează mereu valoarea zero, deși este testată pe date de intrare diferite.
- Concluzii :
  - Cea mai mare provocare și interesantă parte cu privire la această temă a fost să gasesc o complexitate cât mai mică, preferabil liniară, cât mai optimă și de asemenea implementarea a doi algoritmi diferiți pentru aceeași problemă, dar și faptul să folosesc date de intrare netriviiale. Datele fiind foarte importante în testarea algoritmilor.
  - Iar ca direcție viitoare privind extinderea studiului, ar fi aceea de a schimba primul algoritm scris în C, într-unul cu o complexitate mai mică, o idee ar fi implementarea cu arbori și să reușesc să fac complexitatea  $O(n \log n)$ .
  - Tot pentru primul algoritm în C, Algoritm I, se poate optimiza dacă aș stoca un aspersor ca o singură structură care să aibă : position, radius, start, end.

## 6 Referințe bibliografice

- [1] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*. MIT Press, 3rd Edition, 2009, accesat în aprilie 2020
- [2] L<sup>A</sup>T<sub>E</sub>X project site, [https://www.overleaf.com/learn/how-to/Creating\\_a\\_document\\_in\\_Overleaf](https://www.overleaf.com/learn/how-to/Creating_a_document_in_Overleaf), accesat în aprilie 2020.
- [3] Python <https://www.w3schools.com/python/default.asp>, accesat în mai 2020
- [4] Metoda Greedy <https://www.geeksforgeeks.org/greedy-algorithms/>, accesat aprilie 2020
- [5] Random generator [https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_rand.htm](https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm), accesat aprilie 2020