Contents

[[AlgE] Numerele grindină ale lui Lothar Collatz 7](#_Toc21333005)

[[AlgE] Numere palindrom. 9](#_Toc21333006)

[[AlgE] Conjectura lui Goldbach (Csongor) 10](#_Toc21333007)

[[AlgE] Cel mai mic palindrom mai mare decât n [\*\*\*] 13](#_Toc21333008)

[[AlgE] Numere palindrom munte 13](#_Toc21333009)

[[AlgE] Triunghiul lui pascal 14](#_Toc21333010)

[[AlgE] Numere prime (Ana) 15](#_Toc21333011)

[[Tema][AlgE] Numere perfecte (Ana) 16](#_Toc21333012)

[[AlgE] Ultima cifră a numărului n^n 17](#_Toc21333013)

[[AlgE] Formații poker (varianta decizii) 21](#_Toc21333014)

[[AlgE] Algoritmul lui Euclid 28](#_Toc21333015)

[[AlgE] Suma cifrelor unui număr 28](#_Toc21333016)

[[AlgE] Permutarea cifrelor unui număr 28](#_Toc21333017)

[[AlgE] Numărul lui Euler 28](#_Toc21333018)

[[AlgE] Fracții ireductibile (Alex) 29](#_Toc21333019)

[[CG/AlgE] Conversia unei valori ȋntregi ȋn reprezentarea romană 31](#_Toc21333020)

[[CG/AlgE] Conversia unei valori ȋn reprezentarea romană ȋn cifre arabe 33](#_Toc21333021)

[[CG/AlgE] Conversia unui text la reprezentarea morse 34](#_Toc21333022)

[[AlgE] Generarea unui text aleator format din caractere mari, mici și cifre 34](#_Toc21333023)

[[AlgSD1] Numărul cel mai mare și cel mai mic din cifrele unui număr 34](#_Toc21333024)

[[AlgSD1] Suma a două numere ȋntr-un vector 34](#_Toc21333025)

[[AlgSD1] Valori dintr-un vector cu cifre doar de 1 ȋn baza 2 34](#_Toc21333026)

[[AlgSD1] Ordonarea signum!! 35](#_Toc21333027)

[[AlgSD1] Concatenarea a doi vectori 38](#_Toc21333028)

[[AlgSD1] Apartenența unui vector de dimensiune mai mică ȋntr-un vector de dimensiune mai mare 39](#_Toc21333029)

[[AlgSD1] Vectorul de valori unice 41](#_Toc21333030)

[[AlgSD1] Simulare reuniune, intersecție și difernță pe vectori (Marius) 46](#_Toc21333031)

[!! output 47](#_Toc21333032)

[[AlgSD1] Apa dintr-un vector 48](#_Toc21333033)

[[AlgSD1] Numere mari. 5000! (Remus) 52](#_Toc21333034)

[[AlgSD1] Grâul pe tabla de șah 57](#_Toc21333035)

[[AlgSD1] Sortare Bubble (Ana) 59](#_Toc21333036)

[[AlgSD1] Sortare Insertion (Ana) 60](#_Toc21333037)

[[AlgSD1] Sortare Selecție (stg, dr, stg+dr) (Ana) 60](#_Toc21333038)

[[AlgSD1] Counting sort 60](#_Toc21333039)

[[AlgSD1] Ciurul lui Eratostene (Marius) 61](#_Toc21333040)

[[AlgSD2] Camere alăturate (Administrator) 62](#_Toc21333041)

[[AlgSD2] Drumul cel mai scurt ȋn labirint 62](#_Toc21333042)

[[AlgSD2] Parcurgerea matricei ȋn spirala (Remus) 63](#_Toc21333043)

[[AlgSD2] Suma NSEV (Remus) 65](#_Toc21333044)

[[AlgSD2] Topirea unui iceberg (Zsolti) 68](#_Toc21333045)

[[AlgSD2] Automate celulare (Fredkin) 73](#_Toc21333046)

[[AlgSD2] Rotirea unei matrici pe diagonala principală și diagonala secundară (Remus) 74](#_Toc21333047)

[[AlgSD2] Matricea primelor n\*m numere prime (Ana) 76](#_Toc21333048)

[[AlgSD2] Produsul a două matrici (Horatiu) 76](#_Toc21333049)

[[STR] Gestionarea numerelor complexe (Marius) 77](#_Toc21333050)

[[STR] Date. Vârsta ȋn secunde (Zsolti) 80](#_Toc21333051)

[[STR] Date. Distanța dintre două date. (Remus) 85](#_Toc21333052)

[[GRDY] Plata unei sume 87](#_Toc21333053)

[[GRDY] Spectacole (Zsolti) 88](#_Toc21333054)

[[GRDY] Colorarea hărților (Horatiu) 90](#_Toc21333055)

[[GRDY] Codificarea Huffman (Horatiu) 90](#_Toc21333056)

[[GRDY] Subsecvență maximă descrescătoare? (Problema plopilor) (Sara) 90](#_Toc21333057)

[[GRDY] Rucsacul (Marius) 91](#_Toc21333058)

[[GRDY] Săritura calului 93](#_Toc21333059)

[[GRDY] Comis voiajorul (Marius) 94](#_Toc21333060)

[[REC] Fibonacci (Aron) 96](#_Toc21333061)

[[REC] Ieșirea din labirint (Alex) 96](#_Toc21333062)

[[REC] Platouri (Horatiu) 96](#_Toc21333063)

[[REC] Mastermind (Horatiu) 96](#_Toc21333064)

[[REC] Numărul de obiecte (structuri de valori identice amorfe) dintr-o matrice 98](#_Toc21333066)

[[REC] Algoritm de sortare Selection Sort fără cicluri repetitive sau goto. (Aron) 98](#_Toc21333067)

[[BCKT] Generare permutări (Ana) 98](#_Toc21333068)

[[BCKT] Generare aranjamente (Ana) 98](#_Toc21333069)

[[BCKT] Generare combinări (Alex) 99](#_Toc21333070)

[[BCKT] Generare partiții (Ana) 101](#_Toc21333071)

[[BCKT] Problema damelor (Ana) 101](#_Toc21333072)

[[REC/DC] Apa dintr-un vector (Zsolti) 101](#_Toc21333073)

[[MATH/RECDC] Determinant nxn (Marius) 102](#_Toc21333074)

[[MATH/RECDC] Determinant nxn (Remus) 104](#_Toc21333075)

[[RECDC] Turnurile din Hanoi (Extindere) 105](#_Toc21333076)

[[RECDC] Merge Sort (Aron) 105](#_Toc21333077)

[[RECDC] Quiq Sort (Aron) 105](#_Toc21333078)

[[RECDC] minMax (Zsolti) 105](#_Toc21333079)

[[RECDC] Binary search (Alex) 106](#_Toc21333080)

[[RECDC] ridicarea la putere (Horatiu) 108](#_Toc21333081)

[[PD] Fibonacci (Zsolti) 109](#_Toc21333082)

[[PD] Subsecvența de sumă maximă 111](#_Toc21333083)

[[Log] problema celor 2 ouă 111](#_Toc21333084)

[[Log] Formații poker (număr identități) 111](#_Toc21333085)

[[Log] Matricea XOR 111](#_Toc21333086)

[[Log] Bananele și cămila 111](#_Toc21333087)

[[MATH] Aria unui poligon (reprezentarea nu este necesară) 111](#_Toc21333088)

[[SDDU - LSI] TAD Lista SI (Ana) //se considera o clasa Lista cu adaugare la inceput, la final si la mijloc (pozitii random), stergerile cu vector suplimentar etc 111](#_Toc21333089)

[[SDDU - LSI] Lista ordonată (Ana) 111](#_Toc21333090)

[[SDDU - LSI] Numere mari 111](#_Toc21333091)

[[SDDU - COADA] Camere alăturate 111](#_Toc21333092)

[[SDDU - COADA] Drumul cel mai scurt (Zsolti) 111](#_Toc21333093)

[[SDDU - LC] Cavalerii mesei rotunde (Horatiu) 111](#_Toc21333094)

[[RND] Apa dintr-un vector (Alex) 111](#_Toc21333095)

[[RND] Pătrat magic (Aron) 111](#_Toc21333096)

[[RND] Pătrat magic Durer 111](#_Toc21333097)

[[RND] Sudoku 111](#_Toc21333098)

[[RND] Cerc magic (opt) 111](#_Toc21333099)

[[RND] Algoritmul de selecție MC (Zsolti) 112](#_Toc21333100)

[[RND] Automate celulare 114](#_Toc21333101)

[[RND] Automate celulare simetrice (Fredkin 1960) 114](#_Toc21333102)

[[] Spirolaterele lui Franks Odds 114](#_Toc21333103)

[[AG] TAD A.G. (Zsolti) 114](#_Toc21333104)

[[AG] Dispersia unui graf (Remus) 115](#_Toc21333105)

[[GRP] Reprezentări 116](#_Toc21333106)

[[GRP] Intersecții de segmente 116](#_Toc21333107)

[[GRP] Parametrizarea poligoanelor regulate. (RTS) (Remus) 117](#_Toc21333108)

[[GRP] Desenarea unui ceas Analog (Remus) 118](#_Toc21333109)

[[GRP/BCKT] Mulțimi de puncte. Aria Triunghiului minim (Remus) 124](#_Toc21333110)

[[GRP/REC] Fractali geometrici 127](#_Toc21333111)

[[GRP/REC] Mandlebro 127](#_Toc21333112)

[[GRP] Mulțimi de puncte. Acoperirea convexă (Csongor) 129](#_Toc21333114)

[[GRP] Reprezentare de funcții (Zsolti) 129](#_Toc21333115)

[[GRP] Implementarea algoritmilor de procesare a matricelor pe algebra RGB și crearea de filtre. (Marius) 130](#_Toc21333116)

[[GRP] Construirea unui zoom pe o matrice de pixeli pe baza mediei componentelor de culoare. (Marius) 139](#_Toc21333117)

[[GEN] Teorema lui Pick (Csongor) 142](#_Toc21333119)

[[GRAF] BFS, DFS (Csongor) 146](#_Toc21333120)

[[GRAF] Colorarea Hărții (Csongor) 147](#_Toc21333121)

[[GRAF] Algoritmul lui Dijkstra (Csongor) 150](#_Toc21333122)

[[GRAF] Graf Eulerian (Csongor) 153](#_Toc21333123)

[[GRAF] Graf Hamiltonian (Csongor) 155](#_Toc21333124)

[[GRAF] Algoritmul lui Kruskall (Csongor) 158](#_Toc21333125)

[[ARB] 161](#_Toc21333126)

[Bibliography 163](#_Toc21333127)

[TAGS]

[AlgE] Algoritmi elementari

[AlgSD1] Algoritmi pe structuri de date ȋnlănțuite cu 1 dimensiune (Vectori)

[AlgSD2] Algoritmi pe structuri de date ȋnlănțuite cu 2 dimensiuni (Matrici)

[STR] Structuri

[GRDY] Tehnica de programare Greedy

[REC] Recursivitate

[BCKT] BackTracking

[REC/DC] Divide and Conquer

[PD] Programare dinamică

[Log] Probleme de logică

[SDDU - LSI]

[SDDU - LDI]

[SDDU - COADA]

[SDDU - LC]

[RND]

[AG]

[GRP]

[GEN]

[GRAF]

[ARB]

## [AlgE] Numerele grindină ale lui Lothar Collatz

Această problemă presupune următoarea transformare pe un număr n dat: dacă numărul este par acesta se ȋmparte la 2 iar dacă este impar acesta se ȋnmulțește cu 3 și se adaugă 1. Transformarea se repetă până când numărul ajunge la 1.

Ex 1: n =12

După prima iterație n = 6,

După a doua iterație n =3,

După a treia iterație n este impar, deci n =3\*n+1, n =10

Iterația a patra n=5

Iterația a cincea n =16

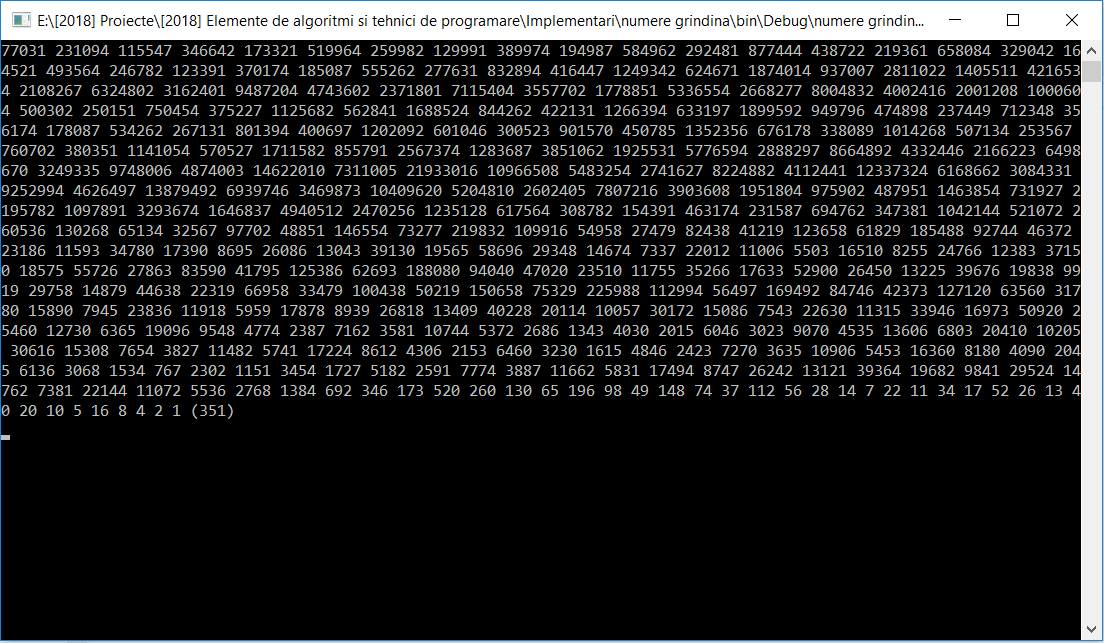
Iterația a șasea n =8

Iterația a șaptea n =4

Iterația a opta n =2

și ȋn final, iterația a noua, n=1.

Ex 2: 77031 parcurge 351 pași până când ajunge la valoarea 1.



De fapt nu se cnoaște nici un număr care să nu “conveargă” tip grindină la 1 (La momentul respectiv s-au prelucrat 27cvadrilioane (2.7E16) de numere și cel mai lung șir de convergență are 1820 de pași (de 15 cifre)) (1 p. 252)

Implementarea nu prezintă nici un fel de dificultate, amintim doar folosirea tipului de dată ulong (2) deoarece plaja de valori a acestui tip de dată ne permite eventual, găsirea acelui număr de 15 cifre. Aplicația are la bază metoda (HailNumbers) care returnează prin lista toReturn toate numerele reprezentând toți pașii prin care trece parametrul formal n până când ajunge la valoarea 1.

|  |
| --- |
| class Program  {  static List <ulong> hailNumbers (ulong n)  {  List<ulong> toReturn = new List<ulong>();  while (n != 1)  {  toReturn.Add(n);  if (n % 2 == 1)  n = 3 \* n + 1;  else  n = n / 2;  }  toReturn.Add(1);  return toReturn;  }  static void Main(string[] args)  {  int max = 0;  ulong no = 0;  for (ulong i = 1; i < 1E5; i++)  {  List<ulong> T = hailNumbers (i);  if (max < T.Count)  {  max = T.Count;  no = i;  }  }    List<ulong> T\_ = hailNumbers (no);  foreach (ulong uL in T\_)  Console.Write(uL + " ");  Console.Write("(" + T\_.Count + ")");  Console.WriteLine();  Console.ReadKey();  }  } |

Programul de față găsește maxiumul de etape de convergență pentru primele 1E5 numere (10000). Teoretic, ca să rezolvăm problema găsirii acelui număr de 15 cifre a cărui etape de convergență urcă la 1820, ar trebui să scriem ȋn loc de 1e5 valoarea 999999999999999, și putem porni de la 1E14, pentru a reduce numărul de valori procesate. Practic acest calcul nu se va executa ȋn timpul vieții noastre, și metoda de a găsi aceste numere mari (sau eventual mai mari) trebuie căutată ȋn conceptul de procesare paralelă (ideea ar fi să ȋmpărțim intervalul de căutare ȋn subintervale mai mici pe care mai multe calculatoare să le poată executa simultan).

## [AlgE] Numere palindrom.

Numerele palindrom sunt acele numere care citite invers (oglindite) au acceași valoare.

Pentru a construi funcția care calculează oglinditul unui număr, trebuie să accesăm cifrele acestuia, exact ȋn ordine inversă și să construim numărul oglindit ȋnmulțind valoarea până atunci obținută cu 10 la care adăugăm cifra curentă:

Ex: n = 2345, og = 0

n%10 = 5, og = 10 \* 0 + 5 = 5; n = n/10 = 234

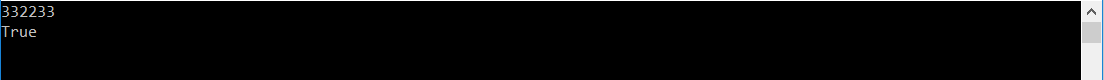
n%10 = 4, og = 10 \* 5 + 4 = 54; n = n/10 = 23

n%10 = 3, og = 10 \* 54 + 3 = 543; n = n/10 = 2

n%10 = 2, og = 10 \* 543 + 2 = 5432; n = n/10 = 0

Așa cum se vede și ȋn exemplu, ȋn timpul procesării cifrelor lui n, acesta ajunge la 0, deci verificarea ogld cu n nu ar avea sens, de aceea salvăm inițial valoarea lui n ȋn variabila save.

|  |
| --- |
| class Program  {  static bool palindrom(ulong n)  {  ulong save = n;  ulong ogld = 0;  while (n > 0)  {  ogld = ogld \* 10 + n % 10;  n /= 10;  }  if (ogld == save)  return true;  return false;  }  static void Main(string[] args)  {  ulong n = ulong.Parse(Console.ReadLine());  Console.WriteLine("{0}", palindrom(n));  Console.ReadKey();  }  } |



## [AlgE] Conjectura lui Goldbach (Csongor)

Un număr Goldbach este un întreg pozitiv care poate fi scris ca suma a două numere prime impare.  
În acest exercițiu, ne propunem sa generăm fiecare sumă de numere prime impare a cărei sumă ne dă ca rezultat numărul introdus de utilizator.

Pentru asta, creăm o clasă numită Goldbach, iar datele vor fi citite din consolă.  
Pentru rezolvarea problemei vom folosi o variabilă globală:

|  |
| --- |
| private List<int> NumerePrime = new List<int>(); |

În această listă, vom stoca numerele prime deja găsite pentru a nu verifica primalitatea unui număr de mai multe ori.

Constructorul nostru nu va conține cod, deaorece putem folosi un obiect de mai multe ori și nu avem variabile care trebuie instanțiate de la început. Pentru generarea sumelor numerelor prime vom folosi următoarele subprograme:

|  |
| --- |
| public void GenerareSecventa(int Numar)  {  Console.Write(Numar + ": ");  DetermminaPerechi(Numar);  } |

Parametrul Numar va fi numărul pentru care vom genera sumele.

|  |
| --- |
| public void DetermminaPerechi(int Numar)  {  for (int i = 3; i <= Numar / 2; i += 2)  {  IncercNumar(Numar, i);  }  Console.WriteLine("\n");  } |

Vom încerca fiecare număr impar începănd de la 3 până la jumătatea numărului.

|  |
| --- |
| public void IncercNumar(int Numar, int NumarDeIncercat)  {  if (NumerePrime.Contains(NumarDeIncercat))  {  if (NumerePrime.Contains(Numar - NumarDeIncercat))  {  AfisPereche(NumarDeIncercat, Numar - NumarDeIncercat);  }  else if (EstePrim(Numar - NumarDeIncercat))  {  NumerePrime.Add(Numar - NumarDeIncercat);  AfisPereche(NumarDeIncercat, Numar - NumarDeIncercat);  }  }  else if (EstePrim(NumarDeIncercat))  {  NumerePrime.Add(NumarDeIncercat);  if (NumerePrime.Contains(Numar - NumarDeIncercat))  {  AfisPereche(NumarDeIncercat, Numar - NumarDeIncercat);  }  else if (EstePrim(Numar - NumarDeIncercat))  {  NumerePrime.Add(Numar - NumarDeIncercat);  AfisPereche(NumarDeIncercat, Numar - NumarDeIncercat);  }  }  } |

Aici se decide defapt dacă o pereche este bună sau nu. În primul rând verificăm dacă lista cu numerele prime conține deja numărul cu care vrem să construim o pereche, dacă da asta înseamnă că numărul este prim, după aceasta verificăm dacă lista conține numărul Numar – NumarDeIncercat, unde acest număr va fi folosit ca un complement, și dacă ea este în listă atunci înseamnă că este prim și putem afișa perechea, dacă ea nu este în listă vom verifica dacă e prim și dacă este atunci stocăm numărul în listă și afișăm perechea. Dacă primul if returnează false atunci verificăm dacă NumarDeIncercat este prim, dacă e prim îl adăugăm în listă, după acesta verificăm dacă Numar – NumarDeIncercat este deja în listă, dacă da atunci afișăm perechea dacă nu atunci verificăm primalitatea numărului, dacă el este prim îl adăugăm în listă și afișăm perechea.

|  |
| --- |
| public bool EstePrim(int Numar)  {  if (Numar == 2) { return true; }  if (Numar % 2 == 0) { return false; }  for (int i = 3; i <= Math.Sqrt(Numar); i += 2)  {  if (Numar % i == 0) { return false; }  }  return true;  } |

Pentru verificarea primalității, vom verifica dacă numărul este par, dacă da atunci returnăm false. Pentru că singurul număr prim și par este numărul 2, întâi returnăm true dacă numărul este 2. După aceasta, verificăm dacă numărul este divizibil cu fiecare număr de la 3 pană la radăcina ei pătrată mergănd câte doi, dacă găsim un număr care este divizibil atunci returnăm false însă dacă nu găsim un număr care sa fie divizibil returnăm true.

|  |
| --- |
| public void AfisPereche(int Numar1, int Numar2)  {  Console.Write("( " + (Numar1) + " : " + (Numar2) + " ) ");  } |

Pentru afișarea unei perechi, vom folosi acest subprogram care primește ca parametri două numere întregi, fiecare fiind un număr din pereche.

Un obiect al clasei va fi instanțiat astfel:

|  |
| --- |
| Goldbach NumereGoldBach = new Goldbach(); |

Iar pentru generarea sumelori unui număr vom folosi următoarea instrucțiune:

|  |
| --- |
| NumereGoldBach.GenerareSecventa(Numar); |

Pentru a genera fiecare pereche între două numere, putem citi datele de la tastatură, și folosind un simplu for generăm perechea fiecărui număr.

|  |
| --- |
| int numarInitial = int.Parse(Console.ReadLine());  int numarFinal = int.Parse(Console.ReadLine());  for (int i = numarInitial; i <=numarFinal; i += 2)  {  NumereGoldBach.GenerareSecventa(i);  } |

## [AlgE] Cel mai mic palindrom mai mare decât n [\*\*\*]

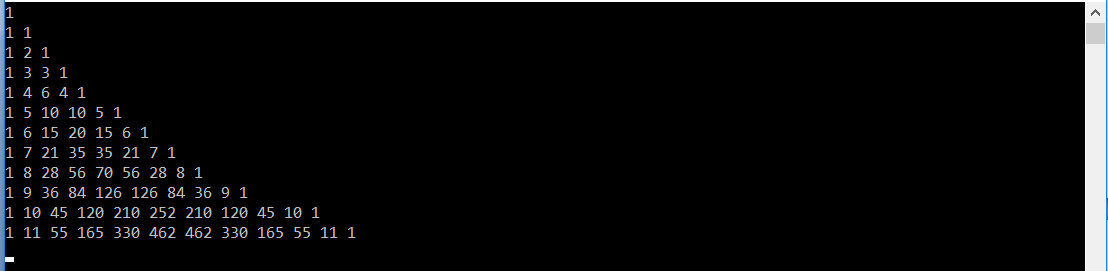
## [AlgE] Numere palindrom munte

Trebuie sa fie crescator si palindrom (12321)

## [AlgE] Triunghiul lui pascal

Pe mai multe rânduri

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  int n;  n = 12;  int[,] m = new int [n,n + 1];  for (int i = 0; i < n; i++)  for (int j = 0; j < n + 1; j++)  m[i, j] = 0;  m[0, 1] = 1;  for (int i = 1 ; i < n; i ++ )  for (int j = 1 ; j <= i + 1 ; j ++ )  m[i,j] = m[i-1,j - 1] + m[i-1,j];  for (int i = 0; i < n; i++)  {  for (int j = 1; j < n + 1; j++)  if (m[i, j] != 0)  Console.Write(m[i, j] + " ");  Console.WriteLine ();  }  Console.ReadKey();  }  } |



## [AlgE] Numere prime. (Ana)

O aplicație pe care o putem realiza cu ajutorul buclelor repetitive este determinarea proprietății de număr prim.

Un număr prim este acel număr care are exact doi divizori (cifra 1 și el însuși).

Vom lucra în funcția Main. Afișăm în consolă linia “n = ” și citim numărul n de la tastatură. Întrucât comanda Console.ReadLine() consideră datele introduse sub forma de string, pentru ca valoarea n să fie un întreg vom folosi funcția int.Parse. Vom număra divizorii numărului n într-o variabilă nrDivizori inițializată cu valoarea 0. Pentru a determina divizorii, vom parcurge un for de la 1 la n unde d este potențialul divizor. În cazul în care numarul n împărțit la d are restul egal cu 0, adică împărțirea se realizează exact, considerăm numarul curent ca divizor al lui n. După încheierea forului, ne vom întreba dacă numarul divizorilor lui n este doi, caz în care afișăm mesajul ‘’Numarul este prim‘’, altfel se va afisa mesajul ‘’Numarul nu este prim’’. Vom încheia algoritlul cu linia de cod Console.ReadKey(); pentru a putea citi rezultatul algoritmului din consolă, altfel aceasta se închide imediat după afișare.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("n = ");  int n = int.Parse(Console.ReadLine());  int nrDivizori = 0;  for (int d = 1; d <= n; d++)  if (n % d == 0) nrDivizori++;    if (nrDivizori == 2) Console.WriteLine("Numarul este prim");  else Console.WriteLine("Numarul nu este prim");  Console.ReadKey();  } |



Dar dacă dorim să verificăm pentru numere mai mari, de tipul long, după micile modificări în program, observăm că acesta se blochează.



Pentru a eficientiza codul, vom porni bucla for de la d=2 și ne vom opri la d=n/2. Din definiția numărului prim știm că numărul prim este acela care are ca divizori pe 1 și pe el însuși. Cu alte cuvinte, un număr care nu are divizori improprii. Astfel, la sfârșitul algoritmului ne vom întreba daca între numerele 2 și jumătatea numărului căutat (știm că n/2 este cel mai mare divizor mai mic decât numărul însuși) se găsesc alți diviziori. Dacă nu se gasesc alți divizori numărul este prim, altfel numărul nu este prim.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("n = ");  long n = long.Parse(Console.ReadLine());  int nrDivizori = 0;  for (long d = 2; d <= sqrt(n); d++)  if (n % d == 0) nrDivizori ++;    if (nrDivizori == 0) Console.WriteLine("Numarul este prim");  else Console.WriteLine("Numarul nu este prim");  Console.ReadKey();  }  } |

Pentru a face programul și mai eficient, vom introduce o variabilă bool EstePrim, care ia valoarea true la începutul programului, și prin care presupunem că numărul căutat este prim. Știm că un numar par nu este prim. Atunci ne întrebăm inițial dacă numărul este par, în cazul în care numărul este par, variabila EstePrim primește valoarea false, altfel vom începe să cautăm alți divizori proprii ai numerelor începând cu valoarea 3. d va parcurge doar numerele impare, întrucât niciun număr care nu se împarte la 2 nu are divizori pari. Parcurgem până la valoarea maximă , deoarece s-a demonstrat că, dacă un număr nu are niciun divizor până la radical din valoarea sa, nu va mai avea alți divizori. În momentul în care găsim un divizor impropriu, variabila EstePrim va lua valoarea false și prin break vom ieși din bucla for. După ieșirea din bucla for, ne vom întreba dacă variabila EstePrim a rămas true, caz în care spunem că numărul este prim, iar în caz contrar, numarul nu este prim.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("n = ");  long n = long.Parse(Console.ReadLine());  bool EstePrim = true;  if (n == 2) EstePrim = true;  else if (n % 2 == 0) EstePrim = false;  else  for (long d = 3; d <= Math.Sqrt(n); d = d + 2)  if (n % d == 0)  {  EstePrim = false;  break;  }  if (EstePrim == true) Console.WriteLine("Numarul este prim");  else Console.WriteLine("Numarul nu este prim");  Console.ReadKey();  } |

Pentru a exemplifica cea mai eficientă cale pentru a determina că un număr este prim sau nu este prin metoda NumărPrim. Aceasta metodă va verifica pentru noi daca numărul salvat în variabila n este prim sau nu. Vom verifica inițial daca numarul n este 1 sau 0, numere care nu sunt prime din definiție, returnăm false. Dacă n este 2, atunci vom returna true. Dacă n se împarte exact la 2, returnăm false. În cazul în care găsim alți divizori mai mari ca 3 și dacă pătratul numărului d îl depășește pe n, atunci numărul nu e prim, iar valoarea returnată va fi false. Folosim pătratul lui d deoarece funcția de calculare a radicalului este ineficientă pentru program, iar inmulțirea este operație elementară. Dacă numărul nu se află în niciunul dintre cazurile de mai sus, atunci numărul este prim și vom returna valoarea true. Această metodă se va apela în Main.

|  |
| --- |
| static bool EstePrim(long n)  {  if (n == 0 || n == 1) return false;  if (n == 2) return true;  if (n % 2 == 0) return false;  for (long d = 3; d \* d <= n; d += 2)  if (n % d == 0)  return false;  return true;  } |



## [AlgE] Numere perfecte. (Ana)

Un număr este perfect dacă suma divizorilor săi mai mici decât el însuși este egală cu acesta.

Îmi propun să exemplific acest tip de algoritm ca program în Main.

Programul începe cu cititul lui n de la tastatură și cu inițializarea valorii sumei cu 0. Vom parcurge cu ajutorul unui for toate numerele de la 1 la n/2, pentru a găsi divizorii numărului n. Odată găsiți, aceștia vor fi adăugați la sumă. La sfârșitul programului, ne vom întreba dacă suma divizorilor este egală cu numărul n, caz în care numărul este perfect, altfel putem spune că n nu este număr perfect.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("număr = ");  int nr = int.Parse(Console.ReadLine());  int SumaDiv = 0;  for (int d = 1; d <= nr / 2; d++)  if (nr % d == 0)  {  SumaDiv = SumaDiv + d;  }  if (SumaDiv == nr) Console.WriteLine("Numarul este perfect");  else Console.WriteLine("Numarul nu este perfect");  Console.ReadKey();  } |



## [AlgE] Ultima cifră a numărului n^n

|  |
| --- |
| Fie n un intreg 2< 100000000, se cere  Adică care este ultima cifră a numărului |
| Exemplu: n= 12,  Exemplu: n =17, |

Rezolvare:

Pentru rezolvare vom scrie un număr n ȋntreg, sub forma

Cu sensul ca ȋmpărțim numărul ȋn partea care se divide la 10 și restul acestei diviziuni

294 = 290 + 4

231237 = 231230 + 7

Astfel avem:

Pe baza desfășurării (3) (binomul lui Newton) avem

Din punct de vedere al ultimei cifre toți (mai puțin ultimul) termeni ai desfășurării sunt 0, deoarece, reprezintă din construcție un număr divizibil cu 10, (adică terminat ȋn cifra 0) iar ridicat la orice putere (1..n) va avea ultima cifră tot 0, și cum

Avem astfel toți termenii

Mai puțin ultimul

Cum

Ne rămâne ultima cifră ca fiind cea care influnețează valoarea căutată

Exemplu

Avem:

Avem

=

Cu alte cuvinte, pentru exemplul dat:

Continuând raționamentul de simplificare, observăm că pentru fiecare număr de forma unde c este o cifră, există repetitiv (cel mult cu perioada 4)

Exemplu pentru c=2

Vom scrie

Calculând repetitiv pentru fiecare cifrăvalorile obținem:

Putem ȋncepe implementarea folosind tabelul de valori anterior:

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  ulong n = ulong.Parse(Console.ReadLine());  switch (n % 10)  {  case 0: Console.WriteLine("0"); break;  case 1: Console.WriteLine("1"); break;  case 2:  switch (n % 4)  {  case 0: Console.WriteLine("6"); break;  case 1: Console.WriteLine("2"); break;  case 2: Console.WriteLine("4"); break;  case 3: Console.WriteLine("8"); break;  }  break;  case 3:  switch (n % 4)  {  case 0: Console.WriteLine("1"); break;  case 1: Console.WriteLine("3"); break;  case 2: Console.WriteLine("9"); break;  case 3: Console.WriteLine("7"); break;  }  break;  case 4:  switch (n % 2)  {  case 0: Console.WriteLine("6"); break;  case 1: Console.WriteLine("4"); break;  }  break;  case 5: Console.WriteLine("5"); break;  case 6: Console.WriteLine("6"); break;  case 7:  switch (n % 4)  {  case 0: Console.WriteLine("1"); break;  case 1: Console.WriteLine("7"); break;  case 2: Console.WriteLine("9"); break;  case 3: Console.WriteLine("3"); break;  }  break;  case 8:  switch (n % 4)  {  case 0: Console.WriteLine("6"); break;  case 1: Console.WriteLine("8"); break;  case 2: Console.WriteLine("4"); break;  case 3: Console.WriteLine("2"); break;  }  break;  case 9:  switch (n % 2)  {  case 0: Console.WriteLine("1"); break;  case 1: Console.WriteLine("9"); break;  }  break;  }  Console.ReadKey();  }  } |

La o analiză mai atentă (ȋn cazul evaluării , nu se folosesc toate datele din tabelul anterior (până acum am dezvoltat de fapt la cazul general ȋn care am construit pentru ), Dar de exemplu pentru cu (n%10), să zicem par, nu vom avea (n%4) impar. Cu alte cuvinte, pentru ultima cifră 2 valorilor posibile vor fi doar {4,6}, și astfel putem rescrie cazurile (Tinând cont ca pentru 3 și 7 avem doar cazurile cu rest modulo impar):

Care ar simplifica o parte din implementare.

Dacă rămânem la cazul general am putea construi o soluție dacă nu mai eficientă (ambii algoritmi prezentați anterior implementează doar o formulă matematică, fără calcule suplimentare repetitive, deci au același timp constant de execuție) ȋn primul rând mai elegantă prin construirea tabelului următor (notat A) cu rezultatele obținute ȋn urma calculului periodic al cifrelor la puterea x.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Rest 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Rest 2 | 0 | 1 | 4 | 9 | 6 | 5 | 6 | 9 | 4 | 1 |
| Rest 3 | 0 | 1 | 8 | 7 | 4 | 5 | 6 | 3 | 2 | 9 |
| Rest 0 | 0 | 1 | 6 | 1 | 6 | 5 | 6 | 1 | 6 | 1 |

Care de altfel rezolvă problema cerută prin simplul apel al elementului de pe linia n%4 și coloana n%10

Exemplu:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  int[,] A = new int[,]  {  {0,1,6,1,6,5,6,1,6,1},  {0,1,2,3,4,5,6,7,8,9},  {0,1,4,9,6,5,6,9,4,1},  {0,1,8,7,4,5,6,3,2,9}  };  ulong n = ulong.Parse(Console.ReadLine());  Console.WriteLine(A[n % 4, n % 10]);  Console.ReadKey();  }  } |

## [AlgE] Formații poker (varianta decizii)

Fie 5 numere a,b,c,d,e ȋntregi ȋn intervalul [2,14] Se cere să se afișeze unul din următoarele cazuri:

* “(0) Nimic” dacă nici o valoare nu este identică
* “(1) O pereche” dacă doar două valori sunt identice
* “(2) Două perechi” dacă două câte două valori sunt identice
* “(3) Trei bucăți” dacă 3 valori sunt identice
* “(4) Full” dacă 3 bucăți sunt identice și ȋn același timp și celelalte două valori rămase sunt identice ȋntre ele
* “(5) Patru bucăți” dacă 4 bucăți sunt identice
* “(6) Cinci bucăți” dacă toate cele 5 bucăți sunt identice

Exemple:

(2,9,9,4,1) : O pereche

(1,4,5,6,3) : Nimic

(8,8,9,1,8) : Trei bucăți

(1,1,2,2,3) : Două perechi

(8,1,8,1,1) : Full

Rezolvare:

Dacă ȋncercăm o abordare directă ar trebui să luăm ȋn calcul pentru fiecare formație ȋn parte toate posibilele scheme de apariție.

Să luăm ȋn considerare cazul ȋn care toate cele cinci sunt egale, acesta fiind și cel mai simplu, avem:

|  |
| --- |
| if ((a == b) && (a == c) && (a == d) && (a == e))  Console.WriteLine("(6) cinci bucati");  else { } |

Unde pe condiția de else ar urma toate celelalte verificări.

Pentru cazul ȋn care trebuie să verificăm dacă patru valori sunt identice, ar trebui să considerăm următoarele posibilități (unde 1 reprezintă valoarea activă din formație)

11110

11101

11011

10111

01111

Ceea ce s-ar scrie astfel:

|  |
| --- |
| if (((a == b) && (a == c) && (a == d)) || ((a == b) && (a == c) && (a == e)) ||  ((a == b) && (a == d) && (a == e)) || ((a == c) && (a == d) && (a == e)) ||  ((b == c) && (b == d) && (b == e)))  Console.WriteLine("(5) patru bucati");  else {...} |

Evident, pe ramura else urmănd celelate verificări,

Dacă luăm ȋn considerare toate variantele (ȋntr-o manieră ordonată) programul poate fi scris relativ ușor și chiar are rol de exercițiu:

Pentru formația de full am avea:

00111 01011 01101 01110 10011 10101 10110 11001 11010 11100

Diferența ȋntre formația de full și cea de 3 bucăți identice constă ȋn valorile 0 care pentru prima formație sunt parte componentă iar pentru a doua nu. Pentru două perechi sunt 15 cazuri [01122 01212 01221 10122 10212 10221 11022 12012 12021 11202 12102 12201 11220 12120 12210], iar pentru o pereche trebuie verificata fiecare variabila cu fiecare. (implementarea de mai jos). ȋn loc să citim valorile de la tastatură am generat aceste valori aleator ȋn intervalul dat și am introdus o variabilă de tip ȋntreg cr care scade de fiecare dată când generăm valorile a,b,c,d,e și crește cu o anumită valoare când se obțin formațiile cerute.

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  int a, b, c, d, e;  int cr = 10;  do  {  cr--;  a = rnd.Next(2, 15);  b = rnd.Next(2, 15);  c = rnd.Next(2, 15);  d = rnd.Next(2, 15);  e = rnd.Next(2, 15);  Console.WriteLine("{0} {1} {2} {3} {4}", a, b, c, d, e);  if ((a == b) && (a == c) && (a == d) && (a == e))  {  Console.WriteLine("(6) cinci bucati");  cr += 100;  }  else  {  if (((a == b) && (a == c) && (a == d)) || ((a == b) && (a == c) && (a == e)) || ((a == b) && (a == d) && (a == e)) || ((a == c) && (a == d) && (a == e)) || ((b == c) && (b == d) && (b == e)))  {  Console.WriteLine("(5) patru bucati");  cr += 50;  }  else  {  if (((a == b) && (c == d) && (c == e)) || ((a == c) && (b == d) && (b == e)) || ((a == d) && (b == c) && (b == e)) || ((a == e) && (b == c) && (b == d)) || ((b == c) && (a == d) && (a == e)) || ((b == d) && (a == c) && (a == e)) || ((b == e) && (a == c) && (a == d)) || ((c == d) && (a == b) && (a == e)) || ((c == e) && (a == b) && (a == d)) || ((d == e) && (a == b) && (a == c)))  {  Console.WriteLine("(4) full");  cr += 20;  }  else  {  if (((c == d) && (c == e)) || ((b == d) && (b == e)) ||((b == c) && (b == e)) || ((b == c) && (b == d)) || ((a == d) && (a == e)) || ((a == c) && (a == e)) || ((a == c) && (a == d)) || ((a == b) && (a == e)) || ((a == b) && (a == d)) || ((a == b) && (a == c)))  {  Console.WriteLine("(3) trei bucati");  cr += 3;  }  else  {  if (((b == c) && (d == e)) || ((b == d) && (c == e)) || ((b == e) && (c == d)) || ((a == c) && (d == e)) || ((a == d) && (c == e)) || ((a == e) && (c == d)) || ((a == b) && (d == e)) || ((a == d) && (b == e)) || ((a == e) && (b == d)) || ((a == b) && (c == e)) || ((a == c) && (b == e)) || ((a == e) && (b == c)) || ((a == b) && (c == d)) || ((a == c) && (b == d)) || ((a == d) && (b == c)))  {  Console.WriteLine("(2) Doua perechi");  cr += 2;  }  else  {  if ((a == b) || (a == c) || (a == d) || (a == e) || (b == c) || (b == d) || (b == e) || (c == d) || (c == e) || (d == e))  {  Console.WriteLine("(1) o pereche");  cr += 1;  }  else  Console.WriteLine("(0) Nimic");  }  }  }  }  }  Console.WriteLine("cr {0}", cr);  Console.ReadKey();  } while (cr > 0);  }  } |



Cu alte cuvinte problema poate fi rezolvată doar folosind verificări de identitate și operatori logici. Pentru o rezolvare mai lizibilă sau mai elegantă (mai eficientă nu avem cum deoarece timpul de execuție a algoritmului anterior este constant ) vom introduce o relație de ordine ȋntre elemente (cei familiarizați cu algoritmii de sortare, da, vom introduce o sortare) cu scopul de a obține următorul considerent: două valori identice vor fi sigur una lângă alta. Introducând această relație de ordine vom reduce considerabil numărul de cazuri pe care trebuie să le discutăm (implementăm).

Sa luăm exemplele anterioare:

Pentru 4 bucăți identice vom avea doar 11110 sau 01111, deoarece dacă sunt identice sunt una lângă alta, pentru full vom avea 11100 sau 00111, pentru 3 bucăți numărul cazurilor se reduce la 11100, 01110 și 00111, respectiv 2 perechi 01122, 11022 și 11220.

Dar să vedem ce ȋnseamnă sortarea ȋn acest moment (Dacă, să presupunem ȋn concurs nu primim posibilitatea să folosim vectori) avem totuși posibilitatea folosirii unui element constitutiv al mai multor algoritmi de sortare, interschimbarea. Cu alte cuvinte trebuie să interschimbăm ȋn așa fel valorile a,b,c,d,e astfel ȋncât să existe aceeași relație de ordine ȋntre fiecare două dintre ele.

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  int a, b, c, d, e;  int cr = 10;  do  {  cr--;  a = rnd.Next(2, 15);  b = rnd.Next(2, 15);  c = rnd.Next(2, 15);  d = rnd.Next(2, 15);  e = rnd.Next(2, 15);  Console.WriteLine("{0} {1} {2} {3} {4}", a, b, c, d, e);  if (a > b) { int t = a; a = b; b = t; }  if (a > c) { int t = a; a = c; c = t; }  if (a > d) { int t = a; a = d; d = t; }  if (a > e) { int t = a; a = e; e = t; }  if (b > c) { int t = b; b = c; c = t; }  if (b > d) { int t = b; b = d; d = t; }  if (b > e) { int t = b; b = e; e = t; }  if (c > d) { int t = c; c = d; d = t; }  if (c > e) { int t = c; c = e; e = t; }  if (c > d) { int t = d; d = e; e = t; }    if ((a == b) && (a == c) && (a == d) && (a == e))  {  Console.WriteLine("(6) cinci bucati");  cr += 100;  }  else  {  if (((a == b) && (a == c) && (a == d)) || ((b == c) && (b == d) && (b == e)))  {  Console.WriteLine("(5) patru bucati");  cr += 50;  }  else  {  if (((a == b) && (c == d) && (c == e)) || ((d == e) && (a == b) && (a == c)))  {  Console.WriteLine("(4) full");  cr += 20;  }  else  {  if (((c == d) && (c == e)) || ((b == c) && (b == d)) || ((a == b) && (a == c)))  {  Console.WriteLine("(3) trei bucati");  cr += 3;  }  else  {  if (((b == c) && (d == e)) || ((a == b) && (d == e)) || ((a == b) && (c == d)))  {  Console.WriteLine("(2) Doua perechi");  cr += 2;  }  else  {  if ((a == b) || (b == c) || (c == d) || (d == e))  {  Console.WriteLine("(1) o pereche");  cr += 1;  }  else  Console.WriteLine("(0) Nimic");  }  }  }  }  }  Console.WriteLine("cr {0}", cr);  Console.ReadKey();  } while (cr > 0);  }  } |

Există o serie de probleme (legate de fundamentele programării) care pot fi rezolvate ușor prin această tehnică de ordonare prin interschimabare repetată [vezi probleme]. Evident cei care cunosc deja structurile de date ȋnlănțuite (ȋn acest caz vectorii) pot să rețină valorile a,b,c,d,e ȋntr-o astfel de structură iar ordonarea acestuia fiind mult mai ușor de implementat.

ȋn particular această problemă se poate rezolva mult mai elegant, luând ȋn calcul că formațiile anterioare diferă una de cealaltă prin numărul de identități (valori identice ȋntre ele).

Dacă ne uităm cu atenție observăm că fiecare formație are un număr unic de identități, ȋn funcție de dificultatea acesteia, formațiile sunt date ȋn ordinea descrescătoare a probabilității de apariție.

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  int a, b, c, d, e;  int cr = 10;  do  {  cr--;  a = rnd.Next(2, 15);  b = rnd.Next(2, 15);  c = rnd.Next(2, 15);  d = rnd.Next(2, 15);  e = rnd.Next(2, 15);  Console.WriteLine("{0} {1} {2} {3} {4}", a, b, c, d, e);  int noid = 0;  if (a == b) noid++;  if (a == c) noid++;  if (a == d) noid++;  if (a == e) noid++;  if (b == c) noid++;  if (b == d) noid++;  if (b == e) noid++;  if (c == d) noid++;  if (c == e) noid++;  if (d == e) noid++;  switch (noid)  {  case 10:  Console.WriteLine("(6) cinci bucati");  cr += 100;  break;  case 6:  Console.WriteLine("(5) patru bucati");  cr += 50;  break;  case 4:  Console.WriteLine("(4) full");  cr += 20;  break;  case 3:  Console.WriteLine("(3) trei bucati");  cr += 3;  break;  case 2:  Console.WriteLine("(2) Doua perechi");  cr += 2;  break;  case 1:  Console.WriteLine("(1) o pereche");  cr += 1;  break;  default:  Console.WriteLine("(0) Nimic");  break;  }  Console.WriteLine("cr {0}", cr);  Console.ReadKey();  } while (cr > 0);  }  } |

Mecanismul anterior funcționează pentru formațiile de poker clasice (cu 5 cărți, adică 5 variabile ȋntregi), dar dacă sunt mai multe valori, apar o serie de formații diferite cu număr de identități identice, și prin urmare acest mecanism nu le poate identifica (De exemplu dacă avem 6 valori, 3 perechi au același număr de identități ca și 3 bucăți (3), sau formația de 3 bucăți cu 3bucăți au tot atâtea identități ca și 4 bucăți (6)). Pentru a rezolva problema pentru formații neconvenționale (cu un număr mai mare de 5 valori) sau eventual chiar pentru o generalizarea problemei de felul (se dau n elemente ȋntregi să se determine numărul maxim de valori identice), variantele anterioare nu mai sunt eficiente, sau chiar adaptabile. Pentru aceste probleme vom introduce conceptul de numărare (capitolul AlgSD1).

Probleme

|  |
| --- |
| [#559 pbinfo] Se dau 5 numere distincte. Să se determine suma celor mai mari 3 dintre ele.  [#009 pbinfo] Să se scrie un program care citeşte de la tastatură trei numere naturale și determină diferenţa dintre cel mai mare şi cel mai mic.  [#793 pbinfo] Se dau trei numere naturale a b c. Să se determine cea mai mare valoare care se poate obține prin înmulțirea a două dintre numere și adunarea rezultatului cu al treilea. |

## [AlgE] Algoritmul lui Euclid

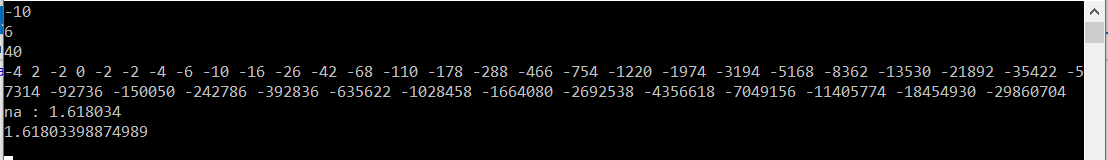
## [AlgE] Suma cifrelor unui număr

## [AlgE] Permutarea cifrelor unui număr

## [AlgE] Numărul lui Euler

**[AlgE] Numărul de aur**

|  |
| --- |
| static void Main(string[] args)  {  long a = int.Parse(Console.ReadLine());  long b = int.Parse(Console.ReadLine());  long s = 0;  int n = int .Parse(Console.ReadLine ());  for (int i = 0; i < n; i++)  {  s = a + b;  a = b;  b = s;  Console.Write( s + " ");  }  float na = (float)s / (float)a;  Console.WriteLine(na);  Console.ReadKey();  } |



## [AlgE] Fracții ireductibile (Alex)

Fracțiile ireductibile sunt acele fracții care simplificate (împărțind atât numitorul cât și numărătorul) cu CMMDC-ul lor, ajung la valori care nu mai pot fi simplificate.

CMMDC sau Cel mai mare divizor comun pentru două sau mai multe numere nenule este cel mai mare număr natural care divide toate numerele date.

Ex: CMMDC al numerelor a = 3, b = 9 și c=21 este 3.

Pentru CMMDC putem folosi algoritmul lui Euclid sau al scaderilor repetate.

|  |
| --- |
| int Euclid(int a, int b)  {  int c;  while (b!=0)  {  c = a % b;  a = b;  b = c;  }  return a;  }  int Cmmdc(int a, int b)  {  while (a != b)  if (a > b) a = a - b;  else b = b - a;  return a;  } |

Pentru a rezolva această problemă, trebuie mai întâi să creăm o funcție CMMDC, care ne returnează cel mai mare divizor comun al numitorului și al numărătorului, valoarea inițiala fiind 1. Putem folosi algoritmul lui Euclid (este mai eficient din punct de vedere al timpului de executie) sau algoritmul cu scăderi repetate.

!!! există patru forme de implementare a algoritmului lui Euclid

Ex: n = 3, m = 21

Exemplu pas cu pas:

1. Se face CMMDC(3,21) = 3

a>b? Nu => a = 3 b = 18 (b-a)

a>b? Nu => a = 3 b = 15

a>b? Nu => a = 3 b = 12

a>b? Nu => a = 3 b = 9

a>b? Nu => a = 3 b = 6

a>b? Nu => a = 3 b = 3

a==b? Da => return 3

2. Se împarte n și m cu CMMDC(3,21), adică n = 3/3 => n = 1, iar m = 21/3 => m = 7.

3. Se afișează răspunsul final: 3/21 = 1/7

Programul întreg:

|  |
| --- |
| class Program  {  static int cmmdc(int a, int b)  {  while (a != b) //Cat timp a diferit de b se executa adica pana ajungem la o valoare egala.  if (a > b) a -= b; //Daca a > b atunci din a scadem b  else b -= a; // altfel, adica daca b<=a atunci din b scadem a.  return a; // returnam valoarea lui a deoarece am iesit din while iar a = b;  }  static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int m = int.Parse(Console.ReadLine());  int cm = cmmdc(n, m);  Console.Write(n + " / " + m + " = " + n / cm + " / " + m / cm);  Console.ReadKey();  }  } |

!! Output

## [CG/AlgE] Conversia unei valori ȋntregi ȋn reprezentarea romană

În primul rând, să ne familiarizăm cu notația romană, și regulile acesteia (4).

1. Simbolurile I,X,C,M reprezintă 1,10,100,1000 și ȋn extensie considerăm simbolurile (X),(C),(M) de 1000 de ori simbolul dat (10.000, 100.000 și 1.000.000). Fiecare din acest simbol apare ȋn repetiție maximă de 3 ori [valoarea maximă pe care o putem scrie astfel (conform legii 1) este 3.999.999] deoarece la 4 milioane am avea deja (M)(M)(M)(M). Valorile V, L, D reprezentând 5,50 și 500 respectiv (V), (L), (D) cu valoarea 5000, 50.000 și 500.000 nu necesită repetiție. De fapt repetarea lor duce la o reprezentare invalidă a numărului.
2. Scrierea ȋn dreapta a unui simbol mai mic (sau egal) fața de un simbol mai mare duce la ȋnsumarea acestora. Ex. VI = 5+1 = 6, MCC = M + C+C =1200. LXV = 50 +10 +5 = 65, III = 1+1+1 =3
3. Scrierea ȋn stânga a unui simbol mai mic fața de un simbol mai mare duce la diferența acestora (cel mai mic din cel mai mare). Ex: IV = 5-1 = 4, XL =50-10 = 40, (C)(M) = 10k-1k =9k. Totdeauna ȋn diferență apar simbolurile cele mai apropiate putere a lui 10 față de simbolul de bază. Următoarele numere nu se scriu așa C(M) = 10000-100 = 9900 (singurul simbol ȋn diferență lângă (M) poate fi (C)), IL =49 (singurul lângă L poate fi X)
4. Ordinea de scriere dedusă din regulile anterioare, este 1= I,2 = II,3 = III, pentru 4 dacă am reprezenta IIII am ignora prima regulă, de aceea 4 se scrie ca fiind IV, 5 = V, 6 = VI, 7 =VII, 8 VIII și pentru 9, aceelași considerent ca și pentru 4, adică IX, 10 = X, după care se repetă 11 =XI, ... până la 39 = XXXIX, 40 = XL ș.a.m.d.

Algoritmic ne vom folosi de faptul că orice multiplu putere a lui 10, a unei valori date se scrie ȋn același mod doar cu simboluri diferite. Adică 4 se scrie ca fiind IV, (5-1), 40 se scrie XL (50-10), 400 se scrie (CD) (500-100), 4000 (M(V)) ș.a.m.d. Sau 7,70,700 ... toate fiind scrise ca simbolul de mijloc (5,50,500 ... ) la care se adaugă de două ori simbolul putere a lui 10 (1,10,100 ...). Cunoscând acest lucru, putem construi o funcție care la modul general returnează ordinea simbolurilor (1,2,3,... 9 multiplicate fiecare cu puterea lui 10). Dacă avem simbolul low (puterea lui ), simbolul med valoarea de mijloc și simbolul hig (puterea lui ) numeralele romane vor arăta așa (+ având sens de concatenare)

1: low,

2: low + low,

3: low + low + low,

4: low + med,

5: med,

6: med + low,

7: med + low + low

8: med + low + low + low

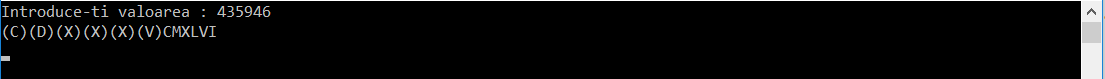
9: low + hig

O dată construită această funcție, nu mai trebuie decât să ȋmpărțim ȋn porțiuni (putere a lui 10) numărul dat și să apelăm funcția ȋn plaja de valori dată de puterea lui 10 la care suntem.

Ex: 435946 = 400000 + 30000 + 5000 +900 +40 + 6, adică

4 cu simbolurile ((M),(D),(C)), 3 cu simbolurile ((C),(L),(X)), 5 cu simbolurile ((X),(V), M), 9 cu simbolurile (M,D,C), 4 cu simbolurile (C,L,X) și 6 cu simbolurile (X,V,I)

Adică (C)(D)+(X)(X)(X)+(V)+CM+XL+VI



|  |
| --- |
| class Program  {  static string[] sr = new string[] { "(M)", "(D)", "(C)", "(L)", "(X)", "(V)", "M", "D", "C", "L", "X", "V", "I" };  static string tcr(int r, int idx)  {  string low = sr[idx];  string med = sr[idx - 1];  string hig = sr[idx - 2];  switch (r)  {  case 1: return low;  case 2: return low + low;  case 3: return low + low + low;  case 4: return low + med;  case 5: return med;  case 6: return med + low;  case 7: return med + low + low;  case 8: return med + low + low + low;  case 9: return low + hig;  default: return "";  }  }  static void Main(string[] args)  {  Console.Write("Introduce-ti valoarea : ");  int a = int.Parse(Console.ReadLine());  int[] v = new int[7];  v[0] = 1000000;  v[1] = 100000;  v[2] = 10000;  v[3] = 1000;  v[4] = 100;  v[5] = 10;  v[6] = 1;  string rez = "";  for (int i = 0; i < a/v[0]; i++)  rez += sr[0];  a = a % v[0];  for (int i = 1; i < 7; i++)  {  rez += tcr(a / v[i], 2 \* i);  a = a % v[i];  }  Console.WriteLine(rez);  Console.ReadKey();  }  } |

## [CG/AlgE] Conversia unei valori ȋn reprezentarea romană ȋn cifre arabe

|  |
| --- |
| #include <iostream>  using namespace std;  int symbol (char c)  {  switch (c)  {  case 'M' : return 1000; break;  case 'D' : return 500; break;  case 'C' : return 100; break;  case 'L' : return 50; break;  case 'X' : return 10; break;  case 'V' : return 5; break;  case 'I' : return 1; break;  }  }  void main ()  {  char s[100];  cin >>s;  int n = strlen (s);  int \* v = new int [n];  for (int i = 0 ; i< n ; i ++ )  v[i] = symbol (s[i]);  int value = v[n-1];  for (int i = 0 ; i < n - 1 ; i ++ )  if (v[i] <v[i+1])  value -= v[i];  else  value += v[i];    cout << value;  system ("pause");  } |

## [CG/AlgE] Conversia unui text la reprezentarea morse

## [AlgE] Generarea unui text aleator format din caractere mari, mici și cifre

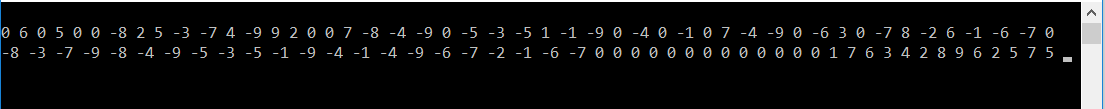
## [AlgSD1] Numărul cel mai mare și cel mai mic din cifrele unui număr

## [AlgSD1] Suma a două numere ȋntr-un vector

## [AlgSD1] Valori dintr-un vector cu cifre doar de 1 ȋn baza 2

## [AlgSD1] Ordonarea signum!!

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static int signum(int n)  {  if (n > 0) return 1;  if (n < 0) return -1;  return 0;  }  static void swap(ref int[] v, int p, int l)  {  int t = v[p];  v[p] = v[l];  v[l] = t;  }  static void Main(string[] args)  {  int n;  int[] v = new int[] {0, 6, 0, 5, 0, 0, -8, 2, 5, -3, -7, 4, -9, 9, 2, 0, 0, 7, -8, -4, -9, 0, -5, -3, -5, 1, -1, -9, 0, -4, 0, -1, 0, 7, -4, -9, 0, -6, 3, 0, -7, 8, -2, 6, -1, -6, -7, 0};  n = v.Length;  Console.WriteLine();  for (int i = 0; i < n; i++)  Console.Write(v[i] + " ");    int lkpm = -1;  int lkpz = -1;    for (int i = 0; i < n; i++)  {  switch (signum(v[i]))  {  case -1:  if ((lkpz == -1) && (lkpm == -1))  {  swap(ref v, 0, i);  lkpm = 0;  }  else  if ((lkpz == -1) && (lkpm != -1))  {  swap(ref v, i, lkpm + 1);  lkpm++;  }  else  if ((lkpz != -1) && (lkpm == -1))  {  swap(ref v, 0, i);  lkpm = 0;  swap(ref v, i, lkpz + 1);  lkpz++;  }  else  {  if (lkpm + 1 < n)  {  swap(ref v, i, lkpm + 1);  lkpm++;  }  if (lkpz + 1 < n)  {  swap(ref v, i, lkpz + 1);  lkpz++;  }  }  break;  case 0:  if ((lkpz == -1) && (lkpm == -1))  {  swap(ref v, 0, i);  lkpz = 0;  }  else  if ((lkpz == -1) && (lkpm != -1))  {  swap(ref v, i, lkpm + 1);  lkpz = lkpm + 1;  }  else  if ((lkpz != -1) && (lkpm == -1))  {  swap(ref v, i, lkpz + 1);  lkpz++;  }  else  {  if (lkpz + 1 < n)  {  swap(ref v, i, lkpz + 1);  lkpz++;  }  }  break;    }  }  Console.WriteLine();  for (int i = 0; i < n; i++)  Console.Write(v[i] + " ");  Console.ReadKey();  }  } |



## [AlgSD1] Concatenarea a doi vectori (Ana)

Concatenarea vectorilor este procesul prin care doi vectori se înlănțuiesc, dând naștere unui vector de dimensiune egală cu dimensiunile celor doi vectori adunate.

Pentru a ajunge la rezultatul dorit, vom folosi doi vectori deja populați astfel încât n este lungimea primului vector, iar m lungimea celui de-al doilea. Vom crea un nou vector, căruia îi vom atribui lungimea n+m. Vom copia elementele din vector1 în VectorConcatenat folosind atribuirea VectorConcatenat[i] = vector1[i] în timp ce parcurgem primul vector. Apoi vom copia elementele din vector2 în VectorConcatenat începând de la poziția n+1, până la sfârșitul vectorului VectorConcatenat, adică încă m pași.

Vom folosi o funcție de afișare a vectorului:

|  |
| --- |
| static void Afisare(int[] v)  {  for (int i = 0; i < v.Length; i++)  Console.Write(v[i] + " ");  Console.WriteLine();  } |

Prin parcurgerea elementelor și afișarea acestora pe rând, cu spații între ele.Vom afișa vectorii pe rând, știind că vctorii se scriu pe rânduri diferite.

|  |
| --- |
| static void Main(string[] args)  {  int[] vector1 = { 3, 2, 5, 8, 6, 1, 9 };  int[] vector2 = { 7, 2, 4, 9, 0, 3, 1, 6, 5, 8 };  int n = vector1.Length, m = vector2.Length;  int[] VectorConcatenat = new int[n + m];  for (int i = 0; i < n; i++)  VectorConcatenat[i] = vector1[i];  for (int i = 0; i < m; i++)  VectorConcatenat[n + i] = vector2[i];  Afisare(vector1);  Afisare(vector2);  Afisare(VectorConcatenat);  Console.ReadKey();  }  static void Afisare(int[] v)  {  for (int i = 0; i < v.Length; i++)  Console.Write(v[i] + " ");  Console.WriteLine();  } |



## [AlgSD1] Apartenența unui vector de dimensiune mai mică ȋntr-un vector de dimensiune mai mare

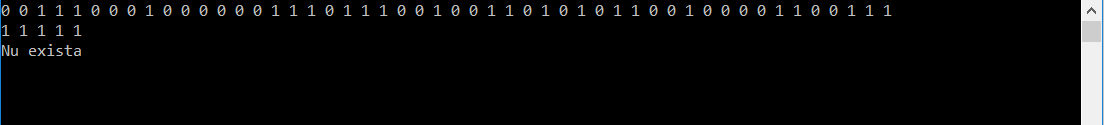
Se dau doi vectori de ȋntregi (v1 și v2), de dimensiune n1 și n2 cu n2<n1, se cere să se afișeze poziția de la care elementele vectorului v2 se regăsesc ȋn ordine, ȋn vectorul v1, eventual dacă acesta se găsește de mai multe ori se vor afișa toate pozițiile de ȋnceput, sau mesajul “nu există” ȋn caz contrar.

Programul generează cei doi vectori (nu avem nevoie să verificăm care dimensiune este mai mare deoarece se specifică) după care ne bazăm pe verificarea (v1[i] == v2[0]) numai ȋn acest caz putem, eventual avea o soluție. Dacă există această identitate, mergem mai departe să verificăm toate elemenetele rămase ale lui v2 cu toate elementele ulterioare poziției i (de unde ȋncepem verificarea) a lui v1. Nu are rost să contorizăm toate identitățile v1[i+j] cu v2[j], deoarece ne este necesar să găsim cel puțin una diferită, ca să nu existe soluție, caz ȋn care ieșim din verificarea curentă (variabila ok reține evenimentul ȋn care am găsit o diferență ȋntre v1[i+j] și v2[j]). Dacă cel puțin o dată variabila ok rămâne true la finalul parcurgerii celui de-al doi-lea vector, ȋnseamna că am găsit cel puțin o soluție pe care o și afișăm, iar variabila found o trecem ȋn true (prin intermediul acestei variabile ne asigurăm că vom afișa fie soluțiile dacă există, fie mesajul corespunzător “nu există”).

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  int n1 = 50; int[] v1 = new int[n1];  int n2 = 3; int[] v2 = new int[n2];    for (int i = 0; i < n1; i++)  {  v1[i] = rnd.Next(2);  Console.Write(v1[i] + " ");  }  Console.WriteLine();  for (int i = 0; i < n2; i++)  {  v2[i] = rnd.Next(2);  Console.Write(v2[i] + " ");  }  Console.WriteLine();  bool ok;  bool found = false;  for (int i = 0; i < n1 - n2 +1; i++)  {  if (v1[i] == v2[0])  {  ok = true;  for (int j = 1; j < n2; j++)  if (v1[i + j] != v2[j])  {  ok = false;  break;  }  if (ok)  {  Console.WriteLine(i);  found = true;  }  }  }  if (!found)  Console.WriteLine("Nu exista");  Console.ReadKey();  }  } |

Mai trebuie specificat că parcurgerea primului vector se face până la indicele n1-n2+1, cu sensul că dacă nu am găsit până la această valoare o identitate (v1[i] == v2[0]), nu mai are rost să căutăm deoarece și dacă am găsi ȋn continuare dimensiunea vectorului al doilea nu mai ȋncape ȋn ce a rămas ȋn vectorul ȋntâi, deci nu mai putem avea soluții.





## [AlgSD1] Vectorul de valori unice

Se dă un vector de ȋntregi, să se construiască un nou vector care conține valorile unice din vectorul inițial.

Ex:

2 23 3 2 1 22 3 2 1 1 0 23

Rezultat:

2 23 3 1 22 0

Eventual

0 1 2 3 22 23

În primul rând trebuie să facem distincție ȋntre problema afișării tuturor numerelor distincte și problema construirii vectorului de valori cu numerele disticte. În primul caz nu ne interesează câte valori sunt (deoarece nu trebuie să construim o structură de date cu aceste valori, trebuie doar să le afișăm).

O metodă elegantă ȋn c# (eventual și ȋn c++, dacă avem posibilitatea folosirii containerelor STL) este folosirea unei structui de date neindexate (o listă simplu ȋnlănțuită) care va reține toate elementele distincte ale vectorului inițial. Introducem ȋn listă prima valoare (aceasta este unică fiind singura) și apoi pentru toate celelalte elemente din vector dacă elementul curent ȋl găsim ȋn listă trecem la următorul fără să modificăm lista iar dacă nu există ȋl adaugăm ȋn listă. Astfel ȋn final lista va conține toate elementele din v care nu se repeat (subprogramul uniq\_list).

|  |
| --- |
| static int[] uniq\_list(int [] v)  {  List<int> toReturn = new List<int>();  toReturn.Add (v[0]);  for (int i = 1; i < v.Length; i++)  {  bool found = false;  foreach (int j in toReturn)  if (j == v[i])  {  found = true;  break;  }  if (!found)  toReturn.Add(v[i]);  }  return toReturn.ToArray();  } |

Putem construi același mecanism și fără liste folosindu-ne de un vector suplimentar (declarat global de dimensiune maximă n) (cu același scop ca și lista definită anterior). Doar că trebuie sa refacem calculul de final astfel ȋncât vectorul să nu conțină toate elementele doar pe cele unice (vectorul ar conține pe ultimile poziții valoarea 0 din definiție, iar aceste valori nu mai sunt rescrise deoarece nu mai există elemente distincte).

Exemplul anterior:

2 23 3 2 1 22 3 2 1 1 0 23

Rezultat:

2 23 3 1 22 0 0 0 0 0 0 0

Pentru a găsi dimensiunea vectorului de valori unice vom folosi o variabilă k, pe care o vom incrementa de fiecare dată când introducem un element nou ȋn vector. Prin metoda Resize<>() asociată obiectelor de tip vector ȋn c# putem trunchia vectorul la câte elemente dorim (ȋn acest caz k). ȋn caz că nu avem acces la metoda Resize, va trebui să definim un vector nou de dimensiune k și să copiem elementele lui toReturn ȋn acesta.

|  |
| --- |
| static int[] uniq\_array(int[] v)  {  int[] toReturn = new int[v.Length];  toReturn[0] = v[0];  int k = 1;  for (int i = 1; i < v.Length ; i++)  {  bool found = false;  for (int j = 0; j < k; j ++ )  if (toReturn[j] == v[i])  {  found = true;  break;  }  if (!found)  {  toReturn[k] = v[i];  k++;  }  }  Array.Resize<int>(ref toReturn, k);  return toReturn;  } |

Putem considera și o altă abordare pe baza conceptului de numărare (acolo unde elementele vectorului ne permit acest lucru). Dacă de exemplu am avea valorile vectorului ȋn intervalul [-100, 100], ȋnchis, ar trebui să numărăm 201 valori (de la -100 până la 100 (inclusiv) + valoarea 0). Vectorul de numărare ȋn acest caz ar avea dimensiunea 201 (0..200 și null pe poziția 201). Ne asigurăm că toate valorile sunt 0 (această etapă poate fi ignorată ȋn momentul când prin definiție vectorul se inițializează trece la valorile default (0 pentru ȋntregi). Pentru vectori dinamici ȋn c++ această etapă este obligatorie), după care trecem la procesarea vectorului v (să numărăm aparițiile acestuia). Vom face scalarea Count[0] va număra valorile -100, count[1] valorile -99 ș.a.m.d. count de [200] va număra valorile 100. După această etapă (având vectorul de numărare) este suficient doar să găsim valorile !=0 din acest vector. Având ȋn vedere numărul restrâns de date ȋntr-un vector de numărare, putem parcurge acest vector de două ori, prima dată să determinăm numărul de elemente diferite de 0, și apoi să construim vectorul cu aceste elemente.

ȋn continuare am parametrizat conceptul de numărare pentru o valoare minimă și maximă asociată elementelor vectorului.

|  |
| --- |
| static int[] uniq\_count(int[] v, int min, int max)  {  int cdim = max - min + 1;  int[] count = new int[max - min + 1];  for (int i = 0; i < v.Length; i++)  count[v[i] - min]++;  int no = 0;  for (int i = 0; i < cdim; i++)  if (count[i] != 0)  no++;  int[] toReturn = new int[no];  int k = 0;  for (int i = 0; i < cdim; i++)  if (count[i] != 0)  {  toReturn[k] = i + min;  k++;  }  return toReturn;  } |

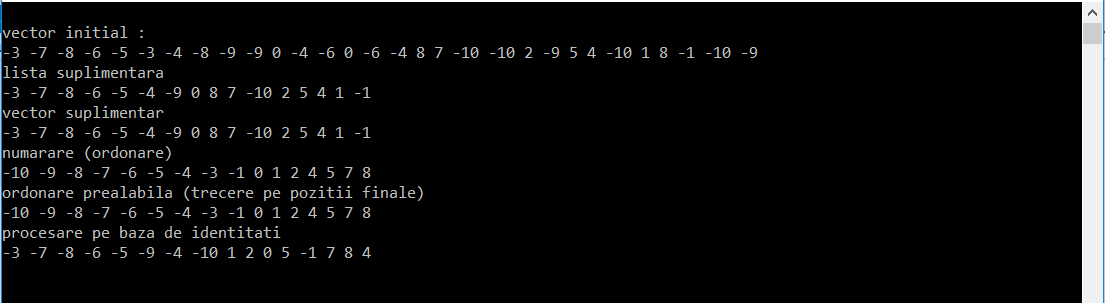
O altă abordare des ȋntâlnită se bazează pe ordonarea prealabilă a vectorului inițial. Tot aici vom renunța la vectorul suplimentar și ne vom folosi de pozițiile finale pentru a muta aici elementele care sunt dubluri.

|  |
| --- |
| static int[] uniq\_orded(int[] v)  {  v = sort(v);  int k = 0;  for (int i = 0; i < v.Length - 1 - k; i++)  {  if (v[i] == v[i + 1])  {  i--;  for (int j = i + 1; j < v.Length - k - 1; j++)  v[j] = v[j + 1];  k++;  }  }  Array.Resize<int>(ref v, v.Length - k);  return v;  } |

Mecanismul de mutare a dublurilor pe ultimile poziții merge și fără ordonarea prealabilă a vectorului. Când găsim două elemente identice, pe al doi-lea ȋl mutăm la sfârșit, dupa care lungimea vectorului descrește cu o unitate și refacem calculul de unde am rămas. ȋn metoda uniq\_noorded\_noarry am folosit variabila k pentru a reține poziția până la care am introdus dubluri (de la capătul vectorului) astfel că dimensiunea activă avectorului va fi totdeauna ȋntre i și v.Lenght - 1 – k. Decrementarea indicelui j are sens ȋn momentul ȋn care valoarea adusă de pe ultima poziție activă are aceeași valoare cu v[j], fără decrementare s-ar păstra dublură ȋntre (v[i] și v[j]), j- - forțează o reevaluare a poziție v[j] și ȋn urma egalitătii acesteia cu v[i], v[j] ar ajunge pe ultima poziție (doar că acum, ultima poziție este cu un indice mai puțin spre stânga ... și tot așa până când v[i]!=v[j]).

|  |
| --- |
| static int[] uniq\_noorded\_noarray(int[] v)  {  int k = 0;  for (int i = 0; i < v.Length - 1 - k; i++)  {  int j = i + 1;  while (j <= v.Length - k-1)  {  if (v[i] == v[j])  {    int t = v[j];  v[j] = v[v.Length - 1 - k];  v[v.Length - 1 - k] = v[j];  k++;  j--;  }  j++;  }  }  Array.Resize<int>(ref v, v.Length - k);  return v;  } |

|  |
| --- |
| class Program  {  static int[] sort(int[] v) {...}  static int[] uniq\_list(int [] v) {...}  static int[] uniq\_array(int[] v) {...}  static int[] uniq\_count(int[] v, int min, int max) {...}  static int[] uniq\_orded(int[] v) {...}  static int[] uniq\_noorded\_noarray(int[] v) {...}  static void afisare(string msg, int[] v)  {  Console.WriteLine(msg);  for (int i = 0; i < v.Length; i++)  Console.Write("{0} ", v[i]);  Console.WriteLine();  }  static Random rnd = new Random();  static void Main(string[] args)  {  int n1 = 30;  int[] v1 = new int[n1];  int[] vsave = new int[n1];  int min = -10;  int max = 10;  for (int i = 0; i < n1; i++)  {  v1[i] = rnd.Next(max - min + 1) + min;  vsave[i] = v1[i];  }  Console.WriteLine();  afisare("vector initial : ", v1);  int[] v2 = uniq\_list(v1);  afisare("lista suplimentara", v2);  int[] v3 = uniq\_array(v1);  afisare("vector suplimentar", v3);  int[] v4 = uniq\_count(v1, min, max);  afisare("numarare (ordonare)", v4);  int[] v5 = uniq\_orded(v1);  afisare("ordonare prealabila (trecere pe pozitii finale)", v5);  int[] v6 = uniq\_noorded\_noarray(vsave);  afisare("procesare pe baza de identitati", v6);    Console.ReadKey();  }  } |



Observații:

## [AlgSD1] Simulare reuniune, intersecție și difernță pe vectori (Marius)

Se dau doi vectori “a” și “b”. Să se simuleze reuniunea, intersecția si diferența a două mulțimi pe acești vectori.

Un mod de a rezolva această problemă este prin folosirea a 2 vectori caracteristici, unul pentru fiecare vector, care va conține numărul aparițiilor fiecărui element.

Numărul de elemente al celor doi vectori caracteristici va fi dat de numărul maxim aflat in vectorul “a” sau “b”. Vom declara o variabila de tip int “max” care va stoca numărul maxim si pe care o vom inițializa cu primul element din vectorul “a”. Vom parcurge vectorul “a” si vom verifica fiecare element daca este mai mare decât variabila “max”. În caz afirmativ, valoarea variabilei “max” va fi schimbată cu elementul respectiv. Vom repeta același procedeu si pentru vectorul “b”.

Acum că știm care este numărul maxim din cei doi vectori, putem să creăm alți doi vectori cu un număr de elemente egal cu variabila “max” +1.

Vom parcurge din nou ambii vectori “a” și “b” astfel că pentru fiecare element vom incrementa elementul din vectorul caracteristic corespunzător de pe poziția valorii elementului.

*Exemplu: elementul „5” de pe poziția „3” din vectorul „a” va incrementa elementul din vectorul caracteristic corespunzător lui „a” de pe poziția „5”.*

Acum că avem și cei doi vectori caracteristici, putem să afișăm reuniunea, intersecția și diferența celor doi vectori.

Vom începe cu reuniunea și vom proceda astfel: pentru fiecare număr de la 0 la „max” inclusiv, vom verifica daca există cel puțin o apariție în vectorul „a” sau „b”, adică verificam dacă elementul din vectorul caracteristic „vca” sau „vcb” de pe poziția „i” nu este egal cu 0 iar, in caz afirmativ vom afișa indexul acestuia reprezentând elementul care se regăsește in cel puțin unul dintre vectori.

Procedeul este similar pentru intersecție doar că vom verifica dacă există cel puțin o apariție în ambii vectori.

În continuare, reprezentarea diferenței o vom împărți în două, elementele care sunt în „a” și nu apar în  
 „b” și elementele care sunt in „b” și nu apar in „a”.

Procedeul este asemănător cu cel folosit pentru reuniune și intersecție. Pentru prima diferență vom verifica pentru fiecare număr de la 0 la „max” inclusiv dacă elementul din vectorul caracteristic corespunzător lui „a” nu este egal cu 0 și elementul din vectorul caracteristic corespunzător lui „b” este egal cu 0, iar pentru a doua diferența vom face exact opusul.

Acest program este realizat în întregime în metoda „Main”.

|  |
| --- |
| static void Main(string[] args)  {  int[] a = { 1,2,3,4,5,6,7,8,9,0};  int[] b = { 7,19,38,10,49,47,29,9,1,4,2,7,3,6,3,6,2,6,3,55};  int i;  int max = a[0];  for (i = 0; i < a.Length; i++)  if (a[i] > max)  max = a[i];  for (i = 0; i < b.Length; i++)  if (b[i] > max)  max = b[i];  int[] vca = new int[max+1];  int[] vcb = new int[max+1];    for (i = 0; i < a.Length; i++)  vca[a[i]]++;  for (i = 0; i < b.Length; i++)  vcb[b[i]]++;    Console.Write("Reuniune: ");  for (i = 0; i <= max; i++)  if (vca[i] != 0 || vcb[i] != 0)  Console.Write(i + " ");    Console.Write("\nIntersectie: ");  for (i = 0; i <= max; i++)  if (vca[i] != 0 && vcb[i] != 0)  Console.Write(i + " ");    Console.Write("\nDiferenta(a-b): ");  for (i = 0; i <= max; i++)  if (vca[i] != 0 && vcb[i] == 0)  Console.Write(i + " ");    Console.Write("\nDiferenta(b-a): ");  for (i = 0; i <= max; i++)  if (vca[i] == 0 && vcb[i] != 0)  Console.Write(i + " ");  Console.ReadKey();  } |

## !! output

## [AlgSD1] Apa dintr-un vector

Se dă un vector de valori ȋntregi care reprezintă altitudini. Să se determine câte unități de apă poate reține vectorul.

Ex: 2, 5, 3, 8, 1, 1, 1, 2, 5, 7, 7, 3, 2, 5, 4

Vectorul de altitudini (valoarea vectorului este notată cu X)

8 ...X...........

7 ...X.....XX....

6 ...X.....XX....

5 .X.X....XXX..X.

4 .X.X....XXX..XX

3 .XXX....XXXX.XX

2 XXXX...XXXXXXXX

1 XXXXXXXXXXXXXXX

Apa ce poate fi reținută de vector (ȋn acest caz) este reprezentată cu O

8 ...X...........

7 ...XOOOOOXX....

6 ...XOOOOOXX....

5 .XOXOOOOXXXOOX.

4 .XOXOOOOXXXOOXX

3 .XXXOOOOXXXXOXX

2 XXXXOOOXXXXXXXX

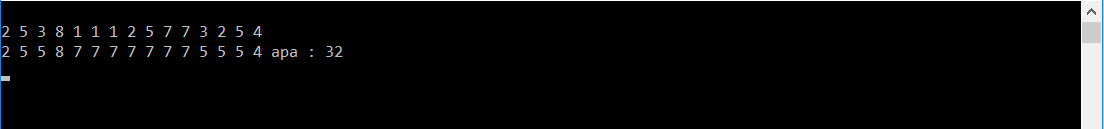
1 XXXXXXXXXXXXXXX

Pentru ȋnceput, ne vom folosi de proprietatea “apei” de a rămâne pe loc acolo unde este ȋnconjurată de valori mai mari ȋn stânga și ȋn dreapta. Vom ȋncerca pentru fiecare poziție din vector să introducem o valoare (unitate) de apă, dacă aceasta rămâne acolo incrementăm valoarea din vector și refacem calculul, dacă nu, trecem la următoarea poziție din vector.

În exemplul anterior pentru valoarea v[0] nu avem vecin mai mare ȋn stânga, deci trecem mai departe (oricum pentru orice valoare nu avem pe margini, de fapt nici nu le vom verifica) la v[1] cu valoarea 5 nu avem ȋn stâga, mergem mai departe, la v[2] cu valoarea 3, avem pentru valoarea 4 vecin și ȋn stânga și ȋn drepata, deci deja avem o unitate de apă și incrementăm v[2] la 4, pentru 5 mai avem o unitate de apă și pentru 6 nu mai avem ȋn stânga, trecem la v[3] ș.a.m.d. Este important doar să observăm că trebuie să verificăm nu doar vecinul imediat, ci și următorul, până când găsim o valoare cel puțin egală sau ajungem peste marginea vectorului.

|  |
| --- |
| class Program  {  static Random rnd = new Random();  static void Main(string[] args)  {  int n = 13;  int apa = 0;  int[] v = new int[n];  for (int i = 0; i < n; i++)  v[i] = rnd.Next(10);  Console.WriteLine();  for (int i = 0; i < n; i++)  Console.Write(v[i] + " ");  bool ok;  for (int i = 1; i < n-1; i++)  {  do  {  ok = true;  int value = v[i] + 1;  bool br = false;  bool bl = false;  int j = i;  while (j >= 0)  {  if (value <= v[j])  {  bl = true;  break;  }  j--;  }  j = i;  while (j < n)  {  if (value <= v[j])  {  br = true;  break;  }  j++;  }  if (bl && br)  {  v[i]++;  apa++;  ok = false;  }  } while (!ok);  }  Console.WriteLine();  for (int i = 0; i < n; i++)  Console.Write(v[i] + " ");  Console.WriteLine("apa : " + apa);  Console.ReadKey();  }  } |

În implementarea anterioară se pot observa cele două instrucțiuni while care caută un vecin de valoare mai mare sau egală cu valoarea curentă. Existența celor doi vecini este reținută ȋn variabilele bl și br. Dacă ambele variabile, la sfârșitul celor două cicluri repetitive, au valoarea true pe aceea poziție din vector putem insera o unitate de apă (apă ++) și trebuie să verificăm valoarea imediat superioară (acest lucru se realizează prin intermediul variabilei ok care forțează verificarea valorii v[i]+1). Toate aceste calcule se fac pentru fiecare index din vector (mai puțin prima și ultima poziție care oricum nu pot reține apă).



Se poate observa ȋn imaginea anterioară modul cum procesează algoritmul datele. Vectorul al doi-lea este rezultatul procesării și au fost introduse 32 de unităti ȋn acesta (oricum am mai ȋncerca să introducem o valoare ȋn vector aceasta sau nu are limită ȋn stânga sau ȋn dreapta).

O primă ȋmbunătățire a algoritmului poate consta ȋn ȋncercarea de a pune (după principiul anterior) apa (câtă ȋncape) pe fiecare coloană:

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  #include <time.h>  using namespace std;  int main ()  {  srand (time (NULL));  int n;  cin >> n;  int \* u = new int [n];  for (int i = 0 ; i < n ; i ++ )  u[i] = rand () %10;  for (int i = 0 ; i < n ; i ++ )  cout << u [i] << " ";  cout << endl;  int apa = 0;  for (int i = 1; i < n-1 ; i ++ )  {  bool in;  do  {  in = false;  bool left = false;  bool right = false;    for (int j = i -1 ; j >= 0 ; j -- )  if (u[j] > u[i])  {  left = true;  break;  }  for (int j = i + 1 ; j < n ; j ++ )  if (u[j] > u[i])  {  right = true;  break;  }  if (left && right)  {  u[i] ++;  apa ++;  in = true;  }  }  while (in);  }  cout << "apa = " << apa << endl;  system("pausE");  return 0;  } |

În secțiunea DC (Divide et Imperra / Divide and Conquer) vom studia o altă abordare a acestei probleme prin ȋmpărțirea vectorului inițial ȋn “bazine” pentru care limitele sunt și maxime de valori iar rezolvarea acestora nu necesită decât o singură parcurgere.

## [AlgSD1] Numere mari. 5000! (Remus)

Factorialul unui număr natural n, notat cu n!, este produsul numerelor naturale mai mici sau egale cu n:

Observăm că factorialul crește exponențial și nu poate fi stocat pe int sau long, deci avem nevoie de o modalitate de a salva numerele foarte mari. Pentru acest program, am ales să salvăm numerele într-un vector de cifre.

Vom crea metoda ntov (number to vector). Tipul de return va fi int[], iar ca și parametru, vom primi un număr int n. Pentru a nu modifica valoarea lui n, vom crea o nouă variabila x egală cu n, și vom calcula numărul de cifre în variabila k. Creăm un vector v de dimensiune k, și parcurgem cifrele numărului x, salvându-le în vectorul v. Folosim un for care parcurge descrescător, pentru a putea pune ultima cifră a lui x pe ultima poziție a lui v, ș.a.m.d.

|  |
| --- |
| static int[] ntov(int n)  {  int x = n, k = 0;  while (x > 0)  {  x /= 10;  k++;  }  int[] v = new int[k];  x = n;  for (int i = k - 1; i >= 0; i--)  {  v[i] = x % 10;  x /= 10;  }  return v;  } |

Vom avea nevoie și de o metodă ce primește ca și parametru un vector, și returnează vectorul cu elementele pe poziții răsturnate (primul element se va pune pe ultima poziție, al doilea pe penultima etc.). Vom parcurge prima jumătate a elementelor vectorului si le vom interschimba cu elementele de pe pozițiile corespunzătoare.

|  |
| --- |
| static int[] Inv(int[] v)  {  for (int i = 0; i < v.Length / 2; i++)  {  int t = v[i];  v[i] = v[v.Length - 1 - i];  v[v.Length - 1 - i] = t;  }  return v;  } |

Trebuie să simulăm înmulțirea (și implicit adunarea) pentru vectori de cifre. Pentru asta, trebuie să ne folosim de metoda clasică de înmulțire a două numere:

**Metoda de adunare**

Aceasta primește doi vectori de cifre ca și parametri, si va returna vectorul de cifre al sumei lor. Suma dintre două numere se face începând cu cifrele de la final, deci pentru ușurință, aplicăm metoda de inversare pentru ambii vectori. Avem nevoie de numărul de cifre maxim și cel minim dintre cei doi vectori, deci vom salva valorile acestea în două variabile min și max. Observăm că oricât de mari sunt cele două numere, suma lor are cel mult max+1 cifre, deci vectorul în care vom salva suma este inițializat cu dimensiunea max+1.

Parcurgem elementele până la min într-un for și salvăm suma de pe poziția i a celor doi vectori în v[i]. De valorile ce nu sunt cifre ne vom ocupa mai târziu. Pozițiile din vector ce nu au înca valori vor primi valorile de pe pozițiile corespunzătoare ale vectorului mai mare. Cum nu știm care este cel mai mare vector, putem scrie două for-uri ce încep cu lungimea celuilalt vector si se termina la lungimea max. Astfel, dacă am ales vectorul mai mic, i va fi inițializat cu lungimea celuilalt vector, care este egală cu max, și când se ajunge la verificare, i<max este fals, deci acel for nu va face nimic. Ceea ce înseamnă că elementele puse în vector vor fi mereu cele corecte.

Următorul pas este rezolvarea problemei valorilor din vector ce nu sunt cifre.

Parcurgem întreg vectorul sumă, și dacă v[i] nu este cifră, pentru următoarea poziție adăugăm 1, iar v[i] va rămâne ultima cifră a sa. Deoarece vom primi la sumă vectori din urma unei înmulțiri, este posibil ca elemente ale vectorului să fie mai mari decât 20. Prin urmare, în loc să adăugăm 1 la următoarea poziție, vom adăuga partea întreagă a lui v[i].

Observăm că valorile salvate în v sunt cifrele în ordine răsturnată, deci vom returna Inv(v).

|  |
| --- |
| static int[] Suma(int[] v1, int[] v2)  {  v1 = Inv(v1);  v2 = Inv(v2);  int min = v1.Length, max = v1.Length;  if (v2.Length < min)  min = v2.Length;  else  max = v2.Length;  int[] v = new int[max + 1];  for (int i = 0; i < min; i++)  v[i] = v1[i] + v2[i];  for (int i = v1.Length; i < max; i++)  v[i] = v2[i];  for (int i = v2.Length; i < max; i++)  v[i] = v1[i];  for (int i = 0; i <= max; i++)  if (v[i] > 9)  {  v[i + 1] += v[i] / 10;  v[i] %= 10;  }  return Inv(v);  } |

**Metoda de înmulțire**

Avem aceiași parametri și tip de return ca și la sumă. Înmulțirea se realizează astfel: luăm cifrele numărului mai mic și le înmulțim pe rând cu celălalt număr, după care rezultatele sunt așezate, unul sub altul, cu ultima cifră sub cifra cu care s-a făcut înmulțirea. Același efect se poate realiza dacă punem 0-uri la final: la primul rezultat nu se pune niciun 0, la al doilea rezultat se pune un 0, la al treilea două 0-uri etc. Cum în informatică indexarea se începe cu 0, putem spune că adăugăm i 0-uri la final. Apoi, aceste rezultate sunt adunate între ele și finalul este rezultatul produsului.

Înmulțirea începe cu ultimele cifre ale numerelor, deci ne vom folosi din nou de metoda de inversare. Avem nevoie să știm care este cel mai mic dintre cele două numere, prin urmare dacă b este mai mare decât a, interschimbăm vectorii. Vom salva dimensiunile vectorilor în două variabile max și min. Observăm că lungimea maximă a produsului dintre două numere este suma lungimilor lor, așadar declarăm vectorul v și îl inițializăm cu lungimea max+min. Parcurgem cifrele primului număr, cu i de la 0 până la min, și creăm un vector auxiliar. Vom păstra primele i poziții de valoare 0, și pe celelalte poziții vom face produs între cifra actuală și întreg numarul de lungime max, deci inițializăm vectorul cu lungimea i+max. Trebuie să parcurgem toate cifrele numărului mai lung, cu j de la 0 la max, și să facem produsul între b[i] și a[j]. Cum primele i poziții sunt 0, rezultatul este salvat pe poziția i+j a vectorului auxiliar. Rezultatul obținut se adună la rezultatul precedent, dar cum cifrele sunt salvate în ordine inversă, vom aduna Inv(v).

Problema valorilor ce nu sunt cifre se rezolvă în metoda de adunare, în concluzie putem să returnăm v.

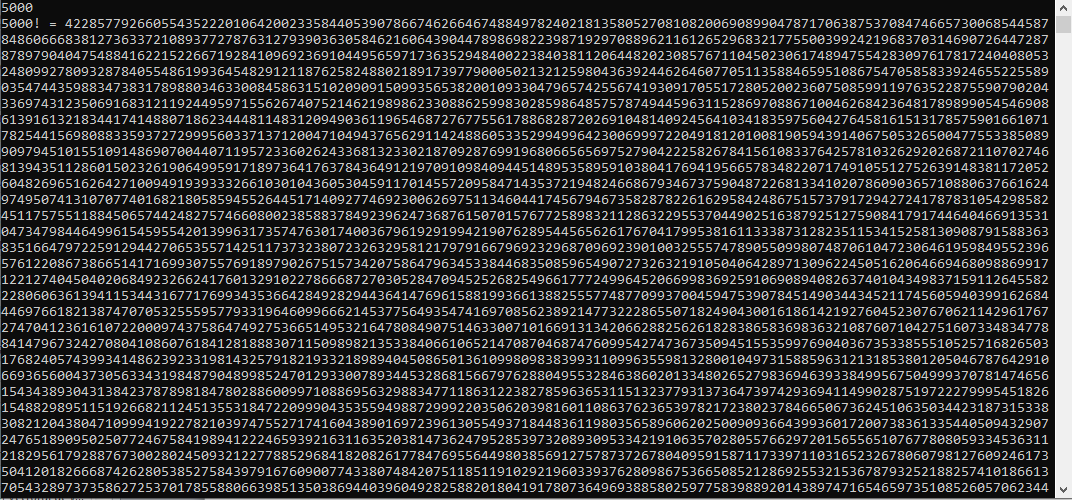
|  |
| --- |
| static int[] Produs(int[] a, int[] b)  {  a = Inv(a);  b = Inv(b);  if (b.Length > a.Length)  {  int[] t = a; a = b; b = t;  }  int max = a.Length, min = b.Length;  int[] v = new int[max + min];  for (int i = 0; i < min; i++)  {  int[] aux = new int[i + max];  for (int j = 0; j < max; j++)  aux[i + j] = b[i] \* a[j];  v = Suma(v, Inv(aux));  }  return v;  } |

Acum avem tot ce ne trebuie pentru a calcula factorialul. Metoda recursivă originală poate calcula până la valoarea lui 20!, dacă folosim valori long în loc de int. Ne vom folosi de această metodă și o vom adapta pentru codul nostru. Tipul de return este int[], iar ca și parametru vom primi int n. Dacă n este 0 sau 1, vom returna vectorul format din cifra 1. În caz contrar, vom face produsul între (n-1)! și n, adică vom apela metoda Produs pentru Factorial(n-1) și vectorul de cifre al lui n, adică ntov(n).

|  |
| --- |
| static int[] Factorial(int n)  {  if (n == 0 || n == 1)  return new int[] { 1 };  return Produs(Factorial(n - 1), ntov(n));  } |

În metoda Main, vom citi n de la tastatură, vom apela metoda Factorial pentru n, și vom afișa rezultatul. Cum de fiecare dată vectorul rezultat a avut dimensiunea maximă posibilă, avem multe 0-uri în fața rezultatului actual. Deci, cât timp întalnim numai 0, indexul va crește. După aceea, vom salva rezultatul într-un string și îl vom afișa. Cu acest program putem calcula valoarea lui 5000! în mai puțin de un minut.

|  |
| --- |
| static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int[] factorial = Factorial(n);  int i = 0;  while (factorial[i] == 0)  i++;  string str = $"{n}! = ";  for (; i < factorial.Length; i++)  str = $"{str}{factorial[i]}";  Console.WriteLine(str);  } |



**[Alg] Calcul cu date abstracte (Remus)**

SymbolsEquation

## [AlgSD1] Grâul pe tabla de șah

Există mai multe origini asociate cu această problemă, ȋn orice caz ȋn vechime undeva ȋn jurul datei ȋn care, Sissa ben Dahir (inventatorului jocului de șah), primește de la regele indian Shirham o recompensă pentru invenția sa (aprox. 1250) (4). Acesta cere un bob de grâu pentru primul pătrat de pe tablă, două boabe de grâu pentru al doilea, 4 pentru al treilea, 8 pentru al patrulea și tot așa până când toate pătratele sunt umplute.

Considerând că greutatea unui bob de grâu este de aproximativ 65mg, calculați greutatea totală de pe tabla de șah (eventual dacă sunt necesari vom considera următorii multiplii 1sac = 50kg grâu, 1 Vagon = 22 Tone de grâu, 1 Tren = 41 vagoane), eventual exprimați această cantitate ȋn multiplii dați.

Pentru rezolvarea acestei probleme să observăm că de fapt căutăm suma unei progresii geometrice cu rația 2.

Avem formula sumei unei progresii geometrice:

Unde

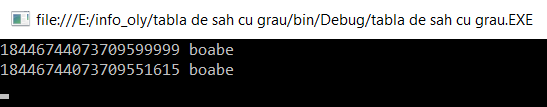
Obținem astfen numărul căutat ca fiind

Calculul se poate observa mai simplu transformând numerele ȋn baza 2

Vom avea:

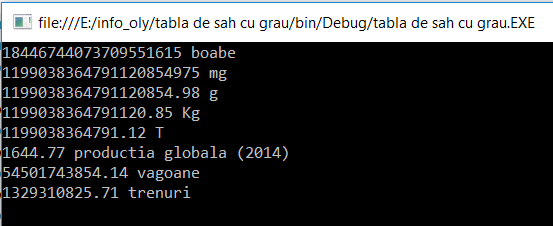
O dată stabilit calculul necesar problema este trivială, pentru implementare vom folosi tipul de dată Decimal (ulong depășește valoarea căutată) (2). Mai trebuie să observăm eroarea de conversie dată de folosirea Math.Pow (baza, exponent) (eroarea se datorează transformării la Decimal a valorii double) față de construcția (naivă) a funcției myPower (bază, exponent)

|  |
| --- |
| class Program    {  static Decimal myPow(int baza, int exponent)  {  Decimal toR = baza;  for (int i = 1; i < exponent; i++)  toR \*= baza;  return toR;  }  static void Main(string[] args)  {  Decimal boabeA = (Decimal)Math.Pow(2, 64) - 1;  Console.WriteLine("{0:00} boabe", boabeA);  Decimal boabeB = myPow(2, 64) - 1;  Console.WriteLine("{0:00} boabe", boabeB);  Console.ReadKey();  }  } |



Nu rămâne decât să ȋnțelegem acest număr (boabeB) (ȋn multiplii dați de problemă)

|  |
| --- |
| class Program  {  static Decimal myPow(int baza, int exponent) {...}  static void Main(string[] args)  {  Decimal boabeB = myPow(2, 64) - 1;  Decimal global2014 = 729000000;  Console.WriteLine("{0:00} boabe", boabeB);    boabeB \*= 65;  Console.WriteLine("{0:00} mg", boabeB);  boabeB /= 1000;  Console.WriteLine("{0:.00} g", boabeB);  boabeB /= 1000;  Console.WriteLine("{0:.00} Kg", boabeB);  boabeB /= 1000;  Console.WriteLine("{0:.00} T",boabeB);  Console.WriteLine("{0:.00} productia globala (2014)", boabeB / global2014);  boabeB /= 22;  Console.WriteLine("{0:.00} vagoane",boabeB);  boabeB /= 41;  Console.WriteLine("{0:.00} trenuri", boabeB);  Console.ReadKey();  }  } |



## [AlgSD1] Sortare Bubble (Ana)

Sortarea este o procedură foarte importantă atunci când lucrăm cu vectori. Una dintre cele cele mai simple metode de sortare este sortarea Bubble sau sortarea prin metoda bulelor.

Numele metodei a fost dat după modul în care elemente mai mici sau mai mari plutesc până la capatul vectorului precum bulele de aer care se ridică din apă.

Înainte de a explica sortarea propriu-zisă, vom crea metode pentru atribuirea valorilor aleatoare într-un vector, cât și de afișare a vectorului.

|  |
| --- |
| static void Initializare(int[] v)  {  Random r = new Random();  for (int i = 0; i < v.Length; i++)  v[i] = r.Next(100);  }  static void Afisare(int[] v)  {  for (int i = 0; i < v.Length; i++)  Console.Write(v[i] + " ");  Console.WriteLine();  } |

Cea mai importantă parte din cod este inteschimbarea elementelor consecutive atunci când acestea nu se află în ordinea aleasă (crescătoare sau descrescătoare). Acest test se realizează parcurgând vectorul. Din moment ce lucrăm cu perechi de elemente, se parcurge vectorul până la penultimul element.

|  |
| --- |
| For (int I = 0; I < v.Length – 1; i++)  if (v[i] > v[i + 1])  {  int aux = v[i];  v[i] = v[i + 1];  v[i + 1] = aux;  } |

Acest proces se va realiza atât timp cât vectorul nu este sortat. Putem verifica aceasta folosind o variabilă bool. Sortarea se repetă până cănd vectorul este ordonat. La fiecare pas presupunem ca vectorul este ordonat, iar la prima interschimbare variabila semafor ok ia valoarea false.

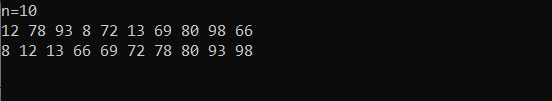
|  |
| --- |
| static void SortareaBubble(int[] v)  {  bool ok;  do  {  ok = true;  for (int i = 0; i < v.Length - 1; i++)  if (v[i] > v[i + 1])  {  int aux = v[i];  v[i] = v[i + 1];  v[i + 1] = aux;  ok = false;  }  } while (!ok);  } |

Pentru o mică optimizare a codului, vom adăuga o variabilă k, inițializată cu 0, care are ca scop blocarea elementelor care se află deja pe pozițiile potrivite de la finalul vectorului. Deci, după fiecare iterație acesta crește. Astfel vectorul se parcurge cu câte o poziție în minus.

|  |
| --- |
| static void SortareaBubble(int[] v)  {  int k = 0;  bool ok;  do  {  ok = true;  for (int i = 0; i < v.Length - 1 - k; i++)  if (v[i] > v[i + 1])  {  int aux = v[i];  v[i] = v[i + 1];  v[i + 1] = aux;  ok = false;  }  k++;  } while (!ok);  } |

Pentru a se realiza programul, metodele trebuie apelate în metoda Main. Se citește n de la tastatură și creăm un vector cu n elemente. Apelam metodele: inițializare, afișare, sortare și, din nou, metoda de afișare.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("n=");  int n = int.Parse(Console.ReadLine());  int[] vector = new int[n];  Initializare(vector);  Afisare(vector);  SortareaBubble(vector);  Afisare(vector);  Console.ReadKey();  } |



## [AlgSD1] Sortare Insertion (Ana)

## Sortarea prin inserție este altă sortare importantă în cadrul programării clasice. Timpul de execuție este de O(n\*log n), fiind la fel de eficientă precum sortarea prin metoda bulelor.

## Această sortare funcționează în felul următor: se parcurg toate elementele începând de la j=1, iar la fiecare pas facem o parcurgere inversă pornind de la i=j până la i>0. Acest lucru asigură ca toate elementele până la valoarea j sunt sortate.

## În cazul în care urmărim să sortăm vectorul crescător ne întrebăm dacă elementul de pe poziția i-1 este mai mare decât cel de pe poziția i, caz în care vom interschimba elementele. În caz contrar vom ieșii din bucla for actuală prin cuvântul cheie break, iar programul se va executa în continuare cu următoarea valoare a lui j. Pentru sortarea descrescătoare se schimbă semnul de comparație din if.

## Pentru a putea urmări mai bine progresul sortării, vom afișa vectorul la fiecare pas al lui i.

## Vom folosi metodele de afișare și initializare de la sortarea bubble, iar în metoda Main se apelează SortareaPrinInsertie în loc de SortareaBubble.

|  |
| --- |
| static void SortarePrinInsertie(int[] v)  {  for (int j = 1; j < v.Length; j++)  for (int i = j; i > 0; i--)  {  if (v[i-1] > v[i])  {  int aux = v[i];  v[i] = v[i - 1];  v[i - 1] = aux;  }  else break;  Afisare(v);  } } |

## 

## [AlgSD1] Sortare Selecție (stg, dr, stg+dr) (Ana)

## Această metodă de sortare este una mai eficientă decât sortarea bubble și cea de inserție, deoarece ordonarea vectorului prin această metodă se realizează prin mai puține interschimbări.

Această sortare se construiește pe baza unei bucle for prin care se parcurge vctorul. Vom folosi o variabilă pivot care ia valoarea elementului curent de pe poziția j. Vom compara acest element cu toate elmentele de după el până la sfârșitul vectorului, iar de fiecare dată când elementul curent și elementul pivot nu sunt în ordine crescătoare (pentru aranjarea crescătoare), valoarea pivotului se schimbă cu valoarea indiceui elementului cu care se află în conflict. După parcurgerea celorlaltor elemente din vector, se face o interschimbare a elementelor de pe poziția j și poziția pivot (prin metoda celor 3 pahare). În altă ordine de ideei găsesc elementul minimul de după poziția j și se interschimbă cu elem de pe poziția j. Astfel primul element a ajuns pe poziția lui, procedeul repetându-se până când toate elementele ajung pe poziția lor pe măsură ce j înaintează în vector.

## Vom folosi metodele de afișare și initializare de la sortarea bubble, iar în metoda Main se apelează SortareaPrinSelectie în loc de SortareaBubble.

|  |
| --- |
| static void SortareaPrinSelectie(int[] v)  {  for (int j = 0; j < v.Length; j++)  {  int pivot = j;  for (int i = j + 1; i < v.Length; i++)  if (v[i] < v[pivot])  pivot = i;  int aux = v[j];  v[j] = v[pivot];  v[pivot] = aux;  }  } |



## [AlgSD1] Counting sort

## [AlgSD1] Ciurul lui Eratostene (Marius)

Se dă un număr n, se cere afișarea numerelor prime de la 0 la n.

Pentru realizarea acestui program vom folosi ciurul lui Eratostene care este un algoritm simplu de descoperire a tuturor numerelor prime până la un întreg specificat.

Algoritmul constă în crearea unei colectii ce va conține numerele de la 2 la n. Se marchează fiecare număr neprim cu 0 astfel:

* Începând de la 2 se selectează toate numerele care sunt multiplu de 2 cu 0 .
* Primul număr nemarcat după 2 este 3 care este un număr prim, se repetă procedeul de marcare a multiplilor și pentru acesta.
* Se repetă acest procedeu până la numărul n/2.
* După executarea acestor pași în colectie vor rămâne doar numerele prime.

Implementare și îmbunătățiri:

Vom crea o funcție de tip void care ia ca parametru o variabilă de tip int "n". În funcție creăm un vector de tip int de dimesiune n+1 și îl inițializăm cu numerele de la 2 la n.

Parcurgem elementele vectorului de la 2 la n/2, verificăm dacă elementul curent nu este egal cu 0, după care parcurgem multiplii elementului curent astfel: într-un for parcurgem elementele de la 2\*i până la n mergând din i în i, verificăm dacă sunt diferite de 0 și le setăm cu valoarea 0. Astfel că vor rămâne doar elementele prime în vector și 0 în locul elementelor neprime. La sfârșitul funcției vom afișa toate elementele diferite de 0.

|  |
| --- |
| public static void Eratostene(int n)  {  int[] c = new int[n + 1];  for (int i = 2; i < n+1; i++)  c[i] = i;  for(int i = 2; i <= n / 2; i++)  if(c[i]!=0)  for (int j = 2 \* i; j <= n; j += i)  if (c[j] != 0)  c[j] = 0;  for(int i=2;i<=n;i++)  if(c[i]!=0)  Console.Write(c[i]+" ");  } |

## [AlgSD2] Camere alăturate (Administrator)

## [AlgSD2] Drumul cel mai scurt ȋn labirint

## [AlgSD2] Parcurgerea matricei ȋn spirala (Remus)

Se dă o matrice de dimensiune nxn. Parcurgeți elementele acesteia în spirală.

Ex.:

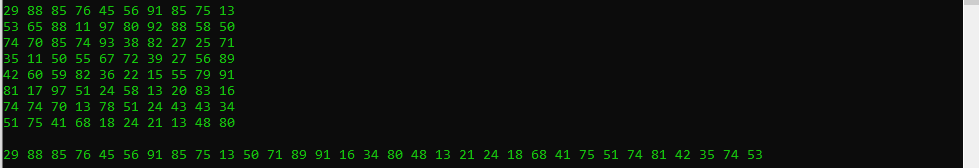
Output: 12, 3, 7, 9, 6, 2, 11, 5, 8.

Înainte să începem, vom crea o metodă pentru a inițializa matricea cu valori aleatoare și o metodă pentru afișarea unei matrice.

|  |
| --- |
| static int[,] Init (int n, int m)  {  int[,] matrix = new int[n, m];  Random r = new Random();  for (int i = 0; i < n; i++)  for (int j = 0; j < m; j++)  matrix[i, j] = r.Next(10, 100);  return matrix;  }  static void Afisare(int[,] matrix)  {  for (int i = 0; i < matrix.GetLength(0); i++)  {  for (int j = 0; j < matrix.GetLength(1); j++)  Console.Write(matrix[i, j] + " ");  Console.WriteLine();  }  Console.WriteLine();  } |

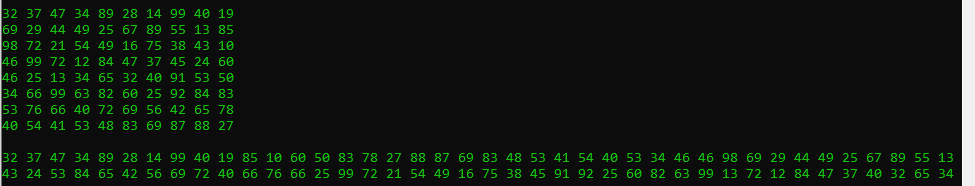
În primul rând, vom studia metoda de parcurgere a conturului unei matrice. Parcurgem elementele de pe prima linie, de pe ultima coloană, de pe ultima linie în ordine inversă, respectiv de pe prima coloană în ordine inversă. Pentru a nu repeta colțurile, la fiecare parcurgere, excludem ultimul element.

|  |
| --- |
| static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int m = int.Parse(Console.ReadLine());  int[,] matrix = Init(n, m);  Afisare(matrix);  for (int i = 0; i < m - 1; i++)  Console.Write(matrix[0, i] + " ");  for (int i = 0; i < n - 1; i++)  Console.Write(matrix[i, m - 1] + " ");  for (int i = m - 1; i >= 1; i--)  Console.Write(matrix[n - 1, i] + " ");  for (int i = n - 1; i >= 1; i--)  Console.Write(matrix[i, 0] + " ");  Console.ReadKey();  } |



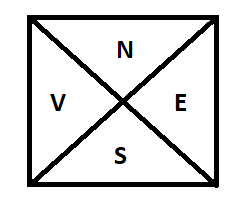
Pentru a parcurge toată matricea în spirală, vom continua parcurgerea și pentru liniile și coloanele interioare, până când ajungem la mijlocul matricei. Acest efect se poate realiza folosind o variabilă k care ia valori de la 0 la n/2 sau de la 0 la m/2, dacă m este mai mic. Vom exclude primele k valori și ultimele k-1 valori de pe fiecare linie.

|  |
| --- |
| for (int k = 0; k < Math.Min(n, m) / 2; k++)  {  for (int i = k; i < m - 1 - k; i++)  Console.Write(matrix[k, i] + " ");  for (int i = k; i < n - 1 - k; i++)  Console.Write(matrix[i, m - 1 - k] + " ");  for (int i = m - 1 - k; i >= k + 1; i--)  Console.Write(matrix[n - 1 - k, i] + " ");  for (int i = n - 1 - k; i >= k + 1; i--)  Console.Write(matrix[i, k] + " ");  } |



## [AlgSD2] Suma NSEV (Remus)

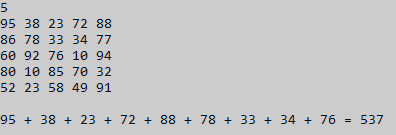
Un lucru important în folosirea matricelor este determinarea corectă a elementelor pe care dorim să le prelucrăm. Un exercițiu bun pentru aceasta este calcularea sumei elementelor din regiunile nord, sud, est si vest ale matricei. Acestea sunt în felul următor:



Vom folosi metodele de inițializare și de afișare de la algoritmul precedent. Pentru ușurință, păstrăm elementele de pe diagonale în toate regiunile.

Dacă parcurgerea, în mod normal, se face cu i și cu j de la 0 la n, putem considera parcurgerea regiunilor ca fiind o parcurgere mai restrânsă a elementelor. i nu este nevoie să meargă până la n, ci doar până la n/2 inclusiv, iar j trebuie să excludă câte un element la început și la sfârșit in plus la fiecare iterație. Acest efect se poate realiza prin parcurgerea elementelor de la i până la n-i. Vom calcula suma acestor elemente și vom crea un string pentru a afișa rezultatul.

|  |
| --- |
| static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int[,] matrix = Init(n, n);  Afisare(matrix);  string ecuatie = "";  int suma = 0;  for (int i = 0; i <= n / 2; i++)  for (int j = i; j < n - i; j++)  {  ecuatie += matrix[i, j] + " + ";  suma += matrix[i, j];  }  ecuatie = ecuatie.Remove(ecuatie.Length - 2) + "= " + suma;  Console.WriteLine(ecuatie);  Console.ReadKey();  } |

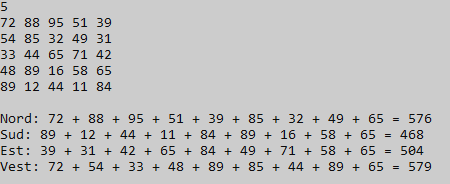


Pentru regiunea sud, elementele încep de la jumătate până la n, și dimensiunea lui j crește. Dar forma regiunii este aceeași. Prin urmare, decât să modificăm parcurgerea, propun să modificăm doar indicii. Linia pe care toate elementele sale sunt afișate este chiar ultima din matrice, adica linia n-1. Următoarea este penultima linie, adică linia n-2. În concluzie, parcurgerea poate să rămână aceeași, dar vom alege elementele de pe poziția n-i-1 și j de fiecare dată.

Pentru regiunile est și vest, observăm că parcurgerea lui j se potrivește pentru linie, nu pentru coloană. Ne folosim de acest lucru, și pentru vest alegem elementul de pe poziția j și i, iar pentru est cel de pe poziția j și n-i-1.

Pentru a nu repeta parcurgerile de 4 ori, vom face toate acțiunile în aceeași parcurgere, salvând valorile necesare în mai multe variabile.

|  |
| --- |
| static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int[,] matrix = Init(n, n);  Afisare(matrix);  string ecuatieN = "", ecuatieS = "", ecuatieE = "", ecuatieV = "";  int sumaN = 0, sumaS = 0, sumaE = 0, sumaV = 0;  for (int i = 0; i <= n / 2; i++)  for (int j = i; j < n - i; j++)  {  ecuatieN += matrix[i, j] + " + ";  sumaN += matrix[i, j];  ecuatieS += matrix[n-i-1, j] + " + ";  sumaS += matrix[n-i-1, j];  ecuatieE += matrix[j, n-i-1] + " + ";  sumaE += matrix[j, n-i-1];  ecuatieV += matrix[j, i] + " + ";  sumaV += matrix[j, i];  }  ecuatieN = ecuatieN.Remove(ecuatieN.Length - 2) + "= " + sumaN;  ecuatieS = ecuatieS.Remove(ecuatieS.Length - 2) + "= " + sumaS;  ecuatieE = ecuatieE.Remove(ecuatieE.Length - 2) + "= " + sumaE;  ecuatieV = ecuatieV.Remove(ecuatieV.Length - 2) + "= " + sumaV;  Console.WriteLine("Nord: " + ecuatieN);  Console.WriteLine("Sud: " + ecuatieS);  Console.WriteLine("Est: " + ecuatieE);  Console.WriteLine("Vest: " + ecuatieV);  Console.ReadKey();  } |



## [AlgSD2] Topirea unui iceberg (Zsolti)

Această problemă este pentru determinarea numărului de pași în care se topește un iceberg în apă. O bucată de gheață se topește dacă are cel puțin 2 din cei 4 vecini ai lui ca fiind apă.

Soluția pe care o vom folosi va fi să rulăm câte un pas până la topirea completă, iar la final vom vedea câți pași am făcut.

Pentru început, vom avea nevoie să codăm spațiul în care ne aflăm, în cazul nostru acesta fiind un tablou bidimensional (unde 0 – apă, 1 – gheață) , inițializat cu valori prestabilite pentru a putea urmări.

static int[,] iceberg;

static void Main(string[] args)

{

iceberg = InitializeazaIcerbergul();

Afiseaza();

}

private static int[,] InitializeazaIcerbergul()

{

return new int[,]{

{1,0,0,0,0,0,0,0,0,0},

{0,0,0,0,0,1,1,0,0,0},

{0,0,1,1,1,1,1,1,0,1},

{0,0,1,1,1,1,1,1,1,0},

{0,1,1,1,1,1,1,1,1,0},

{0,1,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,0,0,0,0},

{0,0,0,0,0,0,0,0,0,0}

};

}

private static void Afiseaza()

{

for (int i = 0; i < iceberg.GetLength(0); i++) {

for (int j = 0; j < iceberg.GetLength(1); j++)

{

if (iceberg[i, j] == 1)

Console.ForegroundColor = ConsoleColor.Blue;

else

Console.ForegroundColor = ConsoleColor.White;

Console.Write("#");

}

Console.WriteLine();

}

}

După rularea codului, ar trebui să apară icebergul în starea inițială:



Următorul lucru pe care trebuie să-l facem este să executăm un pas din topire.

private static void Topeste()

{

int[,] \_iceberg = new int[iceberg.GetLength(0), iceberg.GetLength(1)];

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

\_iceberg[i, j] = iceberg[i,j];

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

{

if (iceberg[i, j] == 1 && NumarVecini(i, j) > 1)

\_iceberg[i, j] = 0;

}

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

iceberg[i, j] = \_iceberg[i, j];

}

Se poate observa că funcția este alcătuită din 3 părți, împărțită de for-urile duble. Pentru a putea face verificări și modificări (topiri) corecte, trebuie să ne asigurăm că nicio modificare nu va încurca verificările și vice-versa. Pentru asta, vom crea o copie a spațiului în care ne aflăm înainte de orice modificare. Prima și ultima parte a funcției se ocupă doar de copierea și resetarea acestuia după modificări. În a două parte se fac modificările (topirile). Condiția reprezintă faptul că în spațiul respectiv [i,j] este gheață și are cel puțin 2 vecini apă.

private static int NumarVecini(int i,int j)

{

int nrVecini = 0;

if (i-1>=0 && iceberg[i-1, j] == 0)

nrVecini++;

if (i+1< iceberg.GetLength(0) && iceberg[i+1, j] == 0)

nrVecini++;

if (j - 1 >= 0 && iceberg[i, j-1] == 0)

nrVecini++;

if (j+1< iceberg.GetLength(1) && iceberg[i, j+1] == 0)

nrVecini++;

return nrVecini;

}

După pasul rulat, icebergul va arată așa:



Iar din punctul acesta, singurul lucru rămas este să se repete acest proces până când icebergul s-a topit complet. Sugerez cititorilor să continue din acest punct și apoi să vadă codul integral pentru verificare.

class Program

{

static int[,] iceberg;

static void Main(string[] args)

{

iceberg = InitializeazaIcerbergul();

Afiseaza();

int nrPasi = 0;

while (PotTopi())

{

Topeste();

nrPasi++;

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("Pasul numarul: "+nrPasi);

Afiseaza();

}

Console.ReadKey();

}

private static void Topeste()

{

int[,] \_iceberg = new int[iceberg.GetLength(0), iceberg.GetLength(1)];

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

\_iceberg[i, j] = iceberg[i,j];

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

{

if (iceberg[i, j] == 1 && NumarVecini(i, j) > 1)

\_iceberg[i, j] = 0;

}

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

iceberg[i, j] = \_iceberg[i, j];

}

private static int NumarVecini(int i,int j)

{

int nrVecini = 0;

if (i-1>=0 && iceberg[i-1, j] == 0)

nrVecini++;

if (i+1< iceberg.GetLength(0) && iceberg[i+1, j] == 0)

nrVecini++;

if (j - 1 >= 0 && iceberg[i, j-1] == 0)

nrVecini++;

if (j+1< iceberg.GetLength(1) && iceberg[i, j+1] == 0)

nrVecini++;

return nrVecini;

}

private static bool PotTopi()

{

for (int i = 0; i < iceberg.GetLength(0); i++)

for (int j = 0; j < iceberg.GetLength(1); j++)

if (iceberg[i, j] == 1)

return true;

return false;

}

private static int[,] InitializeazaIcerbergul()

{

return new int[,]{

{1,0,0,0,0,0,0,0,0,0},

{0,0,0,0,0,1,1,0,0,0},

{0,0,1,1,1,1,1,1,0,1},

{0,0,1,1,1,1,1,1,1,0},

{0,1,1,1,1,1,1,1,1,0},

{0,1,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,1,1,1,1,0,0,0,0},

{0,0,0,0,0,0,0,0,0,0}

};

}

private static void Afiseaza()

{

for (int i = 0; i < iceberg.GetLength(0); i++) {

for (int j = 0; j < iceberg.GetLength(1); j++)

{

if (iceberg[i, j] == 1)

Console.ForegroundColor = ConsoleColor.Blue;

else

Console.ForegroundColor = ConsoleColor.White;

Console.Write("#");

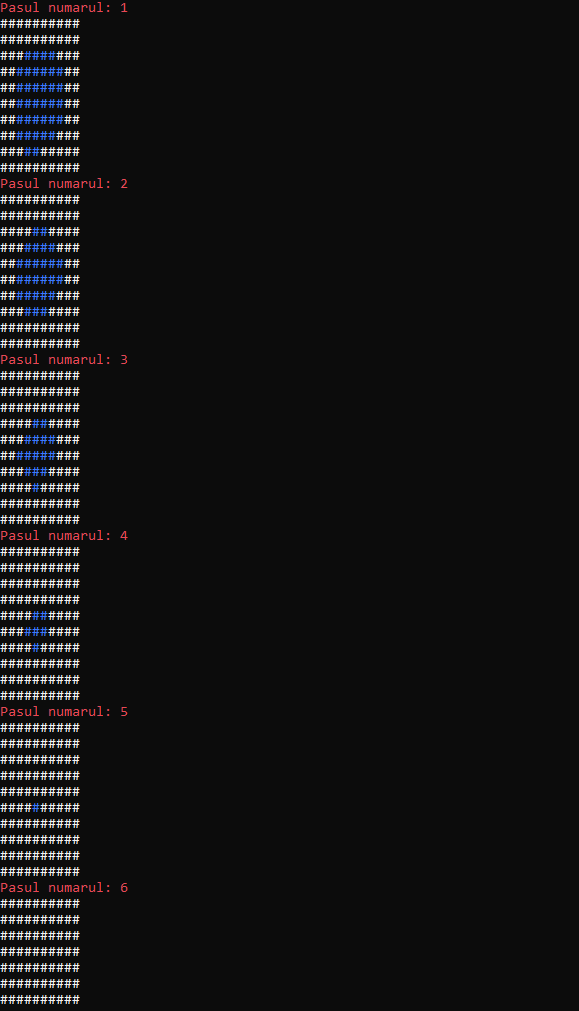
}

Console.WriteLine();

}

}

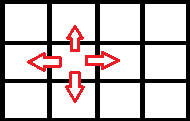
}



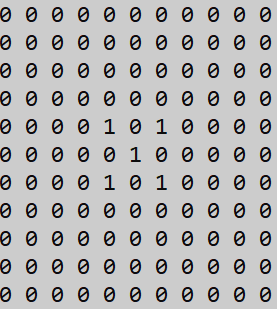
# [AlgSD2] Automate celulare (Fredkin) (Remus)

Automatele celulare sunt matrice ale căror elemente au un număr finit de stări, cum ar fi pornit și oprit. Pentru fiecare element, definim un set de elemente ca fiind vecinii acestuia. Se pornește de la o stare inițială, iar în următoarea generație, fiecare element este determinat de o regulă matematică în funcție de vecinii acestuia. Această regulă nu se modifică în timp și se aplică întregii matrice simultan.

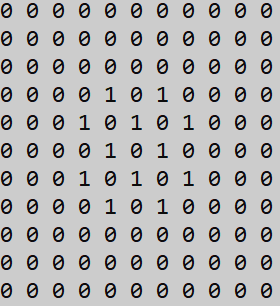
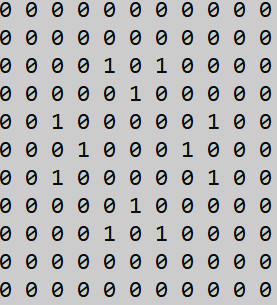
Pentru realizarea programului, am decis ca vecinii elementelor să fie 4: elementele de deasupra, de sub, din stânga și din dreapta fiecărui element. Excepțiile sunt elementele de pe marginea matricei, care au doar 3 sau 2 vecini.

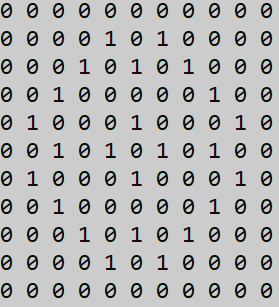
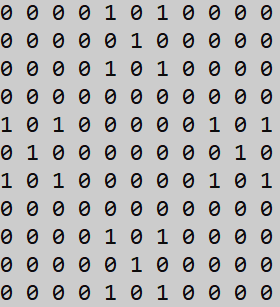


Stările elementelor de pornit și oprit sunt evidențiate prin valorile 0, respectiv 1 atribuite acestor elemente. Pentru starea inițială, vom avea valoarea 1 pentru elementul din centrul matricei și pentru elementele din jurul acestuia care nu sunt vecinii lui.



Regula matematică după care se decid generațiile viitoare este aceasta: pentru fiecare element din generația curentă, dacă suma vecinilor săi este pară, atunci acesta primește valoarea 0, iar în caz contrar, acesta primește valoarea 1. Câteva dintre generațiile ce urmează sunt ilustrate mai jos:

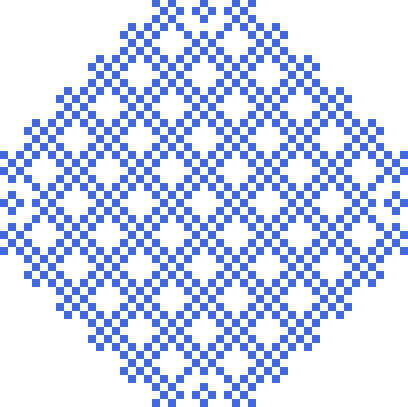
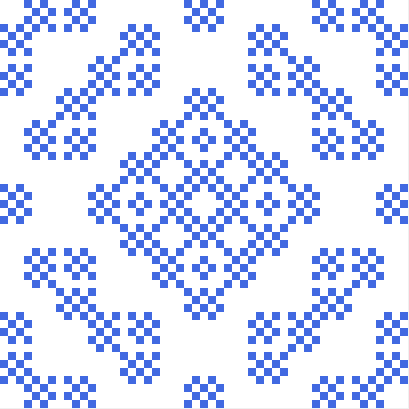
Pentru realizarea programului, avem nevoie de două matrice, a doua având rolul de a stoca valorile următoarei generații. Vom declara o variabilă n pentru dimensiunea matricelor, care trebuie să aibă o valoare impară, pentru a avea o matrice simetrică. Pentru a evita mai multe verificări, dimensiunea matricelor va fi n+2, n+2, iar la prelucrarea și modificarea matricelor, vom face parcurgerile de la 1 până la n inclusiv. Atribuim valoarea 1 elementelor dorite în starea inițială și apelăm metoda de afișare a matricei.

|  |
| --- |
| static int[,] matrix, temp;  static int n;  static void Main(string[] args)  {  n = 11;  matrix = new int[n + 2, n + 2];  temp = new int[n + 2, n + 2];  for (int i = 0; i < n + 2; i++)  for (int j = 0; j < n + 2; j++)  {  matrix[i, j] = 0;  temp[i, j] = 0;  }  matrix[n / 2 + 1, n / 2 + 1] = 1;  matrix[n / 2, n / 2] = 1;  matrix[n / 2 + 2, n / 2] = 1;  matrix[n / 2, n / 2 + 2] = 1;  matrix[n / 2 + 2, n / 2 + 2] = 1;  Afisare();  ...  } |

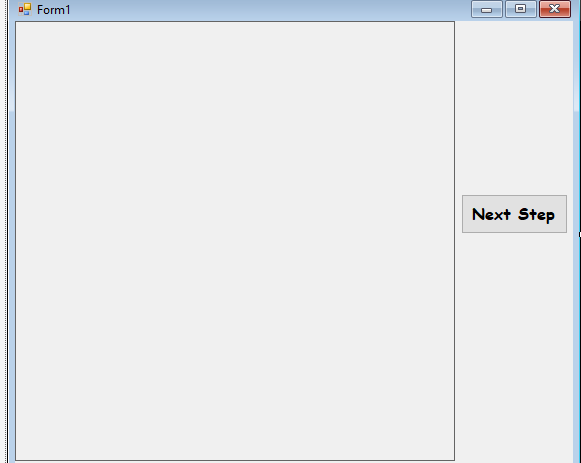
Urmează algoritmul de determinare a următoarei generații. Vom face tot acest lucru într-o buclă do while infinită, la finalul căreia vom pune Console.ReadKey() pentru a vedea cu ușurință rezultatul fiecărei generații. Pentru algoritm, avem nevoie să iterăm cu i și j de la 1 până la n inclusiv și să facem suma vecinilor elementului curent. După aceea, verificăm dacă suma este pară și atribuim valoarea 0 în matricea temporară, sau 1 în caz contrar. La final, parcurgem întreaga matrice și primește valoarea matricei temporare de pe poziția aceea, iar matricea temporară devine 0. Apelăm metoda de afișare.

|  |
| --- |
| do  {  for (int i = 1; i <= n; i++)  for (int j = 1; j <= n; j++)  {  int s = matrix[i - 1, j];  s += matrix[i, j - 1];  s += matrix[i + 1, j];  s += matrix[i, j + 1];  if (s % 2 == 0)  temp[i, j] = 0;  else  temp[i, j] = 1;  }  for (int i = 0; i < n + 2; i++)  for (int j = 0; j < n + 2; j++)  {  matrix[i, j] = temp[i, j];  temp[i, j] = 0;  }  Afisare();  Console.ReadKey();  } while (true); |

Pentru un efect visual, putem crea o aplicație Windows Form. La un moment dat, vom ajunge la următoarele matrice, și multe altele:

Vom avea un formular cu un pictureBox pătrat și un buton pentru a trece la pasul următor.



Vom crea o nouă clasă statică Engine în care se vor crea metodele majore. Declarăm cele două matrice și variabila n, dar de asemenea și un pictureBox display, o matrice de pictureBox pentru afișarea automatului celular, și o variabilă întreagă length ce determină lungimea elementelor din matrice.

În metoda Init, facem toate acțiunile ce trebuie făcute inițial: Inițializăm n, length, display, cele trei matrice și apelăm metoda de afișare. Pentru inițializarea matricei de pictureBox, trebuie să atribuim câmpului Parent variabila display, câmpului Size un nou Size de dimensiunea length, length, iar câmpului Location un nou punct pe poziția i\*length, j\*length. Pentru afișare, parcurgem matricea, iar dacă valoarea acesteia este 0, fundalul pictureBox-ului corespunzător devine alb, și o culoare la alegere în caz contrar.

|  |
| --- |
| public static class Engine  {  public static PictureBox display;  public static PictureBox[,] pbMatrix;  public static int[,] matrix, temp;  public static int n, length;  public static void Init(PictureBox p)  {  n = 51;  length = p.Width / n;  display = p;  pbMatrix = new PictureBox[n, n];  matrix = new int[n + 2, n + 2];  temp = new int[n + 2, n + 2];  for (int i = 0; i < n; i++)  for (int j = 0; j < n; j++)  {  pbMatrix[i, j] = new PictureBox();  pbMatrix[i, j].Parent = display;  pbMatrix[i, j].Size = new Size(length, length);  pbMatrix[i, j].Location = new Point(i \* length, j \* length);  }  matrix[n / 2 + 1, n / 2 + 1] = 1;  matrix[n / 2, n / 2] = 1;  matrix[n / 2 + 2, n / 2] = 1;  matrix[n / 2, n / 2 + 2] = 1;  matrix[n / 2 + 2, n / 2 + 2] = 1;  Afisare();  }  public static void Afisare()  {  for (int i = 1; i <= n; i++)  for (int j = 1; j <= n; j++)  {  if (matrix[i, j] == 0)  pbMatrix[i - 1, j - 1].BackColor = Color.White;  else  pbMatrix[i - 1, j - 1].BackColor = Color.RoyalBlue;  }  } |

Pentru a trece la o altă generație, creăm o metodă în care scriem ceea ce am scris în bucla do while dinainte.

|  |
| --- |
| public static void Step()  {  for (int i = 1; i <= n; i++)  for (int j = 1; j <= n; j++)  {  int s = matrix[i - 1, j];  s += matrix[i, j - 1];  s += matrix[i + 1, j];  s += matrix[i, j + 1];  if (s % 2 == 0)  temp[i, j] = 0;  else  temp[i, j] = 1;  }  for (int i = 0; i < n + 2; i++)  for (int j = 0; j < n + 2; j++)  {  matrix[i, j] = temp[i, j];  temp[i, j] = 0;  }  Afisare();  } |

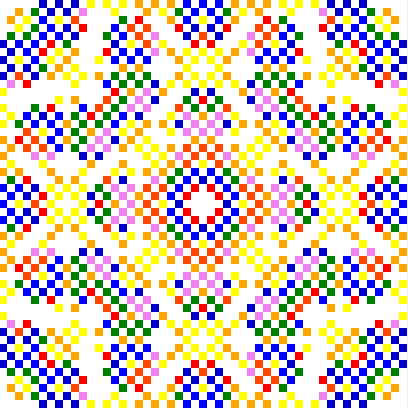
Tot ce rămâne de făcut este să apelăm aceste metode în clasa Form1.

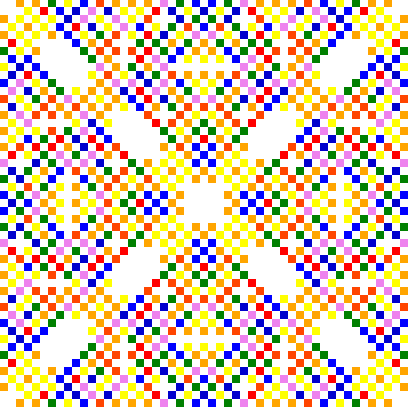
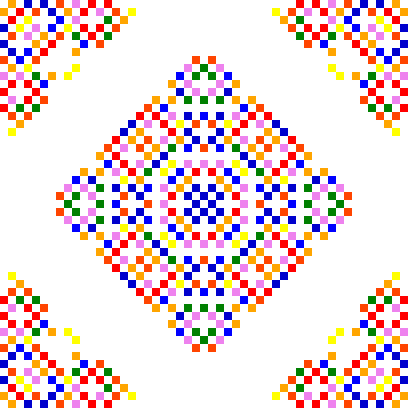
|  |
| --- |
| public partial class Form1 : Form  {  public Form1()  {  InitializeComponent();  Engine.Init(pictureBox1);  }  private void NextStep\_Click(object sender, EventArgs e)  {  Engine.Step();  }  } |

Există o multitudine de feluri în care putem realiza acest program. Tot ce trebuie să facem este să schimbăm legea matematică după care funcționează programul. Dacă dorim sa avem mai multe stări decât acestea două, putem salva acest număr într o variabilă t, valoarea lui temp[i,j] va fi restul împărțirii sumei la variabila t, iar la afișare vom face un switch după valoarea din matrice. În cazul 0, culoarea este albă, iar în fiecare dintre celelalte cazuri se ia o culoare al alegere.

|  |
| --- |
| public static void Step()  {  for (int i = 1; i <= n; i++)  for (int j = 1; j <= n; j++)  {  int s = matrix[i - 1, j];  s += matrix[i, j - 1];  s += matrix[i + 1, j];  s += matrix[i, j + 1];  temp[i, j] = s % t;  }  ...  }  public static void Afisare()  {  for (int i = 1; i <= n; i++)  for (int j = 1; j <= n; j++)  switch(matrix[i, j])  {  case 1: pbMatrix[i-1, j-1].BackColor = Color.Red; break;  case 2: pbMatrix[i-1, j-1].BackColor = Color.Blue; break;  case 3: pbMatrix[i-1, j-1].BackColor = Color.Orange; break;  case 4: pbMatrix[i-1, j-1].BackColor = Color.Green; break;  case 5: pbMatrix[i-1, j-1].BackColor = Color.MediumBlue; break;  case 6: pbMatrix[i-1, j-1].BackColor = Color.Yellow; break;  case 7: pbMatrix[i-1, j-1].BackColor = Color.Violet; break;  case 8: pbMatrix[i-1, j-1].BackColor = Color.OrangeRed; break;  case 9: pbMatrix[i-1, j-1].BackColor = Color.YellowGreen; break;  default: pbMatrix[i-1, j-1].BackColor = Color.White; break;  }  } |

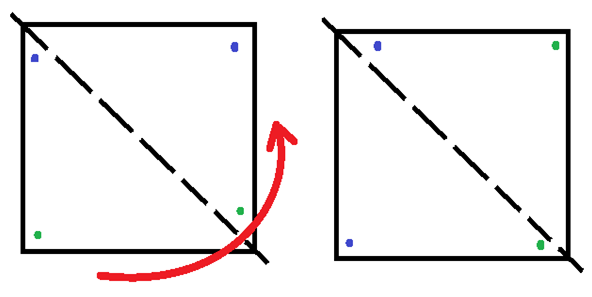
Rezultatele pot fi infinite:

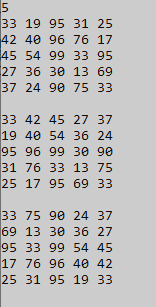
## [AlgSD2] Rotirea unei matrici pe diagonala principală și diagonala secundară (Remus)

Prin rotirea matricei pe o diagonală, ne imaginăm că acea diagonală devine fixă, și matricea se poate roti în jurul ei. Cu acest program, dorim să rotim matricea 180 de grade, adică după rotirea pe diagonala principală, elementul din colțul din stânga jos va fi elementul din dreapta sus, ș.a.m.d.



Pentru a realiza acest lucru, trebuie să parcurgem elementele de deasupra diagonalei și să le interschimbăm cu elementele corespunzătoare. Pentru aceasta, parcurgem cu i de la 0 la n și cu j de la i+1 la n, și interschimbăm elementul de pe poziția i și j cu cel de pe poziția j și i. Pentru diagonala secundară, parcurgem cu j de la 0 la n-i-1, iar elementul cu care trebuie să interschimbăm este cel de pe poziția n-j-1 și n-i-1. La final, fiecare matrice afișată va fi precedenta ei interschimbată.

|  |
| --- |
| static void Main(string[] args)  {  int n = int.Parse(Console.ReadLine());  int[,] matrix = Init(n, n);  Afisare(matrix);  for (int i = 0; i<n; i++)  for(int j=i+1; j<n; j++)  {  int aux = matrix[i, j];  matrix[i, j] = matrix[j, i];  matrix[j, i] = aux;  }  Afisare(matrix);  for(int i=0; i<n; i++)  for(int j=0; j<n-i-1; j++)  {  int aux = matrix[i, j];  matrix[i, j] = matrix[n-j-1, n-i-1];  matrix[n-j-1, n-i-1] = aux;  }  Afisare(matrix);  Console.ReadKey();  } |



## [AlgSD2] Matricea primelor n\*m numere prime (Ana)

Matricea primelor n\*m elemente este o matrice dreptunghiulară, dacă valoarea lui n diferă de cea a lui m (n≠m), unde n reprezintă numărul de linii și m numărul de coloane, care conține pe fiecare poziție câte un număr prim. Aceste numere sunt aranjate în ordine crescătoare, de la stânga la dreapta, iar la o privire mai atentă, putem să citim din matrice al p-ulea număr prim cu condiția ca acel p să fie mai mic sau egal cu n\*m (p<=n\*m).

În prima parte a codului inițializăm matricea, citind n și m de la tastatură. Avem în plus o variabilă k inițializată cu valoarea celui mai mic număr prim existent și anume k=2.

|  |
| --- |
| Console.Write("n = ");  int rand = int.Parse(Console.ReadLine());  Console.Write("m = ");  int coloana = int.Parse(Console.ReadLine());  int[,] matrix = new int[rand, coloana];  int k = 2; |

Parcurgem matricea element cu element, și testăm dacă elementul k este prim. Dacă EstePrim(k) returnează true, adăugăm elementul pe poziția curentă. Cât timp k nu este prim, valoarea lui k crește, elementul de pe poziția curentă se modifică până când se găsește următorul număr prim, iar apoi se trece la următoarea poziție. Verificarea ca numărul k să fie prim este făcută cu funcția EstePrim(k), explicată într-un capitol anterior.

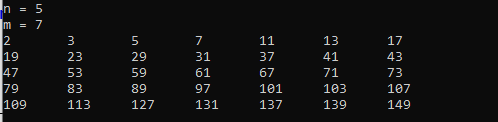
|  |
| --- |
| for (int i = 0; i < n; i++)  for (int j = 0; j < m; j++)  {  while (!EstePrim(k))  k++;  matrix[i, j] = k;  k++;  } |

Iar la sfârșit, se afișează matricea:

|  |
| --- |
| for (int i = 0; i < n; i++)  {  for (int j = 0; j < m; j++)  Console.Write(matrix[i, j] + "\t");  Console.WriteLine();  } |

Codul complet este următorul:

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("n = ");  int n = int.Parse(Console.ReadLine());  Console.Write("m = ");  int m = int.Parse(Console.ReadLine());  int[,] matrix = new int[n, m];  int k = 2;  for (int i = 0; i < n; i++)  for (int j = 0; j < m; j++)  {  while (!EstePrim(k))  k++;  matrix[i, j] = k;  k++;  }  for (int i = 0; i < n; i++)  {  for (int j = 0; j < m; j++)  Console.Write(matrix[i, j] + "\t");  Console.WriteLine();  }  Console.ReadKey();  }  static bool EstePrim(int k)  {  if (k == 0 || k == 1) return false;  if (k == 2) return true;  if (k % 2 == 0) return false;  for (int i = 3; i \* i <= k; i += 2)  if (k % i == 0)  return false;  return true;  } |



## [AlgSD2] Produsul a două matrici (Horatiu)

## [STR] Gestionarea numerelor complexe (Marius)

Se dau 3 numere complexe în reprezentare algebrică, se cere calcularea și afișarea în reprezentare algebrică a sumei, scăderii, înmulțirii, ridicării la putere, a modulului și a conjugatului și să se verifice egalitatea dintre două numere complexe.

Vom realiza acest program cu ajutorul “programării orientate pe obiecte”, astfel ca programul creează o clasă "NumarComplex" care va implementa două proprietăți de tip double (im, re) pentru stocarea valorii imaginare si a valorii reale, precum și câteva metode care vor implementa rezolvarea separată a cerințelor problemei. Pentru a reprezenta vizual un număr complex, vom crea o metodă pentru afișarea numărului complex într-un „string”.

Pentru reprezentarea adunării, scăderii, înmulțirii și a egalității, vom folosi suprascrierea operatorilor.

Astfel că vom începe cu metoda destinata adunării, metodă care va suprascrie operatorul aritmetic “+” care va primi ca argument două variabile de tip “NumarComplex” și care va returna un număr complex format astfel: partea reală a numărului complex returnat va fi egală cu suma părților reale a celor două variabile primite ca argument iar partea imaginară va fi egală cu suma părților imaginare a acestora.

Procedeul este similar pentru scădere.

În cazul înmulțirii se va suprascrie operatorul aritmetic “\*”, metoda va primii ca argument 2 parametri de tip “NumarComplex” si va returna o variabilă de același tip, dar modul in care se va calcula partea reală și cea imaginară va fi diferită, adică partea imaginară va fi rezultatul scăderii produsului părții imaginare a celor două variabile din produsul părții reale iar partea imaginară va fi rezultatul adunării aceleiași operații, cu alte cuvine se va folosi formula:

***z = (a,b)= a + bi*  și  *w = (c,d) = c + di*  este  *zw = (ac-bd,bc+ad) = (ac-bd) + i(bc+ad)***

Metoda egalității va suprascrie operatorul de egalitate “==”, va primi ca argument 2 variabile de tip “NumarComplex” și va returna o variabilă de tip bool (true în cazul în care 2 numere complexe sunt egale, adică partea reală si partea imaginară a variabilelor argument sunt egale, false în caz contrar).

Metoda inegalității ar putea fi rezolvată într-un mod similar dar in acest exemplu vom opta pentru o rezolvare mai simplă, adică variabila bool returnată de metoda va fi egală cu negația apelului metodei create precedent.

Ridicarea la putere s-a realizat prin formula lui Moivre: Math formula

Metoda ridicării la putere primește ca argument o variabilă de tip int “puterea”, vom declara o variabilă double “r” care va fi egală cu radical din suma pătratelor valorii reale și imaginare și o variabila double “teta” care va fi egală cu artg de im/ re. Am ales aici să returnăm o variabilă de tip string egală cu “r” la puterea “putere” înmulțit cu cosinus din “putere” înmulțit cu “teta” la care vom adăuga un string “+i\*” și la care vom adăuga sinus din “putere” înmulțit cu “teta”.

O altă metodă implementată în acest program este cea a conjugatului, care returnează o variabilă de tip „NumarComplex” care va transforma partea imaginară în opusul stării inițiale.

Ultima metodă implementată este cea a modulului care returnează o variabilă de tip double egală cu radical din suma pătratelor părții reale și părții imaginare.

În metoda „Main”, sunt declarate cele 3 variabile de tip număr complex si apelate metodele create anterior.

**Clasa „NumarComplex”:**

|  |
| --- |
| public class NumarComplex  {  private double im;  private double re;  public double Im { get => im; set => im = value; }  public double Re { get => re; set => re = value; }  public NumarComplex(double re=0,double im = 0)  {  this.Re=re;  this.Im=im;  }  public void view()  {  Console.WriteLine();  Console.Write(Re+"+"+Im+"\*i");  }  public static NumarComplex operator+(NumarComplex t1,NumarComplex t2)  {  NumarComplex ad = new NumarComplex(t1.Re + t2.Re, t1.Im + t2.Im);  return ad;  }  public static NumarComplex operator -(NumarComplex t1, NumarComplex t2)  {  NumarComplex sc = new NumarComplex(t1.Re - t2.Re, t1.Im - t2.Im);  return sc;  }  public static NumarComplex operator \*(NumarComplex t1, NumarComplex t2)  {  NumarComplex prod = new NumarComplex(t1.Re \* t2.Re- t1.Im \* t2.Im, t1.Re \* t2.Im + t1.Im \* t2.Re);  return prod;  }  public static bool operator==(NumarComplex t1,NumarComplex t2)  {  if (t1.Re == t2.Re && t1.Im == t2.Im)  return true;  return false;  }  public static bool operator !=(NumarComplex t1, NumarComplex t2)  {  return !(t1 == t2);  }  public string ridicareLaPutere(int putere)  {  Console.WriteLine();  double r = Math.Sqrt(Re \* Re + Im \* Im);  double teta = Math.Atan(Im /Re);  return "\nRidicarea la puterea "+putere+": "+Math.Pow(r,putere)\*(Math.Cos(putere\*teta))+"+i\*"+(Math.Sin(putere\*teta));  }  public NumarComplex conjugat()  {  return new NumarComplex(Re, -Im);  }  public double modul()  {  return Math.Sqrt(Re \* Re + Im \* Im);  }  } |

**Metoda „Main”:**

|  |
| --- |
| static void Main(string[] args)  {  NumarComplex c1 = new NumarComplex();  NumarComplex c2 = new NumarComplex(5);  NumarComplex c3 = new NumarComplex(2,3);  c1.view();  c2.view();  c3.view();  Console.Write("\n\nAdunare: ");  NumarComplex adunare = c1 + c2;  adunare.view();  Console.Write("\n\nScadere: ");  NumarComplex scadere = c3 - c2;  scadere.view();  Console.Write("\n\nInmultire: ");  NumarComplex inmultire = c3 \* c2;  inmultire.view();  Console.WriteLine("\n"+c3.ridicareLaPutere(2));  Console.WriteLine("\nModulul lui c3: "+c3.modul());  Console.Write("\nConjugatul lui c3: ");  c3.conjugat().view();  if(c1==c2)  Console.WriteLine("c1==c2");  else  Console.WriteLine("c1!=c2");  Console.ReadKey();  } |

## [STR] Date. Vârsta ȋn secunde (Zsolti)

**Enunț**: Să se calculeze vârsta dumneavoastră până în data curentă în secunde.

**Soluție:**

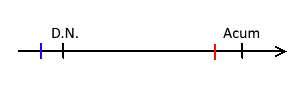
Vom rezolva această problemă pas cu pas, iar apoi vom aduna rezultatele.

Așa că prima oară calculăm diferența de ani, după aceea de zile, de minute și de secunde.

Pentru a ne ușura treaba, vom crea un tip de date pentru date calendaristice.

|  |
| --- |
| struct Date  {  public int an, luna, zi, ore, minute, secunde;  public Date(int \_an, int \_luna, int \_zi, int \_ore, int \_minute, int \_secunde)  {  this.an = \_an;  this.luna = \_luna;  this.zi = \_zi;  this.ore = \_ore;  this.minute = \_minute;  this.secunde = \_secunde;  }  public Date(int \_an, int \_luna, int \_zi): this(\_an, \_luna, \_zi, 0, 0, 0) { }  public override string ToString()  {  return this.zi + ". " + this.luna + " " + this.an;  }  } |

Putem vedea că nu este o structura complexă, este doar cu scopul de a manipula datele mai ușor.

Planul este următorul:

Acum - data curentă

D.N.   - data nașterii

Linia roșie - 01 Ianuarie a anului acesta

Linia Albastra - 01 Ianuarie a anului nașterii

Din nou, sarcina noastră este să calculăm Acum - D.N. în secunde și vom face asta prin a calcula: (roșu - albastru) + (D.N. - albastru) + (Acum - roșu)

Pentru simplicifare vom folosi următoarele notații:

**a** = roșu - albastru

**b** = D.N. - albastru

**c** = Acum - roșu

Acestă abordare ne ușurează treaba semnificativ, deaorece **a**înseamnă doar diferența de ani dintre cele două date, iar **b** și **c** înseamnă doar diferența de zile.

Acum D.N. este egală cu **a** - **b** + **c**, pentru că din poze se poate vedea că **c**-ul nu este **a**, dar **b**-ul este, dar este în plus.

|  |
| --- |
| static void Main(string[] args)  {  Date inceput = new Date(1997, 10, 15, 07, 0, 0);  Date sfarsit = new Date(2019, 2, 1, 15, 50, 10);  int diferentaInZile = Diferenta(inceput, sfarsit);  } |

Vom folosi două date apropiate pentru a putea testa rezultatul.

|  |
| --- |
| private static int Diferenta(Date inceput, Date sfarsit)  {  int a = DiferentaDeAniInZile(inceput.an, sfarsit.an);  int b = NumarDeZileDinAcelasAn(new Date(inceput.an, 1, 1), inceput);  int c = NumarDeZileDinAcelasAn(new Date(sfarsit.an, 1, 1), sfarsit);  return a - b + c;  } |

Deja de aici se poate observă beneficiul abordării noastre, căci în loc de operații pe date calendaristice am trecut pe numere întregi.

|  |
| --- |
| private static int DiferentaDeAniInZile(int inceput, int sfarsit)  {  int totalAni = sfarsit - inceput;  int numarAniBisecti = 0;  for (int i = inceput; i < sfarsit; i++)  if (EAnBisect(i))  numarAniBisecti++;  // fie a = numarul de ani bisecti si b = numarul de ani care nu sunt bisecti  // din asta rezulta ca diferenta de ani in zile este a\*364 + b\*365  // dar b se poate scrie ca numarul total de ani - numarul de ani bisecti  // deci rezulta a\*364 + (total-a)\*365 = a\*364 + total\*365 - a\*365, vom extrage a din ecuatie  // ne va ramane a\*(364 - 365) + total\*365 <=> total\*365 - a  return totalAni \* 365 - numarAniBisecti;  } |

Exercițiu suplimentar: Funcția de mai sus este rezolvat într-un mod educativ și în mod intenționat nu s-a ținut cont de perfomanță. Încearcă să-l optimizezi tu.

După calcularea valorilor a,b și c, vom folosi o funcție care convertește zilele, orele și minutele în secunde, pentru că la sfârșit să putem afișa vârstă în secunde.

**Codul integral:**

|  |
| --- |
| struct Date  {  public int an, luna, zi, ore, minute, secunde;  public Date(int \_an, int \_luna, int \_zi, int \_ore, int \_minute, int \_secunde)  {  this.an = \_an;  this.luna = \_luna;  this.zi = \_zi;  this.ore = \_ore;  this.minute = \_minute;  this.secunde = \_secunde;  }  public Date(int \_an, int \_luna, int \_zi): this(\_an, \_luna, \_zi, 0, 0, 0) { }  public override string ToString()  {  return this.zi + ". " + this.luna + " " + this.an;  }  }  static void Main(string[] args)  {  Date inceput = new Date(1997, 10, 15, 07, 0, 0);  Date sfarsit = new Date(2019, 2, 1, 15, 50, 10);  int diferentaInZile = Diferenta(inceput, sfarsit);  int diferentaInSecunde = ConvertesteInSecunde(diferentaInZile, "zile");  //luam de la inceput  diferentaInSecunde -= ConvertesteInSecunde(inceput.ore, "ore");  diferentaInSecunde -= ConvertesteInSecunde(inceput.minute, "minute");  diferentaInSecunde -= inceput.secunde;  //adaugam la sfarsit  diferentaInSecunde += ConvertesteInSecunde(sfarsit.ore, "ore");  diferentaInSecunde += ConvertesteInSecunde(sfarsit.minute, "minute");  diferentaInSecunde += sfarsit.secunde;  Console.WriteLine(diferentaInSecunde);  Console.ReadKey();  }  private static int ConvertesteInSecunde(int numar, string tip)  {  switch (tip)  {  case "zile":  return numar \* 86400;  case "ore":  return numar \* 3600;  case "minute":  return numar \* 60;  default:  return 0;  }  }  private static int Diferenta(Date inceput, Date sfarsit)  {  int a = DiferentaDeAniInZile(inceput.an, sfarsit.an);  int b = NumarDeZileDinAcelasAn(new Date(inceput.an, 1, 1), inceput);  int c = NumarDeZileDinAcelasAn(new Date(sfarsit.an, 1, 1), sfarsit);  return a - b + c;  }  private static int DiferentaDeAniInZile(int inceput, int sfarsit)  {  int totalAni = sfarsit - inceput;  int numarAniBisecti = 0;  for (int i = inceput; i < sfarsit; i++)  if (EAnBisect(i))  numarAniBisecti++;  // fie a = numarul de ani bisecti si b = numarul de ani care nu sunt bisecti  // din asta rezulta ca diferenta de ani in zile este a\*364 + b\*365  // dar b se poate scrie ca numarul total de ani - numarul de ani bisecti  // deci rezulta a\*364 + (total-a)\*365 = a\*364 + total\*365 - a\*365, vom extrage a din ecuatie  // ne va ramane a\*(364 - 365) + total\*365 <=> total\*365 - a  return totalAni \* 365 - numarAniBisecti;  }  private static bool EAnBisect(int an)  {  return an % 4 == 0;  }  private static int NumarDeZileDinAcelasAn(Date inceput, Date sfarsit)  {  //daca este aceasi luna, atunci ne intereseaza doar diferenta dintre zile  if (inceput.luna == sfarsit.luna)  return sfarsit.zi - inceput.zi;  int suma = 0;  for (int i = inceput.luna; i < sfarsit.luna; i++)  suma += NumarDeZileIntroLunaDupaAn(inceput.an, i); //inceput si sfarsit au acelas an, deci nu conteaza pe care-l pun  //forul de mai sus ruleaza exclusiv pana la luna de sfarsit si ne mai raman cateva zile pe care nu le-am luat inca in calcul  suma += sfarsit.zi;  return suma;  }  private static int NumarDeZileIntroLunaDupaAn(int an, int luna)  {  int[] zileleLuni = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  luna--; //pentru a ca vectorul de mai sus incepe de la 0, iar noi avem prima luna (Ianuarie) fiind 1  if (luna == 2) //daca cere luna Februarie  if (EAnBisect(an)) // este an bisect  return zileleLuni[luna] + 1; // va fii 28+1  return zileleLuni[luna];  } |

## [STR] Date. Distanța dintre două date. (Remus)

În acest program, vom crea un tip de dată definit de utilizator, adică o clasă, pentru simularea unei date. Pentru acest lucru, avem nevoie de 3 variabile întregi pentru a memora anul, luna și ziua. Putem crea doi constructori, unul în care dăm toate valorile întregi, iar altul în care primim data în format string. Pentru afișare, suprascriem metoda ToString().

|  |
| --- |
| class Date  {  public int an, luna, zi;  public Date(int z, int l, int a)  {  zi = z; luna = l; an = a;  }  public Date(string date)  {  zi = int.Parse(date.Split('.')[0]);  luna = int.Parse(date.Split('.')[1]);  an = int.Parse(date.Split('.')[2]);  }  public override string ToString()  {  StringBuilder s = new StringBuilder();  s.AppendFormat("{0}.{1}.{2}", zi, luna, an);  return s.ToString();  } |

Pentru a determina distanța dintre două date în zile, am creat o funcție ce calculează numărul de zile al unei date începând cu anul 0. Astfel, distanța dintre cele două date va fi modul din diferența numărului de zile al celor două date. Numărul de zile se calculează astfel: parcurgem toți anii de la anul 0 până înaintea anului curent, și verificăm dacă acesta este an bisect, adunând 366 de zile, sau 365 în caz contrar. Apoi parcurgem toate lunile mai puțin cea curentă, și verificăm câte zile trebuiesc adăugate pentru luna curentă: dacă luna este 2 și este an bisect, adăugăm 29, iar 28 daca anul nu este bisect, daca luna este 4, 6, 9 sau 11, adăugăm 30, și adăugăm 31 altfel. La final, adăugăm numărul curent de zile.

|  |
| --- |
| public int NrZile()  {  int nrZile = 0;  for (int i = 0; i < an; i++)  {  if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0)  nrZile += 366;  else nrZile += 365;  }  for (int i = 1; i < luna; i++)  {  if (i == 2)  {  if ((an % 4 == 0 && an % 100 != 0) || an % 400 == 0)  nrZile += 29;  else nrZile += 28;  }  else if (i == 4 || i == 6 || i == 9 || i == 11)  nrZile += 30;  else  nrZile += 31;  }  nrZile += zi;  return nrZile;  }  } |

## [STR] Elemente de geometrie cu coordonate

## [GRDY] Plata unei sume

## [GRDY] Spectacole (Zsolti)

**Enunț**: Se organizează mai multe spectacole în aceeași zi, fiecare spectacol având o oră de început și una de sfârșit. Să se scrie un program care, prin metoda Greedy, să ordoneze spectacolele astfel înca să se poată organiza un număr maxim de spectacole fără suprapuneri.

**Soluție**:

1. Acest pas 0 îl vom face doar pentru noi și este total optional, dar ne va ușura treaba. Creăm un tip de date pentru a putea ține codul curat și organizat.

|  |
| --- |
| struct Spectacol  {  public string nume;  public int inceput;  public int sfarsit;  public Spectacol(string \_nume, int \_inceput, int \_sfarsit)  {  this.nume = \_nume;  this.inceput = \_inceput;  this.sfarsit = \_sfarsit;  }  } |

Odată creat, vom putea să ne declarăm lista de spectacole într-un mod mai elegant.

|  |
| --- |
| static void Main(string[] args)  {  Spectacol[] spectacole = {  new Spectacol("A1",8, 10),  new Spectacol("A2",10, 11),  new Spectacol("A3",9, 13),  new Spectacol("A4",12, 13),  new Spectacol("A5",14, 16),  new Spectacol("A6",17, 19)  };  } |

1. Ordonăm crescător tabloul nostru cu spectacole, în funcție de ora de sfârșit.

Pentru că, în momentul de față, algoritmul de sortare este irelavant, vom presupune că există undeva o metodă Ordoneaza, care ne va rezolva acest pas.

|  |
| --- |
| Ordoneaza(spectacole); |

1. Suntem gata să obținem răspunsul la această problemă. Fiindcă nu știm exact numărul de spectacole pe care le putem organiza, vom folosi un tablou dinamic, adică o Listă.

|  |
| --- |
| List<Spectacol> raspuns = new List<Spectacol>(); |

1. Pentru că vectorul nostru cu spectacole este pregătit deja (fiindcă la pasul 1 a fost ordonat), putem ști sigur că primul element este unul bun.

|  |
| --- |
| raspuns.Add(spectacole[0]); |

1. Mergând mai departe, trebuie să găsim următorul spectacol care se poate organiza.

Este organizabil dacă începe exact când sau după ce s-a terminat ultimul spectacol. Pentru a putea ține evidența mai ușor, vom introduce o variabilă care să rețină ultimul spectacol organizat. Astfel, codul de la pasul 3 se va schimba în:

|  |
| --- |
| Spectacol ultimul = spectacole[0];  raspuns.Add(ultimul); |

1. Din acest moment, avem tot ce ne trebuie să construim răspunsul complet.

|  |
| --- |
| for(int i = 1; i < spectacole.Length; i++)  if(ultimul.sfarsit <= spectacole[i].inceput)  {  ultimul = spectacole[i];  raspuns.Add(ultimul);  } |

1. În acest pas, deja am obținut lista de spectacole care se pot organiza. Pentru verificare, vom afișa pe ecran.

|  |
| --- |
| foreach(Spectacol spectacol in raspuns)  Console.Write(spectacol.nume + " "); |

**Codul integral:**

|  |
| --- |
| static void Main(string[] args)  {  …  Ordoneaza(spectacole);  List<Spectacol> raspuns = new List<Spectacol>();  Spectacol ultimul = spectacole[0];  raspuns.Add(ultimul);  for(int i = 1; i < spectacole.Length; i++)  {  if( ultimul.sfarsit <= spectacole[i].inceput )  {  ultimul = spectacole[i];  raspuns.Add(ultimul);  }  }  foreach(Spectacol spectacol in raspuns)  Console.Write(spectacol.nume+" ");  } |

## [GRDY] Colorarea hărților

## [GRDY] Codificarea Huffman

## 

## [GRDY] Rucsacul (Marius)

Se dă o mulțime formată din n obiecte, fiecare fiind caracterizat de o greutate și o valoare. Să se gasească o submulțime de obiecte astfel încât valoarea să fie maximă, iar suma greutăților lor să nu depășească o valoare c(capacitate).

Rezolvarea acestei probleme prin metoda greedy constă în crearea unei liste a cărei elemente sunt formate din 2 numere(un număr care reprezintă valoarea și un număr care reprezintă greutatea). Se crează o listă rucsac care va conține submulțimea dorită și se declară o variabilă care va reține capacitatea rucsacului.

În continuare cât timp mai există spațiu în rucsac se ordonează elementele rămase în listă de obiecte după valoare, se verifică dacă primul obiect din listă de obiecte se poate adauga în rucsac, în caz afirmativ se adaugă obiectul în rucsac, se elimină din lista de obiecte și se scade din capacitatea rămasă a rucsacului. La sfarșit se afisază elementele din rucsac.

Implementare:

Pentru realizarea acestui program s-au creat două liste generice în care se poate introduce doar vectori de tip int, una pentru lista de obiecte și una pentru rucsac. Vectorii introduși în lista de obiecte au o dimensiune de 2 elemente(valoare si greutate). Se declară o variabilă "c" care va reține capacitatea actuală a rucsacului.

Cât timp capacitatea este mai mare decât 0 se sortează elementele listei de obiecte după primul element din vector(elementul valoare) cu ajutorul unui delegat, se verfică dacă primul element poate fi introdus în rucsac, în caz afirmativ se adaugă acel obiect în lista corespunzatoare rucsacului, se șterge elementul din listă de obiecte și se scade din variabila capacitate greutatea obiectului. Se repeta acest procedeu cât timp capacitatea rucsacului este mai mare ca 0 sau nu mai există nici un obiect care poate fi introdus în rucsac. La sfarșit se afisază elementele din lista rucsac.

|  |
| --- |
| static List<int[]> obiecte = new List<int[]>()  {  new int[] { 1,2 },  new int[] { 2,2 },  new int[] { 3,5 },  new int[] { 10,10 }  };  static List<int[]> rucsac = new List<int[]>();  static void Main(string[] args)  {  int c = 16;  while (c > 0)  {  obiecte.Sort(delegate (int[] a, int[] b) { return b[0]-a[0]; });  if (c < obiecte[0][1])  break;  rucsac.Add(obiecte[0]);  c -= obiecte[0][1];  obiecte.RemoveAt(0);  }  foreach (int[] item in rucsac)  Console.WriteLine("Valoare: {0}, Greutate: {1}",item[0],item[1]);  Console.ReadKey();  } |
| Elementele introduse in rucsac vor fi:  Valoare: 10, Greutate: 10  Valoare: 3, Greutate: 5 |

## [GRDY] Comis voiajorul (Marius)

Se dă o matrice pătratică "distance" de dimensiune "n" care reprezintă distanțe dintre anumite locații. Se cere **cel mai scurt traseu posibil care trece prin toate locațiile o singură dată și se intoarce la loacația inițială** și calcularea distanței(costului) parcurse.

Vom declara o matrice pătratică "distance" care conține disțanta dintre locații, un vector "visited" de tip boolean care îl vom folosi ca să verificăm dacă locația a fost vizitată, un vector "traseuFinal" de tip int care va reține locațiile vizitate in ordine(traseul final), o variabilă "distantaMinimă" care va reține distantă minimă dintre 2 noduri, o variabilă care va reține nodul curent temporar si o variabilă care va reține nodul curent final.

Vom incepe prin declararea si inițialzarea variabilei "toatalDistance" cu 0, variabilă care va reține distanța totală necesară călătoriei si declararea si initializare variabilei "varfStart" care va conține locația de unde va incepe călătoria.

Primul element din vectorul "traseuFinal" va fi setat cu valoarea nodului de start, nodul de start este setat ca fiind vizitat in vectorul "visited" și se inițializează variabila "nodCurentTemporar" cu valoarea acestuia.

în continuare vom parcurge toate locațiile, vom inițializa variabila "distanțaMinimă" cu o valoare mare, într-o altă iterație vom parcurge din nou toate locațiile, scopul fiind de a găsi cel mai apropiat nod de cel curent procedând astfel: verificăm daca locația curentă(a doua parcurgere) nu a fost vizitată si disțanta dintre nodul "nodCurentTemporar" si nodul curent(a doua parcurgere) este mai mică decât dinstanța minimă curentă(distantăMinimă), în caz afirmativ "distantăMinimă" este actualizată si variabila "nodCurentFinal" este setată cu valoarea nodului curent(a doua parcurgere). Dupa încheierea a parcurgerei(a doua parcurgere) am aflat nodul cel mai apropiat, îl vom adăuga in vectorul "traseuFinal", vom crește valoarea variabilei "toatalDistance" cu distanța dintre cele 2 locații, vom seta locația ca fiind vizitată si vom seta variabila "nodCurentTemporar" ca fiind egală cu nodul cel mai apropiat "nodCurentFinal".

La final vom adăuga din nou nodul de start la finalul vectorului "traseuFinal", vom crește valoarea variabilei "toatalDistance" și le vom afișa.

|  |
| --- |
| static int n, nodCurentFinal, nodCurentTemporar, distantaMinima;  static int[] traseuFinal = new int[100];  static bool[] visited = new bool[100];  static int[,] distance = new int[,]{  {999,1,2,7,5},  {1,999,4,4,3},  {2,4,999,1,2},  {7,4,1,999,3},  {5,3,2,3,999}};  static void Main(string[] args)  {  int toatalDistance = 0;  n = distance.GetLength(0);  Console.Write("Locatia de start: ");  int varfStart = int.Parse(Console.ReadLine());  traseuFinal[1] = varfStart;  visited[varfStart] = true;  nodCurentTemporar = varfStart;  for (int i = 1; i < n; i++)  {  distantaMinima = 999;  for (int j = 1; j < n + 1; j++)  {  if (!visited[j] && (distance[nodCurentTemporar - 1, j - 1] < distantaMinima))  {  distantaMinima = distance[nodCurentTemporar - 1, j - 1];  nodCurentFinal = j;  }  }  traseuFinal[i + 1] = nodCurentFinal;  toatalDistance = toatalDistance + distantaMinima;  visited[nodCurentFinal] = true;  nodCurentTemporar = nodCurentFinal;  }  traseuFinal[n + 1] = varfStart;  toatalDistance = toatalDistance + distance[nodCurentTemporar - 1, varfStart - 1];  Console.Write("\nTraseul :");  for (int i = 1; i < n + 1; i++)  {  Console.Write(traseuFinal[i] + " ");  }  Console.Write("\nToatalDistance : " + toatalDistance);  Console.ReadKey();  } |
| Locatia de start: 1  Traseul :1 2 5 3 4  ToatalDistance : 14 |

## [REC] Fibonacci

## [REC] Ieșirea din labirint

## [REC] Platouri

## [REC] Mastermind

## [REC] Numărul de obiecte (structuri de valori identice amorfe) dintr-o matrice

## [REC] Algoritmii de sortare fără cicluri repetitive sau goto. (Remus)

Ciclurile repetitive (for, while, etc) sunt atât de des folosite de programatori, încât aproape că se poate spune că nu există alternativă. În acest program, am decis să realizăm câteva sortări folosind numai recursivitate.

În primul rând, ar trebui să ne gândim la alternativa unei bucle for. Aceasta este formată din 3 părți: valoarea de început, condiția de oprire și incrementarea. Acestea se pot realiza și prin recursivitate: valoarea de început este apelarea inițială a metodei cu acea valoare, condiția de oprire se face la fel ca și la orice alt program recursiv, iar incrementarea se face la reapelarea metodei cu valoarea incrementată. Vom folosi aceeași modalitate pentru a realiza metoda de afișare a unui vector:

|  |
| --- |
| static void Afisare(int i)  {  if (i < n)  {  Console.Write(v[i] + " ");  Afisare(i + 1);  }  else  Console.WriteLine();  } |

Pentru orice sortare, vom avea un vector de întregi și lungimea acestuia declarate global. În metoda main, inițializăm vectorul cu valoarea dorită, și apelăm metoda de afișare și sortarea dorită, pe care o vom crea imediat.

|  |
| --- |
| static int n;  static int[] v;  static void Main(string[] args)  {  n = 10;  v = new int[] { 3, 2, 7, 0, 1, 6, 4, 5, 9, 8 };  Afisare(0);  BubbleSort(0);  Console.ReadKey();  } |

**Bubble Sort**

Pentru sortarea bubble, avem nevoie de o variabilă k în care reținem de câte ori am parcurs vectorul până acum și de o variabilă bool pentru a ști dacă vectorul este sortat sau nu. Această variabilă va avea inițial valoarea true, iar la parcurgerea curentă, când se realizează o interschimbare, valoarea se schimbă în false.

Deoarece metoda are un for, vom face verificarea din for la începutul metodei. La fiecare pas, verificăm dacă elementul de pe poziția curentă din vector este mai mic decât cel de pe poziția următoare, caz în care acestea se interschimbă. Pentru a trece pe poziția următoare, apelăm metoda cu valoarea i+1. Dacă nu se îndeplinește condiția de la început, verificăm dacă vectorul nu este sortat, caz în care presupunem din nou că vectorul este sortat și apelăm metoda cu valoarea 0 din nou. Pentru a vedea ce se întâmplă pe parcurs, afișăm vectorul.

|  |
| --- |
| static int k = 0;  static bool isSorted = true;  static void BubbleSort(int i)  {  if (i < n - 1 - k)  {  if (v[i] > v[i + 1])  {  int aux = v[i];  v[i] = v[i + 1];  v[i + 1] = aux;  isSorted = false;  }  BubbleSort(i + 1);  }  else if (!isSorted)  {  Afisare(0);  isSorted = true;  BubbleSort(0);  }  } |



Deși pare că vectorul este sortat imediat, acest lucru se datorează faptului că fiecare număr este aproape de poziția sa. Să vedem ce se întâmplă dacă punem valoarea 0 pe ultima poziție:



**Insertion Sort**

Sortarea insertion conține două for-uri unul în celălalt, ceea ce înseamnă că avem nevoie de două metode recursive. Prima iterează cu j de la 1 la n, deci verificăm dacă j este mai mic decât n și reapelăm cu valoarea j+1. De asemenea, vom afișa vectorul de fiecare dată. Apelăm metoda a doua cu valoarea j, deoarece aceasta iterează cu i de la j la 0. În a doua, verificăm dacă i este mai mare decât 0, caz în care facem verificarea pentru interschimbare între elementele de pe poziția i, respectiv i-1. Reapelăm metoda cu valoarea i-1.

|  |
| --- |
| static void InsertionSort(int j)  {  if (j < n)  {  SortElements(j);  Afisare(0);  InsertionSort(j + 1);  }  }  static void SortElements(int i)  {  if (i > 0)  {  if (v[i] < v[i - 1])  {  int aux = v[i];  v[i] = v[i - 1];  v[i - 1] = aux;  }  SortElements(i - 1);  }  } |



**Selection Sort**

Pentru sortarea selection, avem nevoie de o variabilă position în care reținem valoarea poziției elementului cu valoarea minimă, pentru a ști ce element trebuie interschimbat cu cel de pe poziția curentă. Cum și această metodă foloseste două for-uri, vom folosi și aici două metode recursive. În prima, iterăm cu j de la 0 la n, deci verificăm dacă j este mai mic decât n, iar la final reapelâm metoda cu j+1. Position ia valoarea j, după care apelăm metoda de determinare a poziției celui mai mic element din restul vectorului, adică începând cu j+1. Interschimbăm elementul de pe poziția curentă cu cel de pe position doar dacă position este diferit de j, și afișăm vectorul. În a doua metodă, verificăm dacă i este mai mic decât n, iar la final reapelăm metoda cu i+1. Verificăm dacă elementul de pe poziția i este mai mic decăt vectorul de pe position, caz în care position ia valoarea i.

|  |
| --- |
| static int position;  static void SelectionSort(int j)  {  if (j < n)  {  position = j;  GetPosition(j + 1);  if (position != j)  {  int aux = v[j];  v[j] = v[position];  v[position] = aux;  }  Afisare(0);  SelectionSort(j + 1);  }  }  static void GetPosition(int i)  {  if (i < n)  {  if (v[i] < v[position])  position = i;  GetPosition(i + 1);  }  } |



Poate că pare să fi făcut mulți pași, dar la unele dintre iterații nu s-a făcut nicio interschimbare, în schimb la celelalte două sortări, la fiecare pas se fac mai multe interschimbări, ceea ce face sortarea selection mult mai rapidă decât celelalte sortări.

## [BCKT] Generare permutări (Ana)

## [BCKT] Generare aranjamente (Ana)

## [BCKT] Generare combinări (Alex)

Dându-se două numere naturale **N** şi **P** (1 ≤ **P** ≤ **N**), dorim să generăm toate combinările de **N** luate câte **P** ale mulţimii numerelor naturale. Combinări de N luate câte P se notează , iar

(numărul combinărilor de N luate câte P). Reamintim că, spre deosebire de aranjamente, reprezintă modalităţile de a aranja **N** obiecte în **P** poziţii, ordinea în care se face aranjarea lor neavând importanţă (de exemplu, aranjarea **1 2 3** este acelaşi lucru cu aranjarea **2 1 3**). **N** şi **P** se citesc din fişierul **comb.in**, iar combinările generate se scriu în fişierul **comb.out**, câte una pe o linie.

Ex:

|  |  |
| --- | --- |
| comb.in | comb.out |
| 4 2 | 1 2  1 3  1 4  2 3  2 4  3 4 |

Modul de rezolvare este similar cu cel al problemelor anterioare. Mai mult, datorită faptului că ordinea elementelor nu are importanţă, algoritmul se simplifică mult. Să presupunem că suntem la pasul **K** (1 ≤ **K** ≤ **P**). Asta înseamnă că avem depuse valori în Sol[1], Sol[2], ..., Sol[K – 1]. Considerăm Sol[0] = 0. Deoarece ordinea elementelor nu contează, putem depune numere pe poziţia actuală a stivei începând de la valoarea precedentă, la care se adună unu, adică Sol[K – 1] + 1. Astfel nu mai avem nevoie de vectorul **Fol**, deoarece numerele din stivă vor fi întotdeauna ordonate crescător şi nu vor avea cum să se repete. Mai mult, acest artificiu ne asigură şi că nu vom genera mai multe combinări decât este nevoie.

Algoritmul este cel mai rapid de până acum, complexitatea sa fiind O( ), datorită faptului că nu se va încerca niciodata depunerea unei valori invalide în stivă.

Pentru exemplul dat, algoritmul funcţionează în felul următor: mai întâi se iniţializează Sol[0] cu 0. La pasul K = 1, se depune mai întâi valoarea 1 în stivă:

|  |  |
| --- | --- |
| **1** | **1** |
| **K** | **Sol** |

Se trece la pasul K = 2. Se depune valorea Sol[K – 1] + 1 = 2 în stivă:

|  |  |
| --- | --- |
| **2** | **2** |
| **1** | **1** |
| **K** | **Sol** |



Se trece la pasul **K = 3** şi se afişează conţinuturile stivei, deoarece **K > P**.

Se revine la pasul **K = 2** şi se continuă numărătoarea, depunându-se în stivă valoarea 3.

|  |  |
| --- | --- |
| **2** | **3** |
| **1** | **1** |
| **K** | **Sol** |



La următorul pas, se va afişa din nou stiva. Se continuă în acest mod până ce vor fi afişate toate combinările.

Şi de data aceasta, algoritmul va genera combinările în ordine lexicografică.

|  |
| --- |
| static void Main(string[] args)  {  int n = 5;  int p = 3;  sol = new int[p + 1];  b = new bool[n];  Combinari(1, n, p, sol, b);  }  static public void Combinari(int k, int n, int p, int[] sol, bool[] b)  {  if (k > p)  {  for (int i = 1; i <= p; i++)  Console.Write((sol[i]) + " ");  Console.WriteLine();  }  else  for (int i = sol[k - 1] + 1; i < n; i++)  if (b[i] == false)  {  b[i] = true;  sol[k] = i;  Combinari(k + 1, n, p, sol, b);  b[i] = false;  }  } |

## 

## [BCKT] Generare partiții (Ana)

## [BCKT] Problema damelor (Ana)

## [REC/DC] Apa dintr-un vector (Zsolti)

## [MATH/RECDC] Determinant nxn (Marius)

Se dă o matrice pătratică de dimensiune n. Se cere calcularea determinantului acelei matrici.

Vom crea o funcție care va primi ca argument o matrice și care va returna determinantul. Implementarea algoritmului de calcul al determinantului constă în crearea unei variabile de tip float „det” care va stoca rezultatul calculului, la început fiind inițializată cu 0. Vom verifica gradul matricei iar în cazul în care acesta este egal cu 2 vom aplica formula de calcul a determinantului de grad 2 și vom incrementa variabila „det” cu rezultatul acesteia, în caz contrar(gradul matricei este mai mare ca 2) vom folosi recursivitatea și procedeul matematic de rezolvare al determinantului (dezvoltarea dupa linie).

Pentru a putea implementa dezvoltarea după linie vom avea nevoie de o funcție care să elimine din matrice o linie și o coloana. Această funcție „scurtare” va primi ca argument o matrice și două variabile de tip int care vor stoca linia și coloana care se doresc a fi eliminate „k” și „l” și va returna o matrice nouă, matricea creată în cadrul funcției „aux” de dimensiune mai mică cu 1 decât matricea primită ca argument. Vom parcurge matricea primită ca argument și se copiază fiecare element din aceasta mai puțin linia și coloana care doresc a fi eliminate în noua matrice „aux”.

|  |
| --- |
| static int[,] Scurtare(int[,] matrix,int k, int l)  {  int n = matrix.GetLength(0);  int m = matrix.GetLength(1);  int[,] aux = new int[n - 1, m - 1];  for (int i = 0; i < n - 1; i++)  for (int j = 0; j < m - 1; j++)  {  if (i < k && j < l)  aux[i, j] = matrix[i, j];  if (i < k && j >= l)  aux[i, j] = matrix[i, j + 1];  if (i >= k && j < l)  aux[i, j] = matrix[i + 1, j];  if (i >= k && j >= l)  aux[i, j] = matrix[i + 1, j + 1];  }  return aux;  } |

Cât timp matricea dată ca argument nu a ajuns la dimensiunea de 2, variabila „det” este incrementată pentru fiecare linie cu elementul de pe linia cu indexul 0 și indexul coloanei „i” înmulțit cu -1 sau 1 în funcție de paritatea liniei înmulțit cu rezultatul apelului recursiv al funcției având ca argument linia 0 și coloana de index „i”.

|  |
| --- |
| static float Determinant(int[,] matrix)  {  float det = 0;  if (matrix.GetLength(0) == 2)  det += matrix[0, 0] \* matrix[1, 1] - matrix[0, 1] \* matrix[1, 0];  else  {  for (int i = 0; i < matrix.GetLength(0); i++)  det += (float)Math.Pow(-1, i) \* matrix[0, i] \* Determinant(Scurtare(matrix,0, i));  }  return det;  } |

Vom avea nevoie de o funcție care să afișeze matricea, aceasta va primi ca argument o matrice.

|  |
| --- |
| private static void afisare(int[,] m)  {  for (int i = 0; i < m.GetLength(0); i++)  {  for (int j = 0; j < m.GetLength(1); j++)  Console.Write(m[i,j]+" ");  Console.WriteLine();  }  } |

În metoda main vom declara matricea, vom inițializa elementele sale cu valori aleatorii, o vom afișa și vom afișa determinantul.

|  |
| --- |
| static void Main(string[] args)  {  Random r = new Random();  int i, j;  int n=7;  int[,] m = new int[n,n];  for (i = 0; i < m.GetLength(0); i++)  for (j = 0; j < m.GetLength(1); j++)  m[i, j] = r.Next(10);  afisare(m);  Console.WriteLine();  Console.WriteLine(Determinant(m));  Console.ReadKey();  } |

!!output

## [MATH/RECDC] Determinant nxn (Remus)

## [RECDC] Turnurile din Hanoi (Extindere)

**[RECDC] Babilon**

## [RECDC] Merge Sort

## [RECDC] Quiq Sort

## [RECDC] minMax (Zsolti)

## [RECDC] Binary search (Alex)

Algoritmul de căutare binară este un algoritm de căutare folosit pentru a găsi un element într-o listă ordonată (tablou unidimensional/vector). Algoritmul funcționează pe baza tehnicii Divide et Impera.

Valoarea căutată este comparată cu cea a elementului din mijlocul listei. Dacă este egală cu cea acel element, algoritmul se termină. Daca este mai mare decât acea valoare, algoritmul se reia, de la mijlocul listei până la sfârșit, iar dacă e mai mică, algoritmul se reia pentru elementele de la începutul listei până la mijloc.Întrucât la fiecare pas cardinalul mulțimii de elemente în care se efectuează căutarea se înjumatățește, algoritmul are complexitate logaritmică (log n).

Se consideră un tablou unidimensional (vectorul v) cu n elemente care sunt în ordine crescătoare(sortat crescător) și trei variabile: stanga=început, dreapta=sfârșit, iar mijloc = mijlocul.

Metoda verifică de mai multe ori dacă mijlocul vectorului/tabloului unidimensional este egal cu elementul căutat:

-în cazul în care este egală, variabila mijloc repezinta pozitia elementului în vector

-dacă nu se îndeplinește condiția de egalitate, se trece la verificarea poziției elementului căutat în vector astfel: dacă elementul căutat este mai mic decât elementul din mijlocul vectorului, variabila "dreapta" ia valoarea lui mijloc, iar dacă nu variabila "stanga" ia valoarea lui mijloc.

Algoritmul se repetă cât timp valoarea variabilei stanga este mai mica decât valoarea variabilei dreapta.

|  |
| --- |
| class Program  {  static int Cautare(int stanga, int dreapta, int x, int[] v)  {  int mijloc;  if (stanga > dreapta)  return -1;  else  {  mijloc = (stanga + dreapta) / 2;  if (x == v[mijloc])  return mijloc;  if (x < v[mijloc])  return cautare(stanga, mijloc - 1, x, v);  else  return cautare(mijloc + 1, dreapta, x, v);  }  }  !!poate fi si doar boolean  static void Main(string[] args)  {  int n, x, i;  //Citeste numarul de elemente  Console.Write("n= ");  n = int.Parse(Console.ReadLine());  //Citeste numarul pe care il cauti  Console.Write("(Numarul pe care il cauti)x= ");  x = int.Parse(Console.ReadLine());  //Citeste cele n elemente si le baga in vectorul v.  int[] v = new int[n];  Console.WriteLine("Cititi cele " + n + " elemente(in ordine crescatoare)");  for (i = 1; i <= n; i++)  {  Console.Write("v[" + i + "]=");  v[i] = int.Parse(Console.ReadLine());  }  //Verifica daca exista sau nu valoarea lui x in vectorul v.  if (cautare(1, n, x, v) != -1)  Console.WriteLine("Elementul " + x + " a fost gasit pe pozitia: " + cautare(1, n, x, v));  else  Console.WriteLine("Elementul " + x + " nu a fost gasit in sir.");  Console.ReadKey();  }  } |

## [RECDC] ridicarea la putere (Horatiu)

## [PD] Fibonacci (Zsolti)

**Enunț**: Să se construiască într-un tablou unidimensional primele n numere din șirul lui Fibonacci, folosind programarea dinamică.

**Soluție**:

1. Prima oară vom construi o funcție auxiliară care va rezolva problema. O vom numi Fibonacci și va lua ca parametru n (dimensiunea tabloului) și ne va returna soluția.

|  |
| --- |
| static int[] Fibonacci(int n)  {  int[] v = new int[n];  …  return v;  } |

Iar din funcția Main o vom apela și vom afișa rezultatele primite în consolă.

|  |
| --- |
| static void Main(string[] args)  {  int[] serie = Fibonacci(10);  for(int i=0;i<serie.Length;i++)  Console.Write(serie[i]+' ');  } |

1. Să construim tabloul. Știind că primele două valori din seria Fibonacci sunt 0 și 1, putem să-I presetăm.

|  |
| --- |
| for(int i = 0; i < v.Length; i++)  if (i < 2)  v[i] = i; |

1. Din punctul acesta, putem folosi programarea dinamică și să calculăm următorul număr. Știind proprietatea fundamentală a șirului, adică elementul urmator este egal cu suma ultimelor 2 elemente din șir.

|  |
| --- |
| for(int i = 0; i < v.Length; i++)  if (i < 2)  v[i] = i;  else  v[i] = v[i - 1] + v[i - 2]; |

**Codul integral**:

|  |
| --- |
| static void Main(string[] args)  {  int[] serie = Fibonacci(1);  for(int i=0;i<serie.Length;i++)  Console.Write(serie[i]+" ");  }  static int[] Fibonacci(int n)  {  int[] v = new int[n];  for(int i = 0; i < v.Length; i++)  if (i < 2)  v[i] = i;  else  v[i] = v[i - 1] + v[i - 2];  return v;  } |

## [PD] Subsecvența de sumă maximă

## [Log] problema celor 2 ouă

## [Log] Formații poker (număr identități)

## [Log] Matricea XOR

## [Log] Bananele și cămila

## [MATH] Aria unui poligon (reprezentarea nu este necesară)

## [SDDU - LSI] TAD Lista SI (Ana) //se considera o clasa Lista cu adaugare la inceput, la final si la mijloc (pozitii random), stergerile cu vector suplimentar etc

## [SDDU - LSI] Lista ordonată (Ana)

## [SDDU - LSI] Numere mari

## [SDDU - COADA] Camere alăturate

## [SDDU - COADA] Drumul cel mai scurt (Zsolti)

## [SDDU - LC] Cavalerii mesei rotunde (Horatiu)

## [RND] Apa dintr-un vector (Alex)

## [RND] Pătrat magic (Aron)

## [RND] Pătrat magic Durer

## [RND] Sudoku

## [RND] Cerc magic (opt)

## [RND] Algoritmul de selecție MC (Zsolti)

Numele Monte-Carlo vine de la bine cunscutul casino din Monaco. Acest algoritm ne ajută să găsim rezolvarea la o mare varietate de probleme, dar din cauza faptul că, implică un grad de randomizare, există o marjă destul de mică, pentru răspunsuri incorecte.

Ca exemplificare a algoritmului, vom rezolva următoarea problemă: Într-un sac se află obiecte de diferite dimensiuni. Astfel la o extragere, sunt șanse mai mari să apuci un obiect mai mare decât unul mic. Pentru a ne ușura treaba, vom crea un tip de date pentru a lucra cu aceste obiecte:

struct Obiect

{

public string nume;

public int dimensiune;

public Obiect(string \_nume, int \_dimensiune)

{

this.nume = \_nume;

this.dimensiune = \_dimensiune;

}

}

După aceea, vor urma initializările unde vom crea sacul cu obiecte de diferite mărimi și o instanța a clasei Random și vom calcula dimensiunea sacului.

static void Main(string[] args)

{

Obiect[] sac =

{

new Obiect("caciula verde",2),

new Obiect("caciula rosie",2),

new Obiect("stilou", 1),

new Obiect("scaun", 5),

new Obiect("mar", 3)

};

Random rnd = new Random();

int dimensiuneaSacului = 0;

foreach (Obiect obiect in sac)

dimensiuneaSacului += obiect.dimensiune;

}

Odată având dimensiunea sacului, putem schimba modul în care interpretăm sacul. În loc să avem un scaun de dimensiunea 5, putem spune că avem 5 scaune de dimensiunea 1. Astfel, toate obiectele din sac vor avea dimensiunea 1, apărând de mai multe ori, rezultând ceva ce seamănă cu un tablou:

**Caciula Verde**| **Caciula Verde** | **Caciula Rosie** | **Caciula Rosie**| **Stilou** | **Scaun** | **Scaun** | **Scaun** | **Scaun** | **Scaun** | **Mar** | **Mar**| **Mar**Iar cu ajutorul unui indice luat aleator, putem extrage un obiect dintr-un sac:

int numarAleator = rnd.Next(dimensiuneaSacului);

int i = 0;

while (numarAleator - sac[i].dimensiune > 0) {

numarAleator -= sac[i].dimensiune;

i++;

}

Tabloul este doar o metodă de a privi sacul, deoarece nu am construit efectiv un tablou, deplasarea în acesta fiind reprezentată printr-un indice al sacului. De exemplu, dacă numarAleator = 2, indicele va merge până la 1, deoarece va trece de sac[0] (fiind căciulă verde, cu dimensiunea 2) dar nu și de sac[1] (fiind căciulă roșie, tot de dimensiunea 2). Astfel, obiectul ales va fi Căciulă roșie.

class Program

{

struct Obiect

{

public string nume;

public int dimensiune;

public Obiect(string \_nume, int \_dimensiune)

{

this.nume = \_nume;

this.dimensiune = \_dimensiune;

}

}

static void Main(string[] args)

{

Obiect[] sac =

{

new Obiect("caciula verde",2),

new Obiect("caciula rosie",2),

new Obiect("stilou", 1),

new Obiect("scaun", 5),

new Obiect("mar", 3)

};

Random rnd = new Random();

int dimensiuneaSacului = 0;

foreach (Obiect obiect in sac)

dimensiuneaSacului += obiect.dimensiune;

int numarAleator = rnd.Next(dimensiuneaSacului);

int i = 0;

while (numarAleator - sac[i].dimensiune > 0) {

numarAleator -= sac[i].dimensiune;

i++;

}

Console.WriteLine(sac[i].nume);

Console.ReadKey();

}

}

## [RND] Automate celulare

## [RND] Automate celulare simetrice (Fredkin 1960)

## [] Spirolaterele lui Franks Odds

## [AG] TAD A.G. (Zsolti)

## [AG] Dispersia unui graf (Remus)

1. De mana
2. Aleator
3. Pe cerc
4. Alg genetic

## [GRP] Reprezentări

## [GRP] Intersecții de segmente

## [GRP] Parametrizarea poligoanelor regulate. (RTS) (Remus)

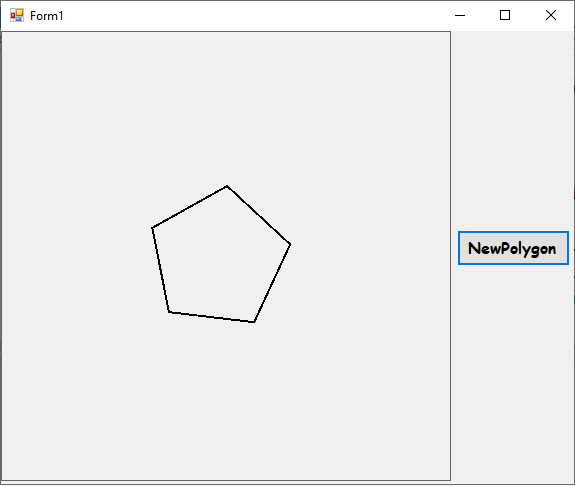
Un poligon este o figură geometrică închisă formată dintr-un număr finit de laturi. Un poligon regulat are toate laturile și toate unghiurile egale. Câteva poligoane regulate cunoscute sunt triunghiul echilateral, pătratul, pentagonul, hexagonul, octagonul, etc. În acest program, vom încerca să creăm o metodă care poate desena orice poligon regulat folosind un minim de informații despre acesta.

În primul rând, această aplicație va fi realizată in Windows Form. Avem nevoie de un pictureBox și de un buton. Parametri ce definesc un poligon regulat sunt numărul de vârfuri și distanța de la oricare punct la centrul de greutate. Vom crea o metodă care primește acești parametri pentru a desena poligonul. Vom avea nevoie de niște obiecte globale pentru desenare: un Bitmap, un Graphics și un Random. Când apăsăm pe buton, se creează un nou bitmap și un nou graphics. Luăm două valori la întâmplare pentru numărul de puncte și pentru rază, și vom apela metoda de desenare a poligonului pe care o vom crea imediat. La final, imaginea din pictureBox devine cea din Bitmap.

|  |
| --- |
| Bitmap bmp;  Graphics grp;  Random r = new Random();  private void button1\_Click(object sender, EventArgs e)  {  bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  grp = Graphics.FromImage(bmp);    int n = r.Next(3, 10);  int length = r.Next(50, pictureBox1.Width / 4);  Poligon(n, length);  pictureBox1.Image = bmp;  } |

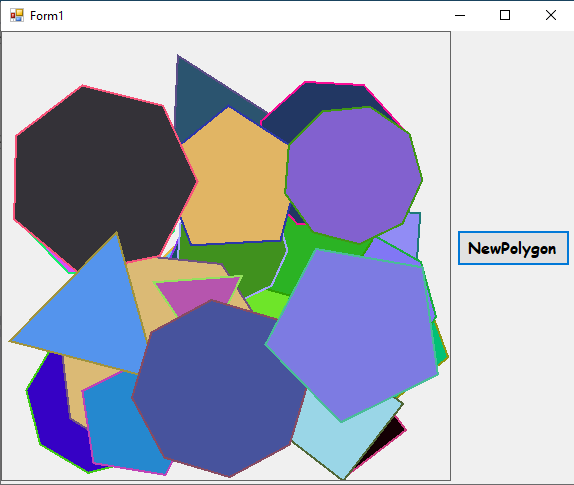
În interiorul metodei, avem nevoie de un unghi egal între toate punctele, care are valoarea 2pi / n. Vom lua un punct pentru centrul poligonului și un vector de n puncte ce trebuiesc calculate. Pentru fiecare punct din vector, calculăm valorile float pentru x și y astfel: luăm valoarea din centru la care adunăm produsul dintre lungimea până la punct și cosinus din theta\*i, respectiv sinus din aceeași valoare. La final, apelăm metoda DrawPolygon a obiectului Graphics.

|  |
| --- |
| void Poligon(int n, int length)  {  double theta = (Math.PI \* 2) / n;  Point center = new Point(pictureBox1.Width / 2, pictureBox1.Height / 2);  PointF[] points = new PointF[n];  for (int i = 0; i < n; i++)  {  float x = center.X + length \* (float)Math.Cos(theta \* i);  float y = center.Y + length \* (float)Math.Sin(theta \* i);  points[i] = new PointF(x, y);  }  grp.DrawPolygon(new Pen(Color.Black, 2), points);  } |



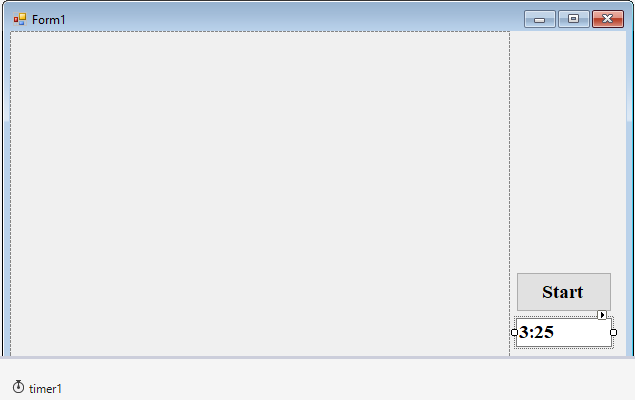
De aici, putem aduce modificări pentru a înfrumuseța aplicația: la o apăsare de buton, nu resetăm bitmap de fiecare dată; centrul poate fi un punct la întâmplare, primit ca și parametru în metodă; pentru a nu avea aceeași rotație de fiecare dată, vom primi ca parametru un unghi inițial al figurii; vom da culori la întâmplare pentru a umple și desena conturul figurilor. Acestea sunt doar niște exemple de lucruri ce se pot face, dar variantele sunt nenumărate.

|  |
| --- |
| public Form1()  {  InitializeComponent();  bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  grp = Graphics.FromImage(bmp);  pictureBox1.Image = bmp;  }  Bitmap bmp;  Graphics grp;  Random r = new Random();  private void button1\_Click(object sender, EventArgs e)  {  int n = r.Next(3, 10);  int length = r.Next(50, pictureBox1.Width / 4);  Point center = new Point();  center.X = r.Next(length, pictureBox1.Width - length);  center.Y = r.Next(length, pictureBox1.Height - length);  double angle = r.NextDouble() \* (Math.PI \* 2) / 3;  Poligon(center, n, length, angle);  pictureBox1.Image = bmp;  }  void Poligon(Point center, int n, int length, double angle)  {  double theta = (Math.PI \* 2) / n;  PointF[] points = new PointF[n];  for (int i = 0; i < n; i++)  {  float x = center.X + length \* (float)Math.Cos(theta \* i + angle);  float y = center.Y + length \* (float)Math.Sin(theta \* i + angle);  points[i] = new PointF(x, y);  }  Color fill = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));  Color draw = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));  grp.FillPolygon(new SolidBrush(fill), points);  grp.DrawPolygon(new Pen(draw, 2), points);  } |



## [GRP] Desenarea unui ceas Analog (Remus)

Pentru acest program, ne propunem să desenăm un ceas analog funcțional în Windows Form Application. Pentru aceasta, vom avea nevoie de un pictureBox, pus la dimensiunea 500, 500, de un timer, de un textBox în care să scriem o oră de început și de un buton de start care să înceapă timer-ul și ceasul, de la ora setată în textBox.



Am creat clasa Engine, în care se vor face toate acțiunile majore în aplicație, și unde avem salvate toate variabilele statice în program. Avem nevoie de un Graphics, un Bitmap, un PictureBox și dimensiunile acestuia, valorile întregi resx și resy, de raza ceasului R, de minutul și ora, de float alpha pentru a calcula unghiul la care se află acele ceasului, și de o valoare float fi în care păstrăm aceeași valoare, 3 pi supra 2, deoarece valoarea unghiului în informatică nu este la fel ca și cea în matematică. Pentru a nu ne repeta, vom declara și culoarea ceasului Color color și pentru a desena acele ceasului, Pen silverPen.

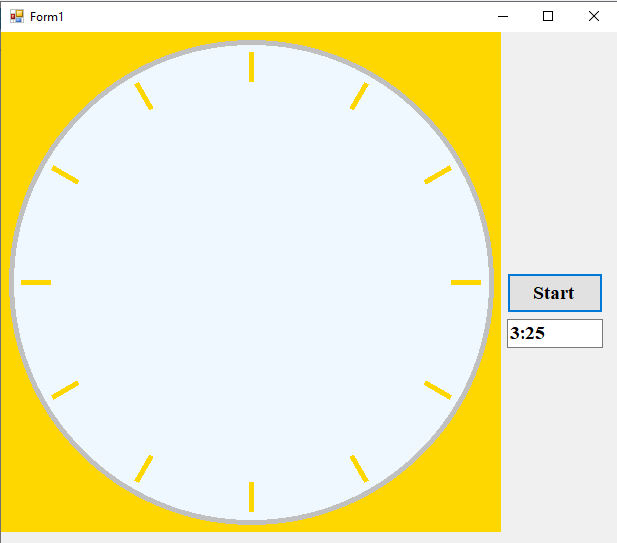
|  |
| --- |
| public static class Engine  {  public static Graphics grp;  public static Bitmap bmp;  public static int resx, resy;  public static PictureBox pB;  public static Color color = Color.Gold;  public static Pen silverPen = new Pen(Color.Silver, 5);  public static int R;  public static float alpha;  public static float fi = (float)(Math.PI \* 3) / 2;  public static int minute = 0, hour = 0; |

La rularea programului, vom apela metoda Init pentru a inițializa aceste valori.

|  |
| --- |
| public static void Init(PictureBox P)  {  pB = P;  resx = P.Width;  resy = P.Height;  bmp = new Bitmap(resx, resy);  grp = Graphics.FromImage(bmp);  grp.Clear(color);  pB.Image = bmp;  } |

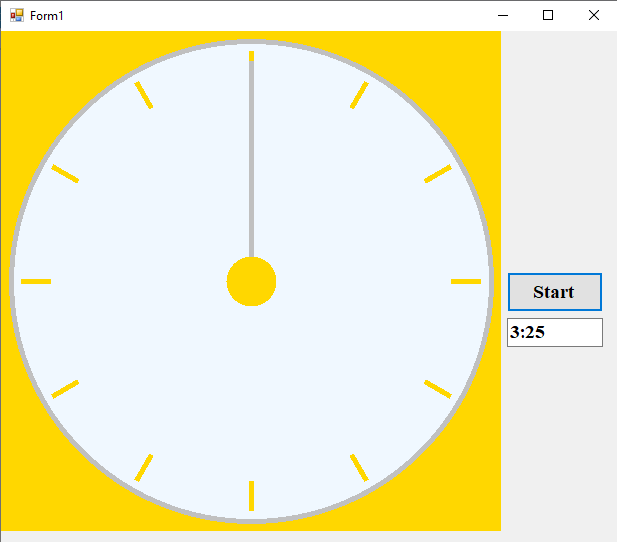
Acum trebuie să scriem o metodă ce va desena ceasul în funcție de oră și minut. Începem cu desenarea ceasului în sine: un cerc peste culoarea aurie, de culoare deschisă, cu mici ace desenate la fiecare dintre cele 12 ore. Atribuim valoare razei R, care va fi jumătate din resx. Pentru a nu avea cercurile până în marginile pictureBox-ului, scădem întâi 20 de pixeli din resx. După aceea, folosim metodele FillEllipse, și DrawEllipse pentru aspect. Acum pentru a desena acele la înclinația potrivită, i se atribuie lui alpha valoarea 2 pi supra 12, și parcurgem cu i de la 0 la 12. De fiecare dată, trebuie calculat x și y al unui capăt al acului, respectiv x și y al celuilalt capăt. Pentru x, începem calculul din mijlocul ceasului, adică de la resx/2, după care calculăm valoarea adăugată astfel: ne trebuie niște distanță de la marginea ceasului până la capătul acului, deci vom înmulți R-10, cu cosinus din i\*alpha+fi. Pentru y, înmulțim cu sinus din același unghi, iar pentru celălalt capăt al acului, se înmulțește R-40 cu aceleași valori. După care se deseneaza o linie de la un punct la altul.

|  |
| --- |
| public static void DrawMap()  {  R = (resx - 20) / 2;  grp.FillEllipse(new SolidBrush(Color.AliceBlue), resx/2 - R, resy/2 - R, 2 \* R, 2 \* R);  grp.DrawEllipse(silverPen, resx/2 - R, resy/2 - R, 2 \* R, 2 \* R);  alpha = (float)Math.PI \* 2 / 12;  for( int i=0; i<12; i++)  {  float x = resx/2 + (R - 40) \* (float)Math.Cos(i \* alpha + fi);  float y = resy/2 + (R - 40) \* (float)Math.Sin(i \* alpha + fi);  float \_x = resx/2 + (R - 10) \* (float)Math.Cos(i \* alpha + fi);  float \_y = resy/2 + (R - 10) \* (float)Math.Sin(i \* alpha + fi);  grp.DrawLine(new Pen(Color.Gold, 5), \_x, \_y, x, y);  } |



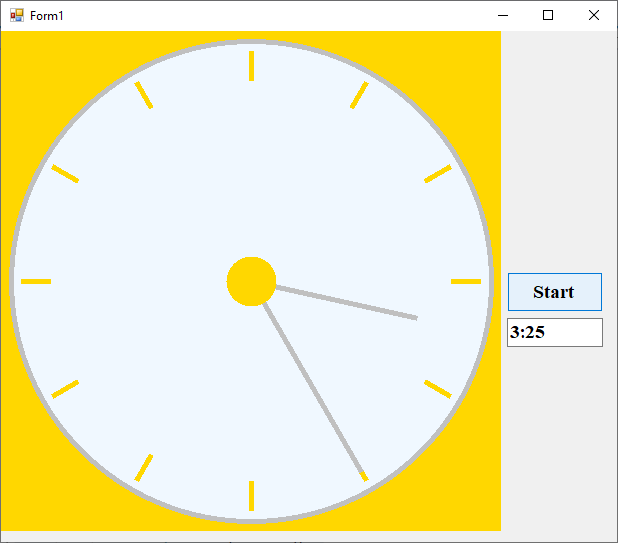
În continuare, trebuie să desenăm un ac argintiu mai lung pentru minute, respectiv unul mai scurt pentru ora. Pentru înclinația corectă, ne trebuie ca alpha să ia valoarea 2 pi supra 60. Calculăm x și y similar, începem din centru, resx/2, apoi înmulțim R-20 cu cosinus din alpha\*minute + fi, iar pentru y aven sinus din aceeasi valoare, și apoi desenăm linia din centru la punctul calculat. Pentru ora, alpha ia valoarea 2 pi supra 12. De data aceasta, avem nevoie de încă un unghi gamma pentru a calcula distanța de acul cu ora actuală, deoarece acesta nu trece direct de la primul ac la al doilea, ci se apropie de al doilea incet. Deci, cu fiecare minut, acul se deplasează foarte puțin. Pe porțiunea unei ore de 1/12 din tot cercul, acul se mișcă de 60 de ori, deci gamma este 2 pi supra 12\*60. Calculul lui x este resx/2 adunat cu produs între R-70 (pentru a fi mai scurt) și cosinus din alpha\*hour + fi + gamma\*minute. După care desenăm acul de la centru până la punctul calculat. Pentru aspect, desenăm un cerc auriu de rază mică în centru peste acele acestea.

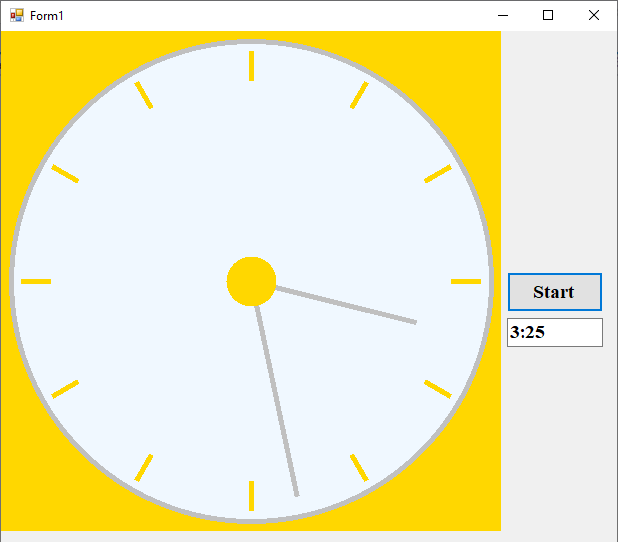
|  |
| --- |
| alpha = (float)(Math.PI \* 2) / 60;  float xm = resx / 2 + (R - 20) \* (float)Math.Cos(alpha \* minute + fi);  float ym = resy / 2 + (R - 20) \* (float)Math.Sin(alpha \* minute + fi);  grp.DrawLine(silverPen, resx/2, resy/2, xm, ym);  float gamma = (float)(Math.PI \* 2) / (12 \* 60);  alpha = (float)(Math.PI \* 2) / 12;  float xo = resx / 2 + (R - 70) \* (float)Math.Cos(alpha \* hour + fi + gamma \* minute);  float yo = resy / 2 + (R - 70) \* (float)Math.Sin(alpha \* hour + fi + gamma \* minute);  grp.DrawLine(silverPen, resx/2, resy/2, xo, yo);  grp.FillEllipse(new SolidBrush(Color.Gold), resx/2-25,resy/2-25,50,50);  pB.Image = bmp;  }  } |



Tot ce mai lipsește este calculul orei. Vom seta implicit ca timer1 să aibă intervalul 1000, adică o secundă, și în fiecare secundă minutul crește. Trebuie să ținem cont că minutul nu poate ajunge la 60, iar ora nu poate ajunge la 12. La fiecare tick, apelăm metoda DrawMap. Atunci când apăsăm pe Start, inițializăm ora și minutul cu valorile din textBox1, pornim timerul și desenăm harta.

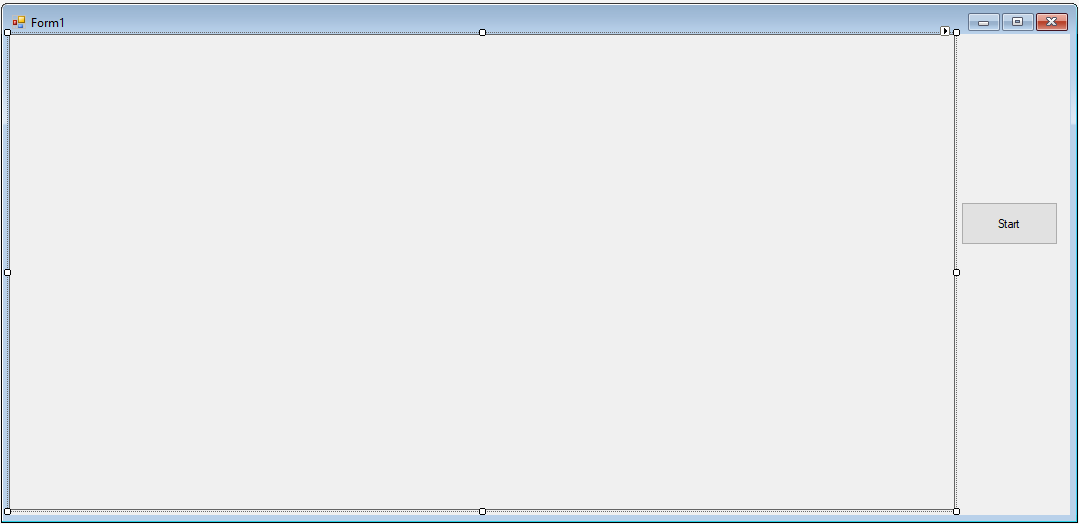
|  |
| --- |
| public partial class Form1 : Form  {  public Form1()  {  InitializeComponent();  }  private void Form1\_Load(object sender, EventArgs e)  {  Engine.Init(pictureBox1);  Engine.DrawMap();  }  private void timer1\_Tick(object sender, EventArgs e)  {  Engine.minute++;  if (Engine.minute == 60)  {  Engine.minute = 0;  Engine.hour++;  }  if (Engine.hour == 12)  Engine.hour = 0;  Engine.DrawMap();  }  private void button1\_Click(object sender, EventArgs e)  {  timer1.Enabled = true;  string s = textBox1.Text;  Engine.hour = int.Parse(s.Split(':')[0]);  Engine.minute = int.Parse(s.Split(':')[1]);  Engine.DrawMap();  }  } |





## [GRP/BCKT] Mulțimi de puncte. Aria Triunghiului minim (Remus)

În acest program, avem ca date de intrare un vector de puncte, și ni se cere să afișăm punctele și să desenăm triunghiul cu aria minimă care se poate forma cu punctele date. Aplicația este realizată în Windows Form, și avem nevoie de un pictureBox și de un buton de Start. Deoarece având o listă de multe puncte poate rezulta ușor în trei puncte coliniare, vom avea un pictureBox cât mai mare.



Vom folosi unele variabile în întregul program, deci le vom declara global. Pentru a desena punctele și triunghiul cu aria minimă, vom avea nevoie de un Graphics, de un Bitmap și de un Pen, declarăm și vectorul de puncte și dimensiunea sa și o variabilă Random pentru a inițializa vectorul cu valori aleatoare de fiecare dată.

|  |
| --- |
| Graphics grp;  Bitmap bmp;  Pen pen = new Pen(Color.Black);  PointF[] points;  int n;  Random r = new Random(); |

La fiecare apăsare de buton, vom reinițializa bmp și grp pentru a nu desena peste ce am desenat înainte. Dăm valoare lui n, inițializăm vectorul și îl parcurgem pentru a genera aleator fiecare punct din vector, și apoi să îl desenăm ca și o elipsă.

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  grp = Graphics.FromImage(bmp);  n = 10;  points = new PointF[n];  for (int i = 0; i < n; i++)  {  points[i] = new PointF(r.Next(pictureBox1.Width), r.Next(pictureBox1.Height));  grp.DrawEllipse(pen, points[i].X-2, points[i].Y-2, 5, 5);  }  ...  } |

Pentru a verifica toate cazurile posibile și pentru a determina care este triunghiul de arie minimă, va trebui să parcurgem toate variantele de 3 puncte distincte din vectorul dat. Acest lucru se realizează folosind backtracking, mai exact generând toate combinările de n elemente luate câte 3. Știind algoritmul pentru combinări, urmează să îl adaptăm problemei noastre. Vectorul soluție va reține indicii punctelor care urmează să fie verificate, dar avem nevoie și de indicii punctelor care formează triunghiul minim curent pentru a ajuta la verificare. În concluzie, avem nevoie de un vector în plus care este inițial format din valorile 0, 1 și 2 (iar prima valoare este de asemenea 0, deoarece aceea nu se ia în calcul).

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  ...  int[] sol = new int[4];  bool[] b = new bool[n];  int[] minCurent = new int[4] { 0, 0, 1, 2 };  Backtracking(1, n, 3, sol, b, minCurent);  ...  } |

În metoda Backtracking, doar porțiunea din if este schimbată de original. Aici se prelucrează datele, se decide ce se face cu noua combinare generată. În cazul acestei probleme, comparăm aria minimului curent cu aria soluției nou generate. Dacă prima este mai mare, atunci valorile din minimul curent se vor schimba cu cele din soluție. Pentru a nu scrie totul pe o singură linie, vom crea noi variabile pentru fiecare dintre punctele minimului curent, si fiecare dintre punctele soluției.

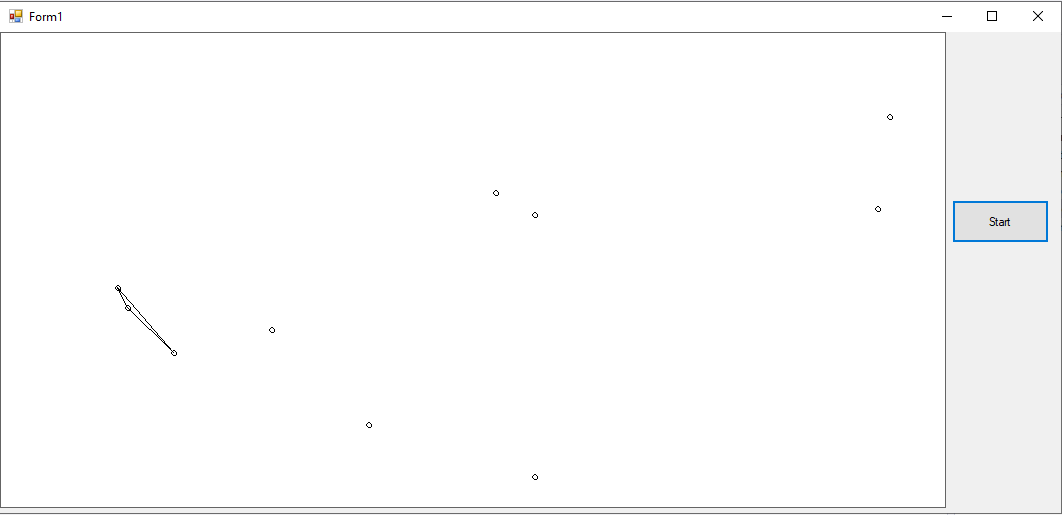
|  |
| --- |
| void Backtracking(int k, int n, int p, int[] sol, bool[] b, int[] minCurent)  {  if (k > p)  {  PointF minA = points[minCurent[1]];  PointF minB = points[minCurent[2]];  PointF minC = points[minCurent[3]];  PointF A = points[sol[1]];  PointF B = points[sol[2]];  PointF C = points[sol[3]];  if(Aria(minA,minB,minC) > Aria(A,B,C))  for(int i = 1; i <= p; i++)  minCurent[i] = sol[i];  }  else  for(int i = sol[k - 1]; i < n; i++)  if (b[i] == false)  {  b[i] = true;  sol[k] = i;  Backtracking(k + 1, n, p, sol, b, minCurent);  b[i] = false;  }  } |

Funcția care calculează aria triunghiului format din trei puncte date este următoarea:

|  |
| --- |
| float Aria(PointF a, PointF b, PointF c)  {  return Math.Abs(a.X \* (b.Y-c.Y) + b.X \* (c.Y-a.Y) + c.X \* (a.Y-b.Y))/2;  } |

După ce avem cele trei puncte generate, trebuie doar să mai desenăm triunghiul. Acest lucru se poate face ușor folosind metoda DrawPolygon a unui obiect de tip Graphics, care primeste ca argument un obiect Pen și un vector de puncte. Declarăm vectorul si îi atribuim valorile punctelor din vectorul original de pe pozițiile date de vectorul obținut prin Backtracking. La final, imaginea afișată în pictureBox va fi Bitmap-ul pe care am desenat.

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  ...  PointF[] triunghi = new PointF[3];  triunghi[0] = points[minCurent[1]];  triunghi[1] = points[minCurent[2]];  triunghi[2] = points[minCurent[3]];  grp.DrawPolygon(pen, triunghi);  pictureBox1.Image = bmp;  } |



## [GRP/REC] Fractali geometrici

## [GRP/REC] Mandlebro

## [GRP] Mulțimi de puncte. Acoperirea convexă (Csongor)

De studiat

## [GRP] Reprezentare de funcții (Zsolti)

## [GRP] Implementarea algoritmilor de procesare a matricelor pe algebra RGB și crearea de filtre. (Marius)

Se dă o imagine, se cere crearea unor filtre, ex. greyscale și afișarea noii imagini.

Vom folosi procesarea matricilor pe algebra RGB (Red, Green, Blue, culorile ce formează fiecare pixel).

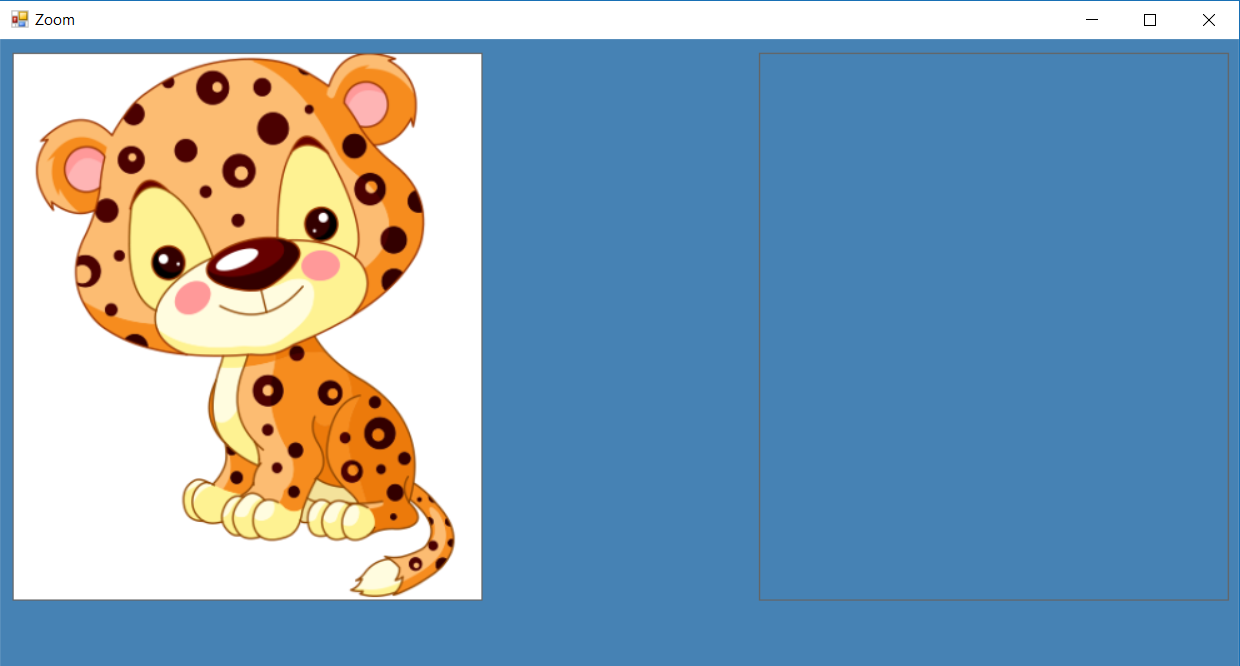
Pentru ca exemplul să fie cât mai demonstrativ, pentru această problemă vom crea 8 filtre.

Vom crea două variabile „picturebox1” și „picturebox2” și două variabile de tip bitmap “sursa” și “destinatie”. În metoda „Form1\_Load” vom inițializa variabila „sursa” cu un nou bitmap căruia îi vom oferi ca argument sursa unei imagini (ex. @"..\..\Resurse\tigru.png") și îi vom da lui „picturebox1” ca imagine variabila „sursa”.

|  |
| --- |
| Bitmap sursa, destinatie;  private void Form1\_Load(object sender, EventArgs e)  {  sursa = new Bitmap(@"..\..\Resurse\tigru.png");  pictureBox1.Image = sursa;  } |

În momentul de față, ar trebui să avem ceva asemănător cu imaginea de mai jos.

**Img1.png**



Acum că avem o imagine încărcată, putem începe sa creăm filtrele. Începem prin adăugarea unui buton și la acționarea acestuia, în „picturebox2” vom afișa imaginea cu un filtru de grayscale.

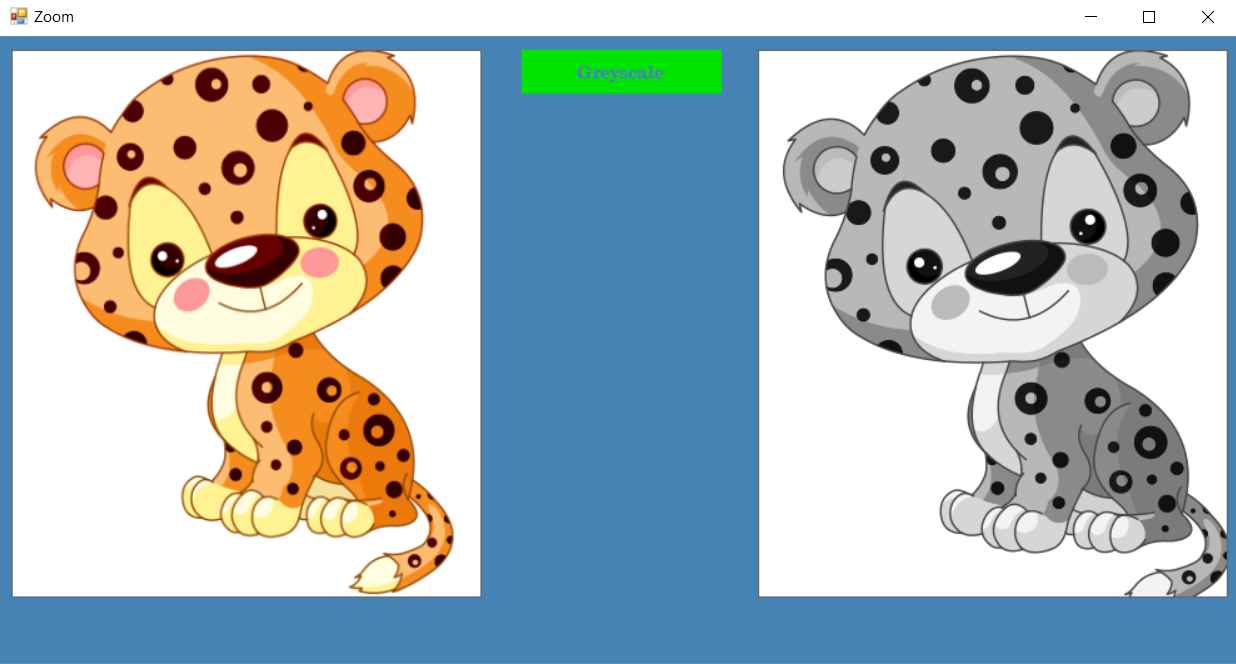
Crearea filtrului greyscale: vom inițializa variabila „destinatie” cu un nou bitmap de aceeași dimensiune ca și imaginea variabilei „sursa”. În continuare vom parcurge fiecare pixel al variabilei „sursa” și folosim metoda „GetPixel” pentru a primi culorile pixelului respectiv. În continuare, într-o variabilă „t” vom stoca media aritmetica a valorii culorilor. Setăm fiecare dintre cele 3 culori ale pixel-ului variabilei „destinatie” corespunzător „sursei” cu „t” cu ajutorul metodei „SetPixel”.

În final, îi oferim variabilei „picturebox2” ca imagine variabila destinație.

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)//greyscale  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  int i, j;  for(i = 0;i < sursa.Width;i++)  {  for(j=0;j<sursa.Height;j++)  {  Color k = sursa.GetPixel(i, j);  int t = (k.R + k.G + k.B) / 3;  Color nk = Color.FromArgb(t, t, t);  destinatie.SetPixel(i, j, nk);  }  }  pictureBox2.Image = destinatie;  } |

In acest momentul ar trebui să avem ceva asemănător cu imaginea de mai jos.

**Img2.png**

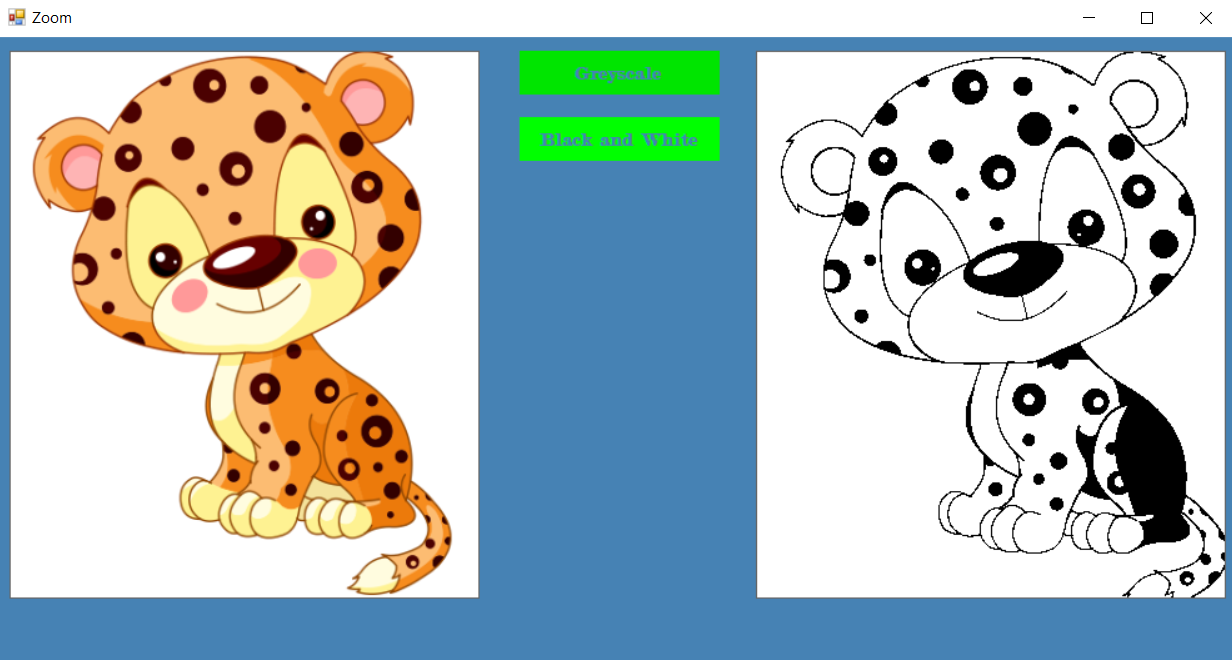


În continuare vom crea un filtru alb-negru, vom adăuga un nou buton în form și vom repeta același procedeu ca la filtrul anterior, doar că după crearea variabilei „t” vom verifica dacă aceasta este mai mică decât 128 (255/2), în caz afirmativ vom seta culoarea pixelul respectiv la alb iar in caz contrar la negru, iar în continuare vom proceda la fel ca la filtrul anterior.

|  |
| --- |
| private void button2\_Click(object sender, EventArgs e)//black and white  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  int i, j;  for (i = 0; i < sursa.Width; i++)  {  for (j = 0; j < sursa.Height; j++)  {  Color k = sursa.GetPixel(i, j);  int t = (k.R + k.G + k.B) / 3;  Color nk;  if (t<128)  nk = Color.FromArgb(0, 0, 0);  else  nk = Color.FromArgb(255, 255, 255);  destinatie.SetPixel(i, j, nk);  }  }  pictureBox2.Image = destinatie;  } |

Programul nostru acum va arata așa:

**Img3.png**



Al treilea filtru pe care îl vom crea este unul de luminozitate(luminozitate ridicata).

Procedeul este similar pentru acest filtru, nu vom mai crea variabila „t” dar vom avea o variabilă „p” care va stoca nivelul luminozității și variabilei „destinatie” îi vom seta culoarea astfel: valorile lui roșu, verde și albastru le vom seta să fie egale cu minimul dintre (valoarea culorii + variabila luminozitate) și 255.

|  |
| --- |
| private void button3\_Click(object sender, EventArgs e)//luminozitate ridicata  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  int i, j;  for (i = 0; i < sursa.Width; i++)  {  for (j = 0; j < sursa.Height; j++)  {  Color k = sursa.GetPixel(i, j);  int p = 50;  Color nk = Color.FromArgb(Math.Min(k.R+p,255), Math.Min(k.G + p, 255), Math.Min(k.B + p, 255));  destinatie.SetPixel(i, j, nk);  }  }  pictureBox2.Image = destinatie;  } |

**Img4.png**

****

Asemănător putem realiza al patrulea filtru care va fi un filtru de luminozitate scăzută.

Acest filtru este opus celui anterior, sigura diferență dintre ele fiind faptul că la calcularea minimului va fi scădere, la filtrul anterior fiind adunare.

|  |
| --- |
| private void button4\_Click(object sender, EventArgs e)//luminozitate scazuta  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  int i, j;  for (i = 0; i < sursa.Width; i++)  {  for (j = 0; j < sursa.Height; j++)  {  Color k = sursa.GetPixel(i, j);  int p = 50;  Color nk = Color.FromArgb(Math.Max(k.R - p, 0), Math.Max(k.G - p, 0), Math.Max(k.B - p, 0));  destinatie.SetPixel(i, j, nk);  }  }  pictureBox2.Image = destinatie;  } |

**Img5.png**



Al cincilea filtru va fi cel de contrast.

Se creează variabilă “p” aceasta reprezentând nivelul contrastului. Se calculează media valorii culorilor, se verifica dacă aceasta este mai mica de 128, iar în caz pozitiv fiecare culoare a pixelului respectiv este setata astfel: exemplu pentru roșu va fi egal cu maximul dintre (roșu – “p” și 0) iar în caz negativ minim dintre (roșu + „contrast” și 255).

|  |
| --- |
| private void button5\_Click(object sender, EventArgs e)//contrast  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  int i, j;  for (i = 0; i < sursa.Width; i++)  {  for (j = 0; j < sursa.Height; j++)  {  Color k = sursa.GetPixel(i, j);  int p = 50;  Color nk;  int t = (k.R + k.G + k.B) / 3;  if (t < 128)  nk = Color.FromArgb(Math.Max(k.R - p, 0), Math.Max(k.G - p, 0), Math.Max(k.B - p, 0));  else  nk = Color.FromArgb(Math.Min(k.R + p, 255), Math.Min(k.G + p, 255), Math.Min(k.B + p, 255));  destinatie.SetPixel(i, j, nk);  }  }  pictureBox2.Image = destinatie;  } |

**Img6.png**

Filtrul următor va fi un filtru pentru culori complementare și se realizează astfel: Se calculează media valorii culorilor și se setează fiecare pixel astfel: din 255 se scade valoarea fiecărei culori.

|  |
| --- |
| private void button9\_Click(object sender, EventArgs e)//culori complementare  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  for (int i = 0; i < sursa.Width; i++)  {  for (int j = 0; j < sursa.Height; j++)  {  Color Get = sursa.GetPixel(i, j);  Color Set;  int t = (Get.R + Get.G + Get.B) / 3;  int rp = Get.R;  int gp = Get.G;  int bp = Get.B;  Set = Color.FromArgb(255 - rp, 255 - gp, 255 - bp);  destinatie.SetPixel(i, j, Set);  }  }  pictureBox2.Image = destinatie;  } |

**Img7.png**

****

În continuare vom crea un filtru roșu.

Se setează valoarea culorilor albastru și verde la 0. Asemănător se pot realiza si filtrele albastru și verde.

|  |
| --- |
| private void button10\_Click(object sender, EventArgs e)//filtru rosu  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  for (int i = 0; i < sursa.Width; i++)  {  for (int j = 0; j < sursa.Height; j++)  {  Color Get = sursa.GetPixel(i, j);  Color Set = Color.FromArgb(Get.R, 0, 0);  destinatie.SetPixel(i, j, Set);  }  }  pictureBox2.Image = destinatie;  } |

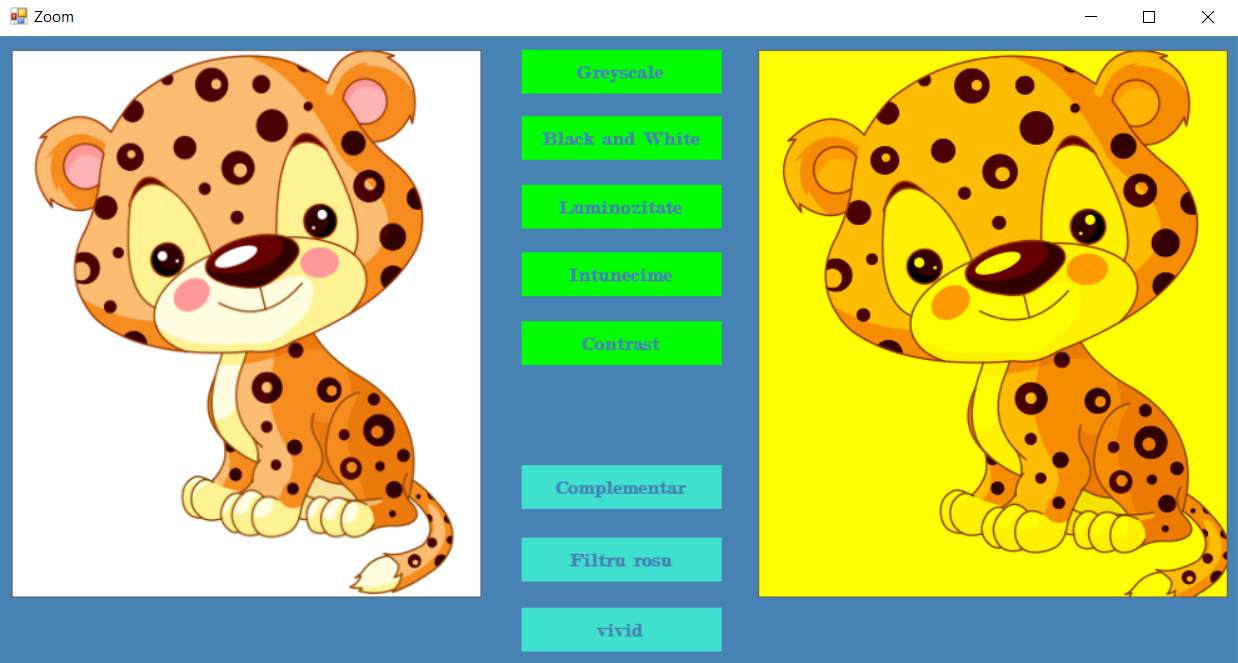
**Img8.png**



Ultimul filtru este cel red-green (vivid colors). Variabila destinație este inițializată/ schimbată cu un bitmap nou, se setează valoarea culorii albastru la 0. Asemănător se poate realiza si filtrele green-blue și red-blue.

|  |
| --- |
| private void button11\_Click(object sender, EventArgs e)//vivid colors  {  destinatie = new Bitmap(sursa.Width, sursa.Height);  for (int i = 0; i < sursa.Width; i++)  {  for (int j = 0; j < sursa.Height; j++)  {  Color Get = sursa.GetPixel(i, j);  Color Set;  int t = (Get.R + Get.G + Get.B) / 3;  Set = Color.FromArgb(Get.R, Get.G, 0);  destinatie.SetPixel(i, j, Set);  }  }  pictureBox2.Image = destinatie;  } |

**Img9.png**



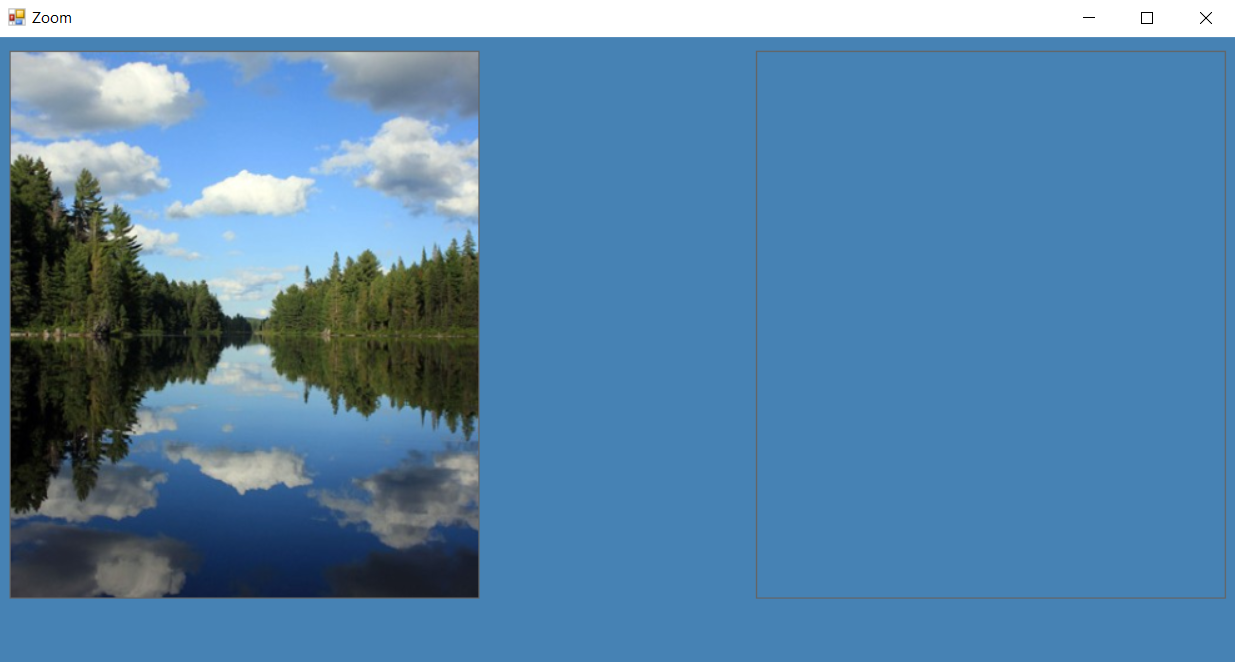
## [GRP] Construirea unui zoom pe o matrice de pixeli pe baza mediei componentelor de culoare. (Marius)

Se dă o imagine, se cere crearea unui zoom.

Vom folosi procesarea matricilor pe algebra RGB și vom crea un program de tip “windows froms application”.

Vom adăuga în form „picturebox1” și „picturebox2” și vom crea două variabile de tip bitmap “sursa” și “destinatie”. În metoda „Form1\_Load” vom inițializa variabila „sursa” cu un nou bitmap căruia îi vom oferi ca argument sursa unei imagini (ex. @"..\..\Resurse\nature.jpg") și îi vom da variabilei „picturebox1” ca imagine variabila „sursa”.

|  |
| --- |
| Bitmap sursa, destinatie;  private void Form1\_Load(object sender, EventArgs e)  {  sursa = new Bitmap(@"..\..\Resurse\nature.jpg");  pictureBox1.Image = sursa;  } |



Vom adăuga un buton în form și la acționarea acestuia vom afișa în variabila „picturebox2” imaginea variabilei „picturebox1” mărită.

Mai jos vom explica secvența de cod executată la acționarea butonului.

Vom începe cu definirea a două variabile de tip int „n”, care va conține lățimea variabilei „sursa”, și „m” care va conține înălțimea. Vom inițializa variabila „destinatie” cu un nou bitmap de dimensiune dublă în comparație cu variabila „sursa”.

În continuare vom seta fiecare pixel par al variabilei „destinatie” cu pixel/ 2 al variabilei „sursa”. Astfel vor rămâne goluri pe pozițiile coloanelor impare și liniilor impare.

Programul constă in 3 parcurgeri ale matricei „sursa”, prima fiind cea prezentata anterior, care a copiat câțiva pixeli în variabila „destinatie” dar care a lăsat câteva goluri, următoarele 2 parcurgeri se vor ocupa de umplerea acestor goluri prin realizarea mediei între culori. Următoarea parcurgere se va concentra pe pixelii de pe linie pară si coloană impară, acești pixeli vor primii culoarea realizată astfel: se creează două variabile “c1” si “c2” de tip Color care vor conține culoarea pixelului de pe poziția(i,j-1) pentru “c1” iar pentru “c2” de pe poziția (i,j+1).

Ultima parcurgere se va concentra pe pixelii rămași, adică cei de pe linie impară și coloană pară, implementarea fiind asemănătoare cu cea a parcurgerii anterioare.

Și acesta este algoritmul de zoom bazat pe media componentelor de culoare, codul executat la acționarea butonului este prezentat mai jos, precum și o imagine care reprezintă rezultatul execuției programului.

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)//zoom  {  int n = sursa.Width;  int m = sursa.Height;  destinatie = new Bitmap(2\*n-1, 2\*m-1);  int i, j;  for (i=0;i<n;i++)  {  for(j=0;j<n;j++)  {  destinatie.SetPixel(2 \* i, 2 \* j,sursa.GetPixel(i, j));  }  }  for (i = 0; i < 2\*n-1; i=i+2)  {  for (j = 1; j < 2\*m-1; j+=2)  {  Color c1 = destinatie.GetPixel(i, j - 1);  Color c2 = destinatie.GetPixel(i, j + 1);  Color c3 = Color.FromArgb((c1.R + c2.R)/2, (c1.G + c2.G) / 2, (c1.B + c2.B) / 2);  destinatie.SetPixel(i, j, c3);  }  }  for (i = 1; i < 2\*n-1; i+=2)  {  for (j = 0; j < 2\*n-1; j++)  {  Color c1 = destinatie.GetPixel(i - 1, j);  Color c2 = destinatie.GetPixel(i + 1, j);  Color c3 = Color.FromArgb((c1.R + c2.R) / 2, (c1.G + c2.G) / 2, (c1.B + c2.B) / 2);  destinatie.SetPixel(i, j, c3);  }  }  pictureBox2.Image = destinatie;  } |

## 

**!! completare**

## [GEN] Teorema lui Pick (Csongor)

În acest exercițiu, ne propunem să calculăm aria unui poligon care are vârfurile în nodurile unei rețele de pătrate.

Pentru asta creăm clasa Pick unde vom citi datele din fișier, acestea fiind stocate în următorul fel: pe prima linie vom avea dimensiunea matricei și numărul de puncte, iar pe celelalte linii vom avea coordonatele punctelor una câte una, unde numerotarea punctelor începe de la 1. De exemplu:

8 4 (vom avea nevoie de o matrice de 8x8) (avem 4 puncte)

0 0 (primul punct are coordonatele x:0 y:0)

0 4 (al doilea punct are coordonatele x:0 y:4)

4 4 (al treilea punct are coordonatele x:4 y:4)

4 0 (ultimul punct are coordonatele x:4 y:0)

În clasa Pick, vom avea opt variablie globale

|  |
| --- |
| public int[,] mat; Îl vom folosi drept sistem cartezian  public Point[] Puncte; Coordonatele punctelor  public int NumarPuncte; Numărul total al punctelor  public int DimMat; Dimensiunea sistemului cartezian  int PuncteInterior=0; Numărul punctelor din interior  int PuncteLatura; Numărul punctelor de pe laturi  float Arie; Aria poligonului |

Avem nevoie de un constructor care are ca parametru un string FisierSursa, care conține destinația fișierului sursă. În constructor vom avea nevoie de TextReader FisierDate, cu care vom citi datele din fișier, iar DimMat va fi egal cu primul număr citit din fișier. Variabila NumarPuncte va fi egal cu al doilea număr pe care îl citim din fișier care ne arată numărul total de puncte, după aceasta citim fiecare linie una câte una pentru a popula sistemul cartezian și vectorul de puncte, iar apoi numărăm fiecare punct din rețea și ne uităm dacă face parte din laturi sau este în interior. La final, calculăm aria astfel: Arie = PuncteLatura / 2 + PuncteInterior - 1; unde PuncteLatura este numărul punctelor de pe laturi și PuncteInterior este numărul punctelor din interior. Punctelor de pe laturi la dăm valoarea 3 și pe a celor din interior valoarea 2 pentru a putea fi vizualizate.

Vom avea nevoie de două structuri pentru rezolvarea problemei: pentru a putea stoca puncte struct Point, iar pentru a putea stoca linii struct Linie.

|  |
| --- |
| public struct Point  {  public double x;  public double y;  public Point(double x, double y)  {  this.x = x;  this.y = y;  }  }  public struct Linie  {  public Point p1, p2;  } |

Pentru a număra punctele de pe laturi, vom calcula ecuația dreptei și verificăm dacă punctul nu iese din segment și verificăm daca ea satisface ecuația dreptei.

|  |
| --- |
| public bool PeLinie(Point punctLinie1, Point punctLinie2, Point punctLinie3)  {  double a = punctLinie2.y - punctLinie1.y;  double b = punctLinie1.x - punctLinie2.x;  double c = a \* (punctLinie1.x) + b \* (punctLinie1.y);  return a \* punctLinie3.x + b \* punctLinie3.y == c  && punctLinie3.x <= Math.Max(punctLinie1.x, punctLinie2.x)  && punctLinie3.x >= Math.Min(punctLinie1.x,punctLinie2.x)  && punctLinie3.y <= Math.Max(punctLinie1.y, punctLinie2.y)  && punctLinie3.y >= Math.Min(punctLinie1.y, punctLinie2.y);  } |

Pentru a verifica dacă un punct este în interior, facem același lucru doar că verificăm fiecare punct căte laturi atinge: dacă ea atinge un număr par de laturi, atunci punctul nu se află în interior. Acesta ne va da fiecare punct din poligon, dar noi vom scădea numărul de laturi din acesta pentru a obține numărul punctelor din interior.

|  |
| --- |
| public static bool PeLinie(Linie segment, Point punct)  {  //Verificăm dacă punctul este pe segment  if (punct.x <= Math.Max(segment.p1.x, segment.p2.x)  && punct.x >= Math.Min(segment.p1.x, segment.p2.x)  && (punct.y <= Math.Max(segment.p1.y, segment.p2.y)  && punct.y >= Math.Min(segment.p1.y, segment.p2.y)))  return true;  return false;  }  public static int Directie(Point a, Point b, Point c)  {  int val = (int)(b.y-a.y) \* (int)(c.x-b.x) - (int)(b.x-a.x) \* (int)(c.y-b.y);  if (val == 0)  return 0; //coliniar  else if (val < 0)  return 2; //sens invers acelor de ceasornic  return 1; //sensul acelor de ceasornic  }  public static bool Intersectie(Linie linia1, Linie linia2)  {  //patru direcții pentru două linii și puncte a celeilate linii  int dir1 = Directie(linia1.p1, linia1.p2, linia2.p1);  int dir2 = Directie(linia1.p1, linia1.p2, linia2.p2);  int dir3 = Directie(linia2.p1, linia2.p2, linia1.p1);  int dir4 = Directie(linia2.p1, linia2.p2, linia1.p2);  if (dir1 != dir2 && dir3 != dir4)  return true; //cele 2 se intersectează  if (dir1 == 0 && PeLinie(linia1, linia2.p1))  return true; //p2 de pe linia 2 este pe linia 1  if (dir2 == 0 && PeLinie(linia1, linia2.p2))  return true; //p1 de pe linia 2 este pe linia 1  if (dir3 == 0 && PeLinie(linia2, linia1.p1))  return true; //p2 de pe linia 1 este pe linia 2  if (dir4 == 0 && PeLinie(linia2, linia1.p2))  return true; //p1 de pe linia 1 este pe linia 2  return false;  }  public static bool InPolygon(Point[] polygon, int numarLaturi, Point punct)  {    Linie exline;  exline.p1 = punct;  exline.p2.x = int.MaxValue; //pentru a creea linia până la infinit  exline.p2.y = punct.y;  int count = 0;  int i = 0;  do  {  Linie side;  side.p1 = polygon[i];  side.p2 = polygon[(i + 1) % numarLaturi];  if (Intersectie(side, exline))  {  if (Directie(side.p1, punct, side.p2) == 0)  return PeLinie(side, punct);  count++;  }  i = (i + 1) % numarLaturi;  } while (i != 0);  return count % 2 == 1; //returnăm true dacă numărul de segmente  //cu care se intersectează este impar  } |

În final pentru a executa programul, introducem toate acestea în constructor.

|  |
| --- |
| public Pick(string FisierSursa)  {  string aux;  TextReader FisierDate = new StreamReader(FisierSursa);  aux = FisierDate.ReadLine();  DimMat = int.Parse(aux.Split(' ')[0]);  NumarPuncte = int.Parse(aux.Split(' ')[1]);  mat = new int[DimMat, DimMat];  Puncte = new Point[NumarPuncte];  PuncteLatura = NumarPuncte;  for(int i=0; i <NumarPuncte; i++)  {  aux = FisierDate.ReadLine();  Puncte[i] = new Point(int.Parse(aux.Split(' ')[1]), int.Parse(aux.Split(' ')[0]));  mat[(int)Puncte[i].x, (int)Puncte[i].y]=1;  }  for(int i=0; i<DimMat; i++)  {  for(int j=0; j<DimMat; j++)  {  if (mat[j, i] != 1)  {  for (int x = 0; x < NumarPuncte-1; x++)  {  if(PeLinie(Puncte[x], Puncte[x+1], new Point(j,i)))  {  mat[j, i] = 3;  PuncteLatura++;  }  }  if (PeLinie(Puncte[0], Puncte[NumarPuncte-1], new Point(j, i)))  {  mat[j, i] = 3;  PuncteLatura++;  }  if (InPolygon(Puncte, NumarPuncte, new Point(j, i)))  {  PuncteInterior++;  mat[j, i] = 2;  }  }  }  }  PuncteInterior += NumarPuncte-PuncteLatura;  Console.WriteLine(PuncteLatura + " " + PuncteInterior);  Arie = PuncteLatura / 2 + PuncteInterior - 1;  Console.WriteLine("Aria este: " + Arie);  ViewMat();  } |

Clasa va fi instanțiată astfel:

|  |
| --- |
| Pick numeleObiectului = new Pick (@"../../FisierCuPuncte.txt"); |

## [GRAF] BFS, DFS (Csongor)

## [GRAF] Colorarea Hărții (Csongor)

În acest exercițiu, ne propunem să colorăm fiecare punct al unui graf în așa fel încât nici unul dintre punctele conectate să nu aibă aceeași culoare. Putem verifica dacă un graf este bipartit dacă numărul de culori folosite este egal cu 2.

Pentru asta, creăm o clasă numită ColorareaHartii unde vom citi datele din fișier, datele vor fi stocate în următorul fel: Pe prima linie vom avea numărul de puncte, iar pe celelate linii vom avea legăturile dintre puncte una câte una, unde numerotarea punctelor începe de la 0. De exemplu:

5 Numărul de puncte

0 1 punctul 0 este legat de punctul 1

0 2 punctul 0 este legat de punctul 2

1 2 punctul 1 este legat de punctul 2

1 4 punctul 1 este legat de punctul 4

2 4 punctul 2 este legat de punctul 4

4 3 punctul 4 este legat de punctul 3

Pentru rezolvarea problemei vom folosi 3 variabile globale:

|  |
| --- |
| int[] CuloriPuncte; Fiecare punct va avea numărul unei culori.  int NumarPuncte; Numărul total de puncte.  int[,] MatDeAdiacenta; Matricea de adiacență pe care o vom popula imediat. |

Avem nevoie de un construcor care are ca parametrii un string FisierSursa , care conține destinația fișierului sursă, și un int PunctInitial , care conține numărul punctului de la care vom începe colorarea. În construcor vom avea nevoie de un TextReader FisierDate cu care vom citi datele din fișier. Variabila NumarPuncte va fi egală cu prima linie pe care o citim din fișier care ne arată numărul total de puncte, MatDeAdiacenta îi dăm dimensiunea de [NumarPuncte, NumarPuncte] și variebilei CuloriPuncte dimensiunea [NumarPuncte]. Vom parcuge fiecare element al vectorului CuloriPuncte pentru a schimba valoarea fiecărui element în -1, unde valoarea -1 semnifică lipsa unei culori, și începem popularea matricei de adiacență pentru care avem nevoie de un string aux în care vom citi, unul câte unul, liniile din fișier până când nu mai sunt alte linii, și schimbăm valoarea punctelor în MatDeAdiacenta la vloarea 1 care reprezintă o conexiune între două puncte.

|  |
| --- |
| public ColorareaHartii(string FisierSursa, int PunctInitial)  {  TextReader FisierDate = new StreamReader(FisierSursa);  NumarPuncte = int.Parse(FisierDate.ReadLine());  MatDeAdiacenta = new int[NumarPuncte, NumarPuncte];  CuloriPuncte = new int[NumarPuncte];  for (int i = 0; i < NumarPuncte; i++)  {  CuloriPuncte[i] = -1;  }  string aux;  while ((aux = FisierDate.ReadLine()) != null)  {  MatDeAdiacenta[int.Parse(aux.Split(' ')[0]),  int.Parse(aux.Split(' ')[1])] = 1;  MatDeAdiacenta[int.Parse(aux.Split(' ')[1]),  int.Parse(aux.Split(' ')[0])] = 1;  }  Colorare(PunctInitial);  } |

Putem începe colorarea. Pentru asta, am creat subprogramul void Colorare(int punct) care are ca parametru un int punct care ne arată punctul de la care să înceapă colorarea. Pentru acest lucru avem nevoie de List<int> aux = new List<int>(); pe care o vom folosi drept o coadă adică vom adăuga la final si vom șterge de la început, vom avea un bool[] vizitat = new bool[NumarPuncte]; cu care vom ține cont de toate punctele deja vizitate. Vom pune punctul de pornire drept vizitat și îl adăugăm în listă, apoi cât timp numărul elementelor din listă este mai mare decât 0, o să dăm o culoare fiecărui punct folosind metoda CuloareMinima, scoatem elementul din listă și parcurgem fiecare element din MatDeAdiacenta de pe linia punctului curent și dacă aceasta are valoarea 1 (adică este conexiune între cele două puncte) și nu este vizitat atunci vom adăuga punctul în listă (pentru calcule următoare) și îl punem ca vizitat.

|  |
| --- |
| public void Colorare(int punct)  {  List<int> aux = new List<int>();  bool[] vizitat = new bool[NumarPuncte];  vizitat[punct] = true;  aux.Add(punct);    while (aux.Count != 0)  {  int punctCurent = aux[0];  CuloriPuncte[punctCurent] = MinColor(punctCurent);  aux.RemoveAt(0);  for (int i = 0; i < NumarPuncte; i++)  {  if (MatDeAdiacenta[punctCurent, i] == 1 && !vizitat[i])  {  aux.Add(i);  vizitat[i] = true;  }  }  }  } |

Pentru a decide ce culoare să îi dăm punctului curent, vom folosi metoda int CuloareMinima(int PunctCurent) care are ca parametru punctul curent și are ca și tip de return int, care va fi numărul culorii. Pentru asta vom avea nevoie de un bool[] culoriGasite = new bool[NumarPuncte]; în care taote elementele sunt false la început, și vom parcurge fiecare valoare din matricea de adiacență de pe linia punctului curent și dacă aceasta are valoarea 1 (există conexiune între cele două puncte) și punctul are culoare (vloarea ei din vectorul CuloriPuncte nu este egală cu -1), atunci punem valoarea culorii găsite drept true în vectorul culoriGasite. După aceasta, vom parcurge fiecare element și vom returna primul indice care are valoare false.

|  |
| --- |
| public int CuloareMinima(int PunctCurent)  {  bool[] culoriGasite = new bool[NumarPuncte];  for (int i = 0; i < NumarPuncte; i++)  {  if (MatDeAdiacenta[PunctCurent, i] == 1 && CuloriPuncte[i] != -1)  {  culoriGasite[CuloriPuncte[i]] = true;  }  }  for (int i = 0; i < NumarPuncte; i++)  {  if (culoriGasite[i] == false)  {  return i;  }  }  return 1;  } |

Pentru a vizualiza culorile punctelor, vom crea metoda void VizualizareCulori() care nu face altceva decât să parcurgă fiecare element al vectorului CuloriPuncte și să scrie pe ecran valoarea vectorului la indicele I, unde i va fi punctul și CuloriPuncte[i] culoarea punctului.

Clasa va fi instanțiată astfel:

|  |
| --- |
| ColorareaHartii numeleObiectului = new ColorareaHartii(@"../../FisierCuPuncte.txt", punctulDeUndeIncepeColorarea); |

Apoi, pentru a vizualiza curlorile punctelor, vom apela metoda VizualizareCulori astfel:   
numeleObiectului.VizualizareCulori();

## [GRAF] Algoritmul lui Dijkstra (Csongor)

În acest exercițiu, ne propunem să determinăm cel mai ieftin drum de la un punt la toate celelalte.

Pentru asta, creăm clasa Dijsktra unde vom citi datele din fișier, acestea fiind stocate în următorul fel: pe prima linie vom avea numărul de puncte, iar pe celelalte linii vom avea legăturile dintre puncte una căte una și costul lor de la un punt la celălalt, unde numerotarea punctelor începe de la 0.

De exemplu:

9 Numărul punctelor

0 1 4 Punctul 0 este legat de punctul 1 și drumul costă 4.

0 7 8 Punctul 0 este legat de punctul 7 și drumul costă 8.

1 7 11 Punctul 1 este legat de punctul 7 și drumul costă 11.

1 2 8 Punctul 1 este legat de punctul 2 și drumul costă 8.

2 8 2 Punctul 2 este legat de punctul 8 și drumul costă 2.

2 3 7 Punctul 2 este legat de punctul 3 și drumul costă 7.

2 5 4 Punctul 2 este legat de punctul 5 și drumul costă 4.

3 4 9 Punctul 3 este legat de punctul 4 și drumul costă 9.

3 5 14 Punctul 3 este legat de punctul 5 și drumul costă 14.

4 5 10 Punctul 4 este legat de punctul 5 și drumul costă 10.

5 6 2 Punctul 5 este legat de punctul 6 și drumul costă 2.

6 8 6 Punctul 6 este legat de punctul 8 și drumul costă 6.

6 7 1 Punctul 6 este legat de punctul 7 și drumul costă 1.

7 8 7 Punctul 7 este legat de punctul 8 și drumul costă 7.

Pentru rezolvarea problemei, vom folosi 3 variabile globale:

|  |
| --- |
| int NumarPuncte; Numărul total de puncte.  int[,] MatDeAdiacenta; Matricea de adiacență pe care o vom popula imediat.  int[] DistanteMinime; Distanța minima găsită până acuma la toate punctele. |

Avem nevoie de un constructor care are ca parametrii un string FisierSursa , care conține destinația fișierului sursă, și un int punctPornire , punctul de la care vom afla distanțele. În constructor, vom avea nevoie de TextReader FisierDate cu care vom citi datele din fișier. Variabila NumarPuncte va fi egală cu prima linie pe care o citim din fișier care ne arată numărul total de puncte, MatDeAdiacenta îi dam dimensiunea de [NumarPuncte, NumarPuncte] și începem popularea matricei de adiacență, pentru care avem nevoie de un string aux în care vom citi unul câte unul liniile din fișier, până când nu mai sunt alte linii, și schimbăm valoarea punctelor în MatDeAdiacenta la vloarea pe care o citim din fișier, care reprezintă o conexiune între două puncte și costul lor de la unul la celălalt. Fiecare index al vectorului DistanteMinime va avea valoarea int.MaxValue înafară de punctul de pornire, unde va fi 0.

|  |
| --- |
| public Dijkstra(string FisierSursa, int punctPornire)  {  TextReader FisierDate = new StreamReader(FisierSursa);  NumarPuncte=int.Parse(FisierDate.ReadLine());  MatDeAdiacenta = new int[NumarPuncte, NumarPuncte];  DistanteMinime = new int[NumarPuncte];  for(int i=0;i<NumarPuncte;i++)  {  DistanteMinime[i] = int.MaxValue;  }  DistanteMinime[punctPornire] = 0;  string aux;  while((aux=FisierDate.ReadLine())!=null)  {  MatDeAdiacenta[int.Parse(aux.Split(' ')[0]),  int.Parse(aux.Split(' ')[1])] = int.Parse(aux.Split(' ')[2]);  MatDeAdiacenta[int.Parse(aux.Split(' ')[1]),  int.Parse(aux.Split(' ')[0])] = int.Parse(aux.Split(' ')[2]);  }  CalcCheapestRoute(punctPornire);  VizualizareDistante();  } |

Pentru calcularea distanțelor, avem nevoie de o funcție void CalculDrumMinim(int punctInitial) care are ca parametru punctul de la care vrem să calculăm distanțele. O să avem nevoie de o listă List<int> aux = new List<int>(); adăugăm punctul inițial în listă, creăm un vector bool[] vizitat = new bool[NumarPuncte]; punem valoarea vectorului la indicele punctului inițial egal cu true și începem parcurgerea în lățime (BFS), cât timp numărul elementelor din listă este mai mare decât 0, scotem elemntul din listă și parcurgem fiecare element din MatDeAdiacenta de pe linia punctului curent, iar dacă aceasta are valoare diferită de 0 (adică este conexiune între cele două puncte) și nu este vizitat, atunci vom adăuga punctul în listă (pentru calcule următoare) și îl punem ca vizitat și ne uităm dacă distanța minimă până la punctul curent plus distanța de la punctul curent până la punctul i este mai mic decât distanța până la punctul I, atunci distanta minimă până la punctul i va fi egală cu distanța minimă până la punctul curent plus distanța de la punctul curent la punctul i. Și dacă distanța minimă până la punctul i plus distnața de la punctul curent până la i este mai mică decat distanța minimă până la punctul curent, atunci distanța minimă până la punctul curent va fi egală cu distanța minimă până la punctul i plus distanța de la punctul curent la i.

|  |
| --- |
| public void CalculDrumMinim(int punctInitial)  {  List<int> aux = new List<int>();  aux.Add(punctInitial);  bool[] vizitat = new bool[NumarPuncte];  vizitat[punctInitial] = true;  while(aux.Count!=0)  {  int punctCurent = aux[0];  aux.RemoveAt(0);  for(int i=0;i<NumarPuncte;i++)  {  if(MatDeAdiacenta[punctCurent,i]!=0)  {  if(!vizitat[i])  {  aux.Add(i);  vizitat[i] = true;  }  if(DistanteMinime[punctCurent] + MatDeAdiacenta[punctCurent,i]  <DistanteMinime[i])  {  DistanteMinime[i] =  DistanteMinime[punctCurent] + MatDeAdiacenta[punctCurent, i];  }  if(DistanteMinime[i] + MatDeAdiacenta[punctCurent,i]  <DistanteMinime[punctCurent])  {  DistanteMinime[punctCurent] =  DistanteMinime[i] + MatDeAdiacenta[punctCurent, i];  }  }  }  }  } |

Clasa va fi instanțiată astfel:

|  |
| --- |
| Dijsktra numeleObiectului = new Dijsktra(@"../../FisierCuPuncte.txt", punctulDeLaCareVremCosturile); |

## [GRAF] Graf Eulerian (Csongor)

În acest exercițiu, ne propunem să căutăm un drum prin care parcurgem fiecare legătura o singură dată.

Pentru asta, creăm clasa Euler unde vom citi datele din fișier, acestea fiind stocate în următorul fel: pe prima linie vom avea numărul de puncte, iar pe celelalte linii vom avea legăturile dintre puncte una câte una, unde numerotarea punctelor începe de la 0. De exemplu:

4 Numărul de puncte

0 1 Punctul 0 este legat de punctul 1

0 1 Punctul 0 este legat de punctul 1

1 2 Punctul 1 este legat de punctul 2

1 2 Punctul 1 este legat de punctul 2

2 3 Punctul 2 este legat de punctul 3

3 1 Punctul 3 este legat de punctul 1

0 3 Punctul 0 este legat de punctul 3

În acest exercițiu, putem avea mai multe legături între două puncte.

Pentru rezolvarea problemei, vom folosi 2 variabile globale:

|  |
| --- |
| public int [,] MatDeAdiacenta; Matricea de adiacență pe care o vom popula imediat. public int NumarPuncte; Numărul total de puncte. |

Avem nevoie de un constructor care are ca parametru un string FisierSursa, care conține destinația fișierului sursă. În constructor, vom avea nevoie de TextReader FisierDate cu care vom citi datele din fișier. Variabila NumarPuncte va fi egală cu prima linie pe care o citim din fișier, care ne arată numărul total de puncte, MatDeAdiacenta îi dăm dimensiunea de [NumarPuncte, NumarPuncte] și începem popularea matricii de adiacență, pentru care avem nevoie de un string aux în care vom citi unul căte unul liniile din fișier, până când nu mai sunt alte linii, și creștem valoarea punctelor în MatDeAdiacenta, deci unde valoarea este mai mare decât 0 avem conexiune.

|  |
| --- |
| public Euler(string FisierSursa)  {  TextReader FisierDate = new StreamReader(FisierSursa);  NumarPuncte = int.Parse(FisierDate.ReadLine());  MatDeAdiacenta = new int[NumarPuncte, NumarPuncte];  string aux;  while((aux=FisierDate.ReadLine())!=null)  {  MatDeAdiacenta[int.Parse(aux.Split(' ')[0]),   int.Parse(aux.Split(' ')[1])] ++;  MatDeAdiacenta[int.Parse(aux.Split(' ')[1]),   int.Parse(aux.Split(' ')[0])] ++;  }  ViewMat();  VerificaEulerian();  } |

Ne vom folosi de proprietățile grafurilor. Algoritmul de determinare sună astfel: Dacă fiecare punct are un număr par de conexiuni, atunci graful are un ciclu eulerian, dacă două dintre puncte au conexiuni de număr impar, atunci graful are un lanț eulerian, iar dacă graful are mai mult de două puncte cu conexiuni impare atunci graful nu conține nici lanț, nici ciclu eulerian. Pentru rezolvarea problemei, vom folosi funcția void VerificaEulerian(). Vom avea nevoie de două variabile, unul pentru a număra punctele cu conexiuni pare int pare și unul pentru a număra punctele cu conexiuni impare int impare, fiecare egal cu 0. Verificăm numărul de conexiuni între puncte începând de la punctul 0 folosind un alt int numarConexiuni și folosindu-ne de MatDeAdiacenta. Dacă un punct are mai mult de 0 conexiuni și numărul de conexiuni este par, atunci creștem valoarea variabilei pare, dacă un punct are mai mult de 0 conexiuni și numărul de conexiuni este impar, atunci creștem valoarea variabilei impare. La final, verificăm dacă numărul punctelor impare este 0, dacă numărul punctelor pare este mai mare decât 0 atunci graful are un ciclu, dacă numărul elementelor impare este 2 atunci graful conține lanț, altfel graful nu conține nimic.

|  |
| --- |
| public void VerificaEulerian()  {  int pare = 0, impare = 0;  for(int i=0;i<NumarPuncte;i++)  {  int numarConexiuni = 0;  for(int j=0;j<NumarPuncte;j++)  {  if (MatDeAdiacenta[i, j] == 1) { numarConexiuni++; }  }  if(numarConexiuni!=0&&numarConexiuni%2==0){pare++;}  if(numarConexiuni!=0&&numarConexiuni%2!=0){impare++;}  }  if(impare==0&&pare>0)  {  Console.WriteLine("Graful contine Ciclu");  }  else if(impare==2&&pare>0)  {  Console.WriteLine("Graful contine Lant");  }  else  {  Console.WriteLine("Graful nu contine nimic");  }  } |

Clasa va fi instanțiată astfel:

|  |
| --- |
| Euler numeleObiectului = new Euler(@"../../FisierCuPuncte.txt"); |

## [GRAF] Graf Hamiltonian (Csongor)

În acest exercitiu, ne propunem să căutăm un drum astfel încât să parcurgem fiecare punct o singură dată.

Pentru asta, creăm clasa Hamilton unde vom citi datele din fișier, acestea fiind stocate în următorul fel: pe prima linie vom avea numărul de puncte și pe celelalte linii vom avea legăturile dintre puncte una câte una, unde numerotarea punctelor începe de la 0. De exemplu:

5 Numărul de puncte

0 1 Punctul 0 este legat de punctul 1

0 3 Punctul 0 este legat de punctul 3

1 2 Punctul 1 este legat de punctul 2

1 3 Punctul 1 este legat de punctul 3

1 4 Punctul 1 este legat de punctul 4

2 4 Punctul 2 este legat de punctul 4

3 4 Punctul 3 este legat de punctul 4

Pentru rezolvarea problemei, vom folosi 2 variabile globale:

|  |
| --- |
| public int [,] MatDeAdiacenta; Matricea de adiacență pe care o vom popula imediat. public int NumarPuncte; Numărul total de puncte. |

Avem nevoie de un constructor care are ca parametru un string FisierSursa, care conține destinația fișierului sursă. În constructor vom avea nevoie de TextReader FisierDate cu care vom citi datele din fișier. Variabila NumarPuncte va fi egală cu prima linie pe care o citim din fișier, care ne arată numărul total de puncte, MatDeAdiacenta îi dăm dimensiunea de [NumarPuncte, NumarPuncte] și începem popularea matricei de adiacență, pentru care avem nevoie de un string aux în care vom citi unul câte unul liniile din fișier, până când nu mai sunt alte linii și schimbăm valoarea punctelor în MatDeAdiacenta la vloarea 1, care reprezintă o conexiune între două puncte, după care creăm o listă List<int> Puncte = new List<int>(); în care vom stoca drumurile găsite și un int nr = 0; care numără câte puncte sunt deja în drum și un vector bool[] Vizitat = new bool[NumarPuncte]; în care vom nota punctle deja în drum cu true. Pornim o funcție recursivă de la fiecare punct pentru a găsi fiecare lanț și ciclu hamiltonian. Cănd începem algoritmul de la fiecare punct, punem variabila nr să fie 1 pentru că avem deja un punct în drum, vectorul Vizitat va avea la indicele i valoarea true, punem punctul în listă și pornim funcția recursivă BackTrack cu parametrii (i, Vizitat,nr, Puncte,i) și la ieșire punem valoarea lui Vizitat la indicele i false și scoatem punctul din listă pentru ca la următoarea iterație totul să înceapă de la zero.

|  |
| --- |
| public Hamilton(string FisierSursa)  {  TextReader FisierDate = new StreamReader(FisierSursa);  NumarPuncte = int.Parse(FisierDate.ReadLine());  MatDeAdiacenta = new int[NumarPuncte, NumarPuncte];  string aux;  while((aux=FisierDate.ReadLine())!=null)  {  MatDeAdiacenta[int.Parse(aux.Split(' ')[0]),   int.Parse(aux.Split(' ')[1])] = 1;  MatDeAdiacenta[int.Parse(aux.Split(' ')[1]),   int.Parse(aux.Split(' ')[0])] = 1;  }  bool[] Vizitat = new bool[NumarPuncte];  int nr = 0;  List<int> Puncte = new List<int>();  for (int i=0;i<NumarPuncte;i++)  {  nr = 1;  Vizitat[i] = true;  Puncte.Add(i);  BackTrack(i, Vizitat,nr, Puncte,i);  Vizitat[i] = false;  Puncte.RemoveAt(Puncte.Count - 1);  }  } |

Pentru funcția recursivă void BackTrack, avem nevoie de cinci parametrii, un int punctCurent care ne arată punctul la care suntem momentan, bool[]vizitat pentru a vedea la care puncte am fost deja, int numarIterare pentru a vedea câte puncte sunt deja în listă, List<int> Puncte pentru a vedea drumul după ordinea punctelor și un int punctInitial cu care vom decide dacă un drum este lanț sau ciclu. Dacă numarIterare este egal cu NumarPuncte, atunci înseamnă că am găsit un drum hamiltonian, iar dacă ultimul punct adăugat are conexiune la punctInitial, înseamnă că acest drum este defapt un ciclu. Dacă numarIterare nu este egal cu NumarPuncte, atunci vom parcurge linia matricei de adiacență la punctCurent și daca valoarea ei este 1 și acesta nu fost vizitat, îl punem ca vizitat, îl adăugăm în listă și reîncepem funcția cu parametrii i (noul punct curent), vizitat (noul vector pentru a evita punctele deja vizitate) , numarIterare+1 (pentru că am găsit un punct nou), Puncte (noua listă de puncte ) și punctInitial (care rămâne mereu același căt timp suntem în subprogram), la revenire punem vizitat la indicele i egal cu false și scoatem ultimul element din listă.

|  |
| --- |
| public void BackTrack(int punctCurent, bool[]vizitat, int numarIterare, List<int> Puncte, int punctInitial)  {  if (numarIterare == NumarPuncte)  {  Console.WriteLine("Drum hamiltonian gasit");  for(int i=0;i<Puncte.Count;i++)  {  Console.Write(Puncte[i]+" ");  }  if (MatDeAdiacenta[punctCurent, punctInitial] == 1)  {  Console.Write("Este Ciclu!");  }  else  {  Console.Write("Este Lant!");  }  Console.WriteLine();  }  else  {  for (int i = 0; i < NumarPuncte; i++)  {  if (MatDeAdiacenta[punctCurent, i] == 1&&!vizitat[i])  {  vizitat[i] = true;  Puncte.Add(i);  BackTrack(i, vizitat,numarIterare+1,Puncte,punctInitial);  vizitat[i] = false;  Puncte.RemoveAt(Puncte.Count-1);  }  }  }  } |

Clasa va fi instanțiată astfel:

|  |
| --- |
| Hamilton numeleObiectului = new Hamilton(@"../../FisierCuPuncte.txt"); |

## [GRAF] Algoritmul lui Kruskall (Csongor)

În acest exercițiu, ne propunem să determinăm arborele de cost minim dintr-un graf.

Pentru ast,a creăm clasa Kruskall unde vom citi datele din fișier, acestea fiind stocate în următorul fel: pe prima linie vom avea numărul de puncte, iar pe celelalte linii vom avea legăturile dintre puncte una câte una și costul lor de la un punct la celălalt, unde numerotarea punctelor începe de la 1. De exemplu:

7 Numărul punctelor

1 2 2 Punctul 1 este legat de punctul 2 și drumul costă 2.

1 3 3 Punctul 1 este legat de punctul 3 și drumul costă 3.

1 4 3 Punctul 1 este legat de punctul 4 și drumul costă 3.

2 3 4 Punctul 2 este legat de punctul 3 și drumul costă 4.

2 5 3 Punctul 2 este legat de punctul 5 și drumul costă 3.

3 4 5 Punctul 3 este legat de punctul 4 și drumul costă 5.

3 5 1 Punctul 3 este legat de punctul 5 și drumul costă 1.

4 6 7 Punctul 4 este legat de punctul 6 și drumul costă 7.

5 6 8 Punctul 5 este legat de punctul 6 și drumul costă 8.

6 7 9 Punctul 6 este legat de punctul 7 și drumul costă 9.

Pentru rezolvarea problemei, vom folosi 3 variabile globale:

|  |
| --- |
| int NumarPuncte; Numărul total de puncte.  int[,] MatDeAdiacenta; Matricea de adiacență pe care o vom popula imediat.  List<Edge> muchii= new List<Edge>(); O listă de muchii unde vom stoca muchiile citite din fișier. |

Vom avea nevoie de o clasă Edge care va stoca punctul de început, punctul destinație și costul pentru a ajunge de la unul la celălalt.

|  |
| --- |
| public class Edge  {  public int start, final, valoare;  public Edge(int start,int final,int valoare)  {  this.start = start;  this.final = final;  this.valoare = valoare;  }  } |

Avem nevoie de un constructor care are ca parametru un string FisierSursa, care conține destinația fișierului sursă. În constructor, vom avea nevoie de TextReader FisierDate, cu care vom citi datele din fișier. Variabila NumarPuncte va fi egală cu prima linie pe care o citim din fișier, care ne arată numărul total de puncte, MatDeAdiacenta îi dăm dimensiunea de [NumarPuncte, NumarPuncte] și începem popularea matricei de adiacență, pentru care avem nevoie de un string aux în care vom citi unul câte unul liniile din fișier, până când nu mai sunt alte linii, și schimbăm valoarea punctelor în MatDeAdiacenta în valoarea 1, ceea ce semnifică conexiunea între două puncte, iar în acest timp adăugăm fiecare muchie în lista de muchii. După acesta sortăm muchiile dupa valoarea lor în ordine crescătoare și calculăm arborele de cost minim.

|  |
| --- |
| public Kruskall(string FisierSursa)  {  TextReader FisierDate = new StreamReader(FisierSursa);  NumarPuncte = int.Parse(FisierDate.ReadLine());  MatDeAdiacenta = new int[NumarPuncte, NumarPuncte];  string buffer;  while((buffer=FisierDate.ReadLine())!=null)  {  int s = int.Parse(buffer.Split(' ')[0])-1;  int e = int.Parse(buffer.Split(' ')[1])-1;  int v = int.Parse(buffer.Split(' ')[2]);  MatDeAdiacenta[s, e] = 1;  MatDeAdiacenta[e, s] = 1;  muchii.Add(new Edge(s, e, v));  }  SortareMuchii();  DetermArboreMinim();  } |

Pentru sortarea muchiilor, folosim un simplu delegat pe metoda Sort din clasa List.

|  |
| --- |
| public void SortareMuchii()  {  ed.Sort(delegate (Edge a, Edge b)  {  return a.valoare.CompareTo(b.valoare);  });  } |

Pentru determinarea arborelui de cost minim, vom folosi funcția DetermArboreMinim în care vom avea nevoie de o listă de muchii arbore și adăugăm primul element din muchii în acesta, după care adăugăm fiecare muchie din lista muchii unul câte unul în lista arbore. Dacă muchia adăugată cauzează garful să aibă ciclu, atunci scoatem muchia respectivă din arbore.

|  |
| --- |
| public void DetermArboreMinim()  {  List<Edge> arbore = new List<Edge>();  arbore.Add(muchii[0]);  for(int i=1;i<muchii.Count;i++)  {  arbore.Add(muchii[i]);  int [,] mataux = new int[NumarPuncte,NumarPuncte];  for(int j=0;j<arbore.Count;j++)  {  mataux[arbore[j].final, arbore[j].start] = 1;  mataux[arbore[j].start, arbore[j].final] = 1;  }  if(ECiclu(mataux,muchii[i].start))  {  arbore.RemoveAt(arbore.Count-1);  }  }  for(int i=0; i<arbore.Count;i++)  {  Console.WriteLine(arbore[i].start + " "   + arbore[i].final + " " + arbore[i].valoare);  }  } |

Pentru a determina dacă un graf conține ciclu, vom folosi funcția ECiclu care are ca parametrii o matrice int[,] MatAux, care este matricea de adiacență a arborelui, și un int start, care ne zice de unde să incepem parcurgerea. Avem nevoie de List<int> aux și de bool[] vizitat = new bool[NumarPuncte]. Adăugăm punctul de pornire în listă și, cât timp lista are elemente, scoatem primul element din listă, punem punctul drept vizitat și parcurgem MatAux pe linia  
punctului curent. Dacă valoarea acesteia este 1 și nu este vizitat, verificăm dacă ea este deja în listă, daca da returnăm valoarea true (adică arborele are ciclu), altfel adăugăm punctul în listă. Dacă am reușit să golim lista, înseamnă ca ea nu are ciclu și returnăm false.

|  |
| --- |
| public bool ECiclu(int[,] MatAux, int start)  {  List<int> aux = new List<int>();  bool[] vizitat = new bool[NumarPuncte];  aux.Add(start);  while (aux.Count != 0)  {  int curent = aux[0];  vizitat[curent] = true;  aux.RemoveAt(0);  for (int i = 0; i < NumarPuncte; i++)  {  if (!vizitat[i] && MatAux[curent, i] == 1)  {  if (aux.Contains(i))  {  return true;  }  else  {  aux.Add(i);  }  }  }  }  return false;  } |

Clasa va fi instanțiată astfel:

|  |
| --- |
| Kruskall numeleObiectului = new Kruskall(@"../../FisierCuPuncte.txt"); |

## [ARB]

[AlgE]Fracții ireductibile

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | O proprietate interesanta a fractiilor ireductibile este aceea ca oricare dintre ele se poate obtine dupa urmatoarele reguli:  1) pe primul nivel se afla fractia 1/1  2) pe al 2-lea nivel se afla in stanga fractia 1/2 si in dreapta fractia 2/1  3) pe fiecare nivel k, sub fractia i/j de pe nivelul k-1 se plaseaza fractia i/(i+j) in stanga si fractia (i+j)/j in dreapta.  Primele 3 niveluri astfel obtinute sunt:  Nivelul 1: 1/1  Nivelul 2: 1/2 2/1  Nivelul 3: 1/3 3/2 2/3 3/1  Dandu-se o fractie oarecare prin numaratorul n si numitorul m (n,m intre 1 si 2000000000), determinati pe ce nivel se afla fractia data sau fractia ireductibila echivalenta cu fractia data.  Exemple: Fractia 12/8 se afla pe nivelul 3 (este echivalenta cu fractia ireductibila 3/2).  Fractia 13/8 se afla pe nivelul 6 | | |

|  |
| --- |
| #include<iostream>  using namespace std;  int cmmdc(int a, int b)  {  while(b)  {  int r=a%b;  a=b;  b=r;  }  return a;  }  int main()  { int n,m,x,y,c,k=1;  cin>>n>>m;  c=cmmdc(n,m);  n=n/c;  m=m/c;  while(n!=1 || m!=1)  {  if(n>m)  {  y=m;  x=n-y;  }  else  {  x=n;  y=m-x;  }  k++;  n=x;  m=y;  if(n==1) { k=k+m-1; m=1;}  else if(m==1) { k=k+n-1; n=1; }  }  cout<<k;  return 0;  } |

[REC/DC]MinMAX

|  |
| --- |
| Scrieti o functie minmax care sa determine elementul minim si elementul maxim dintr-un tablou cu elemente intregi folosind metoda divide et impera. |

|  |
| --- |
| void minmax(int v[], int s, int d, int &minn, int &maxx)  {  if(s==d) minn=maxx=v[s];  else  {  int a,b,x,y;  minmax(v,s,(s+d)/2,a,x);  minmax(v,(s+d)/2+1,d,b,y);  if(a<b) minn=a;  else minn=b;  if(x>y) maxx=x;  else maxx=y;  }  } |

# Bibliography

1. **Ivan, Moscovich.** *A doua mare carte a jocurilor minții.* București : Litera, 2017.

2. [Online] [Cited: 2 10, 2018.] https://stackoverflow.com/questions/24874495/largest-data-type-to-store-numbers.

3. [Online] [Cited: 2 10, 2018.] https://en.wikipedia.org/wiki/Binomial\_theorem.

4. [Online] [Cited: 2 10, 2108.] https://en.wikipedia.org/wiki/Wheat\_and\_chessboard\_problem.