Tema 6 - Tabele de repartizare

1. **Implementare tabelă de repartizare**. Construiţi o clasă HashTable (sau HashMap) potrivită, care să includă operaţiile de inserţie şi căutare. Elementele stocate vor fi de tip (cheie, valoare). Folosiţi pair din stl. Rezolvarea coliziunilor se va realiza prin liste înlănţuite sau prin dublă repartizare. În cazul dublei repartizări, dacă factorul de încarcare al tabelei depăşeşte 0.7, se cere redimensionarea tabelei (aproximativ dublul dimensiunii iniţiale) şi redistribuirea elementelor în noua tabelă (*rehashing*). În funcţia main citiţi dintr-un fişier n elemente de tip pereche (cheie-valoare) (n>20), repartizaţi elementele în tabelă, apoi permiteţi căutarea elementelor. (3p).

   Punctaj suplimentar:

   - template <cheie, valoare> (0.5p).
   - supraîncărcarea operatorului [ ] pentru acces la elemente (1.5p).
   - utilizarea unei funcţii de repartizare pentru chei de tip şir de catactere (0.5p)

2. Se consideră un vector de caractere (citit din fişier). Să se verifice, daca este posibilă formarea unui palindrom cu toate caracterele din şir. Implementaţi eficient folosind **unordered_map** din stl. (1p)

3. Să se afişeze toate cvadruplurile de numere naturale (a, b, c, d) cu fiecare număr mai mic decât 10.000, pentru care $a^2 + b^2 = c^2 + d^2$. Rezolvaţi în complexitate cât mai bună folosind **unordered_map** (2p)

4. Se consideră o listă înlănţuită L de numere reale. Să se elimine în mod eficient valorile care se repetă, folosind **unordered_set**. (1p)

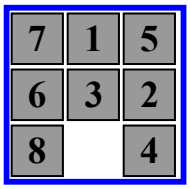
5. **Happy number**. Un *număr fericit* este un număr definit astfel:

   - se porneşte de la k, se înlocuieşte k cu suma pătratelor cifrelor sale.
   - se repetă procesul până când numărul devine 1 sau ciclează la infinit (suma nu va deveni 1 niciodată).
   - dacă procesul se termină cu 1, numărul este fericit.

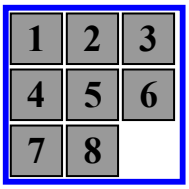
Scrieţi o funcţie care returnează *true* dacă k este un număr fericit şi *false* altfel. (1.5p)

6. Se consideră un şir de cuvinte citite dintr-un fişier. Scrieţi o funcţie, care să grupeze anagramele. Se consideră anagramă un cuvânt obţinut prin rearanjarea literelor altui cuvânt. Folosiţi structurile de date din stl învăţate, aşa încât să obţineţi eficienţa cea mai bună. (1-3p) (punctaj în funcţie de rezolvare)
   Exemplu: Se consideră cuvintele {car, rac, cos, amin, arc, soc, polca, lac, cal, pocal,mina, copil, anim}. Atunci se vor grupa: {car, rac, arc}, {cos, soc}, {amin, mina, anim}, {lac, cal}, {pocal, polca}, {copil}.

7. **Rezolvare puzzle**: Se consideră o jocul următor. Se consideră 8 plăcuţe pătrate numerotate de la 1 la 8, plasate într-o ramă pătrată de dimensiune 3×3. O poziţie este liberă. Orice plăcuţă vecină cu poziţia liberă poate fi glisată pe această poziţie. Se cere şirul de configuraţii (afişat), care duc la aranjarea numerelor în ordine crescătoare, cu poziţia liberă în colţul dreapta-jos al cadrului. Un exemplu este prezentat în figură. (3p)



Configurație inițială



Configurație finală

   Folosiţi algoritmul A*, precum şi structurile de date din stl potrivite (priority_queue, unordered_map, vector etc.)
   Costul unei configuraţii = numărul de mutări care s-a efectuat din configuraţia iniţială până la aceasta.
   Euristica 1: - numărul de plăcuţe care încă nu se află pe poziţia potrivită.
   Păstraţi configuraţiile deja generate împreună cu o legătură (definită în mod potrivit) către configuraţia din care s-a obţinut (parintele) într-un unordered_map, pentru a putea apoi genera şirul de configuraţii de la cea iniţială la soluţie. Pentru exemplul din fighură $h_1 = 9$
   Euristica 2: suma distanţelor de la fiecare piesă la poziţia corectă (city-block distance). Pentru exemplul din figură avem $h_2 = 3+1+2+2+2+2+1+3 = 18$. Euristica 2 este mai eficientă decât 1, pentru ca se apropie mai bine de costul real.
   Indicaţie - pentru simplificare, consideraţi la fiecare pas, că se mută poziţia liberă sus, jos, la stânga sau la dreapta.