Ion Melania-Victoria, 341

Tema 1 laborator SDA

Problema Cifre 4 (20p)

Procesul gândirii:

1) Inițializare: Creăm o coadă pentru BFS și adăugăm cifrele inițiale (2, 3, 5,7). Avem si un vector boolean (inițializat cu False) care urmărește resturile pe care le-am vizitat deja. Acesta este folosit pentru a evita procesarea repetată a acelorași resturi.
2) BFS: Extragem fiecare element din coadă și verificăm dacă am vizitat deja acest rest. Daca nu, verificăm dacă restul împărțirii numarului la P este N, iar in caz afirmativ avem solutia, altfel adăugăm la coadă noi numere generate prin înmulțirea numărului curent cu 10 și adăugarea fiecărei cifre din lista (pentru a obține toate combinațiile posibile).

Repetare: Repetăm pasul 2 până când găsim o soluție sau coada devine goală (ceea ce înseamnă că nu există soluție si afisam -1).

Complexitate:

1) Complexitatea timpului este O(b^d), unde b este factorul de ramificare (în cazul nostru, 4 - numărul de cifre disponibile) și d este adâncimea soluției.
2) Complexitatea spațiului este, de asemenea, O(b^d), deoarece în cel mai rău caz, coada și vectorul de vizitate pot stoca un număr mare de elemente.

Motivația Alegerii Structurii de Date - Coada:

Am ales să folosesc o structură de date de tip coadă din următoarele motive:

1) BFS Implementare: BFS se bazează pe o coadă pentru a explora nivel cu nivel. Coada asigură că procesăm numerele în ordinea în care le generăm, garantând că primul număr care îndeplinește condiția este și cel mai mic.
2) Simplu și eficient: Coada este simplă de implementat și eficientă în termeni de performanță pentru acest tip de algoritm.

Problema Nrpits (20p)

Procesul gândirii:

1) Inițializare: Se utilizează o stivă pentru a păstra numerele secvenței si un număr foarte mare, max, este plasat în stivă la început pentru a evita erori legate de verificarea elementului de vârf al stivei.

2) Pentru fiecare număr din secventa (retinuta in vectorul “sir”) il vom compara cu elementul de pe vârful stivei astfel:

   -Dacă acesta este mai mare decât elementul din vârf, se elimină elementele din stivă până când se găsește un element mai mare decât “a” sau până când stiva are mai puțin de două elemente.

   În acest proces, de fiecare dată când se elimină un element din stivă (cu excepția cazului când stiva ajunge la mai puțin de două elemente), se consideră că s-a găsit un "pit" și se incrementează contorul count_pits.
3) Logica "Pit (gropilor) ":

   -Un "pit" este detectat când un element mai mic (a) este plasat între două elemente mai mari (elementele din stivă).

   -Astfel, când se elimină elemente din stivă (care sunt mai mici decât a), înseamnă că acele elemente erau "pits" pentru că au fost mai mici decât elementele anterioare și decât a.

Complexitate:

1) Complexitatea timpului pentru acest algoritm este O(n), unde n este numărul de elemente în secvență. Fiecare element este procesat o singură dată, iar operațiunile de adăugare și eliminare din stivă sunt realizate în timp constant.
2) Complexitatea spațiului este, de asemenea, O(n), deoarece în cel mai rău caz, toate elementele pot fi stocate în stivă înainte de a fi eliminate.

Motivația Alegerii Structurii de Date - Stiva:

1) Accesul la Elementul Anterior: Stiva este aleasă pentru a reține elementele într-o manieră care permite verificarea ușoară a relației dintre un element și predecesorii săi imediați. Aceasta permite adăugarea și eliminarea elementelor în timp constant (O(1)), ceea ce este crucial pentru eficiența acestui algoritm.
2) Simplificarea Logicii: Stiva ajută la simplificarea logicii de determinare a "pit-urilor". Când un element nou este mai mare decât vârful stivei, eliminăm elemente din stivă, însemnând că acele elemente erau "pits".

Problema Lastk (20p)

Procesul gândirii:

Se inițializează un min-heap cu primul element A și se iterează prin generarea numerelor de la 2 la n folosind formula dată. Pentru fiecare număr generat:

- Dacă heap-ul are mai puțin de k elemente, se adaugă noul număr.
- Dacă noul număr este mai mare decât cel mai mic număr din heap (rădăcina heap-ului), se înlocuiește acest minim cu noul număr.

La final, se extrag numerele din heap, acestea fiind cele mai mari k numere din șir.

Complexitate:

1) Complexitatea de timp: O(n log k) - fiecare inserție sau eliminare din heap necesită O(log k), iar acest lucru se face pentru fiecare dintre cele n elemente.
2) Complexitatea spațiului: O(k) - heap-ul stochează un maxim de k elemente în orice moment.

Motivația Alegerii Structurii de Date - Min-Heap:

1) Eficiență în Timp: Întrucât ne dorim să determinăm cele mai mari k numere din șir, min-heap-ul este ideal deoarece ne oferă acces imediat la cel mai mic element (rădăcina heap-ului). Aceasta este o caracteristică importantă pentru că, în timpul generării șirului, putem compara fiecare element nou cu cel mai mic element din heap. Dacă noul element este mai mare, îl înlocuim, asigurându-ne astfel că menținem întotdeauna cele mai mari k numere.

2) Menținerea Ordinii după Inserții/Eliminări: Heap-ul are proprietatea de reordonare după fiecare inserție sau eliminare, menținându-și structura. Aceasta înseamnă că, după fiecare operație, putem fi siguri că rădăcina heap-ului va fi întotdeauna cel mai mic element dintre cele mai mari k numere. Acest lucru face min-heap-ul extrem de eficient pentru scopul nostru de a filtra și păstra doar elementele relevante.