Resurse examen SD

Seria 15

June 25, 2025

Contents

1 Sortări

Algoritmii de sortare pot fi clasificați după complexitate, spațiu, stabilitate și dacă se bazează pe comparații:

1.1 Clasificare

- **Elementari:** Insertion, Selection, Bubble
- **Divide et Impera:** Merge Sort, Quick Sort
- **Heap-based:** Heap Sort
- **Non-comparative:** Counting Sort, Radix Sort, Bucket Sort

1.2 Stabilitate

Un algoritm de sortare este *stabil* dacă menține ordinea relativă a elementelor egale. Acest lucru este important când sortăm obiecte cu multiple chei.

1.3 Tabel de complexități

Algorithm	**Time Complexity**			**Space (worst)**
	Best	**Average**	**Worst**	
Quicksort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Mergesort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log n)$	$\Theta\big(n(\log n)^2\big)$	$O\big((n \log n)^2\big)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n+k)$
Radix Sort	$\Omega(n\,k)$	$\Theta(n\,k)$	$O(n\,k)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(n+k)$
Cubesort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

Table 1: Array Sorting Algorithms: time and space complexities

1.4 Informații generale

- Sortările prin comparație au limită inferioară $\Omega(n \log n)$.
- Algoritmii non-comparativi (Counting, Radix) pot ajunge la $O(n)$ în condiții favorabile.
- Alegerea pivotului în Quick Sort influențează performanța:
  - pivot ales random
  - mediana din 3/5/7
  - mediana medianelor
- Pentru vectori mici, sortarea cu algoritmi $O(n^2)$ poate fi mai eficientă decât în $O(n \log n)$ (constanta poate fi mare).

1.5 Counting sort

Se creează un vector de frecvență cu valorile pe care trebuie să le sortăm. Valorile din vector sunt numărate, iar vectorul este reconstruit în ordine sortată prin parcurgerea vectorului de frecvență.

Se aplică când elementele sunt numere întregi într-un interval $[0, k]$, (până la 10^6)

1.6 Bucket sort

Se creează k buckets și fiecare element x din vectorul de intrare este distribuit în găleata indexată de o funcție de mapare (de ex. $\lfloor k \cdot \frac{x-a}{b-a} \rfloor$ pentru valori în $[a, b]$). Fiecare găleată este sortată intern (adesea prin Insertion Sort), apoi se concatenează conținutul găleților pentru a obține vectorul sortat.

1.7 Radix sort

Sortarea se face pe mai multe trepte de cifre: pentru fiecare poziție de cifră (de la LSD la MSD sau invers) se aplică Counting Sort pentru a grupa elementele după valoarea cifrei curente. După procesarea tuturor cifrelor, vectorul devine complet sortat.

1.8 Quicksort

Algoritmul Quick Sort folosește o abordare divide et impera: se selectează un pivot, iar vectorul este repartizat în două subsecțiuni, una cu elemente mai mici și cealaltă cu elemente mai mari decât pivotul. Se aplică recursiv același procedeu pe cele două subsecțiuni și, ulterior, se combină rezultatele pentru a obține vectorul sortat.

1.9 Mergesort

Merge Sort divide recursiv vectorul în jumătăți până se obțin secvențe cu un singur element (sau 2, depinde de implementare). Apoi, aceste secvențe sunt interclasate (merge) după ce au fost sortate, fiind combinate treptat pentru a forma vectorul sortat final.

2 Complexități

În studiul algoritmilor, folosim trei noțiuni fundamentale pentru a măsura creşterea funcţiei de timp $T(n)$ în raport cu dimensiunea de intrare n:

- **O** (Big-O): reprezintă o limită superioară asimptotică.

$$T(n) = O(f(n)) \iff \exists\, c > 0,\ n_0 :\ \forall n \geq n_0,\ T(n) \leq c\, f(n).$$

- **Ω** (Big-Omega): reprezintă o limită inferioară asimptotică.

$$T(n) = \Omega(f(n)) \iff \exists\, c > 0,\ n_0 :\ \forall n \geq n_0,\ T(n) \geq c\, f(n).$$

- **Θ** (Big-Theta): când există simultan limite superioară și inferioară de același ordin.

$$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) \ \wedge\ T(n) = \Omega(f(n)).$$

Astfel, $\Theta(f(n))$ indică creşterea „exactă” (în sens asimptotic), iar O şi Ω doar conturul superior, respectiv inferior.

3 Hash-uri și Tabele de Dispersie

3.1 Definiții și noțiuni introductive

Definition 3.1 (Funcție de hash)**.** O *funcție de hash* este o funcție matematică

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

care transformă orice element din universul de chei $\mathcal{U}$ într-o valoare de dimensiune fixă. Rezultatul $h(x)$ se numește *cod hash* al lui x. Funcția trebuie să fie eficient de calculat și să distribuie uniform cheile.

- **Hash prin diviziune:** $h(x) = x \bmod p$, unde p este un număr prim apropiat de m.
- **Hash prin multiplicare:** Hash-ul prin multiplicare standard folosește formula

  $$h_a(K) = \left\lfloor \frac{(aK \bmod W)}{W/M} \right\rfloor,$$

  care produce o valoare în $\{0, \ldots, M-1\}$. Parametrul a se alege coprim cu W și cu o reprezentare binară „aleatorie" a biților. În cazul special când $W = 2^w$ și $M = 2^m$, formula devine

  $$h_a(K) = \left\lfloor \frac{(aK \bmod 2^w)}{2^{w-m}} \right\rfloor,$$

- **Proprietate dorită:** ipoteza dispersiei uniforme simple, adică pentru orice chei distincte x, y avem

  $$\Pr[h(x) = h(y)] = \frac{1}{m}.$$

3.2 Tabelă cu adresare directă

Pentru N chei dintr-un univers mic $\{1, \ldots, N\}$, putem implementa direct o tăblă de dispersie printr-un vector binar:

$$A[1 \ldots N], \quad A[i] = \begin{cases} 1 & \text{dacă } i \text{ este prezent,} \\ 0 & \text{altfel.} \end{cases}$$

Operații:

- `insert(x)`: $A[x] \leftarrow 1$, cost $O(1)$.
- `find(x)`: returnează $A[x]$, cost $O(1)$.
- `delete(x)`: $A[x] \leftarrow 0$, cost $O(1)$.

3.3 Tabele de dispersie cu coliziuni

Pentru universuri mai mari sau chei arbitrare, folosim o funcție de hash h și rezolvăm coliziunile prin:

1. **Listă înlănțuită (chaining).**
2. **Adresare deschisă (open addressing).**

3.3.1 Chaining

Fie $T[0 \ldots m-1]$ un vector de liste înlănțuite. La inserare, adăugăm x în lista $T[h(x)]$. Căutarea parcurge lista, cost amortizat $O(1 + \alpha)$, unde $\alpha = n/m$.

3.3.2 Adresare deschisă

Coliziunile se rezolvă prin sondaj:

Lineară: $h(x,i) = (h_0(x) + i) \bmod m$.

Dublu hashing: $h(x,i) = (h_1(x) + i \cdot h_2(x)) \bmod m$.

În cel mai rău caz, operațiile costă $O(m)$, dar dacă $\alpha < 1$ și funcțiile sunt bine alese, costul mediu este $O(1)$.

3.4 Dispersie universală

Definition 3.2 (Familie universală)**.** O familie de funcții de dispersie $\mathcal{H}$ este *universală* dacă pentru orice $x \neq y$ din $\mathcal{U}$,

$$\big|\{h \in \mathcal{H} : h(x) = h(y)\}\big| = \frac{|\mathcal{H}|}{m}.$$

Consecință: pentru $n \leq m$ și h ales aleator din $\mathcal{H}$, numărul mediu de coliziuni pentru o cheie fixă este mai mic decât 1.

3.5 Algoritmul Rabin–Karp pentru pattern matching

Pentru a găsi un șablon (pattern) P de lungime m într-un text T de lungime n, folosim un hash pentru substring.

- Alegem o bază b (de obicei dimensiunea alfabetului) și un număr M.
- Calculăm hash-ul pattern-ului (substringu-ului)

$$h_P = \sum_{i=0}^{m-1} P[i]\, b^{m-1-i} \bmod M.$$

- Rolling hash-ul fiecărei ferestre de text de lungime m:

$$H_0 = \sum_{i=0}^{m-1} T[i]\, b^{m-1-i} \bmod M,$$

$$H_{i+1} = \big(b\,(H_i - T[i]\, b^{m-1}) + T[i+m]\big) \bmod M.$$

Dacă $H_i = h_P$, facem o verificare explicită $T[i..i+m-1] = P$ pentru a evita *false positives*. Complexitatea totală este $O(n+m)$ în medie.

3.6 Complexități

Structură	**Insert**	**Find**	**Delete**
Directă	$O(1)$	$O(1)$	$O(1)$
Chaining	$O(1+\alpha)$	$O(1+\alpha)$	$O(1+\alpha)$
Open addressing	$O(1/(1-\alpha))$	$O(1/(1-\alpha))$	$O(1/(1-\alpha))$

Table 2: Complexități medii pentru tabele de dispersie