

Analiza timpului de rulare al algoritmilor

Metode teoretice

- relații matematice pentru cazul cel mai defavorabil (uneori și mediu, mai rar: cel mai favorabil)
- de regulă, nu timpul fizic, ci numărul de operații de un anumit tip (ex. pt. sortare: numărul de comparații, numărul de interschimbări)

Evaluare practică

- prin rularea programelor pe diverse seturi de date (aleatoare sau cu anumite proprietăți), și măsurarea timpilor de execuție

Discuțăm:

- funcții pentru generarea de numere (pseudo)aleatoare
- funcții legate de măsurarea timpului

Funcții pentru manipularea timpului (time.h)

Tipuri definite pentru reprezentarea timpului: `clock_t` și `time_t` (sunt de fapt tipuri aritmetice, de ex. `unsigned` sau `unsigned long`)

```
clock_t clock(void);
returnează timpul scurs de la lansarea programului, în unități de ceas
date de constanta CLOCKS_PER_SEC (în standardul POSIX, 1 milion)
e o aproximație dependentă de granularitatea ceasului de timp real
poate interveni depășire (pe sistem de 32 de biți, după cca 72 min.)
```

```
time_t time(time_t *timer);
returnează o valoare aritmetică reprezentând data/ora curentă
(în UNIX, numărul de secunde trecute de la 1 ian. 1970 UTC)
dacă argumentul pointer e nenul, valoarea e stocată și la acea adresă
```

```
double difftime(time_t time1, time_t time0);
returnează diferența exprimată în secunde, ca double
Pentru reprezentări descompuse (zi/oră/min./etc.): tipul struct tm
(vezi detalii în standard)
```

Funcții pentru numere pseudoaleatoare (stdlib.h)

Numerele generate sunt *pseudoaleatoare* (de fapt deterministe, bazate pe un algoritm, dar cu distribuție cât mai uniformă) (numere cu adevărat aleatoare ar trebui să fie bazate pe fenomene fizice, ex. aruncarea unei monede sau descompunerea unor particule)

```
int rand(void);
returnează un număr pseudoaleator între 0 și RAND_MAX (min. 32767)
pt. un număr aleator între 1 și N putem folosi 1 + rand() % N
```

```
void srand(unsigned int seed);
reinițializează generatorul de numere pseudoaleatoare cu valoarea dată
următorul număr va fi generat de rand() pornind de la această valoare
fără apelarea ei, două rulări generează același șir de valori cu rand()
se poate folosi de ex. cu srand(unsigned)time(NULL);
```

Tabele de dispersie (hashtables)

- pentru *regăsirea eficientă* a unui obiect când acesta nu are o valoare numerică de identificare utilizabilă direct ca indice într-un tablou
- ideea: găsirea unei funcții h cu o *valoare numerică unică* pentru fiecare obiect considerat, într-un domeniu restrâns (utilizabil ca indice) \Rightarrow fiecare obiect x e memorat într-un tablou la indicele $h(x)$
- matematic: o funcție parțială $h: D \rightarrow V$, unde D e domeniul tuturor obiectelor posibile, iar domeniul de valori V e $0, 1, \dots, N-1$.
- ex. pt. compilator: D e mulțimea tuturor identificatorilor
- practic, $|D| \gg |V|$, deci h nu poate fi injectivă pe D , dar avem nevoie de valori distincte doar pt. submulțimea $D_u \subseteq D$ a obiectelor efectiv utilizate (ex. identificatorii dintr-un anumit program C)
- se caută funcții de dispersie (hash functions) cu proprietăți cât mai bune (distribuție uniformă \Rightarrow probabilitate mică de valori egale)

Exemple de funcții de dispersie

- funcții simple, calculate rapid, folosind (aproape) toate caracterele
 - adesea cu deplasări pe biți (în loc de înmulțiri, și pt. uniformizare)
- Exemple pentru șiruri (`char *s; len=strlen(s);` parcurs secvențial)

```
for (h=len; len--;) h = ((h<<7) ^ (h<<27)) ^ *s++; /* Knuth */
for (h=5381; c=*s++; ) h += (h << 5) + c; /* Bernstein */
for (h=0; c=*s++; ) h = (h<<6) + (h<<16) - h + c; /* SDBM */
```

Pentru alte obiecte: calcule cu întregii obținuți grupând octeții câte 4

În toate cazurile: valoarea finală luată modulo dimensiunea tabloului

- inserăm elementul la indicele respectiv, și îl căutăm tot acolo

- Și funcțiile bune au *coliziuni* (valori egale pt. obiecte diferite)
- \Rightarrow trebuie rezolvate (dezambiguate) pentru a permite regăsirea corectă
- la ce indice inserăm/căutăm elementul dacă la $h(x)$ se găsește altul?

Tabele de dispersie deschise și închise

Tabele de dispersie închise (closed hashing)

- dacă la indicele $idx=h(x)$ se găsește alt obiect y , se caută succesiv după o anumită regulă: *secvențial* ($idx++$), *liniar* ($idx+=i$), cu a doua funcție ($idx+=h2(x)$), până se găsește obiectul sau o intrare vidă
- nu pot conține mai multe obiecte decât dimensiunea tabloului
- \Rightarrow la depășire, obiectele trebuie redistribuite într-un tablou mai mare
- la ștergere, intrarea în tablou trebuie marcată "șters", nu "vid", pentru a permite căutarea corectă (până la găsim sau "vid")

Tabele de dispersie deschise (open hashing)

- o intrare în tablou: *listă* de obiecte cu aceeași valoare pentru h
- \Rightarrow hashing + căutare liniară în listă (scurtă pentru funcții bune)
- necesită alocare dinamică pentru elementele listei (v. exemplu)
- și aici, tablou cu dimensiune cel puțin comparabilă cu nr. de obiecte