Sisteme de programe pentru timp real

Universitatea “Politehnica” din Bucuresti
2003-2004
Adina Magda Florea
http://turing.cs.pub.ro/sptr_2004

Continut curs

- Complexitatea calculului
- Algoritmi de prelucrare a sirurilor de caractere
- Tehnici de criptare - decriptare
- Tehnici de compresie a datelor
- Algoritmi evoluati pe grafuri
- Retele neurale
- Algoritmi genetici
- Programare Web

Notare

- Examen final: 60%
- Laborator (inclusiv temele de casa): 20%
- Tema de semestru: 20%

- Minimum 6 prezente la laborator (conditie de trecere)

Curs Nr. 1

- Complexitatea algoritmilor
- Probleme NP-complete
- Prelucrarea sirurilor de caractere

1. Complexitatea calculului

Analiza algoritmilor

- **o “dimensiune” N**
- **timpul mediu (average case)**
- **timpul cel mai nefavorabil (worst case)**
- **limita a performantei**
- **deducerea timpului mediu**
- **identificarea operatiilor abstracte**

Clasificarea algoritmilor

Timpul de executie a unui program

- **C1 * f(N) + C2**

- *Ce inseamna “a fi proportional cu” ?*

Timp de executie proportional cu:

- Constant
- log N
- N
- N log N
- N^2
- N^3
- 2^N

Complexitatea calculului

- studiul **cazului cel mai nefavorabil** (worst case) ignorind factorii constanti
- notatia **O mare**
- **DEFINTIE:** O functie $g(N)$ se spune a fi $O(f(N))$ daca exista constantele pozitive C_0 si N_0 astfel incat $g(N) < C_0 * f(N)$ pentru orice $N > N_0$.
- **limita superioara** a performantelor unui algoritm

Complexitatea calculului

- ***O(f(N))*** este o limita superioara si, in realitate, poate fi mult mai scazuta;
- intrarea ce determina cazul cel mai nefavorabil poate sa nu apara niciodata in practica;
- constanta $\mathbf{C_0}$ nu se cunoaste si s-ar putea sa nu fie mica;
- constanta $\mathbf{N_0}$ nu se cunoaste si s-ar putea sa nu fie mica.

Complexitatea calculului

- **notatia Ω**
- **limita inferioara** a performantelor unui algoritm $\boldsymbol{\Omega(f_1(N))}$ ("big omega")
- **DEFINTIE:** O functie $g(N)$ se spune a fi $\Omega(f1(N))$ daca exista o constanta pozitiva C_0 astfel incit $g(N) \geq C_0 * f_1(N)$ pentru un numar infinit de mare de valori ale lui N.

Complexitatea calculului pentru algoritmi recursivi

- Descompunerea recursiva a problemei in subprobleme
- Timpul de executie pentru acesti algoritmi este determinat de dimensiunea si numarul subproblemelor, precum si de costul descompunerii
- C_N = complexitatea pentru o intrare de dimensiune N

- $\mathbf{C_N \approx N}$

La fiecare apel recursiv se elimina cate un element de intrare: $C_N = C_{N-1}+1$

```
int sum(int n)
    { if (n==) return 1;
      return n+sum(n-1);
    }
```

- $\mathbf{C_N \approx N^2/2}$

Un algoritm care necesita parcurgerea intrarii si elimina un element la fiecare apel recursiv: $C_N = C_{N-1} + N$, pentru $N \geq 2$, cu $C_1 = 1$

```
bubble(int a[], int N) /* bubble sort
recursiv */
```

- $\mathbf{C_N \approx \log N}$

Algoritmul injumatateste intrarea la fiecare apel recursiv: $C_N = C_{N/2} + 1$, pentru $N \geq 2$, cu $C_1 = 0$ (pt deducere $N=2^n$)

```
int binsearch(int a[], int key, int l,
  int r)
```

- $\mathbf{C_N \approx 2N}$

Algoritmul imparte intrarea in doua parti la fiecare pas si apoi prelucreaza recursiv fiecare parte ("Divide and conquer"): $C_N = 2*C_{N/2} + 1$, pentru $N \geq 2$, cu $C_1 = 0$

```
rigla(int l, int r, int h
/* Desenarea marcajelor pe o rigla */
```

- **$C_N \approx N\log N$**

Algoritm recursiv care isi imparte intrarea in doua parti la fiecare pas, prelucreaza recursiv fiecare parte si face o trecere liniara peste intrare, fie inainte, fie dupa divizarea intrarii ("Divide and conquer") :

$C_N = 2 * C_{N/2} + N$, pt $N \geq 2$, cu $C_1 = 0$.

```
quicksort(int a[], int l, int r)
```

		Def, Mediu
Insertion sort	**N**	**$N^2/4$, $N^2/8$**
Bubble sort		**$N^2/4$, $N^2/4$**
Quicksort		**N^2, N logN**
Heapsort		**N logN, N logN**
Mergesort		**N logN, N logN (sp ≈ N)**
Sequential search		**N, N/2**
Binary search		**logN, logN**
Binary tree search		**N, logN**
Shortest path	**V, E**	**E sau V^2**
Depth-first search		**V+E sau V^2**
Breadth-first search		**V+E sau V^2**
Min. spanning tree		**(E+V) log V – pr. q.**
		E log E - Kruskal

2. Probleme NP-complete

- "grele" si "usoare"
- Problema usoara: "Exista o cale de la *x* la *y* cu cost ≤ *M* ?
- Problema grea: "Exista o cale de la *x* la *y* cu cost ≥ *M* ?
- Algoritmi "eficienti" si algoritmi "ineficienti"

Probleme NP-complete

- **P:** multimea tuturor problemelor ce pot fi solutionate prin algoritmi **deterministi**, in timp polinomial
- **NP**: multimea tuturor problemelor ce pot fi rezolvate prin algoritmi **nedeterministi**, in timp polinomial
- **NP =? P**

Clasa problemelor NP-complete

- **Clasa de echivalenta**: daca o problema din NP poate fi rezolvata eficient, atunci toate pot fi rezolvate la fel
- Se poate demonstra echivalenta problemelor NP-complete din punct de vedere al algoritmilor de rezolvare
- Pentru a demonstra ca o problema X din NP este NP-completa este necesar sa se demonstreze ca o problema NP-completa (Y, cunoscuta) poate fi transformata in timp polinomial - **redusa polinomial** la problema X

- Presupunem ca problema ciclului hamiltonian (**CH**) este NP-completa *(Y)* si dorim sa vedem daca cea a comis-voiajorului (**CV**) este NP-completa *(X)*
- *Trebuie ca Y sa fie redusa polinomial la X*
- Instanta a **CH** => se construieste o instanta a **CV**:
- $orase_{CV}$ = noduri din $graf_{CH}$
- distante perechi $orase_{CV}$ = 1 daca exiata arc_{CH}
  - » = 2 daca nu exista arc_{CH}

  Se cere **CV** sa gaseasca un ciclu care include toate orasele si are suma ponderilor ≤ V, numarul de noduri din graf – acesta trebuie sa corespunda unui ciclu hamiltonian

  Am demonstrat ca CV este NP-completa

Exemple de probleme NP-complete

- Este o **expresie booleana** data realizabila ?
- Dandu-se un graf neorientat, exista un **ciclu hamiltonian** continut in el (ciclu ce include toate nodurile) ?
- **Problema comis-voiajorului**: Fiind dat un graf cu ponderi intregi pozitive si un intreg K, sa se gaseasca un ciclu care include toate nodurile si a carui suma a ponderilor sa fie ≤ K (sau minima).
- Exista un **K-clique** intr-un graf neorientat G? (*K-clique* – un subgraf complet - orice pereche de puncte din subgraf este legata de un arc - al lui G cu K noduri)

Este un graf neorientat **colorabil cu k culori** ? (G este *k-colorabil* daca exista o atribuire de intregi 1, 2, ... k, reprezentand culori, pentru nodurile din G, astfel incat nu exista doua noduri adiacente de aceeasi culoare. *Numarul cromatic* al lui G este cel mai mic k pentru care G este k-colorabil.)

Exista o **acoperire de noduri** de dimensiune K intr-un graf neorientat ? (G = (V, E), o *acoperire de noduri* este un subset S ⊆ V astfel incit fiecare arc din G este incident intr-un nod din S, card(S) = K.)

- **Problema izomorfismului subgrafurilor**: este un graf neorientat G izomorf cu un subgraf al unui alt graf neorientat G' ?

Problema Knapsack: Fiind data secventa de intregi S = i_1, i_2, ... i_n si un intreg K, exista o subsecventa a lui S a carei sume este exact K ?

Pentru o familie de multimi S_1, S_2, … S_n, exista o **acoperire de multimi** formata dintr-o sub-familie de multimi disjuncte ?

- **Problema planificarii taskurilor**: Dandu-se un *"deadline"* si o multime de taskuri de lungimi variabile, ce trebuie executate pe doua procesoare identice, se pot aranja aceste taskui in asa fel incit *"deadline"*-ul sa fie respectat ?

Cum se poate evita complexitatea?

- algoritm polinomial dar suboptimal; rezulta o solutie apropiata de cea mai buna
- se foloseste direct algoritmul exponential daca dimensiunea intrarii nu este mare
- algoritm exponential cu euristici

3. Algoritmi de identificare a sirurilor de caractere

- Introducere
- Algoritmul cautarii directe
- Algoritmul Boyer-Moore

Curs nr. 2

- Algoritmul Rabin-Karp
- Algoritmul Knuth-Morris-Prat
- Algoritmul deplaseaza-aduna

Algoritmi de identificare a sirurilor de caractere

3.1 Introducere

- Identificarea (recunoasterea) sabloanelor in siruri de caractere
- sablon – p, lungime M
- sir – a, lungime N
- Fiind dat un sir de lungime N

$$a = a_1a_2a_3 \ldots a_N,$$

si un sablon de lungime M

$$p = p_1p_2p_3 \ldots p_M,$$

se cere sa se gaseasca multimea de pozitii

$$\{ i \mid 1 \le i \le N-M+1 \text{ a.i. } a_ia_{i+1} \ldots a_{i+M-1} = p \}$$

3.2 Algoritmul cautarii directe

```
int cautsablon1(char *p, char *a)
  { int i = 0, j = 0;   /* i este pointerul in text */
    int M = strlen(p);  /* j este pointerul in sablon*/
    int N = strlen(a);

  do
   {
    while ((i < N) && (j<M) && (a[i]==p[j])) {i++;j++;}
    if (j==M) return i-M;
    i- = j-1;
    j = 0;
    } while (i<N);
  return i;
  }
```

Algoritmul cautarii directe

```
int cautsablon2(char *p, char *a)
  { int i, j;
    int M = strlen(p);
    int N = strlen(a);

    for (i=0, j=0; j<M && i<N; i++, j++)
      while (a[i] != p[j])
      { i- = j-1;
        j = 0;
      }
    if (j==M)
      return i-M;
    else
      return i;
  }
```

Algoritmul cautarii directe

- **Proprietate:** Algoritmul cautarii directe necesita, in cazul cel mai nefavorabil, un numar de **maxim N*M** comparatii de caractere.
- Timpul de rulare este proportional, pentru cazul cel mai defavorabil, cu M*(N-M+1), si cum M << N obtinem valoarea aproximativa N*M.

3.3 Algoritmul Boyer-Moore

- Potrivit daca intoarcerea in text nu este dificila

```
UN EXEMPLU DE_CAUTARE RAPIDA
UTARE
       UTARE
           UTARE
                 UTARE
                   UTARE
```

Algoritmul Boyer-Moore

- Vectorul **skip** – pentru fiecare caracter din alfabet
- arata, pentru fiecare caracter din alfabet, cat de mult se sare (se deplaseaza sablonul la dreapta) in sir daca acest caracter a provocat nepotrivirea
- rutina **initskip** – initializeaza skip:
- `skip[index(p[j])] = M-j-1, j=0,M-1` pentru caracterele din sablon
- `skip[index(c)] = M` pentru restul de caractere din alfabet

Algoritmul Boyer-Moore

```
int caut_BM(char *p, char*a)
 { int i, j, t;
   int M=strlen(p), N = strlen(a);
   initskip(p);

   for (i=M-1, j=M-1; j>=0; i--, j--)
     while (a[i]!=p[j])
      { t = skip[index(a[i])];
        i+ = (M-j > t) ? M-j : t;
        if (i >= N) return N;
        j = M-1;

   return i+1;
 }
```

Algoritmul Boyer-Moore

- De ce trebuie

```
i+ = (M-j > t) ? M-j : t;
```

Exemplu

```
         i                 t=skip[R] = 0
X  Y  Z  R  R  A  C  R     M-j=M-M+2 = 2
R  R  A  C  R              i= i + 2
         j=M-2
```

Algoritmul Boyer-Moore

- Combinarea a doua metode pentru a afla exact cu cat trebuie deplasat la dreapta sirul: metoda prezentata si metoda vectorului *next* din algoritmul Knuth-Morris-Prat, in varianta de la dreapta la stinga, pentru ca apoi sa se aleaga in *skip* cea mai mare valoare.
- In conditiile acestei modificari, este valabila urmatoarea proprietate.
- **Proprietate:** Algoritmul **Boyer-Moore** face cel mult M+N comparatii de caractere si foloseste N/M pasi, daca alfabetul nu este mic si sablonul este scurt.