Complexitatea problemelor

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2015/2016

Plan

De ce definim complexitatea unei probleme

Pentru o problemă rezolvabilă pot exista mai mulți algoritmi care să o rezolve.

De fapt dacă există unul, atunci există o infinitate. (De ce?)

Ce putem spune despre eficiența rezolvării unei probleme?

Definițiile de la eficiența algoritmilor pot fi ușor transferate la probleme.

Definiția complexității $O(f(n))$ a unei probleme

Oferă o margine superioară pentru efortul computațional necesar rezolvării unei probleme.

Definition

Problema P are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$ dacă există un algoritm A care rezolvă P și $T_A(n) = O(f(n))$.

Definiția complexității $\Omega(f(n))$ a unei probleme

Oferă o margine inferioară pentru efortul computațional necesar rezolvării unei probleme.

Definition

P are complexitatea timp în cazul cel mai nefavorabil $\Omega(f(n))$ dacă orice algoritm A care rezolvă P are $T_A(n) = \Omega(f(n))$.

Algoritm optim pentru o problemă

Definition

A este algoritm optim (din punct de vedere al complexității timp pentru cazul cel mai nefavorabil) pentru problema *P* dacă

- *A* rezolvă *P* și
- *P* are complexitatea timp în cazul cel mai nefavorabil $\Omega(T_A(n))$.

Plan

Problema sortării

Considerăm cazul particular al sortării tablourilor:

SORT
Input n și tabloul $a = [v_0, \ldots, v_{n-1}]$.
Output tabloul $a' = [w_0, \ldots, w_{n-1}]$ cu proprietățile: $w_0 \leq \cdots \leq w_{n-1}$ și $w = (w_0, \ldots, w_{n-1})$ este o permutare a secvenței $v = (v_0, \ldots, v_{n-1})$; .

Notații:
SORTED(a): tabloul a este sortat
$\mathrm{Perm}(v, w)$: w este o permutare a lui v

Sortare prin interschimbare (BubbleSort) 1/2

caracterizarea *SORTED*(a)

$$SORTED(a) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

unde $n = a.size()$. (Aceasta e parte a domeniului problemei.)

De la domeniul problemi la algoritm:

```
for (i=0; i < n-1; ++i) {
  if (a[i] > a[i+1]) {
    swap (a, i, i+1);
```

Sortare prin interschimbare (BubbleSort) 2/2

Procesul de restabilire de mai sus trebuie repetat până nu mai sunt inversiuni:

```
while (posibil să mai existe inversiuni) {
  for (i=0; i < n-1; ++i) {
    if (a[i] > a[i+1]) {
      swap (a, i, i+1);
}
```

(Acesta este pseudocod!)

Testul *posibil să mai existe inversiuni* poate fi verificat ținând minte poziția ultimei inversiuni:

BubbleSort: algoritmul

```
bubbleSort(a, n) {
  ultim = n-1;
  while (ultim > 0) {
    n1 = ultim;
    ultim = 0;
    for (i=0; i < n1; ++i) {
      if (a[i] > a[i+1]) {
        swap (a, i, i+1);
        ultim = i;
      }
    }
  }
}
```

```
swap(a, i, j) {
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Evaluarea algoritmului BubbleSort 1/2

Corectitudine

Invariant bucla `while`: a[`ultim+1 .. n − 1`] include cele mai mari `n−1−ultim` elemente din a și *SORTED*(a[`ultim+1 .. n−1`])

Invariant bucla `for`: a[j] $\leq$ a[i] pentru $j = 0, \ldots, i$.

`swap` menține proprietatea $\mathrm{Perm}(u, u')$, unde u este valoarea variabilei a înainte de `swap` și u' cea de după

Evaluarea algoritmului BubbleSort 1/2

Timp de execuție

- dimensiune instanță: n (= `a.size()`)
- operații măsurate: comparațiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvența de intrare este ordonată descrescător
- numărul de comparații pentru acest caz este

$$(n-1)+(n-2)+\cdots+1=\frac{(n-1)n}{2}=O(n^2)$$

Sortare prin inserție directă (InsertSort) 1/2

Principiul de bază

```
for j in 1 .. n-1
  inserează a[j] în a[0..j-1] a.î. SORT(a[0..j])
```

(Acesta este un pseudo-cod!)

Sortare prin inserție directă (InsertSort) 2/2

Analiza domeniului problemei

Poziția i pe care trebuie inserat $a[j]$:

- $i = j$ dacă $a[j] \geq a[j-1]$;
- $i = 0$ dacă $a[j] < a[0]$;
- $0 < i < j$ și satisface $a[i-1] \leq a[j] < a[i]$

$\Longrightarrow$ $a[i..j-1]$ trebuie deplasate la dreapta cu o poziție!

– condiția pentru deplasarea la dreapta: $i \geq 0 \wedge a[i] > a[j]$

Algoritmic:

```
i = j - 1;
temp = a[j];
while ((i >= 0) && (a[i] > temp)) {
  a[i+1] = a[i];
  i = i -1;
}
```

InsertSort: algoritmul

```
insertSort(a, n) {
  for (j = 1; j < n;  j = j+1) {
    i = j - 1;
    temp = a[j];
    while ((i >= 0) && (temp < a[i])) {
      a[i+1] = a[i];
      i = i -1;
    }
    if (i != j-1) a[i+1] = temp;
  }
}
```

Evaluarea algoritmului InsertSort 1/2

Corectitudine

Invariantul buclei `for`: $\text{Perm}(u, v) \wedge SORTED(a[0..j-1])$, unde u este valoarea curentă a variabilei a

Invariantul buclei `while`: $a[i+1], \ldots, a[j-1] > temp$.

Invariantul buclei `while` și $\texttt{a[i]} \leq \texttt{temp} \vee \texttt{i} < 0$ asigură determinarea corectă a lui i, i.e. $SORTED(a[0..j])$.

Evaluarea algoritmului InsertSort 2/2

Timp de execuție

- dimensiune instanță: n (= `a.size()`)
- operații măsurate: comparațiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvența de intrare este ordonată descrescător
  – căutarea poziției i în subsecvența $a[0 .. j-1]$ necesită $j-1$ comparații
- numărul de comparații pentru acest caz este

$$1 + 2 = \cdots + (n-1) = \frac{(n-1)n}{2} = O(n^2)$$

Selecția sistematică

Analiza domeniului problemei Proprietatea *MAXHEAP*(a):

$$(\forall i \geq 0) 2i+1 < n \implies a[i] \geq a[2i+1] \wedge$$
$$2(i+1) < n \implies a[i] \geq a[2(i+1)]$$

$MAXHEAP(\mathrm{a}) \implies \max \mathrm{a} = \mathrm{a}[0]$

Ideea algoritmului:

- se presupune *MAXHEAP*(a)
- dacă facem interschimbarea `swap(a,0,n-1)`, noua valoarea $a[n-1]$ e pe locul ei final și tabloul rămas de sortat este $\mathrm{a}[0..n-2]$
- $\mathrm{a}[0..n-2]$ se sortează în aceeași manieră

Ideea algoritmului mai algoritmică

```
heapSort(a, n) {
  stabileşte MAXHEAP( a)
  for (r = n-1; r > 0; --r) {
    swap(a, 0, r);
    restabileşte MAXHEAP( a[0..r − 1])
}
```

(Acesta este pseudocod!)

Stabilirea proprietății de max-heap

Analiza domeniului problemei

- $MAXHEAP(\mathsf{a}, \ell)$:

$$(\forall i \geq \ell) 2i+1 < n \implies a[i] \geq a[2i+1] \wedge$$
$$2(i+1) < n \implies a[i] \geq a[2(i+1)]$$

- $\ell \geq n/2 \implies MAXHEAP(\mathsf{a}, \ell)$
- dacă $MAXHEAP(\mathsf{a}, \ell-1)$ putem stabili $MAXHEAP(\mathsf{a}, \ell)$ inserând $a[\ell-1]$ în $\mathsf{a}[\ell..n-1]$

De la domeniul problemei la algoritm:

```
j = ℓ;
while (există copil/copii a/ai lui j) {
   k = indexul copilului cu valoare maximă;
   if (a[j] < a[k]) swap(a, j, k);
   j = k;
}
```

Algoritmul HeapSort

```
insertInHeap(a, n, ℓ) {
  isHeap = false; j = ℓ;
  while ((2*(j-1) <= n-1) && ! isHeap) {
    k = 2*j +1;
    if ((k < n-1) && (a[k] < a[k+1]))  k = k+1;
    if (a[j] < a[k]) swap(a, j, k); else  isHeap = true;
    j = k;
  }
}
heapSort(a, n) {
  for (l = (n-1)/2; l >= 0; l = l-1)
    insertInHeap(a, n, l);
  r = n-1;
  while (r >= 1) {
    swap(a, 0, r);
    insertInHeap(a, r, 0);
    r = r - 1;
  }
}
```

Evaluarea algoritmului HeapSort 1/2

Corectitudine Se bazează pe corectitudinea implementării operațiilor peste *max-heap*.

invariantul instrucțiunii `while` din `insertInHeap`: $(\forall i \geq \ell)$ dacă j nu este în arborele cu rădăcina în i, atunci $MAXHEAP(\mathrm{a}, i)$

invariantul lui `for` din `heapSort`: $MAXHEAP(\mathrm{a}, \ell)$

invariantul instrucțiunii `while` din `heapSort`: $MAXHEAP(\mathrm{a}[0..r-1]) \wedge SORTED(\mathrm{a}[r..n-1])$

Evaluarea algoritmului HeapSort 2/2

Timp de execuție

- dimensiune instanță: n (= `a.size()`)
- operații măsurate: comparațiile care implică elementele tabloului
- cazul cel mai nefavorabil: greu de spus
  – complexitatea timp al operației `insertInHeap`: $O(\log n)$
  – dar construcția max-heap-ului necesită $O(\log n)$ (de fapt $\Theta(n)$
  – complexitatea lui `while`:
  $O(log(n-1) + O(\log n - 2) + \cdots + O(\log 1) = O(n \log n)$
- numărul de comparații pentru acest caz este $O(n \log n)$

Alți algoritmi de sortare

Exerciții pentru seminar.

Două întrebări despe algoritmii de sortare

- care este numărul minim de comparații executate în cazul cel mai nefavorabil?
- care algoritmi de sortare realizează minimul de comparații, i.e. care algoritmi sunt optimali?

Pentru a putea răspunde la cele două întrebări trebuie mai întâi să precizăm modelul de calcul peste care sunt construiți acești algoritmi.

Arborii de decizie pentru sortare: intuitiv

Presupunere: $a_i \neq a_j$ dacă $i \neq j$

Notație: $i \,?\, j \equiv$ se compară a[i] cu a[j]

Construcția arborelui ce reprezintă comparațiile făcute de un algoritm:

- vârfurile interne conțin comparații $i \,?\, j$;
- subarborele din stânga conține comparațiile făcute în cazul $a_i < a_j$;
- subarborele din dreapta conține comparațiile făcute în cazul $a_i > a_j$;
- vârfurile externe (frontiera) conțin permutări

Algoritmi reprezentați ca arborii de decizie (pentru sortare)

Definition

Arbore de decizie pentru n elemente:

- vârfurile interne: $i \,?\, j$
- vârfurile pe frontieră: permutări ale mulțimii $\{0, 1, \ldots, n-1\}$

Definition

Calculul unui arbore de decizie t pentru intrarea $a = (a_0, \ldots, a_{n-1})$:
drum de la rădăcină la un vârf pe frontieră cu proprietatea

- dacă $a_i < a_j$: copilul din stânga lui $i \,?\, j$ devine vârf curent;
- altfel copilul din dreapta devine vârf curent

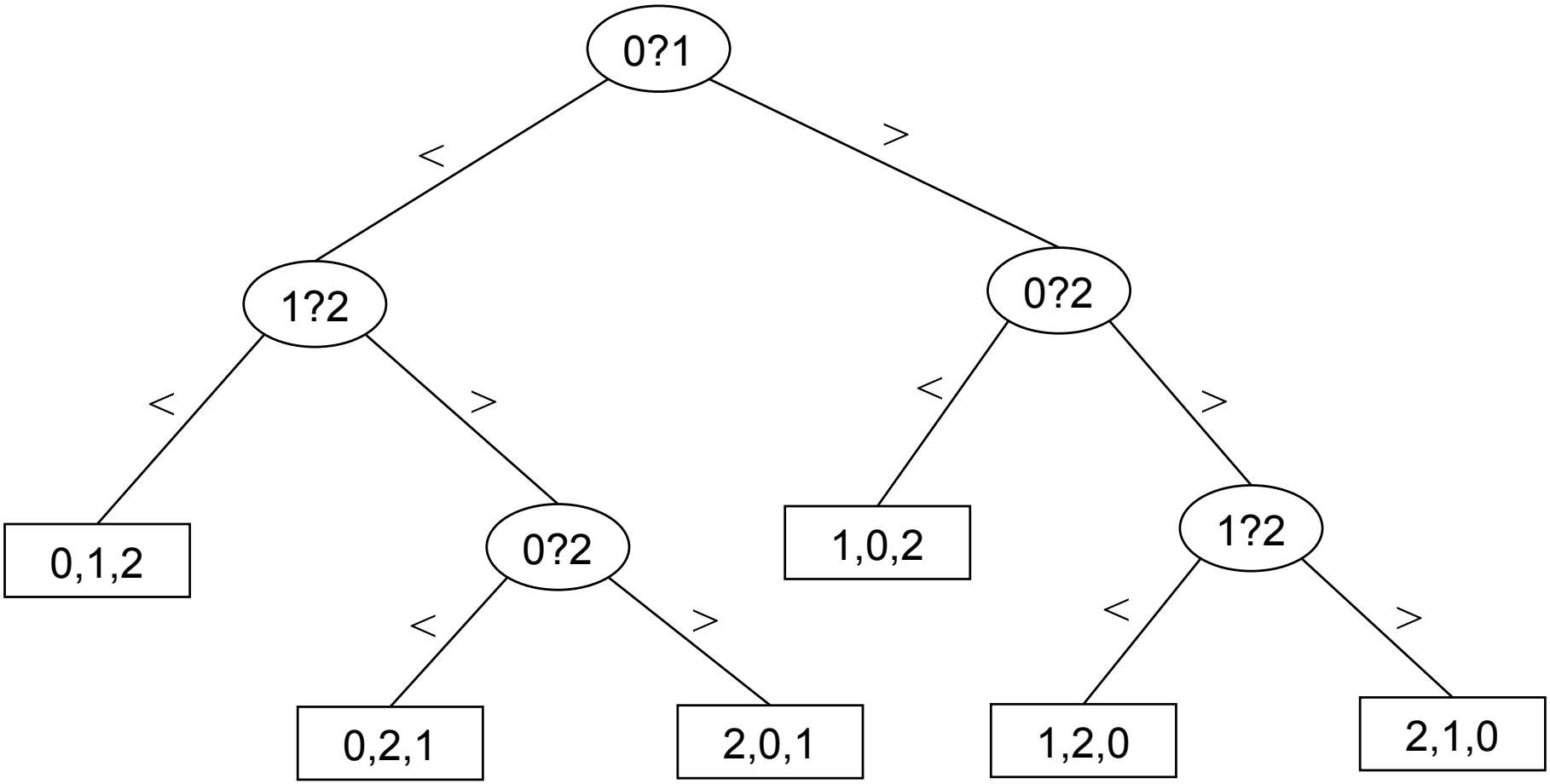
Arbori de decizie pentru sortare

Definition

t (pentru n elemente) rezolvă problema sortării:
$\forall$ intrare $a = (a_0, \ldots, a_{n-1})$
calculul lui t pentru a se termină în π a. î. $a_{\pi(0)} < \cdots < a_{\pi(n-1)}$

arbore de decizie pentru sortare: arbore de decizie care rezolvă problema sortării

Arborele de decizie pentru InsertSort



Complexitatea sortării

Timpul de execuție minim pentru cazul cel mai nefavorabil:

$$T(n) = \min_{t} \max_{\pi} \text{length}(\pi, t)$$

Theorem

Problema sortării are timpul de execuție pentru cazul cel mai nefavorabil $\Omega(n \log n)$ în modelul arborilor de decizie pentru sortare.

Corollary

Algoritmul HeapSort este optimal în modelul arborilor de decizie pentru sortare.

Plan

Problema căutării

Instance o mulțime univers $\mathcal{U}$, o submulțime $S \subseteq \mathcal{U}$ și un element a din $\mathcal{U}$;
Question $a \in S$?

Presupunem că $\mathcal{U}$ este total ordonată și mulțimea S este reprezentată de tabloul $\mathtt{s}[0..\mathtt{n}-1]$ cu $s[0] < \cdots < s[n-1]$.

Algoritm generic divide-et-impera de căutare: ideea

- se determină m cu $p \leq m \leq q$;
- dacă $a = s[m]$ atunci căutarea se termină cu succes;
- dacă $a < s[m]$ atunci căutarea continuă cu subsecvența $(s[p], \ldots, s[m-1])$;
- dacă $a > s[m]$ atunci căutarea continuă cu subsecvența $(s[m+1], \ldots, s[q])$;

Cei mai cunoscuți dintre acestia sunt:

- Căutare liniară (secvențială). Se alege $m = p$.
- Căutare binară. Se alege $m = \lceil \frac{p+q}{2} \rceil$.
- Căutare Fibonacci. Se presupune $q + 1 - p = Fib(k) - 1$

Algoritm generic divide-et-impera de căutare

```
pos(s, n, a) {
  p = 0; q = n - 1;
  2: alege m între p şi q
  while ( (a != s[m]) && (p < q)) {
    if (a < s[m]) q = m -1; else  p = m + 1;
    5: alege m între p şi q
  }
  if (a == s[m]) return m; else return -1;
}
```

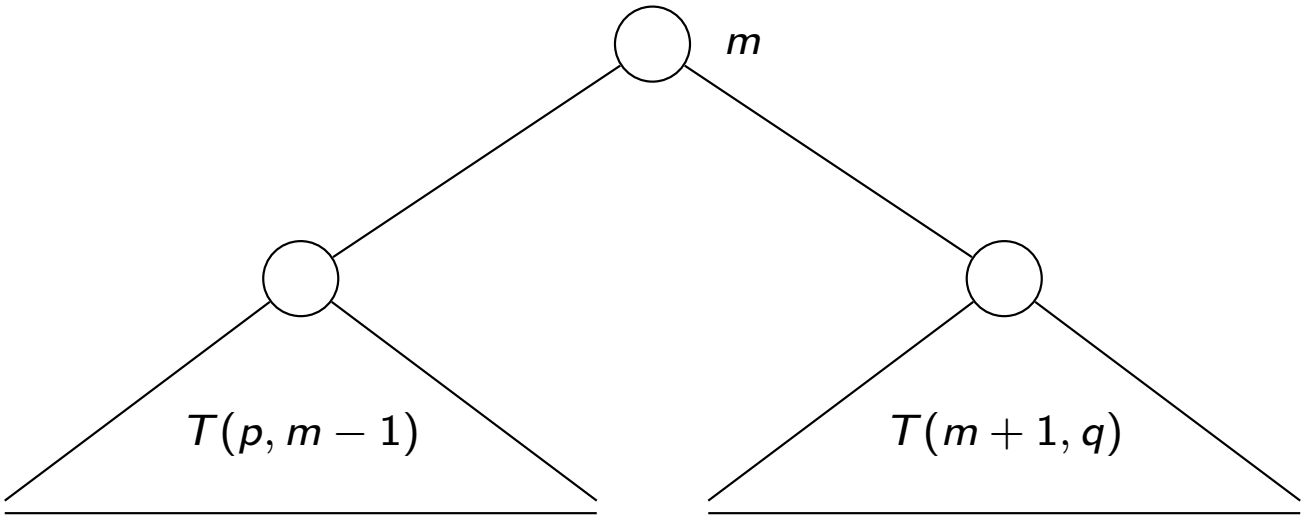
Algoritmi reprezentați ca arbori de decizie (pentru căutare)

Definition

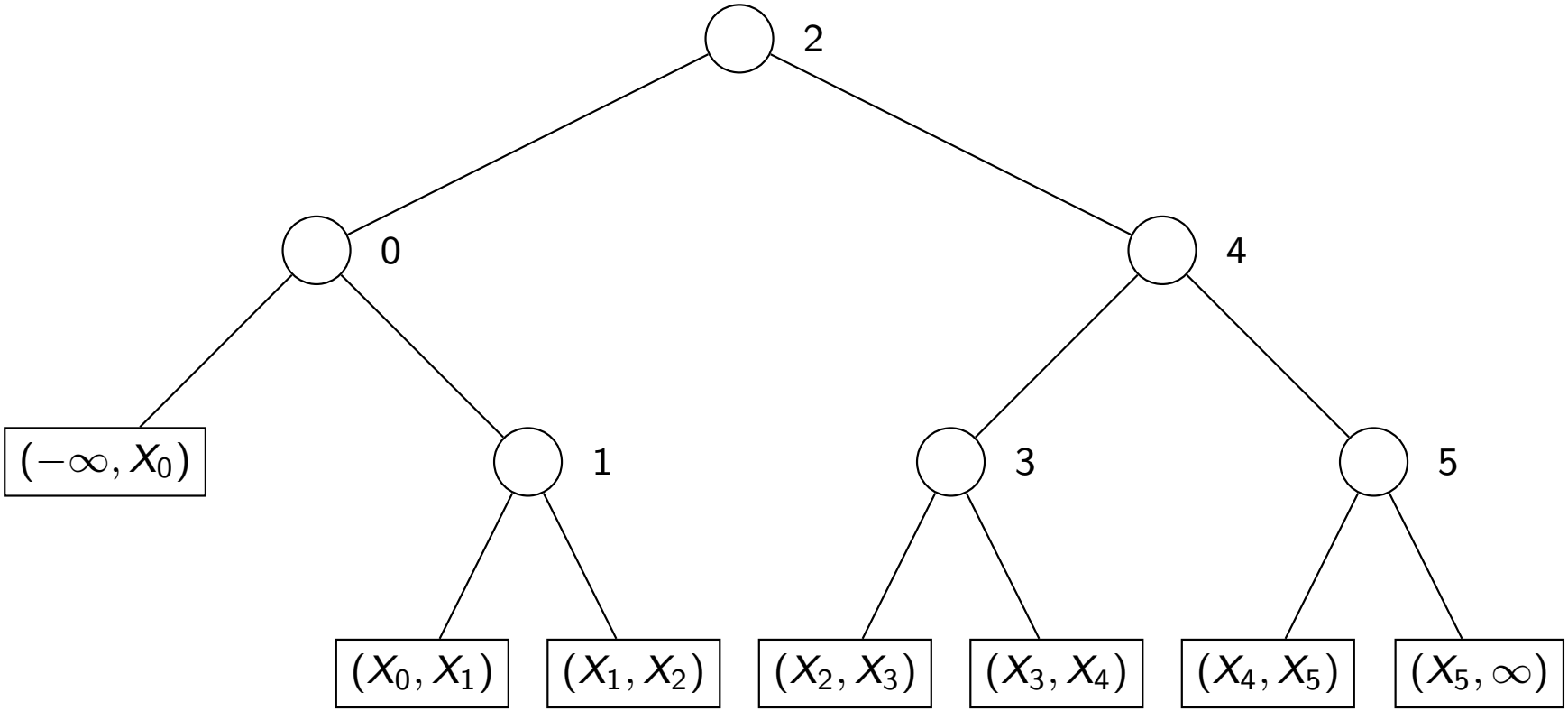
Arborele de decizie pentru căutare de dimensiune n: $T(0, n-1)$, unde $T(p, q)$ este definit recursiv astfel:

- dacă $p > q$ atunci $T(p, q)$ este arborele vid;
- altfel, rădăcina este m e dat de instr. 2 sau 5, subarborele stâng este $T(p, m-1)$ și cel drept este $T(m+1, q)$
- vârfuri pe frontieră: $(-\infty, X_0), (X_0, X_1), \ldots, (X_{n-1}, +\infty)$ în această ordine de la stânga la dreapta

$T(p, q)$ grafic



Exemplu de arbore de decizie pentru căutarea binară



Algoritmi reprezentați ca arbori de decizie (pentru căutare)

Definition

Calculul unui arbore de decizie pentru intrarea $[x_0, \ldots, x_{n-1}], a$:
parcurgerea unui drum de la rădăcină spre frontieră cu proprietatea

1. dacă vârful curent este (X_i, X_{i+1}) (i.e., extern), atunci $a \in (x_i, x_{i+1})$ și calculul se termină cu insucces;
2. dacă vârful curent este m și $a = x_m$, atunci calculul se termină cu succes;
3. dacă $a < x_m$ atunci rădăcina subarborelui stâng devine vârf curent;
4. dacă vârful curent este m și $a > x_m$, atunci rădăcina subarborelui drept devine vârf curent.

Cazul particular al căutării binare

Lemma

Fie t arborele de decizie pentru căutare cu n vârfuri corespunzător căutării binare. Dacă $2^{h-1} \leq n < 2^h$, atunci înălțimea lui t este h.

Corollary

Timpul de execuție pentru cazul cel mai nefavorabil al căutării binare este $O(\log_2 n)$.

Proprietăți ale arborilor de decizie pentru căutare

Definition

Lungimea internă a lui t: $\mathrm{IntLength}(t)$ =suma lungimilor drumurilor de la rădăcină la vârfurile interne.
Lungimea externă a lui t: $\mathrm{ExtLength}(t)$ =suma lungimilor drumurilor de la rădăcină la vârfurile de pe frontieră (pendante).

Lemma

Fie t un arbore de decizie pentru căutare cu n vârfuri interne. Atunci:

$$\mathrm{ExtLength}(t) - \mathrm{IntLength}(t) = 2n.$$

Lemma

Lungimea internă minimă a unui arbore de decizie cu n vârfuri interne este:

$$(n+1)(h-1) - 2^h + 2$$

Complexitatea căutării

Theorem

Problema căutării are timpul de execuție în cazul cel mai nefavorabil $\Omega(\log n)$ în modelul arborilor de decizie pentru căutare.

Corollary

Căutarea binară este optimă în modelul arborilor de decizie pentru căutare.

Plan

Motivație

Mentalitate: "Dacă știu să rezolv problema Q, pot utiliza acel algoritm să rezolv P?"

Intuitiv: Problema P se reduce la Q dacă un algoritm care rezolvă Q poate ajuta la rezolvarea lui P.

Aplicații:

- proiectarea de algoritmi
- demonstrarea limitelor: daca P este dificilă atunci și Q este dificilă
- clasificarea problemelor

Reducerea Turing/Cook

Problema P se reduce polinomial la problema (rezolvabilă) Q, notăm $P \propto Q$, dacă se poate construi un algoritm care rezolvă P după următoarea schemă:

1. se consideră la intrare o instanță p a lui P;
2. preprocesează în timp polinomial intrarea p
3. se apelează algoritmul pentru Q, posibil de mai multe ori (un număr polinomial)
4. se postprocesează rezultatul dat de Q în timp polinomial

Dacă pașii de preprocesare și postprocesare necesită $O(g(n))$ timp, atunci scriem $P \propto_{g(n)} Q$.

Exemplu: MAX $\propto$ SORT

Fie MAX problema determinării elementului maxim dintr-o mulțime:

Input	O mulțime S total ordonată.
Output	Cel mai mare element din S.

Următorul algoritm rezolvă MAX:

1. reprezintă S cu un tablou s (preprocesare);
2. apelează un algoritm de sortare pentru s;
3. întoarce ultimul element din s (postprocesarea);

$\propto$ nu e întotdeauna o "reducere de la o problemă mai complexă la una mai simplă" !!!

$\propto$ e mai degrabă "transformare" ...

Variante pentru submulțimea de sumă dată

SSD1

Input O mulțime S de numere întregi, M număr întreg pozitiv.

Output Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.

Question Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

SSD3

Instance O mulțime S de numere întregi, M un număr întreg pozitiv.

Question Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

Exemplu: SSD1 $\propto$ SSD2

SSD1

Input O mulțime S de numere întregi, M număr întreg pozitiv.

Output Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.

Question Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

1. nu există preprocesare;
2. caută binar pe M^* în intervalul $(0, M]$ apelând un algoritm care rezolvă SSD2;

Acesta este un exemplu de reducerea unei probleme de optim la versiunea ei ca problemă de decizie.

Exemplu: SSD2 $\propto$ SSD1

SSD1

Input O mulțime S de numere întregi, M număr întreg pozitiv.

Output Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K < M$.

Question Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

1. nu există preprocesare;
2. calculează $M^* \leq M$ apelând un algoritm care rezolvă SSD1;
3. dacă $M^* \geq K$ întoarce 'DA', altfel întoarce 'NU';

Exemplu: SSD3 $\propto$ SSD1

SSD1

Input O mulțime S de numere întregi, M număr întreg pozitiv.

Output Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD3

Instance O mulțime S de numere întregi, M un număr întreg pozitiv.

Question Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

1. nu există preprocesare;
2. calculează $M^* \leq M$ apelând un algoritm care rezolvă SSD1;
3. dacă $M^* = M$ întoarce 'DA', altfel întoarce 'NU';

Reducerea Karp

Se consideră P și Q probleme de decizie.
Problema P se reduce polinomial la problema (rezolvabilă) Q, notăm $P \propto Q$, dacă se poate construi un algoritm care rezolvă P după următoarea schemă

1. se consideră la intrare o instanță p a lui P;
2. preprocesează în timp polinomial intrarea p
3. se apelează (o singură dată) algoritmul pentru Q
4. răspunsul pentru Q este același cu cel al lui P (fără postprocesare)

Dacă pasul de preprocesare necesită $O(g(n))$ timp, atunci scriem $P \propto_{g(n)} Q$.

Reducerea Karp este un caz particular de reducere Turing/Cook.

Exemplu: SSD3 $\propto$ SSD2

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.

Question Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

SSD3

Instance O mulțime S de numere întregi, M un număr întreg pozitiv.

Question Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

1. nu există preprocesare;
2. apelează un algoritm care rezolvă SSD2 pentru instanța S, M, M;

Exemplu: SUBSET $\propto$ DISJOINT

SUBSET
Instanță Două mulțimi S_1 și S_2 ($S_1, S_2 \subseteq \mathcal{U}$, $\mathcal{U}$ mulțime univers).
Întrebare $S_1 \subseteq S_2$?

DISJOINT
Instanță Două mulțimi S_1 și S_2.
Întrebare $S_1 \cap S_2 = \emptyset$?

SUBSET $\propto$ DISJOINT:

1. se consideră la intrare o instanță S_1, S_2 a lui SUBSET;
2. calculează $t(S_1, S_2) = S_1, \overline{S_2}$
3. întoarce rezultatul întors de un algoritm care rezolvă DISJOINT pentru instanța $S_1, \overline{S_2}$.

Reducerea: proprietăți

Theorem

a) Dacă P are complexitatea timp $\Omega(f(n))$ și $P \propto_{g(n)} Q$ (versiunea Karp) atunci Q are complexitatea timp $\Omega(f(n) - g(n))$.

b) Dacă Q are complexitatea $O(f(n))$ și $P \propto_{g(n)} Q$ (versiunea Karp) atunci P are complexitatea $O(f(n) + g(n))$.