Outline

Cuprins

1 Recapitulare

Ce s-a discutat la cursul trecut

- problemă rezolvată de un algoritm
  - problemă P
    instanţă (= input): $p \in P$
    rezultat (= output): $P(p)$
  - A rezolvă P:
    σ_p = stare ce codifică $p \in P$
    $\langle A, \sigma_p \rangle \Rightarrow^* \langle \cdot, \sigma' \rangle$, unde σ' codifică $P(p)$
- problemă de decizie: (instance, question)
- problemă rezolvabilă (decidabilă): problemă rezolvată de un algoritm

Ce s-a discutat la cursul trecut

- dimensiunea unei instanţe: dacă $p \in P$, atunci $size_d(p) = size_d(\sigma_p) = \sum_{var \mapsto val \in \sigma} size(val)$, $d \in \{log, unif\}$
- complexitatea timp/spaţiu:
  $\langle A, \sigma_p \rangle = \langle A_0, \sigma_0 \rangle \Rightarrow \langle A_1, \sigma_1 \rangle \Rightarrow \ldots \Rightarrow \langle A_n, \sigma_n \rangle = \langle \cdot, \sigma' \rangle$
  $T_d(A, p) = \sum_i time_d(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$
  $S_d(A, p) = \max_i size_d(\langle A_i, \sigma_i \rangle) = \max_i size_d(\sigma_i)$
- complexitatea în cazul cel mai nefavorabil $n = \{p \in P \mid size_d(p) = n\}$
  $T_d(A, n) = \max\{T_d(A, p) \mid size_d(p) = n\}$
  $S_d(A, n) = \max\{S_d(A, p) \mid size_d(p) = n\}$

Mai mult despre mărimea unei instanţe
Considerăm algoritmul:

```
//@input: m >= 0
//@output: s == m
s = 0;
for (i = 1; i <= m; ++i)
  s = s + 1;
```

- mărimea uniformă ($n = O(1)$): timp de execuţie constantă!!??
- mărimea logaritmică ($n = O(\log m)$): timp $= \sum_{i=1}^{m} \log n = m \log m = O(2^n)$!!?? (s-au considerat doar comparaţiile)
- cel mai firesc este să luăm $n = m$ (egală cu input-ul)

Mai mult despre mărimea unei instanţe

- testul de primalitate

  ```
  isPrime(n) {
    ...
  }
  ```

  De regulă se consideră că dimensiunea instanţei este n
  Agrawal et al., 2004: $T(n) = O(\log^{15/2} n \cdot \mathrm{poly}(\log \log n)) = O(\log^{15/2+\varepsilon} n)$.
  Acest rezultat este foarte important, deoarece arată că dacă se consideră dimensiunea logaritmică a instanţei ($= \log n$), atunci complexitatea timp este mărginită de un polinom.

- cel mai mare divizor comun

  ```
  gcd(a, b) {
    ...
  }
  ```

  input-ul este format din două variabile a şi b
  $n = a \cdot b$ numărul de înmuţiri: $O(5 \log_{10} \min(a, b))$
  (k înmuţiri $\implies \max(a, b) \geq \mathit{fib}(k + 2)$, $\min(a, b) \geq \mathit{fib}(k + 1)$)

2 Complexitatea problemelor computaţionale

De ce definim complexitatea unei probleme computaţionale
Până acum am clasificat problemele în rezolvabile şi nerezolvabile.

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

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

Am văzut cum se măsoară eficienţa unui algoritm.
Ce putem spune despre eficienţa rezolvării unei probleme?

Definiţiile de la eficienţa algoritmilor pot fi uşor transferate la probleme. De exemplu, complexitatea timp a unei probleme se referă la complexitatea timp a algoritmilor care rezolvă problema. Pentru fiecare versiune a definiţiei pentru algorimi (în cazul cel mai nefavorabil, medie, cost uniform, cost logaritmic,. . .), vom avea una corespunzătoare pentru probleme.

Definiţia complexităţii $O(f(n))$ a unei probleme
Oferă o margine superioară pentru efortul computaţional 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 şi $T_A(n) = O(f(n))$.

Pentru a arăta că o problemă P are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$, este suficient de găsit un algoritm A care rezolvă P şi să arătăm că A are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$. Valoarea lui $f(n)$ ne dă o margine superioară pentru timpul necesar rezolvării unei instanţe de dimensiune n.

Definiţia complexităţii $\Omega(f(n))$ a unei probleme
Oferă o margine inferioară pentru efortul computaţional 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 $T_A(n) = \Omega(f(n))$.

Acest tip de informaţie este mult mai dificil de obţinut deoarece trebuie arătat că nu există algoritmi care să rezolve orice instanţă de dimensiune n într-un timp mai mic decât $f(n)$ multiplicat cu o constantă. Vom studia doar două probleme pentru care există dovedită această margine inferioară: sortarea şi căutarea.

Algoritm optim pentru o problemă

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

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

Se poate dovedi că un algoritm este optim din punct de vedere al timpului de execuţie numai dacă se cunoaşte limita inferioară pentru problemă; din acest motiv se cunosc puţini algoritmi optimi.

3 Complexitatea sortării

Problema sortării
Considerăm cazul particular al sortării tablourilor:

SORT	
Input	n şi tabloul $a = [v_0, \dots, v_{n-1}]$.
Output	tabloul $a' = [w_0, \dots, w_{n-1}]$ cu proprietăţile: $w_0 \leq \cdots \leq w_{n-1}$ şi $w = (w_0, \dots, w_{n-1})$ este o permutare a secvenţei $v = (v_0, \dots, v_{n-1})$.

Notaţii:
$sorted(\mathtt{a})$: tabloul `a` este sortat, i.e. $\mathtt{a}[0] \leq \cdots \leq \mathtt{a}[n-1]$
$\mathrm{perm}(v, w)$: w este o permutare a lui v

Sortare prin interschimbare (BubbleSort) 1/2
Metoda bubble sort se bazează pe următoarea definiţie a predicatului *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**.)
Dacă a[i] > a[i + 1], spunem că perechea $(i, i+1)$ formează o **inversiune**. În cazul unei inversiuni proprietatea a[i] ≤ a[i + 1] poate fi stabilită printr-o interschimbare. De aici derivă foarte simplu un algoritm care restabileşte relaţia de ordine corectă între elementele care formează inversiuni:

```
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!)

Segmentul din tablou de la sfârşit care nu include versiuni la o ieraţile while, nu va include inversiuni nici la iteraţiile următoare. De aici rezultă ca testul *nu mai există inversiuni* poate fi verificat ţinând minte poziţia 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 iniţial $(v_0, \ldots, v_{n-1})$ ordonate crescător (i.e., avem *sorted*(a[ultim+1 .. n−1]))

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

Singura instrucţiune care modifică tabloul a este `swap` şi aceasta menţine proprietatea perm(u, u'), unde u este valoarea variabilei `a` înainte de `swap` şi u' cea de după. .

Evaluarea algoritmului BubbleSort 1/2
Timp de execuţie

- dimensiune instanţă: n (= `a.size()`)
- operaţii măsurate: comparaţiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvenţa de intrare este ordonată descrescător
- numărul de comparaţii pentru acest caz este $(n-1)+(n-2)+\cdots+1 = \dfrac{(n-1)n}{2} = O(n^2)$

Sortare prin inserţie directă (InsertSort) 1/2
Principiul de bază al algoritmului de sortare prin inserţie este următorul: Se presupune că subsecvenţa `a`$[0 \mathrel{..} j-1]$ este sortată. Se caută în această subsecvenţă locul i al elementului $a[j]$ şi se inserează $a[j]$ pe poziţia i. Procesul de mai sus trebuie repetat pentru $j = 1, \ldots, n-1$:

```
for (j=1; j < n; ++j )
  inserează a[j] în a[0..j-1] a.î. sorted(a[0..j])
```

(Acesta este un pseudocod!)

Sortare prin inserţie directă (InsertSort) 2/2
Analiza domeniului problemei
Poziţia i pe care trebuie inserat $a[j]$ este determinată astfel:

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

Aşadar elementele de pe poziţiile $i..j-1$ trebui deplasate la dreapta cu o poziţie. Condiţia pentru deplasarea la dreapta este $i \geq 0 \wedge a[i] > a[j]$ ($a[j]$ cel iniţial): 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) {
    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`: este format din perm(u, v), unde u este valoarea curentă a variabilei `a` (reamintim că v este valoarea iniţială), şi din proprietatea că primele $j - 1$ sunt ordonate crescător:

Invariantul buclei `while`: elementele mutate $a[i + 1..j - 1$ sunt mai mari decât $a[j]$ iniţial (= *temp*): $a[i + 1], \ldots, a[j - 1] > temp$.

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

Evaluarea algoritmului InsertSort 2/2
Timp de execuţie

- dimensiune instanţă: n (= `a.size()`)
- operaţii măsurate: comparaţiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvenţa de intrare este ordonată descrescător
  – căutarea poziţiei i în subsecvenţa $\texttt{a}[0 \,..\, j - 1]$ necesită $j - 1$ comparaţii
- numărul de comparaţii pentru acest caz este $1 + 2 = \cdots + (n - 1) = \dfrac{(n-1)n}{2} = O(n^2)$

Selecţia sistematică
Analiza domeniului problemei Se bazează pe structura de date de tip *max-heap*. Proprietatea *maxheap*(`a`):

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

$maxheap(\texttt{a}) \implies \max \texttt{a} = \texttt{a}[0]$

Ideea algoritmului:

- se presupune *maxheap*(a)

- dacă facem interschimbarea `swap(a,0,n-1)`, noua valoarea $a[n-1]$ e pe locul ei final şi tabloul rămas de sortat este $\mathtt{a}[0..n-2]$

- $\mathtt{a}[0..n-2]$ se sortează în aceeaşi 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 proprietăţii de max-heap
Analiza domeniului problemei

- *maxheap*$(\mathtt{a}, \ell)$:

$$\begin{aligned}(\forall i \geq \ell) 2i+1 < n &\implies a[i] \geq a[2i+1] \wedge \\ 2(i+1) < n &\implies a[i] \geq a[2(i+1)]\end{aligned}$$

- $\ell \geq n/2 \implies$ *maxheap*$(\mathtt{a}, \ell)$

- dacă *maxheap*$(\mathtt{a}, \ell - 1)$ putem stabili *maxheap*$(\mathtt{a}, \ell)$ inserând $a[\ell - 1]$ în $\mathtt{a}[\ell..n-1]$

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 operaţiilor peste *max-heap. invariantul instrucţiunii* `while` *din* `insertInHeap`: $(\forall i \geq \ell)$ dacă j nu este în arborele cu rădăcina în i, atunci $\mathit{maxheap}(\mathtt{a}, i)$

invariantul lui `for` *din* `heapSort`: $\mathit{maxheap}(\mathtt{a}, \ell)$

invariantul instrucţiunii `while` *din* `heapSort`: $\mathit{maxheap}(\mathtt{a}[0..r-1]) \wedge \mathit{sorted}(\mathtt{a}[r..n-1])$

Evaluarea algoritmului HeapSort 2/2
Timp de execuţie

- dimensiune instanţă: n $(= \mathtt{a.size}())$
- operaţii măsurate: comparaţiile care implică elementele tabloului
- cazul cel mai nefavorabil: greu de spus
  – complexitatea timp al operaţiei `insertInHeap`: $O(\log(n-\ell))$
  – dar construcţia max-heap-ului necesită $O(n \log n) = O(\log \frac{n-1}{2}) + \cdots + O(\log n)$ (se poate arăta că de fapt e $\Theta(n)$, a se vedea Cormen et al., 6.3)
  – complexitatea lui `while`: $O(\log(n-1)) + O(\log(n-2)) + \cdots + O(\log 1) = O(n \log n)$
- numărul de comparaţii pentru acest caz este $O(n\ \log n)$

Alţi algoritmi de sortare
Exerciţii pentru seminar.

Două întrebări despre algoritmii de sortare
Algoritmii de sortare prezentaţi până acum se bazează pe executarea a două operaţii primitive: compararea şi interschimbarea a două elemente. Deoarece orice interschimbare este, în general, precedată de o comparaţie (prin care se decide dacă interschimbarea este necesară) putem spune că operaţiile de comparare domină calculul oricărui algoritm prezentat până acum.

Ne punem următoarele două întrebări:

- care este numărul minim de comparaţii executate în cazul cel mai nefavorabil?

- care algoritmi de sortare realizează minimul de comparaţii, 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 construiţi aceşti algoritmi.

Arborii de decizie pentru sortare: intuitiv
Pentru simplitate vom presupune $a_i \neq a_j$ dacă $i \neq j$. Deoarece răspunsul dat de comparaţia $i\,?\,j$ are numai două posibilităţi de alegere, rezultă că putem reprezenta cele două multi-mulţimi de comparaţii prin intermediul unui arbore binar:

- vârfurile interne conţin comparaţii $i\,?\,j$;
- subarborele din stânga conţine comparaţiile făcute în cazul $a_i < a_j$;
- subarborele din dreapta conţine comparaţiile făcute în cazul $a_i > a_j$;
- vârfurile externe (frontiera) conţin permutări

Algoritmi reprezentaţi ca arborii de decizie (pentru sortare)

Definition
Un **arbore de decizie** pentru n elemente este un arbore binar în care vârfurile interne sunt etichetate cu perechi de forma $i\,?\,j$, iar vârfurile de pe frontieră sunt etichetate cu permutări ale mulţimii $\{0, 1, \ldots, n-1\}$.

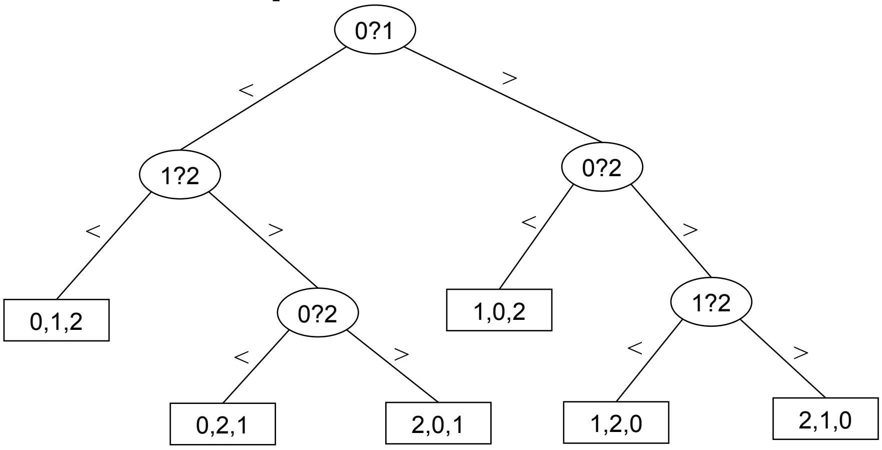
Definition
Fie t un arbore de decizie de dimensiune n şi secvenţa $a = (a_0, \ldots, a_{n-1})$. **Calculul** lui t pentru intrarea a constă în parcurgerea unui drum de la rădăcină la un vârf de pe frontieră definit astfel:

- Iniţial se pleacă din rădăcină.
- Presupunem ca vârful curent este $i\,?\,j$. Dacă $a_i < a_j$ atunci copilul din stânga lui $i\,?\,j$ devine vârf curent; altfel copilul din dreapta devine vârf curent.
- Calculul se opreşte dacă vârful curent este pe frontieră.

Arbori de decizie pentru sortare

Definition
Fie t un arbore de decizie pentru n elemente. Spunem că t **rezolvă problema sortării** dacă pentru orice intrare $a = (a_0, \ldots, a_{n-1})$, calculul lui t pentru a se termină într-un vârf cu permutarea π astfel încât $a_{\pi(0)} < \cdots < a_{\pi(n-1)}$. Un arbore de decizie care rezolvă problema sortării va mai fi numit şi **arbore de decizie pentru sortare** iar modelul de calcul va fi numit **modelul arborilor de decizie pentru sortare**.

Arborele de decizie pentru InsertSort



Complexitatea sortării
Notaţii:
$ADS(n)$ = mulţimea arborilor de decizie pentru sortarea secvenţelor de lungime n
$Fr(t)$ = frontiera arborelui de decizie t
$length(\pi, t)$ = lungimea în t de la rădăcina la $\pi \in Fr(t)$ Putem defini acum timpul de execuţie minim pentru cazul cel mai nefavorabil prin expresia:

$$T(n) = \min_{t \in ADS(n)} \max_{\pi \in Fr(t)} length(\pi, t)$$

unde length(π, t) reprezintă lungimea drumului de la rădăcină la vârful pe frontieră etichetat cu π în arborele de decizie pentru sortare t.

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

Demonstraţie. Un arbore de decizie de dimensiune n care rezolvă problema sortării are $n!$ vârfuri pe frontieră. Un arbore de înălţime k are cel mult 2^k vârfuri pe frontieră. De aici rezultă

$$2^{T(n)} \geq n!$$

care implică $T(n) \geq \log_2(n!) = \Theta(n \log_2 n)$. sfdem

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

4 Complexitatea căutării divide-et-impera

Problema căutării

Instance o mulţime univers $\mathcal{U}$, o submulţime $S \subseteq \mathcal{U}$ şi un element a din $\mathcal{U}$;
Question $a \in S$?

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

Algoritm generic divide-et-impera de căutare: ideea
Mai întâi generalizăm problema presupunând că se caută a în secvenţa $(s[p], \ldots, s[q])$. Reamintim că are loc $s[p] < \cdots < s[q]$. Algoritmii de căutare bazaţi pe paradigma divide-et-impera au o descriere recursivă definită după următoarea strategie:

- 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 subsecvenţa $(s[p], \ldots, s[m-1])$;
- dacă $a > s[m]$ atunci căutarea continuă cu subsecvenţa $(s[m+1], \ldots, s[q])$;

În funcţie de modul de alegere a valorii m prin instrucţiunile 2 şi 5, se disting mai mulţi algoritmi de căutare. Cei mai cunoscuţi dintre acestia sunt:

- **Căutare liniară** (secvenţială). Se alege $m = p$.
- **Căutare binară**. Se alege $m = \lceil \frac{p+q}{2} \rceil$.
- **Căutare Fibonacci**. Se presupune $q + 1 - p = Fib(k) - 1$ unde $Fib(k)$ este la k-lea număr Fibonacci. Se alege m astfel încât $m-p = Fib(k-1)-1$ şi $q - m = Fib(k-2) - 1$.

Algoritm generic divide-et-impera de căutare

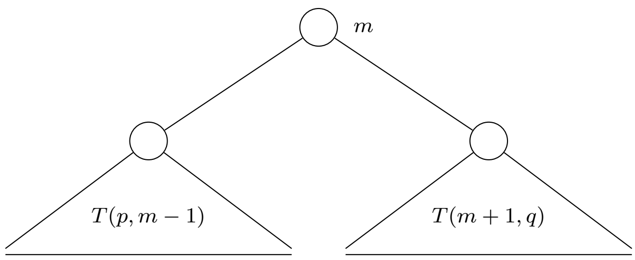
```
pos(s, n, a) {
  p = 0; q = n - 1;
  2: alege  m între  p şi  q
  while ( (a != s[m]) && (p < q)) {
    if (a < s[m]) q = m -1; else  p = m + 1;
    5: alege  m între  p şi  q
  }
  if (a == s[m]) return m; else return -1;
}
```

Algoritmi reprezentaţi ca arbori de decizie (pentru căutare)

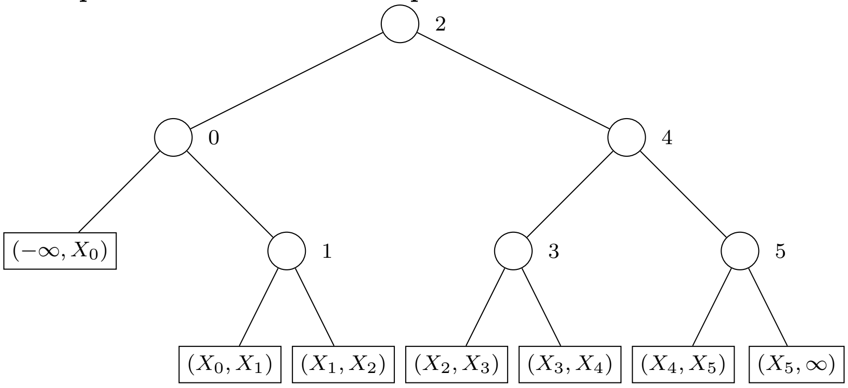
Definition
Arborele de decizie pentru căutare de dimensiune n ataşat unui algoritm bazat pe metoda divide-et-impera este definit după cum urmează:

- Mai întâi se defineşte recursiv arborele $T(p, q)$ astfel:
  - dacă $p > q$ atunci $T(p, q)$ este arborele vid;
  - altfel, rădăcina este m calculat de instrucţiunea 2 sau 5, iar subarborele stâng este $T(p, m - 1)$ şi cel drept este $T(m + 1, q)$.
- Arborele de decizie pentru căutare de dimensiune n este $T(0, n-1)$ la care se adaugă vârfurile externe având ca etichete intervalele $(-\infty, X_0), (X_0, X_1), \ldots, (X_{n-1}, +\infty)$ în această ordine de la stânga la dreapta, unde $X_0, \ldots, X_{n-1}$ sunt n variabile.

$T(p,q)$ grafic



Exemplu de arbore de decizie pentru căutarea binară



Algoritmi reprezentaţi ca arbori de decizie (pentru căutare)

Definition

Calculul unui arbore de decizie pentru intrarea $x_0, \ldots, x_{n-1}, a$, unde $x_0 < \cdots < x_{n-1}$, constă în:

1. etichetarea nodurilor interne cu $x_0, \ldots, x_{n-1}$ astfel încât lista inordine ne dă ordinea crescătoare a etichetelor, şi
2. parcurgerea unui drum de la rădăcină spre frontieră determinat astfel: dacă vârful curent v este etichetat cu x_m (reamintim că m este dat de instrucţiunea 2 sau 5 din schema procedurală divide-et-impera) atunci:
   (a) dacă v este vârf extern etichetat cu (X_i, X_{i+1}) atunci $a \in (x_i, x_{i+1})$ (variabila X_i este interpretată ca având valoarea x_i) şi calculul se **termină cu insucces**;
   (b) dacă $a = x_m$ atunci calculul se **termină cu succes**;
   (c) dacă $a < x_m$ atunci rădăcina subarborelui stâng devine vârf curent;
   (d) dacă $a > x_m$ 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 înălţimea lui t este h.

Demonstraţie. Procedăm prin inducţie după n. Dacă $n = 1$, atunci sfirmaţia din lemă este evident adevărată. Presupunem $n > 1$. Valoarea m corespunzătoare rădăcinii este $\lceil \frac{n}{2} \rceil$. Din definiţia părţii întregi superioare rezultă următoarele inegalităţi:

$$2^{h-2} \leq m < 2^{h-1} + 1. \tag{1}$$

Subarborii rădăcinii au m şi respectiv $n - m - 1$ vârfuri astfel încât $m - 1 \leq n - m - 1 \leq m$. Dacă $m = n - m - 1$, atunci $m = \frac{n-1}{2}$. Deoarece $n \leq 2^h - 1$, rezultă că $m \leq 2^{h-1} - 1 < 2^{h-1}$. Dacă $m - 1 = n - m - 1$, atunci $m = \frac{n}{2} < 2^{h-1}$. Rezultă $m < 2^{h-1}$ în toate cazurile. Aplicând ipoteza inductivă, rezultă că subarborele cel mai înalt (cel cu m vârfuri şi aflat la stânga rădăcinii) are înălţimea $h - 1$. Din definiţia înălţimii arborelui binar, rezultă că înălţimea lui t este h. $\boxed{\text{sfdem}}$

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

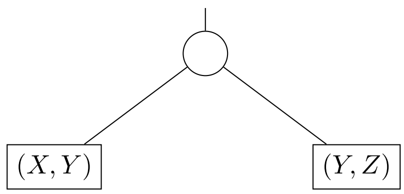
Proprietăţi ale arborilor de decizie pentru căutare

Definition
Fie t un arbore de decizie pentru căutare. **Lungimea internă** a lui t, notată IntLength(t), este suma lungimilor drumurilor de la rădăcină la vârfurile interne. **Lungimea externă** a lui t, notată ExtLength(t), este suma lungimilor drumurilor de la rădăcină la vârfurile de pe frontieră (pendante).

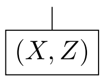
Lemma
Fie t un arbore de decizie pentru căutare cu n vârfuri interne. Atunci:

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

Demonstraţie. Procedăm prin inducţie după n. Pentru $n = 1$ avem IntLength$(t) = 0$ şi ExtLength$(t) = 2$. Presupunem $n > 1$. Fie i un vârf intern cu ambii fii pe frontieră. Înlocuim subarborele:



cu subarborele:



Noul arbore t' este un arbore de decizie cu $n-1$ vârfuri interne şi conform ipotezei inductive avem

$$\text{ExtLength}(t') - \text{IntLength}(t') = 2(n-1).$$

Deoarece $\text{ExtLength}(t) = \text{ExtLength}(t') + k + 2$ şi $\text{IntLength}(t) = \text{IntLength}(t') + k$, unde k este lungimea drumului de la rădăcină la i, rezultă:

$$\begin{aligned} \text{ExtLength}(t) - \text{IntLength}(t) &= \text{ExtLength}(t') + k + 2 - (\text{IntLength}(t') + k) \\ &= 2n - 2 + k + 2 - k \\ &= 2n. \end{aligned}$$

$\boxed{\text{sfdem}}$

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

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

unde $h = \lceil \log_2(n+1) \rceil$.

Demonstraţie. Considerăm formula:

$$x + x^2 + \cdots + x^k = \frac{x^{k+1} - x}{x-1}.$$

Derivăm:

$$1 + 2x + \cdots + kx^{k-1} = \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Înmulţim cu x:

$$x + 2x^2 + \cdots + kx^k = x \cdot \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Luăm $k = h - 1$ şi $x = 2$:

$$2 + 2 \cdot 2^2 + \cdots + (h-1) \cdot 2^{h-1} = 2^h(h-2) + 2.$$

Lungimea internă minimă a unui arbore de decizie este suma primilor n termeni din seria:

$$0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + \cdots$$

şi corespunde arborelui binar complet. Dacă presupunem $n + 1 = 2^h$ atunci această sumă este egală cu:

$$2 + 2 \cdot 2^2 + \cdots + (h-1)2^{h-1} = 2^h(h-2) + 2.$$

Pentru cazul general trebuie să scădem suma drumurilor care unesc rădăcina cu vârfurile de pe nivelul h care lipsesc:

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

$\boxed{\text{sfdem}}$

Corollary
Lungimea externă minimă a unui arbore de decizie este

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

Complexitatea căutării

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

Demonstraţie. Arborele binar cu lungimea internă minimă considerat în lema 43 are şi înălţime minimă. Acum concluzia teoremei rezultă din faptul că $2^{h-1} \leq n < 2^h$, unde n este numărul de noduri iar h este înălţimea arborelui. $\boxed{\text{sfdem}}$

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

Demonstraţie. Se observă că arborele de decizie asociat căutării binare are lungimea externă minimă. $\boxed{\text{sfdem}}$

5 Reducerea polinomială a problemelor computaţionale

Motivaţie
Mentalitate: "Dacă ştiu 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.

Aplicaţii:

- proiectarea de algoritmi
- demonstrarea limitelor: daca P este dificilă atunci şi 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 instanţă 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 paşii de preprocesare şi 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 mulţime:

Input	O mulţime 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);

Deoarece algoritmul de mai sus este mai complex decât algoritmul care determină maximul enumerând toate elementele din S, rezultă că $\propto$ nu este întotdeauna o "reducere de la o problemă mai complexă la una mai simplă". De aceea termenul de *"transformare"* este mai potrivit. Menţinem totuşi şi termenul de reducere pentru că aşa este cunoscut în literatură.

Variante pentru submulţimea de sumă dată

SSD1

Input	O mulţime S de numere întregi, M număr întreg pozitiv.
Output	Cel mai mare număr întreg M^* cu proprietăţile $M^* \leq M$ şi există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance	O mulţime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.
Question	Există număr întreg M° cu proprietăţile $K \leq M^\circ \leq M$ şi $\sum_{x \in S'} x = M^\circ$ pentru o o submulţime oarecare $S' \subseteq S$?

SSD3

Instance	O mulţime S de numere întregi, M un număr întreg pozitiv.
Question	Există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

Exemplu: SSD1 $\propto$ SSD2

SSD1

Input	O mulţime S de numere întregi, M număr întreg pozitiv.
Output	Cel mai mare număr întreg M^* cu proprietăţile $M^* \leq M$ şi există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance	O mulţime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.
Question	Există număr întreg M° cu proprietăţile $K \leq M^\circ \leq M$ şi $\sum_{x \in S'} x = M^\circ$ pentru o o submulţime 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 mulţime S de numere întregi, M număr întreg pozitiv.
Output	Cel mai mare număr întreg M^* cu proprietăţile $M^* \leq M$ şi există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.
SSD2	
Instance	O mulţime S de numere întregi, M, K două numere întregi pozitive cu $K < M$.
Question	Există număr întreg M° cu proprietăţile $K \leq M^\circ \leq M$ şi $\sum_{x \in S'} x = M^\circ$ pentru o o submulţime 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 mulţime S de numere întregi, M număr întreg pozitiv.
Output	Cel mai mare număr întreg M^* cu proprietăţile $M^* \leq M$ şi există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.
SSD3	
Instance	O mulţime S de numere întregi, M un număr întreg pozitiv.
Question	Există o submulţime $S' \subseteq S$ 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 şi 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 instanţă 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 acelaşi 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 mulţime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.
Question	Există număr întreg M° cu proprietăţile $K \leq M^\circ \leq M$ şi $\sum_{x \in S'} x = M^\circ$ pentru o o submulţime oarecare $S' \subseteq S$?

SSD3	
Instance	O mulţime S de numere întregi, M un număr întreg pozitiv.
Question	Există o submulţime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

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

Exemplu: 3-SUM $\propto$ 3-COLLINEAR

3-SUM	
Instanţă	O mulţime de n întregi.
Întrebare	Există 3 numere ale căror sumă este 0?

3-COLLINEAR	
Instanţă	O mulţime cu n puncte în plan.
Întrebare	Există 3 puncte coliniare (pe aceeaşi linie)?

3-SUM $\propto$ 3-COLLINEAR:

1. se consideră la intrare o instanţă $S = \{a_0, a_1, \ldots, a_{n-1}\}$ a lui 3-SUM;
2. calculează $t(S) = \{(a_0, a_0^3), (a_1, a_1^3), \ldots, (a_{n-1}, a_{n-1}^3)\}$
3. întoarce rezultatul întors de un algoritm care rezolvă 3-COLLINEAR pentru instanţa $t(S)$.

Lemma 1. *Dacă a, b, c sunt distincte, atunci $a + b + c = 0$ ddacă (a, a^3), (b, b^3) şi (c, c^3) sunt coliniare.*

Reducerea: proprietăţi

Theorem
a) Dacă P are complexitatea timp $\Omega(f(n))$ şi $P \propto_{g(n)} Q$ (versiunea Karp) atunci Q are complexitatea timp $\Omega(f(n) - g(n))$.[2ex] b) Dacă Q are complexitatea $O(f(n))$ şi $P \propto_{g(n)} Q$ (versiunea Karp) atunci P are complexitatea $O(f(n) + g(n))$.