Determinism

Algoritm determinist:

- fiecare actiune/operatie are un rezultat unic determinat
- **serialitate**: pentru orice moment de timp t din cursul executiei exista o singura actiune efectuata la momentul t

Algoritm nedeterminist:

- exista actiuni/operatii al caror rezultat nu este unic definit ci are valori intr-o multime **finita** de posibilitati
- **paralelism**: structura arborescenta de operatii; operatiile de pe o cale din arbore sunt efectuate serial; operatiile de pe cai diferite sunt efectuate în paralel; la un moment de timp t se executa mai multe actiuni pe diverse cai
- nu are implementare practica

Operatii caracteristice algoritmilor nedeterministi

- <u>choice(A)</u> – ramifica copia curenta a algoritmului in cardinal(A) copii
  - A = multime **finita!**
  - pt fiecare valoare din A - copie a algoritmului care continua cu acea valoare
  - variabilele locale ale algoritmului sunt clonate pt fiecare copie
  - copiile continua in paralel si independent una de alta
- <u>fail</u> – copia curenta se termina cu insucces; restul copiilor continua executia
- <u>success</u> – copia curenta se termina cu succes; celelalte copii sunt terminate (practic executia intregului algoritm se termina cu succes)

Obs1: valoarea de return a algoritmului este success (cand una din cai se termina cu success) sau fail (cand toate caile se termina cu fail) => algoritmii nedeterministi de decizie sunt functionali.
Obs2: algoritmii de optim pot fi modelati ca apeluri succesive ale unor **algoritmi de decizie** (de exemplu determinarea arborelui de acoperire minim pe un arbore fara costuri pe muchii: exista arbore de acoperire din 1 muchie? daca nu, exista din 2 muchii? etc)

Complexitatea temporala a unui algoritm nedeterminist (s.n. <u>complexitate angelica</u>)

- suma complexitatilor operatiilor din secventa/calea cea mai **scurta** care termina algoritmul cu **success**
- daca toate caile intorc fail, complexitatea este suma complexitatilor de pe calea cea mai lunga incheiata cu fail

Obs: choice(A) are complexitate O(1).

Etape in executia unui algoritm nedeterminist

- **generare** (choice – genereaza cate o copie pt fiecare candidat la a fi solutie)
- **testare** (fiecare candidat generat este testat daca e o solutie corecta)

Obs: partea nedeterminista a algoritmului corespunde etapei de generare.

Exemple de algoritmi nedeterministi

- cautarea unui element intr-un vector
- test daca un numar natural este neprim

- sortarea unui vector de elemente strict pozitive
- plasarea reginelor pe tabla de sah

Cautarea unui element intr-un vector

```
// V = vectorul, n = nr de elemente din vector, e = elementul cautat
caut(V, n, e) {
	i = choice(1..n)	// generare - pozitia elementului cautat
	if (V[i]=e) success	// testare - este elementul cautat?
	fail
}
```

Test daca un numar natural este neprim

```
// n = numarul testat
neprim(n) {
	i = choice(2..floor(sqrt(n)))	// generare – posibilii divizori
	if (n mod i=0) success		// testare – este i divizor pentru n?
	fail
}
```

Sortarea unui vector de elemente strict pozitive

```
// V = vectorul, n = nr de elemente din vector
sort(V, n) {
	for i=1 to n Aux[i]=0		// in Aux vom crea vectorul sortat

	for i=1 to n {
		j = choice(1..n)	// generare - pozitia ocupata de V[i]
		if (Aux[j]>0) fail	// testare - nu e deja ocupata?
		Aux[j] = V[i]
	}

	for i=1 to (n-1)
		if (Aux[i]>Aux[i+1]) fail	// testare - e sortat?

	success				// daca niciun test nu a dat fail, success
}
```

Plasarea reginelor pe tabla de sah

```
// Sol = vectorul de solutii, i = coloana, j = linia
ataca (Sol, i, j) {
	for k=1 to (i-1) {
		if (Sol[k]=j) return true
		if (|Sol[k]-j| = |k-i|) return true
	}

	return false
}

// n = dimensiunea tablei de sah
regine (n, Sol) {
	for i=1 to n {
		j=choice(1..n)		// generare - linia reginei de pe col i
		if (ataca(Sol,i,j)) fail	// testare - se ataca?
		Sol[i] = j
	}
	success
}
```

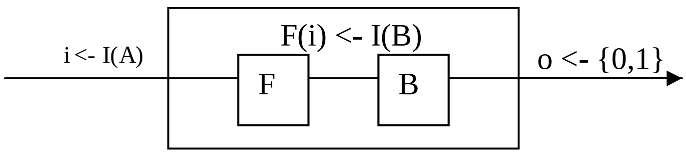
Exercitiu: care este complexitatea temporala/spatiala a algoritmilor nedeterministi de mai sus?

Tractabilitate
P = PTIME = clasa problemelor rezolvabile prin algoritmi deterministi polinomiali
NP = NPTIME = clasa problemelor rezolvabile prin algoritmi nedeterministi polinomiali

Obs: NP nu inseamna ca nu este P! NP vine de la “nondeterministic polynomial time”, nu de la “not P”. In fapt, $P \subseteq NP$ (orice algoritm determinist polinomial poate fi usor transformat intr-un algoritm nedeterminist polinomial), iar daca P = NP ramane in continuare o problema deschisa (cel mai plauzibil este ca nu sunt egale, insa nu s-a putut demonstra inca).

Problema tractabila: rezolvabila printr-un algoritm determinist polinomial
Problema intractabila: toti algoritmii deterministi care o rezolva sunt supra-polinomiali

Reducerea polinomiala



A = algoritm pt problema PA (pe care o reduc)
B = algoritm pt problema PB (la care reduc)
F = algoritm **determinist polinomial** de transformare a intrarilor pt A in intrari pt B

Daca exista F algoritm determinist cu complexitate polinomiala care transforma **orice** instanta i a problemei PA intr-**o** instanta F(i) a problemei PB a.i. **A(i)=1** $\Leftrightarrow$ **B(F(i))=1**, atunci spunem ca PA se reduce polinomial la PB: **PA $\leq_p$ PB**

Obs0: diferenta dintre reducerea polinomiala si reducerea Turing este ca aici F trebuie sa fie determinist polinomial. Observatiile 1 si 2 sunt identice cu cele de la reducerea Turing, insa merita readuse in atentie.

Obs1: faptul ca reducerea nu merge (neaparat) si de la PB la PA se vede in sublinierea cuvintelor “orice” si “o”. Cand reducem PA la PB luam o intrare oarecare pt PA si **construim** in mod convenabil o intrare pt PB.

Obs2: demonstratia faptului ca o problema se reduce polinomial la alta nu e completa daca nu demonstrati **implicatia in ambele sensuri**! Intai trebuie demonstrat ca A(i)=1 => B(F(i))=1, apoi ca B(F(i))=1 => A(i)=1.

Consecinte ale stabilirii unei relatii de reducere polinomiala:
1) $\mathbf{A \leq_p B \wedge B \in P \Rightarrow A \in P}$ (complex(F) + complex(alg(B)) polinomiala)
2) $\mathbf{A \leq_p B \wedge A \notin P \Rightarrow B \notin P}$ (altfel ar deveni si A tractabila)

Aceste consecinte se folosesc pt a demonstra apartenenta unor probleme la o anumita clasa de probleme, folosind reduceri (de) la probleme a caror (in)tractabilitate este cunoscuta.

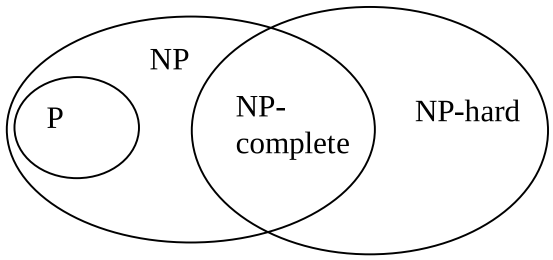
Clase de probleme

P = { A | ∃ un algoritm determinist polinomial care rezolva A }

NP = { A | ∃ un algoritm nedeterminist polinomial care rezolva A }

NP-hard = { A | $\forall$ Q $\in$ NP • Q $\leq_p$ A }

NP-complete = { A | A$\in$ NP $\wedge$ A$\in$ NP-hard }



Strategii de a demonstra apartenenta la o clasa de probleme

- **P:** se construieste un algoritm determinist polinomial
- **NP:** se construieste un algoritm nedeterminist polinomial SAU se arata ca solutia poate fi verificata in timp polinomial
- **NP-hard**: se gaseste o problema cunoscuta ca NP-hard si se reduce aceasta problema la problema curenta (nu invers!)
- **NP-complete**: se arata ca e NP si NP-hard