Algoritmi ricorsivi su alberi

Algoritmi ricorsivi su alberi: dimensione

Calcolo della dimensione d = numero di nodi

- Caso base: albero vuoto: d = 0
- Caso induttivo: d = 1 + dimensione(sottoalbero sx) + dimensione(sottoalbero dx)

```
Dimensione( u ):
  IF (u == null) {
    RETURN 0;
  } ELSE {
    dimensioneSX = Dimensione( u.sx );
    dimensioneDX = Dimensione( u.dx );
    RETURN dimensioneSX + dimensioneDX + 1;
  }
```

invocare con
u = radice

Profondità di un nodo

- Radice ha profondità pari a 0
- I suoi figli hanno profondità pari a 1, ecc.
- Se un nodo ha profondità pari a p,
  i figli hanno profondità pari a p+1

```
p = 0;
WHILE (u.padre != null) {
  p = p + 1;
  u = u.padre;
}
```

Algoritmi ricorsivi su alberi: altezza

altezza = max profondità raggiunta dalle foglie
(massima distanza delle foglie dalla radice)

Calcolo efficiente dell'altezza h:

- Caso base: foglia ha h=0 (ma non lo usiamo….)
- Passo induttivo: h = 1+ max altezza dei figli
- Caso base per **null** --> h = -1 (usiamo questo!)

```
Altezza( u ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    altezzaSX = Altezza( u.sx );
    altezzaDX = Altezza( u.dx );
    RETURN max( altezzaSX, altezzaDX ) + 1;
  }
```

invocare con
u = radice

Problema decomponibile
(divide et impera sugli alberi)

Codici precedenti: stessa struttura!

- **Caso base**: per u = **null** o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli u.sx e u.dx
- **Ricombinazione**: ottieni il risultato con Ricombina

```
Decomponibile(u):
  IF (u == null) {
    RETURN Decomponibile(null);
  } ELSE {
    risultatoSX = Decomponibile(u.sx);
    risultatoDx = Decomponibile(u.dx);
    RETURN Ricombina(risultatoSX, risultatoDx);
  }
```

invocare con u = radice

Problemi decomponibili
(divide et impera sugli alberi)

Analisi di complessità per un albero di n nodi:

se Ricombina richiede R tempo:

Decomponibile richiede **O(R n) tempo**

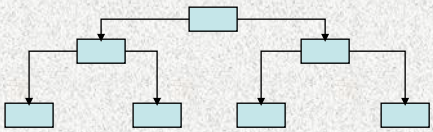
(nei nostri esempi, *R* = *costante*)

```
Decomponibile(u):
  IF (u == null) {
    RETURN Decomponibile(null);
  } ELSE {
    risultatoSX = Decomponibile(u.sx);
    risultatoDx = Decomponibile(u.dx);
    RETURN Ricombina(risultatoSX, risultatoDx);
  }
```

Problema decomponibile: albero T completamente bilanciato

- Albero *binario* **completo** ≡ ogni nodo interno ha **sempre due figli non vuoti**
- Albero **completamente bilanciato** ≡ **completo** e tutte le **foglie** hanno la **stessa profondità**

Esempio:



- T(u) = sottoalbero di T radicato in u

Albero completamente bilanciato

- Problema: algoritmo che restituisce TRUE se un albero è completamente bilanciato
  - Soluzioni?
  - Ricorsivamente?
  - È sufficiente testare che il sottoalbero sx e dx siano *completamente bilanciati*?
    - Entrambi devono *anche* avere la stessa profondità

Problema decomponibile: bilanciato

- Risolviamo un problema più generale per T(u), calcolandone *anche* l'altezza oltre che a dire se è completamente bilanciato o meno.
- La ricorsione restituisce una coppia <booleano, altezza>
- Tempo di risoluzione: O(n) tempo per n nodi

```
CompletamenteBilanciato( u ):
  IF (u == null) {
    RETURN <TRUE, -1>;
  } ELSE {
    <bilSX,altSX> = CompletamenteBilanciato( u.sx );
    <bilDX,altDX> = CompletamenteBilanciato( u.dx );
    completamenteBil = bilSX && bilDX && (altSX == altDX);
    altezza = max(altSX, altDX) + 1;
    RETURN <completamenteBil,altezza>;
  } ⟨post: restituisce TRUE come prima componente ↔ T(u) è completamente bilanciato⟩
```

FINE

Lucidi tratti da
Crescenzi • Gambosi • Grossi,
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2006
http://algoritmica.org