

## Nota

Questi lucidi non sono da considerare come sostitutivi né dei testi di riferimento né degli appunti presi alle lezioni, e non sono quindi assolutamente sufficienti per la preparazione dell'esame. Sono piuttosto da considerare una traccia dettagliata degli argomenti trattati a lezione. Bisogna anche tener conto del fatto che parte del materiale inserito nei lucidi è protetto da copyright e può essere reso disponibile ad uso personale e solo per scopi didattici (si veda la nota successiva relativa alle video lezioni ). Non è quindi possibile ridistribuire questo materiale e fornire ad altri la propria password di accesso.

September 16, 2023

1 / 519

# Algoritmi e Strutture Dati

AA. 2022-23

# Algoritmi e Strutture Dati

- Orario: 3 ore lunedì, 2 ore martedì
- Programma: = 90 % anni scorsi
- Libro adottato: Cormen et al. *Introduzione agli algoritmi e strutture dati* 3a Ed. McGraw-Hill
- Verifica:
  - ① 1 prova intermedia (data da confermare): 31 Ottobre
  - ② 2 prova intermedia (data da confermare): 12 Dicembre
  - ③ Verifica: prova scritta (o due prove intermedie) + orale
- Ric. studenti: informazioni nella pagina del docente (Cercachi UNIFI) ed eventuali modifiche negli avvisi sulla pagina del corso di studi

# Algoritmi e Strutture Dati vs Lab. Algoritmi

## Laboratorio di Algoritmi

- Non ci sono lezioni frontali (lavoro a casa)
- Alcuni argomenti saranno trattati nelle lezioni di algoritmi (probabilmente alla fine del corso)
- Esercizi pratici e semplici programmi Python
  - ▶ Consegnata su Moodle nella pagina di Algoritmi e Strutture Dati

## Informazioni Generali

- Comunicazioni: sito web, pinco.pallino at stud.unifi.it durante ricevimento studenti
- Precedenze d'esame
  - ▶ Per **partecipare alla prova scritta** è necessario aver superato il corso integrato "Fondamenti di Informatica / Programmazione" ed "Analisi Matematica I" (o "Elementi di Analisi Matematica")
  - ▶ Dettagli nella pagina web del corso

# Verifica Laboratorio di Algoritmi

- Laboratorio di Algoritmi (a partire dall'appello successivo)
  - ① Esercizi da svolgere a casa: esperimenti con relazione su esercizi indicati dal docente
  - ② Orale: condivisione dell'ambiente di sviluppo online e discussione del codice
  - ③ Dettagli nella pagina Moodle del corso
- Chiave di accesso Moodle:
  - ▶ ASD: AlgMor003

## Algoritmo - in generale

Descrizione di una sequenza di azioni che un esecutore deve compiere per risolvere un problema

### Esempio (ricetta)

**Problema:** cucinare un dolce

**Input:** ingredienti

**Esecutore:** cuoco

**Algoritmo:** ricetta

**Output:** piatto cucinato

### Altro esempio (calcolo $R_t$ )

**Problema:** calcolare il tasso di contagio in una pandemia

**Input:** dati sul numero di contagiati

**Algoritmo:** varie alternative (es: ▶ Sito INFN)

**Output:** valore  $R_t$

## Algoritmo - definizione più formale

Una procedura di calcolo ben definita che prende un valore (o un insieme di valori) in **ingresso** e genera un valore (o un insieme di valori) come **uscita**

Risolve uno specifico **problema computazionale** per **ogni** possibile istanza

## Problema computazionale

- La sua **descrizione** specifica in termini generali la relazione ingresso/uscita desiderata
- L'**algoritmo** descrive un modo per ottenere tale relazione
- Un **programma** infine implementa l'algoritmo

## Problemi computazionali

### Problema computazionale: descrizione

- La **descrizione** specifica in termini generali la relazione ingresso/uscita desiderata
- Viene descritto in genere in termini **matematici**

### Problema computazionale: algoritmo

- L'**algoritmo** descrive un modo per ottenere tale relazione
- Viene descritto in genere con **pseudo-codice**

### Problema computazionale: programma

- Un **programma** infine implementa l'algoritmo
- Viene descritto in genere con un programma in un linguaggio di **alto livello** (es. C, C++, Python)

# Problemi computazionali: esempi

## Minimo di un insieme ( $S$ )

$$m = \min(S)$$

$$\Rightarrow m \in S : \forall s \in S : m \leq s$$

**Input:**  $S$     **Output:**  $m$

## Ricerca in un insieme

$$i = \text{Ricerca}(S, v)$$

$$\Rightarrow i \in \{1, \dots, n\} : S_i = v; 0 \text{ altrimenti}$$

**Input:**  $S$     **Output:**  $i$

## Ordinamento

**Input:** una sequenza di  $n$  numeri  $\langle a_1, a_2, \dots, a_n \rangle$

**Output** una permutazione  $\langle a'_1, a'_2, \dots, a'_n \rangle$  della sequenza di input tale che  $\langle a'_1 \leq a'_2 \leq \dots \leq a'_n \rangle$

# Algoritmi: esempi

## Algoritmi ?! per alcuni degli esempi

- **Minimo:**

Confronta ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo

- **Ricerca:**

Confronta  $v$  con tutti gli elementi di  $S$  e restituisci la posizione corrispondente se trovato;  
restituisci 0 se nessun elemento corrisponde

- **Ordinamento:**

Prendi il valore più piccolo lo metti da parte, poi passi al successivo fino a terminare tutti i valori

# Problemi ...

## Descrizione

- Algoritmi descritti in linguaggio naturale non sono precisi
  - ▶ Serve un linguaggio più formale indipendente dall'implementazione (programma)

## Valutazione algoritmi

- Esistono algoritmi “migliori” di quelli proposti?
  - ▶ Come valuto “migliore”?

## Caratteristiche comuni a molti problemi algoritmici

- **Molte soluzioni** tra cui scegliere la “migliore”
- **Molte applicazioni** per ogni algoritmo

## Corso focalizzato su:

### 1. Analisi degli algoritmi

- Studio *teorico* delle prestazioni e dell'utilizzo di risorse degli algoritmi
  - ▶ l'algoritmo risolve il problema in modo **efficiente**?
  - ▶ potrei fare meglio?

### 2. Studio della correttezza degli algoritmi

- Un algoritmo termina con l'output corretto *per ogni* istanza di input? (**risolve** il problema computazionale)?
  - ▶ alcuni problemi non possono essere risolti
  - ▶ alcuni problemi vengono risolti in modo approssimato

### 3. Utilizzo degli algoritmi

- Posso risolvere problemi con "piccole modifiche" di algoritmi noti
- *Scelta* di algoritmi alternativi
- Conoscere le interfacce di ingresso-uscita degli algoritmi

# Dato vs tipo di dato

## Dato

- Valore che una variabile può assumere

## Tipo di dato

- Modello matematico, costituito da:
  - ▶ un insieme di valori che la variabile può assumere
  - ▶ un insieme di operazioni ammesse su questi valori
- Si trascurano i dettagli implementativi
- Tipi di dati **primitivi** forniti dal linguaggio
  - ▶ `int (+,-,*,/,%), boolean (!, &&, ||)`
  - ▶ matrice bidimensionale (memorizzata “per righe”)

# Strutture di dati

- Estende il concetto di “tipo di dato” elementare a dati più complessi
- Un modo per memorizzare e organizzare i dati e semplificare l’accesso e la modifica
- Include:
  - ▶ organizzazione dati in memoria
  - ▶ parte algoritmica (accesso e modifica)

# Strutture di dati

## Organizzazione in memoria dei dati

Come "organizzo i byte" per memorizzare i dati

### Insieme di operatori per:

- **accedere** alla struttura
  - ▶ leggere elementi
  - ▶ cercare elementi
- **manipolare** la struttura:
  - ▶ aggiungere elementi
  - ▶ rimuovere elementi
  - ▶ modificare elementi
  - ▶ modificare la struttura
- 

## Strutture dati elementari (nei linguaggi di alto livello ed. C)

- **Vettore di memoria**: elementi dello stesso tipo
- **Array**: vettore su più dimensioni
- **Record**: elementi di tipo diverso con nome (struct del C)

## Altre strutture dati (talvolta presenti in linguaggi di alto livello - es. Dizionario in Python)

- **Insieme**
  - ▶ insieme ordinato (sequenza, lista)
  - ▶ insieme dinamico
- **Pila** (stack)
- **Coda** (queue)
  - ▶ coda con priorità
- **Dizionario**
- **Grafo**
  - ▶ albero

# Insiemi - insiemi ordinati

## Differenze tra **insiemi** in informatica e in matematica

- Un insieme matematico **non** cambia  
(per aggiungere e ad  $S$ :  $S' = S \cup e$ )
- In informatica: **insieme dinamico** (cresce, cambia nel tempo)
- Ogni elemento è rappresentato da un oggetto con:
  - ▶ **attributi** e
  - ▶ **dati satelliti**
- Diversi algoritmi possono usare diverse operazioni sugli insiemi
  - ▶ la migliore implementazione dipende dalle operazioni richieste

## Possibili operazioni per SET S

- `Size(S)`
- `Contains(S, x)` ( $x$  è un elemento)
- `Search(S, k)` ( $k$  è una chiave)
- `Insert(S, x)`
- `Delete(S, x)`
- Operazioni tipiche su insiemi
  - ▶ `Union(A, B)`
  - ▶ `Intersection(A, B)`
  - ▶ `Difference(A, B)`

## Insiemi ordinati (sequenza)

- Ordine nella sequenza è importante

## Possibili operazioni per sequenza S

- First(S)
- Last(S)
- Next(S, pos)
- Prev(S, pos)
- Insert(S, pos, x)
- Delete(S, pos)
- Read(S, pos)
- Write(S, pos, x)
- Min(S) / Max(S)

## Stringa

- Tipo particolare di insieme ordinato (contiene caratteri alfanumerici)
- Operazioni specifiche:
  - ▶ Concat(S,str)
  - ▶ Prefix(S,len)
  - ▶ Suffix(S,len)
- Varie implementazioni
  - ▶ Terminato con un NULL, come in C:
 

c	a	t	×
---	---	---	---
  - ▶ Indicando prima la lunghezza, come in Pascal:
 

3	c	a	t
---	---	---	---
  - ▶ Con lunghezza fissa:
 

c	a	t	×		
---	---	---	---	--	--
- Vantaggi e svantaggi?

## Stack e Code

- Insiemi dinamici in cui
  - ▶ la posizione di Insert
  - ▶ l'elemento rimosso da Delete
- sono predeterminati

## Stack

- *Last In, First Out (LIFO)*

## Coda

- *First In, First Out (FIFO)*

## Stack

## Operazioni

- IsEmpty(S)
- Push(S, v)
- Pop(S)
- Top(S)
- Uso:
  - ▶ Passaggio dei parametri e indirizzo di ritorno per chiamate di funzione

# Coda

## Operazioni

- IsEmpty(S)
- Enqueue(S, v)
- Dequeue(S)
- Top(S)
  
- Uso:
  - ▶ In SO gestione dei processi in attesa di utilizzare una risorsa
  - ▶ A volte coda con priorità

# Coda con priorità

## Operazioni

- Insert(Q, x)
- Delete(Q, x)
- FindMin(Q) (Max)
- ExtractMin(Q) (Max)

## Implementazione

- Ad esempio con **heap binario**
- Lo vedremo poi...

# Dizionario (Map)

- Relazione univoca (associa ad ogni elemento di  $A$  un solo elemento di  $B$ , ma non viceversa)
- Un insieme  $D$  di coppie (*chiave, valore*)
  - ▶ Relazione  $R : A \rightarrow B$
  - ▶ Insieme  $A$  è il dominio (chiavi)
  - ▶ Insieme  $B$  è il codominio (valori)

## Operazioni

- $\text{Search}(D, k)$
- $\text{Insert}(D, k, v)$
- $\text{Delete}(D, k)$

# Grafi e alberi

- Rappresentazione grafica di una interrelazione tra oggetti/concetti
- Albero è un caso particolare di grafo

## Grafi

- Insiemi di nodi e insieme di archi che connettono i nodi (es: il Web)
- Operazioni:
  - ▶ Visita: ispezione completa di tutti i nodi di un albero o di un grafo
  - ▶ Altre operazioni in seguito

## Albero ordinato (definizione formale in seguito)

- un tipo particolare di grafo
- un nodo è designato come **radice**
- i rimanenti nodi, se esistono, sono partizionati in insiemi ordinati e disgiunti, anch'essi alberi ordinati

# Alberi

Ci sono vari concetti che sono indicati come “albero”

Disambiguazione Wikipedia:

- pianta perenne dotata di fusto
- simbolo araldico e religioso (negli stemmi)
- **una struttura dati usata per organizzare un insieme di dati**
- **in matematica, un grafo non orientato connesso e aciclico**
- genealogico
- negli apparati meccanici, un organo atto a trasferire un moto
- componente della distribuzione dei motori a quattro tempi
- parte di una nave o imbarcazione a vela

Possibili implementazioni per strutture dati

## Possibili implementazioni di alcune strutture dati

Messaggio principale

- ci sono vari modi per realizzare una struttura dati
- in certi casi è preferibile un modo ...
- ... in altri un altro approccio
- la scelta dipende dall'uso
- ... ma bisogna saper scegliere

# Implementazione **Array**

Esempio in C:      `int A[10][20]`

## Memorizzazione “standard” (per righe)

Data la matrice  $A[R][C]$

$A[i][j]$  è memorizzato in:  $i \cdot C + j$   
 (primo elemento in posizione 0 )

## Matrici sparse

Memorizzo le triplettre  $(i, j, val)$  solo se  $A[i][j] \neq 0$

## Spazio occupato da matrice $n \times n$ (interi 4 byte)

- occupa  $4 \cdot n^2$  byte se rappresentata per righe
- e se rappresentata per colonne?
- occupa  $v \cdot 3 \cdot 4$  byte se è sparsa e contiene  $v$  valori
- conviene se  $v \cdot 3 < n^2$
- come memorizzo le triplettre? Con il dizionario

## Quanto può essere grande una matrice?

- esempio: matrice dei link delle pagine Web (vedi poi per grafi)
- stima: 1 miliardo di pagine Web
- matrice avrebbe  $10^9 \cdot 10^9 = 10^{18}$  elementi
- se ogni elemento occupa un byte si ha un exabyte  
 (...mega, giga, tera, peta, exa...)
- (5 exa: stima del totale di parole mai pronunciate da essere umano...)

# Implementazione Insiemi (e dizionari)

- Insieme → Collezione di oggetti
- Dizionario → Insieme di associazioni chiave-valore
- Implementazioni ...

## Sappiamo **implementare** insiemi?

- Consideriamo **solo** alcune strutture dati viste a Fondamenti di Informatica:
  - ▶ Tupla, Vettore di memoria, Lista concatenata (ordinata o no)

## Insiemi realizzati con **vettori booleani**

- Insieme
  - ▶ interi  $1 \dots n$
  - ▶ collezione di  $n$  possibili oggetti memorizzati in un vettore
- Rappresentazione dell'insieme:
  - ▶ vettore booleano con  $n$  elementi
- **Vantaggi**
  - ▶ semplice!
  - ▶ efficiente verificare se un elemento appartiene all'insieme
  - ▶ efficienti Unione e Intersezione
- **Svantaggi**
  - ▶ occupazione di memoria  $O(n)$ , indipendente dalla dimensione dell'insieme
  - ▶ molte operazioni inefficienti  $O(n)$
- Nota:  $O(n) \approx$  proporzionale con  $n$  (poi una definizione formale)

## Insiemi realizzati con liste concatenate non ordinate

### • Vantaggi

- ▶ occupazione di memoria proporzionale alla dimensione dell'insieme
- ▶ operazioni di inserimento:  $O(1)$

### • Svantaggi

- ▶ operazioni di ricerca e cancellazione:  $O(n)$
- ▶ operazioni di unione, intersezione e differenza:  $O(n \cdot m)$

## Insiemi realizzati con liste concatenate ordinate

### • Vantaggi

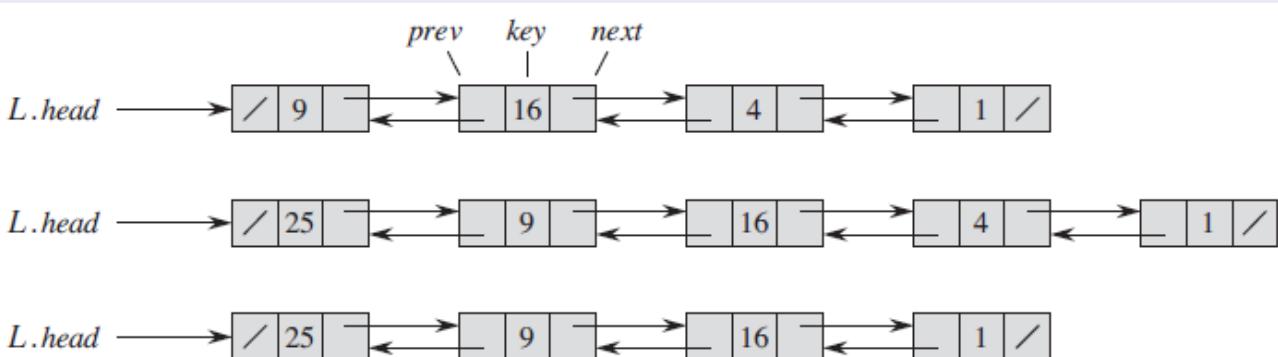
- ▶ Occupazione di memoria proporzionale alla dimensione dell'insieme
- ▶ Operazioni di unione, intersezione e differenza =  $O(n)$

### • Svantaggi

- ▶ Operazioni di ricerca, inserimento e cancellazione:  $O(n)$

## Insieme ordinato (sequenza)

### Implementazione con lista concatenata



### Varianti:

- Bidirezionale / monodirezionale
- Con sentinella / senza sentinella
- Circolare / non circolare

## Realizzazione di insiemi con strutture dati **complesse**

- Saranno presentate nel corso
- Alberi di ricerca (bilanciati)
  - ▶ ricerca, inserimento, cancellazione:  $O(\log n)$
  - ▶ viene mantenuto l'ordinamento
  - ▶ elencare tutti gli elementi:  $O(n)$
- Tabelle hash
  - ▶ ricerca, inserimento, cancellazione:  $O(1)$
  - ▶ viene perso l'ordinamento
  - ▶ elencare tutti gli elementi:  $O(m)$ , dove  $m$  è la dimensione della tabella

## Implementazione Insiemi (e dizionari)

- Insieme → Collezione di oggetti
- Dizionario → Insieme di associazioni chiave-valore
- Implementazioni ...

	search	insert	delete	min	scan	sorted
Vet. bool	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$	Si
Lista non ordinata	$O(n)$	$O(1)$ *	$O(n)$	$O(n)$	$O(n)$	No
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Si
Hash (medio) mem. interna	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	No
Hash (medio) mem. esterna	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	No
ARN	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Si

\* sapendo che l'elemento da inserire non è già presente

$N$  numero massimo di elementi **memorizzabili**,  $n$  elementi **memorizzati**  
 $m$  dimensione tabella hash.  $m \geq n$  con memorizzazione interna

# Pila / Coda

## Stack

Possibili implementazioni:

- liste bidirezionali
- vettore di memoria

## Coda

Possibili implementazioni

- Liste
  - ▶ puntatore head per estrazione
  - ▶ puntatore tail per inserimento
- Array circolari

# Dizionario

## Implementazioni

- **array ordinato**
- **array non ordinato mantenuto con la tecnica del raddoppiamento-dimezzamento**
- **struttura circolare doppiamente collegata**
- albero binario di ricerca
- albero rosso-nero
- tavola ad accesso diretto
- tavola hash con indirizzamento aperto
- tavola hash con liste di collisione
- tavola hash perfetta
- albero B
- tecniche map-reduce

# Albero

## Possibili realizzazioni

- Realizzazione con vettore dei figli. Rischio di sprecare memoria se molti nodi hanno grado minore del grado massimo
- Realizzazione con puntatori padre/primo-figlio/fratello  
Ha una lista dei figli (fratelli)
- Realizzazione con vettore dei padri  
L'albero è rappresentato da un vettore i cui elementi contengono l'indice del padre (unico)

Ne parleremo abbondantemente il seguito

# Grafo

## Rappresentazioni

- con lista di archi
- con liste di adiacenza
- con liste di incidenza
- con matrice di adiacenza
- con matrice di incidenza

Ne parleremo abbondantemente il seguito

# Sommario

## Abbiamo visto...

### • Algoritmi

- ▶ Utilità e definizioni di algoritmi
- ▶ Obiettivi del corso relativi allo studio di algoritmi

### • Strutture dati

- ▶ Stretto legame tra
  - ★ Memorizzazione dati
  - ★ Accesso ed utilizzo
- ▶ Caratteristiche principali strutture dati
- ▶ Diverse implementazioni con diverse peculiarità
- ▶ Richiamo strutture dati note

## Insertion sort e oltre...

AA. 2022-23

## Il problema dell' ordinamento

**Input:** una sequenza di  $n$  numeri  $A = \langle a_1, a_2, \dots, a_n \rangle$

**Output:** una permutazione

$A' = \langle a'_1, a'_2, \dots, a'_n \rangle$  di  $A$  t.c. :

$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

- Sequenze memorizzate in **array**
- Numeri → **chiavi**

## Descrivere algoritmi

- Italiano (Inglese)
- Pseudo-codice

## Insertion sort

**Buon** algoritmo per **pochi** elementi

Ordina un *mazzo di carte*:

- in una mano ordino le carte (all'inizio è “vuota”)
- prendo una carta e la inserisco nella posizione corretta
  - ▶ confrontandola con ogni carta nella mano
- in ogni momento le carte nella mano sono ordinate

## Pseudocodice per INSERTION-SORT

- Input:  $A[1..n]$ 
  - ▶ primo elemento: 1
  - ▶ In C, C++, Java, Python primo elemento: 0
- $A$  è **ordinato sul posto**

## INSERTION-SORT( $A$ )

```

1  for  $j \leftarrow 2$  to  $A.length$ 
2       $key \leftarrow A[j]$ 
3      // Inserisci  $A[j]$  nella sequenza ordinata  $A[1..j - 1]$ 
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] \leftarrow A[i]$ 
7           $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow key$ 

```

## Correttezza

- Un algoritmo si dice **corretto** se, per ogni istanza di input, termina con l'output **giusto**
- Mostrare che un algoritmo funziona con un determinato input **non** è una dimostrazione di correttezza!

- Per algoritmi **iterativi** si usa spesso un **invariante di ciclo**
- Per INSERTION-SORT:

*“All'inizio di ogni iterazione del ciclo for il sottoarray  $A[1..j - 1]$  è ordinato ed è formato dagli stessi elementi che erano originariamente in  $A[1..j - 1]$ , ma ordinati”*

## Correttezza con invariante di ciclo

- Bisogna **dimostrare tre cose** sull'invariante:
  - ▶ **Inizializzazione:** è vera prima della prima iterazione del ciclo
  - ▶ **Conservazione:** se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione
  - ▶ **Conclusione:** quando il ciclo termina l'invariante ci fornisce una proprietà utile per mostrare che l'algoritmo è corretto

## Analogo all'**induzione matematica**

- Inizializzazione
- Conservazione
- Conclusione non c'è nell'*induzione matematica*:  
il passo induttivo è *usato all'infinito*

## Induzione matematica

Voglio mostrare che una **proprietà**  $P(n)$  vale per **qualunque**  $n \in \mathbb{N}$

- $U = \{n \in \mathbb{N} : \text{vale } P(n)\}$
- Passo base: **dimostro** che vale  $P(1)$  (o  $P(0)$ )
- Passo induttivo **suppongo** che  $P(n)$  valga per un generico  $n$  e dimostro che vale anche  $P(n + 1)$   
 $(n \in U \Rightarrow n + 1 \in U)$
- Concludo che l'insieme  $U$  coincide con tutto l'insieme dei numeri naturali

## Esempio: serie aritmetica

$$P(n) = 1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$$

- Passo base:

$P(n)$  vera per  $n = 1$ :

$$P(1) = \frac{1 \cdot (1+1)}{2} = \frac{1 \cdot 2}{2} = 1$$

- Passo induttivo:

$$P(n+1) = 1 + 2 + \dots + n + (n+1) = P(n) + (n+1) =$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} =$$

$$\frac{(n+1)((n+1)+1)}{2} = P(n+1)$$

- Conclusione:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2} \quad \forall n \in \mathbb{N}.$$

**IC:** “All'inizio di ogni iterazione del ciclo `for` il sottoarray  $A[1..j-1]$  è ordinato ed è formato dagli stessi elementi che erano originariamente in  $A[1..j-1]$ , ma ordinati”

`INSERTION-SORT( $A$ )`

```

1  for  $j \leftarrow 2$  to  $A.length$ 
2       $key \leftarrow A[j]$ 
3      // Insert  $A[j]$  in sorted sequence  $A[1..j-1]$ 
4       $i \leftarrow j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] \leftarrow A[i]$ 
7           $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow key$ 

```

## Invariante di ciclo

**IC:** “All’inizio di ogni iterazione del ciclo *for* il sottoarray  $A[1..j-1]$  è ordinato ed è formato dagli stessi elementi che erano originariamente in  $A[1..j-1]$ , ma ordinati”

- Inizializzazione (vera prima della prima iterazione):

...

...

- Conservazione (se vera prima di iterazione  $j$  vera anche alla fine):

...

...

- Terminazione (vera dopo la fine dell’ultima iterazione):

...

...

## Python

```
for j in range(1,len(A)):
    print(A)
    key = A[j]
    i = j-1
    while i >= 0 and A[i] > key:
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

## Complessità di un algoritmo

Analisi delle **risorse** impiegate da un algoritmo per risolvere un problema, in funzione della dimensione e del tipo di input

### Risorse

- **Tempo**: per completare l'algoritmo
  - ▶ misurato con il cronometro
  - ▶ misurato contando il numero di operazioni elementari
- **Spazio**: quantità di memoria utilizzata
- **Banda**: quantità di bit spediti (algoritmi distribuiti)

### Perché stimare la complessità in tempo degli algoritmi?

- per stimare il tempo impiegato per un dato input
- per stimare il più grande input gestibile in tempi ragionevoli
- per confrontare l'efficienza di algoritmi diversi
- per ottimizzare le parti più importanti

## Problemi...

- Come definire la dimensione dell' input?
- Come misurare il tempo?

# Dimensione input

## Costo logaritmico

- numero di bit necessari per rappresentare l'input
  - ▶ Es.: moltiplicazione numeri di  $n$  bit

## Costo uniforme

- Numero di elementi nell'input
  - ▶ Es.: minimo in un vettore di  $n$  elementi
- Potrebbero esserci più valori da considerare  
(es. numero di *nodi ed archi* di un grafo)

# Definizione di tempo

## Tempo $\equiv$ wall-clock time

Dipende da:

- bravura del programmatore
- linguaggio di programmazione
- codice generato dal compilatore
- processore, memoria (cache, primaria, secondaria), sistema operativo, processi in esecuzione

## Tempo $\equiv$ numero operazioni elementari

Operazione elementare: eseguita in tempo “costante”.  
Esempi (operazioni elementari?)

- $a* = 2$  ?
- $Math.cos(d)$  ?
- $\min(A, n)$  ?

# Modello di calcolo

Rappresentazione astratta di un calcolatore

- **Astrazione**
- **Realismo**
- **Potenza matematica**

## Random Access Machine (RAM)

**Singolo processore**

- istruzioni eseguite **in successione** (no concorrenti)
- istruzioni comuni, tempo costante (no **ordinamento**)

**Memoria**

- quantità infinita di celle di dimensione finita
- accesso in tempo costante (RAM)

**Dati**

- interi e virgola mobile (no “matrice”)
- non ci preoccupiamo della *precisione*
- parole di dimensioni limitate

# Analisi di insertion sort

INSERTION-SORT( $A$ )

	Cost Times
1 <b>for</b> $j \leftarrow 2$ <b>to</b> $A.length$	$c_1 \quad n$
2 $key \leftarrow A[j]$	$c_2 \quad n - 1$
3        // Insert $A[j]$ in sorted sequence $A[1..j - 1]$	$0 \quad n - 1$
4 $i \leftarrow j - 1$	$c_4 \quad n - 1$
5 <b>while</b> $i > 0$ and $A[i] > key$	$c_5 \quad \sum_{j=2}^n t_j$
6 $A[i + 1] \leftarrow A[i]$	$c_6 \quad \sum_{j=2}^n (t_j - 1)$
7 $i \leftarrow i - 1$	$c_7 \quad \sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] \leftarrow key$	$c_8 \quad n - 1$

$t_j$ : numero di volte che viene eseguito il test del while

Tempo di esecuzione dell'algoritmo:

- Linea  $i$  impiega  $c_i$

$$\text{costo} = \sum_{\text{istruzioni}} c_i \cdot (\text{volte che viene eseguita})$$

$$T(n) = \text{tempo esecuzione di INSERTION-SORT} =$$

$$= c_1 \cdot n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

- $T(n)$  dipende da  $t_j$  che dipendono da valori in input

# Caso migliore

## Array già ordinato

- Sempre  $A[i] \leq key \Rightarrow \forall j \ t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $\Rightarrow T(n) = a \cdot n + b$   
con costanti  $a$  e  $b$  (dipendono da  $c_i$ )  
 $\Rightarrow T(n)$  è una **funzione lineare** di  $n$

# Caso peggiore

## Array ordinato al contrario

- Sempre  $A[i] > key$  nel test del while
- $\Rightarrow$  Confronta  $key$  con tutti i  $j-1$  elementi alla sinistra di  $j$
- Un altro test dopo i  $j-1$  test  $\Rightarrow t_j = j$

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j = \left( \sum_{j=1}^n j \right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n (j - 1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

Avendo posto  $k = j - 1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) =$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) =$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

- $T(n) = an^2 + bn + c$  con costanti  $a, b, c$
- $T(n)$  è una **funzione quadratica** di  $n$

## Caso peggiore

- Tempo di esecuzione più lungo per ogni input di dim  $n$ :
  - ▶ **Limite superiore** per ogni input
  - ▶ Frequenti  
Es: ricerca di un elemento assente
  - ▶ **Caso medio** spesso **altrettanto cattivo del peggiore**

## Caso medio

- Input:  $n$  numeri a caso
- In "media"  $A[j] <$  di metà degli elementi in  $A[1..j-1]$
- $\Rightarrow t_j = j/2$
- Circa metà del caso peggiore, ma sempre una funzione quadratica di  $n$

## Tasso di crescita: astrazione semplificativa

Guarda solo il **termine principale** di una formula ignorando costanti

### insertion sort

$$T(n) = an^2 + bn + c \Rightarrow an^2 \Rightarrow n^2$$

- Il tempo di esecuzione **cresce come**  $n^2$   
non è **uguale** a  $n^2$
- *Tasso di crescita:*  $n^2 \rightarrow \Theta(n^2)$
- Usato per confrontare algoritmi

### Divide et impera

## Algoritmi iterativi e ricorsivi

### Algoritmi iterativi

- Insertion sort è **incrementale**: ordinato  $A[1..j - 1]$ , si sistema  $A[j]$   
 $\Rightarrow A[1..j]$  è ordinato

### Algoritmi ricorsivi: **divide et impera**

- **divide** il problema in sottoproblemi
- sottoproblemi risolti **ricorsivamente (impera)**
- soluzioni dei sottoproblemi **combinare** per risolvere il problema originale
- **caso base**: sottoproblemi piccoli si risolvono direttamente

# Fattoriale

Il **fattoriale** di un numero naturale  $n$  è il prodotto dei numeri interi positivi minori o uguali a  $n$ :

- Iterativo:

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

- oppure:

$$n! = n \cdot (n-1) \cdots 2 \cdot 1$$

- Ricorsivo:

$$n! = \begin{cases} 1 & \text{se } n = 1 \\ n \cdot (n-1)! & \text{se } n > 1 \end{cases}$$

# Fattoriale

## Iterativo

$\text{FATT}(n)$

```

1 if  $n = 1$ 
2     return 1
3  $Fat \leftarrow 1$ 
4 for  $i \leftarrow 2$  to  $n$ 
5      $Fat \leftarrow Fat \cdot i$ 
6 return  $Fat$ 
```

## Alternativa

$\text{FATT}(n)$

```

1  $Fat \leftarrow 1$ 
2 for  $i \leftarrow 1$  to  $n$ 
3      $Fat \leftarrow Fat \cdot i$ 
4 return  $Fat$ 
```

# Fattoriale

## Altra alternativa

$\text{FATT}(n)$

```

1  $Fat \leftarrow 1$ 
2 for  $i \leftarrow n$  downto 1
3      $Fat \leftarrow Fat \cdot i$ 
4 return  $Fat$ 
```

## Ricorsivo

$\text{FATT}(n)$

```

1 if  $n = 1$ 
2     return 1
3 else
4     return  $n \cdot \text{FATT}(n - 1)$ 
```

# Fattoriale

## Fattoriale iterativo

$\text{FATT}(n)$

```

1 if  $n = 1$ 
2     return 1
3  $Fat \leftarrow 1$ 
4 for  $i \leftarrow 2$  to  $n$ 
5      $Fat \leftarrow Fat \cdot i$ 
6 return  $Fat$ 
```

## Possiamo "studiarlo"?

- E' corretto?
  - ▶ invariante di ciclo
- Quanto tempo impiega?
  - ▶ ciclo su  $n$  valori
  - ▶ operazioni costanti
  - ▶  $\rightarrow$  costa  $c \cdot n = n$

# Fattoriale

## Fattoriale ricorsivo

$\text{FATT}(n)$

```

1 if  $n = 1$ 
2     return 1
3 else return  $n \cdot \text{FATT}(n - 1)$ 

```

## Possiamo "studiarlo" ?

- E' corretto?
  - ▶ sulla base della definizione ricorsiva
- Quanto tempo impiega?
  - ▶ costo costante ad ogni chiamata
  - ▶ quante chiamate?
  - ▶ mi aiuta l'albero di ricorsione
  - ▶ costa  $c \cdot n = n$

# Merge sort: ordina $A[p \dots r]$

## Divide - Impera - Combina

- **Divide**  $A[p \dots r]$  in  $A[p \dots q]$  e  $A[q + 1 \dots r]$ ,  
 $q$  è il punto di mezzo di  $A[p \dots r]$
- **Impera** ordinando ricorsivamente  $A[p \dots q]$  e  $A[q + 1 \dots r]$
- **Combina** fondendo i due sotto-array ordinati  $A[p \dots q]$  e  $A[q + 1 \dots r]$  in un singolo array ordinato  $A[p \dots r]$

$\text{MERGE-SORT}(A, p, r)$

```

1 if  $p < r$                                 // Controlla il caso base
2      $q \leftarrow \lfloor (p + r)/2 \rfloor$           // Divide
3      $\text{MERGE-SORT}(A, p, q)$                   // Impera
4      $\text{MERGE-SORT}(A, q + 1, r)$               // Impera
5      $\text{MERGE}(A, p, q, r)$                   // Combina

```

Chiamata iniziale:  $\text{MERGE-SORT}(A, 1, n)$

MERGE( $A, p, q, r$ )

```

1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  // Crea array  $L[1..n_1 + 1]$  e  $R[1..n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5     $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7     $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] \leftarrow L[i]$ 
15      $i \leftarrow i + 1$ 
16   else  $A[k] \leftarrow R[j]$ 
17      $j \leftarrow j + 1$ 

```

**Tempo di esecuzione**

- Primi due loop:  $\Theta(n_1 + n_2) = \Theta(n)$
- Ultimo *for*:  $n$  iterazioni ciascuna in tempo costante  
 $\Rightarrow \Theta(n)$
- **Totale:**  $\Theta(n)$

**Correttezza?**

Esercizio: *Invariante di ciclo per mostrare la correttezza di MERGE*

# Analizzare algoritmi divide et impera

## Ricorrenza

- Equazione di ricorrenza descrive il tempo di esecuzione totale  
 $T(n) = \text{tempo esecuzione per problema di dimensione } n$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

- Se  $n \leq c$  per qualche costante  $c \Rightarrow$   
**caso base** risolto in **tempo costante**:  $\Theta(1)$
- Altrimenti **divide** in  $a$  sottoproblemi di dimensione  $1/b$
- $D(n)$ : tempo per **dividere**
- $C(n)$ : tempo per **combinare**

## Analizzare merge sort

- Supponiamo che  $n$  sia una potenza di 2  
 $\Rightarrow$  due sottoproblemi di dimensione  $n/2$
- Caso base:  $n = 1$
- Se  $n \geq 2$ , i tempi per i passi divide et impera sono:
  - ▶ **Divide**: Calcola  $q \Rightarrow D(n) = \Theta(1)$
  - ▶ **Impera**: Risolve 2 sottoproblemi di dim  $n/2 \Rightarrow 2 \cdot T(n/2)$
  - ▶ **Combina**: fonde un array con  $n$  elementi
- $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$

## Ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

## Risolvere la ricorrenza di merge-sort

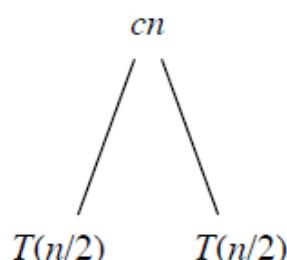
- Dal teorema dell'esperto (lo vedremo poi) soluzione:  
 $T(n) = \Theta(n \lg n)$
- $\lg n$  indica  $\log_2 n$
- Più veloce di insertion sort ( $\theta(n^2)$  caso peggiore)
- Su **input piccoli** insertion sort può essere più veloce
- Per input grandi abbastanza merge sort in genere più veloce

### Senza il teorema

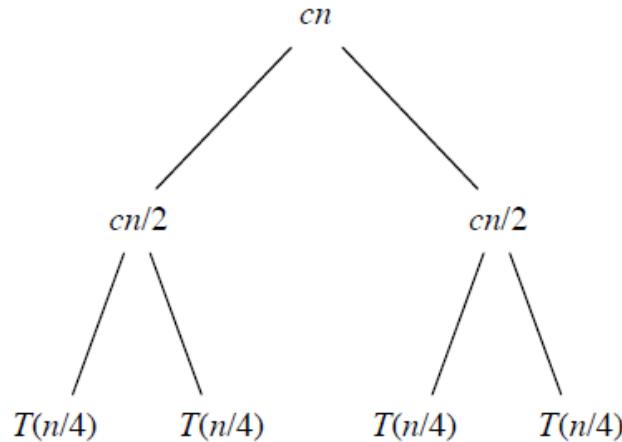
- costante  $c$ : massimo tra  
tempo di esecuzione per il *caso base* e  
tempo per *divide* e *combina* per ogni elemento
- **Ricorrenza:**

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

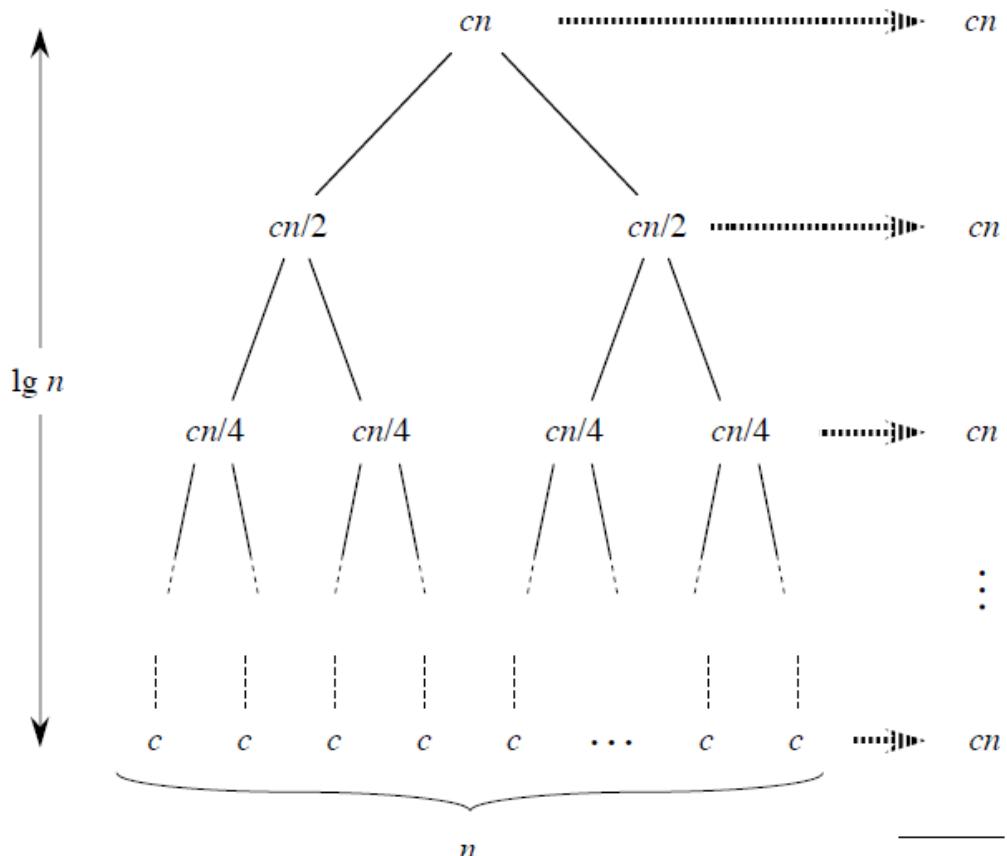
- **Albero di ricorsione** mostra espansioni successive
- Problema originario:  $c \cdot n + 2$  sotto-problemi  $T(n/2)$ :



- Ogni sotto-problema di dim  $n/2$  costa  $cn/2 + 2$  sottoproblemi  $T(n/4)$ :



Si continua ad espandere fino ad arrivare a problemi di dimensione 1:



- Ogni livello ha costo  $c \cdot n$ :
  - ▶ 1 livello:  $c \cdot n$
  - ▶ 2 livello: 2 sottoproblemi con costo  $c \cdot n/2$  ( $2 \cdot c \cdot n/2 = c \cdot n$ )
  - ▶ 3 livello: 4 sottoproblemi con costo  $c \cdot n/4$  ( $4 \cdot c \cdot n/4 = c \cdot n$ )
  - ▶ → costo per livello costante
- $\lg n + 1$  livelli (altezza:  $\lg n$ )
  - ▶ Ad ogni livello dimezzo i dati:  
al livello  $i$  dimensione  $\frac{n}{2^i}$  (radice ha dim.  $n$ )
  - ▶ Caso base (foglia) con un solo elemento:  
 $\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n = \lg n$
- **Costo totale**

$\lg n + 1$  livelli, ognuno costa  $cn \Rightarrow$  totale  $c n \lg n + c n$   
 $c \Rightarrow \Theta(n \lg n)$

## Crescita delle funzioni

# Esercizi...

## Ricerca in un array (2.1-3)

- **Input:** una sequenza di  $n$  numeri  $A = \langle a_1, a_2, \dots, a_n \rangle$  e un valore  $v$
- **Output:** un indice  $i$  tale che  $v = A[i]$  o  $NIL$  se  $v$  non si trova in  $A$
- Scrivere lo **pseudocodice** dell'algoritmo semplice (che scorre tutti i valori)
- Utilizzando un'invariante di ciclo dimostrare che l'algoritmo è corretto

## Ricerca Binaria 2.3-5

- Se la sequenza  $A$  è ordinata possiamo *confrontare il punto centrale della sequenza con  $v$  ed escludere metà sequenza da ulteriori controlli*
- Scrivere pseudocodice iterativo e ricorsivo
- Dimostrare che nel caso peggiore è  $\Theta(\lg n)$
- Chiamata iniziale con parametri  $A, v, 1, n$

## Selection sort 2.2-2

Algoritmo per ordinare  $n$  numeri memorizzati nell'array  $A$

- Trovare il più piccolo elemento di  $A$  e scambiarlo con l'elemento in  $A[1]$
- trovare il secondo elemento più piccolo di  $A$  e scambiarlo con  $A[2]$
- continuando per i primi  $n - 1$  elementi di  $A$

### Esercizio

- Scrivere lo pseudocodice.
- Quale invariante di ciclo conserva questo algoritmo?
- Perché basta eseguirlo solo per i primi  $n - 1$  elementi e non per  $n$ ?
- Esprimere nella notazione  $\Theta$  i tempi di esecuzione dei casi migliore e peggiore dell'algoritmo

- Come modificare quasi tutti gli algoritmi in modo da avere un buon tempo di esecuzione nel caso migliore?
- Tante possibili risposte.

## Crescita delle funzioni

- Come aumenta il tempo di esecuzione di un algoritmo al crescere della dimensione dell'input **al limite**?
- Per input piccoli il tempo di esecuzione potrebbe essere diverso

## Notazione asintotica

- Astraendo termini di ordine inferiore e fattori costanti
- Un modo per confrontare la “dimensione” di funzioni:

$$\begin{array}{ll} O \approx \leq & \Omega \approx \geq \\ \Theta \approx = & \\ o \approx < & \omega \approx > \end{array}$$

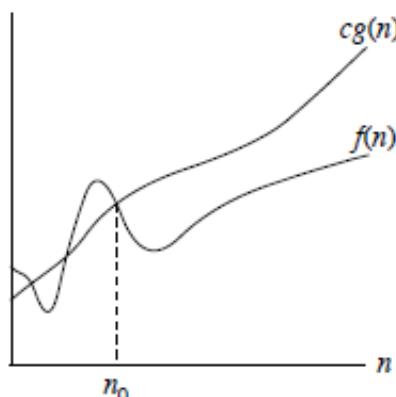
## Nota

- Ogni funzione nelle definizioni deve essere **asintoticamente non negativa**

## Notazione $O$

$$O(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\}$$

$O(g(n))$  è un **insieme di funzioni**



- $g(n)$  è un **limite asintotico superiore** per  $f(n)$
- Se  $f(n) \in O(g(n))$ , si scrive  $f(n) = O(g(n))$

**Esempio**

$2n^2 = O(n^3)$ , con  $c = 1$  e  $n_0 = 2$

**Esempi di funzioni in  $O(n^2)$** 

$n^2$

$n^2 + n$

$n^2 + 1000n$

$1000n^2 + 1000n$

**anche**

$n$

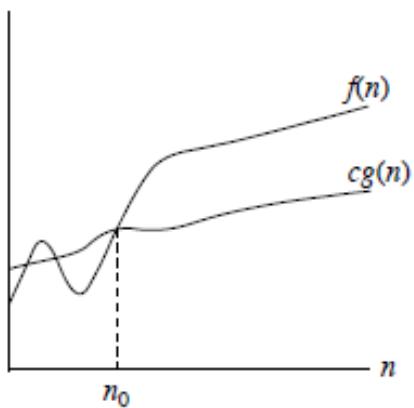
$n/1000$

$n^{1.9999}$

$\frac{n^2}{\lg \lg \lg n}$

**Notazione  $\Omega$** 

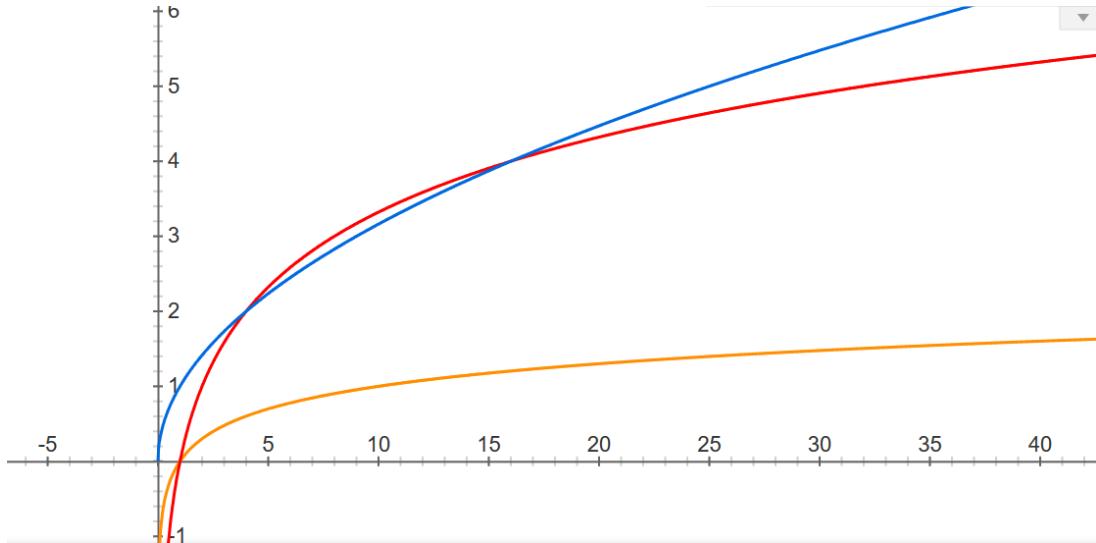
$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}$



$g(n)$  è un **limite asintotico inferiore** per  $f(n)$

**Esempio**

$\sqrt{n} = \Omega(\lg n)$  con  $c = 1$  e  $n_0 = 16$   
 $\sqrt{16} = 4$ ,  $\lg 16 = 4(2^4 = 16)$



blu:  $\sqrt{n}$ , arancione:  $\log(n)$ , rosso:  $\lg(n)$

**Esempi di funzioni in  $\Omega(n^2)$** 

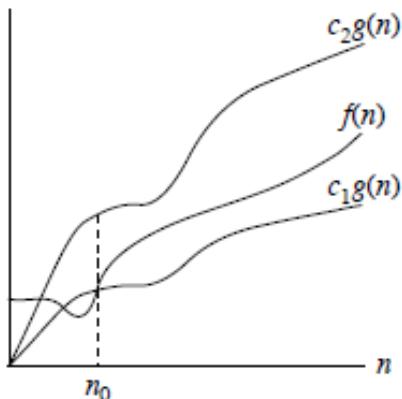
- $n^2$
- $n^2 + n$
- $n^2 - n$
- $1000n^2 + 1000n$
- $1000n^2 - 1000n$

**anche**

- $n^3$
- $n^{2.0001}$
- $n^2 \lg \lg \lg n$
- $2^{2^n}$

## Notazione $\Theta$

$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$



$g(n)$  è un **limite asintoticamente stretto** per  $f(n)$

## Teorema:

$f(n) = \Theta(g(n))$  sse  $f(n) = O(g(n))$  e  $f(n) = \Omega(g(n))$

## Notazione asintotica in equazioni

**Nota:** Le costanti e i termini di ordine inferiore non sono importanti

### Nel lato destro

$O(n^2)$  indica una **qualche funzione anonima nell'insieme  $O(n^2)$**

- $f(n) = O(n^2) \rightarrow$  vuol dire  $f(n) \in O(n^2)$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$   
vuol dire  $2n^2 + 3n + 1 = 2n^2 + f(n)$  per qualche  $f(n) \in \Theta(n)$   
(in particolare:  $f(n) = 3n + 1 \Rightarrow$  elimino dettagli dalle formule)

### Nel lato sinistro

- Indipendentemente da come scelgo le funzioni anonime a sinistra, posso scegliere quelle a destra per rendere l'equazione valida
- $2n^2 + \Theta(n) = \Theta(n^2)$  vuol dire:  
 $\forall$  funzione  $f(n) \in \Theta(n) \exists$  una funzione  $g(n) \in \Theta(n^2)$  t.c.  
 $2n^2 + f(n) = g(n)$

**Notazione  $o$** 

$o(g(n)) = \{f(n) : \forall c > 0 \in \mathbb{R} \exists n_0 > 0 \in \mathbb{N} \text{ t.c.}$   
 $0 \leq f(n) < c \cdot g(n) \forall n \geq n_0\}$

oppure

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

**Esempi di funzioni in  $o(n^2)$** 

$$n^{1.9999} = o(n^2)$$

$$n^2 / \lg n = o(n^2)$$

$$n^2 \neq o(n^2)$$

$$n^2 / 1000 \neq o(n^2)$$

**Notazione  $\omega$** 

$\omega(g(n)) = \{f(n) : \forall c > 0 \in \mathbb{R} \exists n_0 > 0 \in \mathbb{N} \text{ t.c.}$   
 $0 \leq cg(n) < f(n) \forall n \geq n_0\}$

Oppure

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

**Esempi di funzioni in  $\omega(n^2)$** 

$$n^{2.0001} = \omega(n^2)$$

$$n^2 \lg n = \omega(n^2)$$

$$n^2 \neq \omega(n^2)$$

## Confronto di funzioni: proprietà relazionali

### Transitività :

$f(n) = \Theta(g(n))$  e  $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

Lo stesso per  $O$ ,  $\Omega$ ,  $o$  e  $\omega$

### Riflessività :

$f(n) = \Theta(f(n))$

Analogamente per  $O$  e  $\Omega$

### Simmetria:

$f(n) = \Theta(g(n))$  sse  $g(n) = \Theta(f(n))$

### Simmetria trasposta:

$f(n) = O(g(n))$  sse  $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$  sse  $g(n) = \omega(f(n))$

## Assenza di tricotomia

### Numeri Reali

- Per i numeri reali vale la proprietà di tricotomia
- Dati  $a, b \in \mathbb{R}$  vale una e sola delle seguenti relazioni
  - ▶  $a < b$
  - ▶  $a = b$
  - ▶  $a > b$

### Per il confronto tra funzioni?

- Intuitivamente:
  - ▶  $O$  equivale a  $\leq$
  - ▶  $\Omega$  equivale a  $\geq$
  - ▶ ...
- ma non vale la tricotomia
  - ▶ Esempio:  $n^{1+\sin n}$  e  $n$  ( $1 + \sin n$  oscilla tra 0 e 2)

# Logaritmi

## Notazioni:

- $\lg n = \log_2 n$  (logaritmo binario)
- $\ln n = \log_e n$  (logaritmo naturale)
- $\lg^k n = (\lg n)^k$
- $\lg \lg n = \lg(\lg n)$  (composizione)
- Le funzioni logaritmiche si applicano soltanto al termine successivo nella formula,  
 $\lg n + k$  vuol dire  $(\lg n) + k$  e non  $\lg(n + k)$

## Nell'espressione $\log_b a$

- Se  $b$  è costante l'espressione è strettamente crescente quando  $a$  cresce
- Se  $a$  è costante l'espressione è strettamente decrescente quando  $b$  cresce

## Identità utili

Per tutti i reali  $a > 0$ ,  $b > 0$ ,  $c > 0$  e  $n$  (con base del logaritmo  $\neq 1$ ):

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

# Note sulla crescita di funzioni

## Quanto cresce $\log \log n$ ?

- Mooooooolto poco
- Supponiamo di considerare la base 10. Si ha  $\log \log 10^{100} = 2$
- Ma quanto è grande  $10^{100}$ ?
- E' un googol (da cui deriva il nome google)
- $70! = 1.198$  googol.
- Il numero **stimato** di **atomi** nell'universo visibile è compreso tra  $10^{72}$  e  $10^{87}$

# Ricorrenze

# Ricorrenze

## Algoritmo generico divide-et-impera

- Caso base:
  - ▶ problema  $P$  (dim 1) in  $\Theta(1)$  passi
- Caso ricorsivo:
  - ▶ dividere problema  $P$  (dim  $n > 1$ )
    - ★ in  $a$  sotto-problemi  $P$  (dim  $n/b$ )
  - ▶ combinare le soluzioni dei sottoproblemi in  $f(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ aT(n/b) + f(n) & \text{se } n > 1 \end{cases}$

## Obiettivo

**ottenere una formula chiusa per  $T(n)$**

## Ricorrenza

Equazione (o disequazione) che descrive una funzione in termini del suo valore con input più piccoli

In genere in termini di:

- uno o più casi base e
- se stessa, con argomenti più piccoli

## Esempi

Vedremo

- soluzioni esatte a ricorrenze "esatte", poi
- con funzioni asintotiche nella definizione della ricorrenza
- con funzioni asintotiche nella soluzione della ricorrenza

## Fattoriale

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(n-1) + 1 & \text{se } n > 1 \end{cases}$$

**Soluzione:**  $T(n) = n!$

## Dimostrazione per induzione

**Suppongo** che sia vero per  $k < n$  ( $T(k) = k$  per  $k < n$ ) e **verifico**:

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= (n-1) + 1 = n \end{aligned}$$

## Metodi di soluzione

- **Metodo di sostituzione** ipotizzo una soluzione e mostro che è corretta con *induzione matematica*
- **Metodo dell'albero di ricorsione.** I **nodi** rappresentano i costi ai vari livelli della ricorsione
- **Metodo dell'esperto** fornisce soluzione per:

$$T(n) = aT(n/b) + f(n)$$

- Ricorrenza e soluzione spesso espresse in notazione asintotica
  - ▶ Es:  $T(n) = 2T(n/2) + \Theta(n)$ ; soluzione  $T(n) = \Theta(n \lg n)$ .
- Se serve soluzione esatta (non asintotica) devo dettagliare le condizioni al contorno

# Metodo di sostituzione

## Due passi principali

- ① Ipotizzare la soluzione
- ② Con induzione matematica trovo le costanti e **dimosto** che la soluzione è corretta

## Soluzione Esatta

Ricorrenza con una funzione esatta  $\Rightarrow$  soluzione è esatta

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \end{cases}$$

- ① **Ipotesi:**  $T(n) = n \lg n + n$

- ② **Induzione:**

**Caso base**  $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

**Passo induttivo:** Ipotesi induttiva  $T(k) = k \lg k + k \quad \forall k < n$   
uso ipotesi induttiva per  $k = (n/2)$

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(\frac{n}{2} \lg \frac{n}{2} + \frac{n}{2}\right) + n \quad (\text{ipotesi induttiva}) \\ &= n \lg \frac{n}{2} + n + n = n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n = n \lg n + n \end{aligned}$$

## Soluzione Asintotica (usata più spesso)

$$T(n) = \begin{cases} O(1) & \text{se } n \text{ sufficientemente piccolo} \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

- Soluzione con notazione asintotica:  $T(n) = \Theta(n \lg n)$
- Non si mostra in dettaglio il caso base
  - ▶  $T(n)$  è costante  $\forall n$  costante
  - ▶ Interessa soluzione asintotica  $\Rightarrow$  posso sempre scegliere un caso base  $O(1)$

## Risolvere $T(n) = 2T(n/2) + \Theta(n)$ col metodo di sostituzione

- Dare un **nome** alla costante per  $\Theta(n)$
- Mostrare il limite superiore ( $O$ ) e inferiore ( $\Omega$ ) separatamente

## Limite superiore di $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

- $T(n) \leq 2T(n/2) + cn$  per qualche costante positiva  $c$   
(definizione di  $O()$ )
- **Ipotesi**  $T(n) = O(n \lg n)$   
(cioè  $T(n) \leq d \cdot n \cdot \lg n$  per  $d > 0$  costante)
- Se trovo  $d$  costante t.c. la ricorrenza è verificata  $\Rightarrow$  verifico che  $T(n) = O(n \lg n)$
- Nella sostituzione non ho funzioni asintotiche, ma funzioni che dipendono da  $d$  e  $c$  che quindi posso risolvere in forma esatta

## Sostituzione

**Ipotizzo** che  $T(n) = O(n \lg n)$  cioè che

$T(k) \leq d \cdot k \cdot \lg k$  per qualche costante positiva  $d$  (e  $k < n$ )

$$\begin{aligned}
 T(n) &\leq 2T\left(\frac{n}{2}\right) + c \cdot n \\
 &\leq 2(d \cdot \frac{n}{2} \lg \frac{n}{2}) + c \cdot n \\
 &= d \cdot n \lg \frac{n}{2} + c \cdot n \\
 &= d \cdot n \lg n - d \cdot n + c \cdot n \\
 &\leq d \cdot n \lg n \quad \text{valido se } -dn + cn \leq 0, \text{ cioè } d \geq c
 \end{aligned}$$

$$\Rightarrow T(n) = O(n \lg n)$$

## Limite inferiore

$T(n) \geq 2T(n/2) + cn$  per qualche costante positiva  $c$   
(definizione di  $\Omega()$ )

**Ipotesi:**  $T(n) \geq d \cdot n \lg n$  per qualche costante positiva  $d$ .

## Sostituzione

$$\begin{aligned}
 T(n) &\geq 2T\left(\frac{n}{2}\right) + c \cdot n \\
 &\geq 2(d \cdot \frac{n}{2} \lg \frac{n}{2}) + c \cdot n \\
 &= d \cdot n \lg \frac{n}{2} + c \cdot n \\
 &= d \cdot n \lg n - d \cdot n + c \cdot n \\
 &\geq d \cdot n \lg n \quad \text{valido se } -dn + cn \geq 0, \text{ cioè } d \leq c
 \end{aligned}$$

## Quindi

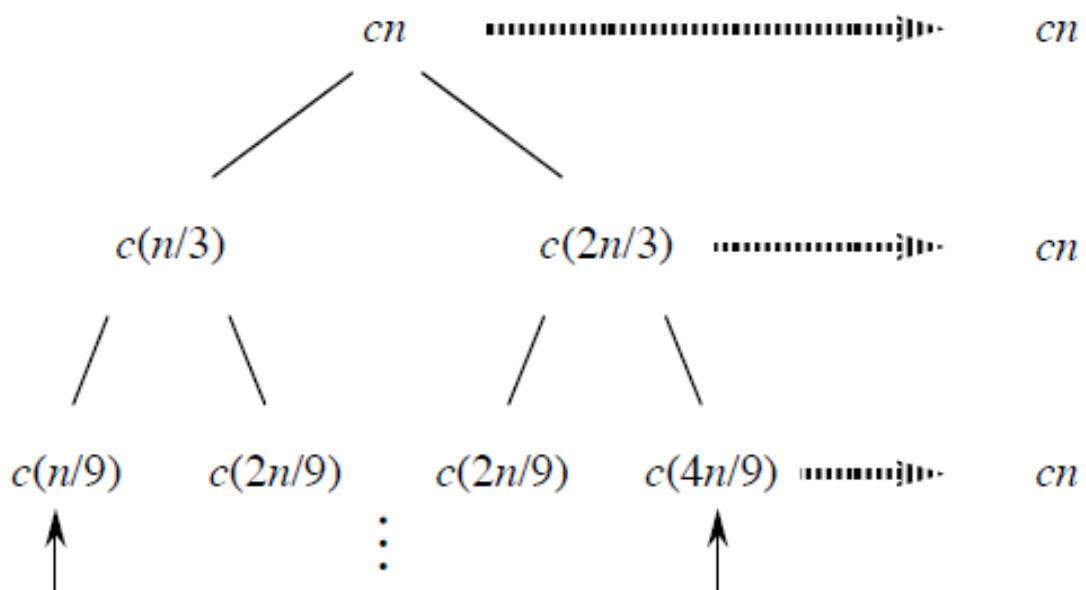
$T(n) = \Omega(n \lg n) \Rightarrow T(n) = \Theta(n \lg n)$   
( $d = c$  per  $O$  e  $\Omega$  non sono sempre uguali)

## Metodo dell'albero di ricorsione

- Intuisco un'ipotesi che verifico col metodo di sostituzione
- Nodo → costo di qualche sotto problema
- Sommo i costi all'interno di ogni livello e poi di tutti i livelli

## Esempio

- $T(n) = T(n/3) + T(2n/3) + \Theta(n)$
- Limite **superiore** ( $O$ ):  $T(n) \leq T(n/3) + T(2n/3) + cn$
- Limite **inferiore**: ( $\Omega$ )  $T(n) \geq T(n/3) + T(2n/3) + cn$
- Albero:



## Caratteristiche albero

- $\log_3 n$  livelli pieni  
Altezza a sx ( $x$ ) t.c.  
 $3^x = n \Rightarrow x = \log_3 n$
- Altezza albero:  $\log_{3/2} n$
- Ogni livello  $\leq cn$
- Idea per **limite superiore**:  $\leq dn \log_{3/2} n = O(n \lg n)$
- Idea per **limite inferiore**:  $\geq dn \log_3 n = \Omega(n \lg n)$
- ... molto complicato fare i conti esatti con l'albero!

### 1. Limite superiore - Ipotesi: $T(n) \leq dn \lg n$

$$\begin{aligned}
 T(n) &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \leq d\left(\frac{n}{3}\right) \lg\left(\frac{n}{3}\right) + d\left(\frac{2n}{3}\right) \lg\left(\frac{2n}{3}\right) + cn \\
 &= (d\left(\frac{n}{3}\right) \lg n - d\left(\frac{n}{3}\right) \lg 3) + (d\left(\frac{2n}{3}\right) \lg n - d\left(\frac{2n}{3}\right) \lg\left(\frac{3}{2}\right)) + cn \\
 &= d n \lg n - d\left(\left(\frac{n}{3}\right) \lg 3 + \left(\frac{2n}{3}\right) \lg\left(\frac{3}{2}\right)\right) + cn \\
 &= d n \lg n - d\left(\left(\frac{n}{3}\right) \lg 3 + \left(\frac{2n}{3}\right) \lg 3 - \left(\frac{2n}{3}\right) \lg 2\right) + cn \\
 &= d n \lg n - dn\left(\lg 3 - \frac{2}{3}\right) + cn \leq \\
 &\leq d n \lg n
 \end{aligned}$$

se  $-dn\left(\lg 3 - \frac{2}{3}\right) + cn \leq 0$  quindi per  $d \geq \frac{c}{\lg 3 - \frac{2}{3}}$

$$\Rightarrow T(n) = O(n \lg n)$$

## 2. Limite inferiore - Ipotesi: $T(n) \geq d n \lg n$

- **Sostituzione:** Stessa del limite superiore, ma cambiando  $\leq$  con  $\geq$
- Corretto se
  - ▶  $0 < d \leq \frac{c}{\lg 3 - 2/3}$
- Quindi,  $T(n) = \Omega(n \lg n)$
- Poichè
  - ▶  $T(n) = O(n \lg n)$
  - ▶  $T(n) = \Omega(n \lg n)$
- $\Rightarrow T(n) = \Theta(n \lg n)$

## Metodo dell'esperto

### Teorema dell'esperto

- Sia  $T(n)$  una funzione definita sui naturali dalla ricorrenza:  

$$T(n) = aT(n/b) + f(n)$$

( $n/b$  indica  $\lfloor n/b \rfloor$  o  $\lceil n/b \rceil$ )
- $a \geq 1, b > 1$  costanti e  $f(n) > 0$  asintoticamente

**Note:**

- $a \geq 1$
- $b > 1$
- $f(n) > 0$  per  $n > n_0$

**Allora**  $T(n)$  può essere asintoticamente limitata nei seguenti modi:

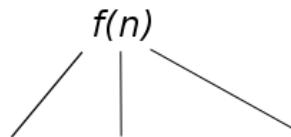
- **Caso 1:** Se  $f(n) = O(n^{\log_b a - \epsilon})$  per qualche costante  $\epsilon > 0$   
allora:  $T(n) = \Theta(n^{\log_b a})$
- **Caso 2:** Se  $f(n) = \Theta(n^{\log_b a})$   
allora:  $T(n) = \Theta(n^{\log_b a} \lg n)$
- **Caso 3:** Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  per qualche costante  $\epsilon > 0$  e  
 $f(n)$  t.c.  $af(n/b) \leq cf(n)$  per qualche costante  $c < 1$  e  $\forall n \geq n_0$   
allora:  $T(n) = \Theta(f(n))$

- Confronta  $n^{\log_b a}$  con  $f(n)$
- Soluzione dipende dalla **più grande**
- I tre casi **non coprono tutte** le funzioni possibili
- Condizione di regolarità : voglio essere sicuro che quando scendo nell'albero  $f()$  diventa più piccola

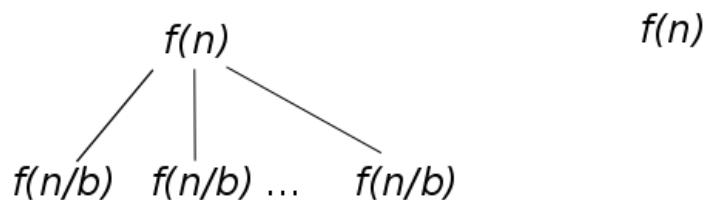
## Intuizione

del teorema dell'esperto  
a partire dall'albero di ricorsione

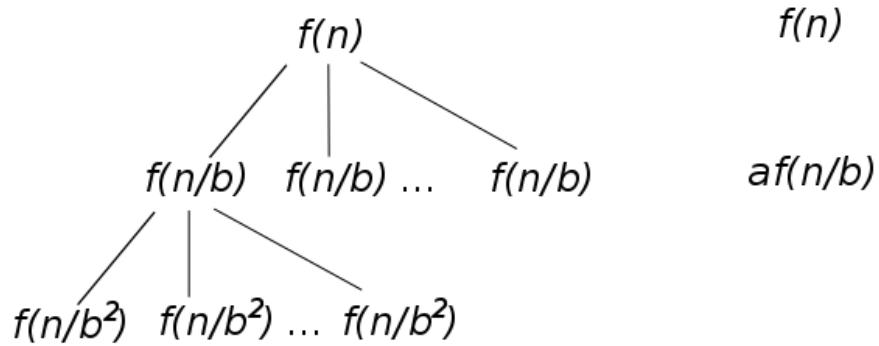
$$T(n) = aT(n/b) + f(n)$$



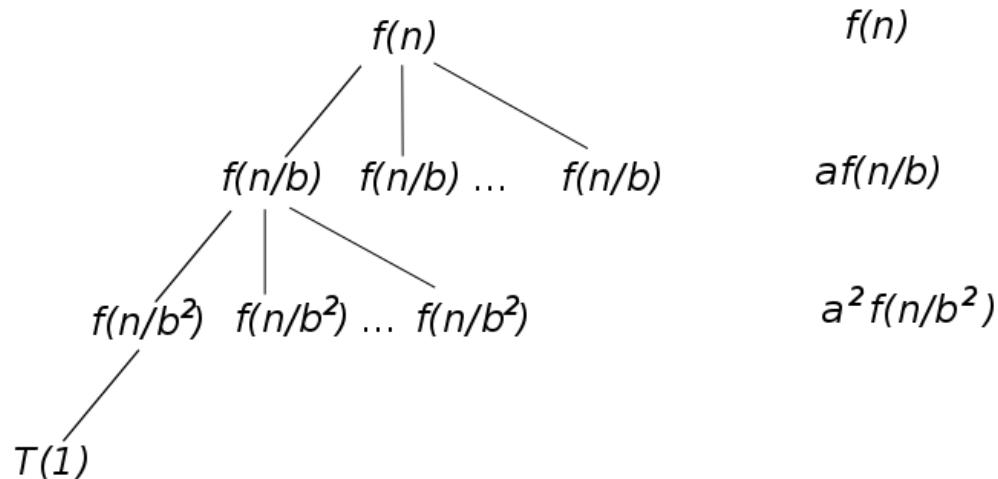
$$T(n) = aT(n/b) + f(n)$$



$$T(n) = aT(n/b) + f(n)$$



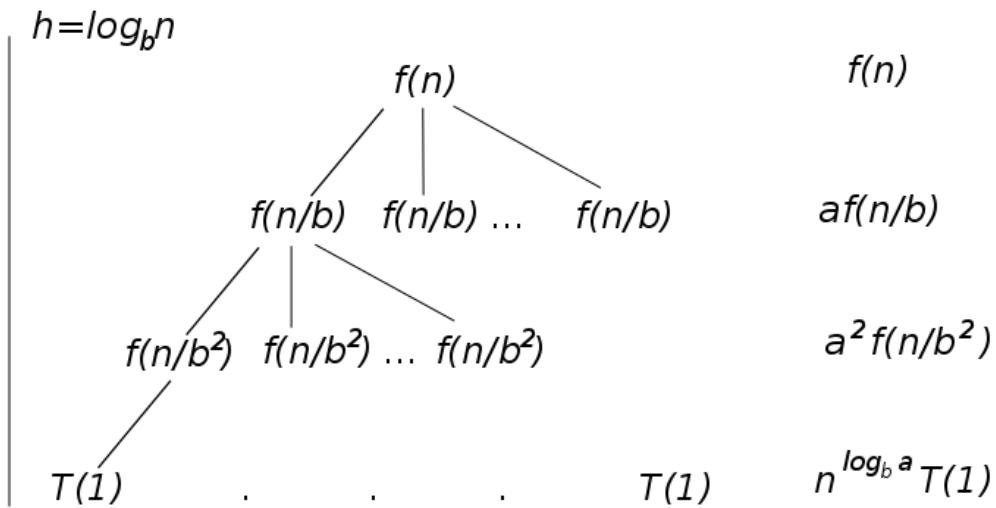
$$T(n) = aT(n/b) + f(n)$$



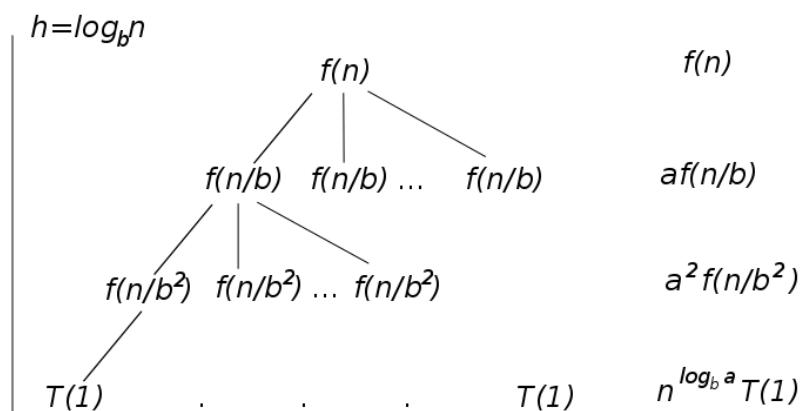
Quante foglie?

- Altezza albero: quando  $n/b^h = 1 \Rightarrow n = b^h \Rightarrow h = \log_b n$
- Foglie =  $a^h = a^{\log_b n} = n^{\log_b a}$

$$T(n) = aT(n/b) + f(n)$$



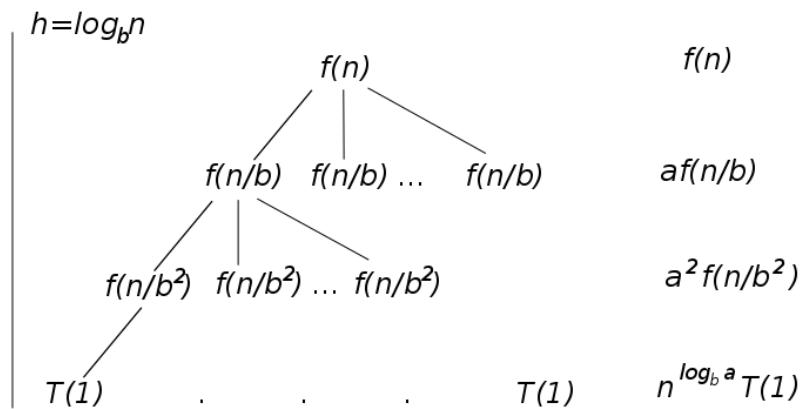
## Caso 1



$$\exists \epsilon > 0 : f(n) = O(n^{\log_b a - \epsilon})$$

- Costo cresce dalla radice alle foglie
- Costo dominato dalle foglie
- $\Rightarrow \Theta(n^{\log_b a})$

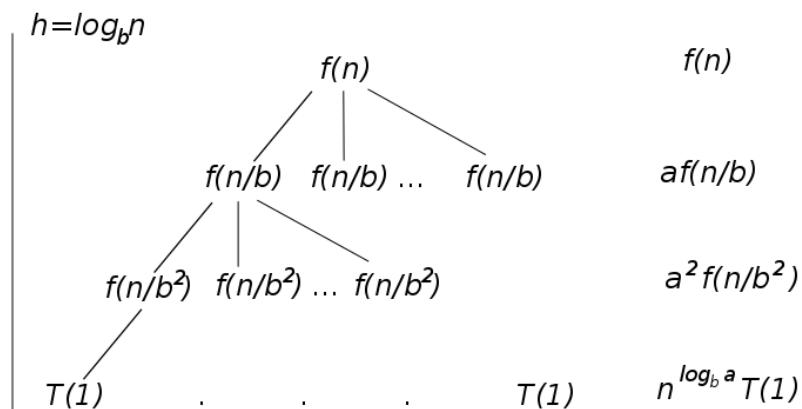
## Caso 2



$$f(n) = \Theta(n^{\log_b a})$$

- Il costo è all'incirca lo stesso in ognuno dei  $\log_b n$  livelli
- $\Rightarrow \Theta(f(n) \lg_n) = \Theta(n^{\log_b a} \lg_n)$

## Caso 3



$$\exists \epsilon > 0 : f(n) = \Omega(n^{\log_b a + \epsilon})$$

- Costo decremente dalla radice alle foglie
- Costo dominato dalla radice
- $\Rightarrow \Theta(f(n))$

## Condizione di regolarità del caso 3

- Vale sempre se  $f(n) = n^k$  e  $f(n) = \Omega(n^{\log_b a + \epsilon})$  per qualche costante  $\epsilon > 0$
- $\Rightarrow$  OK se  $f(n)$  è un polinomio
- **Dimostrazione**
  - ▶  $f(n) = \Omega(n^{\log_b a + \epsilon})$  e  $f(n) = n^k \Rightarrow$   
 $n^k > n^{\log_b a + \epsilon} = n^{\log_b a} \cdot n^\epsilon > n^{\log_b a} \Rightarrow$   
 $k > \log_b a$
  - ▶ i due lati come esponenti con base  $b$ :  
 $b^k > b^{\log_b a} = a \Rightarrow a/b^k < 1$
  - ▶  $a, b, k$  costanti  $\Rightarrow$  scelgo  $c = a/b^k \Rightarrow c < 1$  costante
  - ▶  $a f(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n)$   
 $\Rightarrow$  condizione di regolarità soddisfatta

## Esempi

- $T(n) = 4T(\frac{n}{2}) + n$   
 $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n$   
**Caso 1:**  $f(n) = O(n^{2-\epsilon})$  con  $\epsilon = 1 \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = 4T(\frac{n}{2}) + n^2$   
 $\Rightarrow n^{\log_b a} = n^2; f(n) = n^2$   
**Caso 2:**  $f(n) = \Theta(n^2) \Rightarrow T(n) = \Theta(n^2 \lg n)$
- $T(n) = 4T(\frac{n}{2}) + n^3$   
 $\Rightarrow n^{\log_b a} = n^2; f(n) = n^3$   
**Caso 3:**  $f(n) = \Omega(n^{2+\epsilon})$  con  $\epsilon = 1$   
e condizione regolarità  $4(\frac{n}{2})^3 \leq c n^3$  con  $c = \frac{1}{2}$   
 $\Rightarrow T(n) = \Theta(n^3)$

## Ricerca binaria

Trovare un elemento in un array ordinato:

- **Divide:** Verificare l'elemento di mezzo
- **Impera:** Cercare ricorsivamente in un sotto-array
- **Combina:** Banale

Ricorrenza per ricerca binaria:

$$T(n) = 1 + T(n/2) + \Theta(1)$$

- Da teorema dell'esperto:  
 $n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow$   
 Caso 2  $\Rightarrow T(n) = \Theta(\lg n)$

## Potenza intera

**Problema:** Calcolare  $a^n$ , dove  $n \in \mathbb{N}$

Algoritmo **intuitivo**:  $\Theta(n)$

**Algoritmo divide et impera**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{se } n \text{ è pari} \\ a^{n/2} \cdot a^{n/2} \cdot a & \text{se } n \text{ è dispari} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n)$$

# Cenni di calcolo delle probabilità

## (distribuzione di probabilità discreta)

## Calcolo delle probabilità

### Probabilità

- La probabilità è definita su uno **spazio degli eventi**  $S$   
 $S$  è un insieme di **eventi elementari** (esito di un esperimento)
- Per il lancio di due monete (ognuna  $T$  o  $C$ ):  
 $S = \{TT, TC, CT, CC\}$

### Eventi

- Un **evento** è un *sottoinsieme* di  $S$  (es. due lanci uguali:  $\{TT, CC\}$ )
  - ▶ Un evento si realizza quando si realizza uno qualsiasi degli eventi elementari che lo compongono
- A ciascun evento elementare si assegna una probabilità (reale compreso tra 0 e 1)
  - ▶ la somma delle probabilità degli eventi elementari è 1
  - ▶ la probabilità di un evento è la somma delle probabilità degli eventi elementari che lo costituiscono

# Definizioni di probabilità

## Interpretazione classica

- Si assegna la stessa probabilità a tutti gli eventi elementari
- $\Rightarrow$  la probabilità di un evento  $E$  è il rapporto tra il numero di eventi elementari che lo compongono (*casi favorevoli*) e il numero totale di eventi elementari (*casi possibili*)

$$P(E) = \frac{\text{casi favorevoli}}{\text{casi possibili}}$$

- ▶ bisogna scegliere opportunamente gli eventi elementari
- ▶ molto usata nel nostro corso
- Non è detto che gli eventi elementari siano equiprobabili (probabilità di prendere 30L ad un esame...)

# Definizioni di probabilità

## Interpretazione frequentista

- Per assegnare la probabilità ad un evento  $E$  si ripete l'esperimento  $n$  volte e si osserva il numero di volte in cui  $E$  si è verificato

$$P(E) = \text{frequenza relativa} = \frac{\text{casi favorevoli su } n}{n}$$

- Basata sulla **legge empirica del caso**: *la frequenza relativa di un evento tende a stabilizzarsi all'aumentare del numero delle prove effettuate*
  - ▶ è una legge empirica, non un teorema!
  - ▶ è un caso particolare della *legge dei grandi numeri*
- La probabilità di un evento  $E$  è il limite delle frequenze relative dei casi favorevoli quando il numero delle prove tende all'infinito

$$P(E) = \lim_{n \rightarrow \infty} \frac{\text{casi favorevoli su } n}{n}$$

# Definizioni di probabilità

## Interpretazione assiomatica

- Una **distribuzione di probabilità**  $P(\ )$  è una **relazione** che associa gli eventi di  $S$  ai numeri reali t.c. sono soddisfatti gli **assiomi di probabilità**:
  - ①  $P(E) \geq 0 \forall$  evento  $E$
  - ②  $P(S) = 1$
  - ③  $P(A \cup B) = P(A) + P(B)$  se  $A$  e  $B$  sono **mutuamente esclusivi**
- $P(E)$  è la probabilità dell'evento  $E$  (possibilità che un evento incerto si manifesti)

# Leggi della Probabilità

## Probabilità dell'evento contrario

- $\bar{E}$  è l'evento contrario di  $E$   
(tutti gli eventi elementari che non sono in  $E$ )
- La somma di tutti gli eventi elementari è 1  $\Rightarrow$   
 $P(\bar{E}) = 1 - P(E)$

# Leggi della Probabilità

## Probabilità totale per eventi incompatibili

- $A$  e  $B$  sono incompatibili se non hanno eventi elementari in comune ( $A \cap B = \emptyset$ )
- Probabilità che si verifichi uno dei due eventi (probabilità totale):  

$$P(A \text{ o } B) = P(A \cup B) = P(A) + P(B)$$

## Probabilità totale per eventi compatibili

- $A$  e  $B$  sono compatibili se hanno eventi elementari in comune (devo contarli una volta sola)
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- Esempio: estrazione di una carta da un mazzo con 40 carte
  - ▶  $A =$  "estrai una carta di cuori";  $B =$  "estrai una figura"
  - ▶  $A \text{ o } B =$  "estrai una carta di cuori o una figura"
  - ▶  $A \text{ e } B =$  "estrai una figura di cuori"
  - ▶ 
$$P(A \cup B) = \frac{10}{40} + \frac{12}{40} - \frac{3}{40}$$

# Leggi della Probabilità

## Probabilità composta per eventi indipendenti

- Due eventi sono **indipendenti** se il realizzarsi di uno **non influenza** la probabilità del realizzarsi dell'altro
- probabilità che si realizzino entrambi (probabilità composta):  

$$P(A \cap B) = P(A) \cdot P(B)$$

## Probabilità composta per eventi dipendenti

- indico con  $P(B|A)$  la probabilità **condizionata** di  $B$  dato  $A$  (probabilità che si verifichi  $B$  supponendo che si sia verificato  $A$ )
- $$P(A \cap B) = P(A) \cdot P(B|A)$$

# Probabilità condizionata e indipendenza

## Definizione di probabilità condizionata

- Si ha *a priori* qualche informazione sull'esito di un esperimento
- $P(A|B) = \frac{P(A \cap B)}{P(B)}$
- Due eventi sono **indipendenti** se  $P(A \cap B) = P(A) \cdot P(B)$ 
  - ▶ In tal caso  $P(A|B) = P(A)$

## Teorema di Bayes

- Dalla definizione di probabilità condizionata e dalla legge commutativa ( $A \cap B = B \cap A$ ) si ha:
- $P(A \cap B) = P(B)P(A|B) = P(A)P(B|A)$
- Quindi  $P(A|B) = \frac{P(A)P(B|A)}{P(B)}$

# Variabili Casuali

## Variabile casuale (o stocastica, o aleatoria)

- E' una variabile che può assumere valori diversi in dipendenza di un qualche fenomeno casuale
  - ▶ usiamo le lettere maiuscole per indicare la variabile, e le lettere minuscole per i suoi possibili valori

## Esempi

- $X = \{ \text{numero di teste in tre lanci di monete} \}$ 
  - ▶ un suo possibile valore (esito) sarà:  $x = 2$
- $X = \{ \text{risultato del lancio di un dado} \}$ 
  - ▶ la variabile casuale ha 6 possibili valori: 1, 2, 3, 4, 5, 6 (equiprobabili)
- esempio più complesso: risultato del lancio di due dati
  - ▶ valori compresi tra 2 e 12 **NON** equiprobabili

# Distribuzione di probabilità

- I possibili esiti di un esperimento e le loro probabilità sono sintetizzati in distribuzioni di probabilità
- Una variabile casuale si riferisce all'esito di un fenomeno casuale
- A ogni possibile esito è associata una probabilità di verificarsi
- La distribuzione di probabilità di una variabile casuale elenca **i suoi possibili valori e le relative probabilità**

# Distribuzione di probabilità

## per variabili casuali discrete

- La variabile casuale  $X$  è detta **discreta** se assume un numero finito o un'infinità numerabile di valori  $\{x_1, x_2, \dots, x_n\}$  (è definita su un insieme discreto)
- La distribuzione di probabilità di una variabile casuale discreta  $X$  assegna una probabilità  $P(X = x_i)$  (o  $P(x)$ ) a ogni possibile valore assunto  $x \in \{x_1, x_2, \dots, x_n\}$ 
  - ▶  $\forall x \quad 0 \leq P(x) \leq 1$
  - ▶  $\sum_{i=1}^n P(X = x_i) = 1$
- Anche chiamata funzione di massa (o funzione massa di probabilità o densità discreta):  $p(x) = P(X = x)$

# Sintesi di una distribuzione

- Per descrivere le caratteristiche di una distribuzione di probabilità si possono usare sintesi numeriche
  - ▶ ad esempio media, mediana, quartili, varianza e deviazione standard (o scarto quadratico medio)
  
- Le misure che ricorrono maggiormente sono:
  - ▶ la **media** per descrivere la tendenza centrale della distribuzione;
  - ▶ la **deviazione standard** e la **varianza** per descrivere la sua variabilità

## Valore atteso

### di una distribuzione discreta

- Il valor medio (o valore atteso) per una variabile casuale indica *cosa ci si aspetta* in media in una lunga serie di osservazioni
- Il valore atteso della variabile casuale discreta  $X$  è definito come:  

$$\mu = E[X] = \sum_{i=1}^n x_i P(x_i)$$

### Esempio: lancio di due monete

- variabile casuale  $X = \{\text{numero di teste}\} = \{0, 1, 2\}$
- Valor atteso di  $X$ 
  - ▶  $P(0) = 0.25, P(1) = 0.50, P(2) = 0.25$
  - ▶  $\rightarrow \mu = E[X] = \sum_{i=1}^3 x_i P(x_i) = 0 \cdot P(0) + 1 \cdot P(1) + 2 \cdot P(2) = 1$
  - ▶ ci aspettiamo che in media si ottenga una testa in due lanci consecutivi

## Linearità del valore atteso

- Se  $E[X]$  e  $E[Y]$  sono definiti si ha:  $E[X + Y] = E[X] + E[Y]$
- Se  $X$  è una variabile casuale, qualsiasi funzione  $g(x)$  definisce una nuova variabile casuale  $g(X)$

$$E[g(X)] = \sum_x g(x)P\{X = x\}$$

- In particolare se  $g(x) = ax$  si ha  $E[aX] = aE[X]$  (i valori attesi sono lineari)

## Variabilità distribuzione discreta

Data la definizione del valor medio allora si avrà per le misure di variabilità:

- **Varianza** di una variabile casuale discreta  
 $\sigma^2 = E[X - \mu]^2 = \sum_{i=1}^n (x_i - \mu)^2 P(x_i)$
- **Deviazione standard** di una variabile casuale discreta  
 $\sigma = \sqrt{\mu} = \sqrt{\sum_{i=1}^n (x_i - \mu)^2 P(x_i)}$

## Distribuzione Uniforme Discreta

### Variabile aleatoria Uniforme Discreta

- Se  $S$  è finito e  $\forall s \in S$  si ha  $P\{s\} = 1/|S|$
- Esempio: **moneta perfetta**
- La funzione densità di probabilità è:

$$f_Y(y) = \begin{cases} \frac{1}{N} & \text{se } y = 1 \dots N \\ 0 & \text{altrimenti} \end{cases}$$

# Variabili casuali indicatrici

- Dato uno spazio dei campioni  $S$  e un evento  $A$ , si definisce la variabile casuale indicatrice:

$$I\{A\} = \begin{cases} 1 & \text{se si verifica } A \\ 0 & \text{se non si verifica } A \end{cases}$$

## Lemma

Per un evento  $A$ , sia  $X_A = I\{A\}$   
Allora  $E[X_A] = Pr\{A\}$

## Dim

$\bar{A}$  complemento di  $A$ :

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot Pr\{A\} + 0 \cdot Pr\{\bar{A}\} \quad (\text{def valore atteso}) \\ &= Pr\{A\} \end{aligned}$$

Valore atteso: media ponderata dei possibili valori di  $I\{A\}$

Metodo per conversione tra **probabilità** e **valori attesi**

# Esempi

## Numero atteso di teste lanciando una moneta una volta

- Moneta imparziale
- Spazio dei campioni:  $S = \{T, C\}$
- $Pr\{T\} = Pr\{C\} = 1/2$
- Variabile casuale indicatrice:  
 $X_T = I\{T\}$   
 $X_T$  conta il numero di teste in un lancio
- Dal lemma:  
 $E[X_T] = Pr\{T\} = 1/2$

## Numero atteso di teste in $n$ lanci

- Questo numero NON è una probabilità !
- $X$ : variabile casuale per il numero di teste in  $n$  lanci
- Variabile casuale indicatrice
  - ▶  $X_i = I\{i\text{-mo lancio è evento } T\}$
  - ▶  $X = \sum_{i=1}^n X_i$
  - ▶ Numero atteso di teste ( $T$ ):  

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n Pr\{T\} = \sum_{i=1}^n 1/2 = n/2$$
- Linearità valori attesi:  
 valore atteso della somma = somma dei valori attesi

# Analisi Probabilistica e Algoritmi Randomizzati

## Il problema delle assunzioni

### Il problema delle assunzioni

#### Scenario

- Agenzia per assumere un impiegato
- L'agenzia intervista un candidato al giorno
- Dopo il colloquio si decide subito se assumere (e si licenzia l'attuale impiegato)
  - ▶ **Costo colloquio:**  $c_c$  per candidato
  - ▶ **Costo assunzione:**  $c_a$  per candidato
- Supponiamo  $c_a > c_c$  (forse  $c_a \gg c_c$ )
- Assumiamo sempre il miglior candidato visto

#### Obiettivo

- Determinare il costo

## Pseudocodice

- Candidati numerati  $1, \dots, n$
- Il primo candidato è sempre assunto

HIRE-ASSISTANT( $n$ )

```
best ← 0      // Candidato 0 e' fittizio
for  $i \leftarrow 1$  to  $n$ 
    colloquio con candidato  $i$ 
    if (candidato  $i$  migliore di best)
        best ←  $i$ 
        assumi candidato  $i$ 
```

## Paradigma comune (non solo assunzioni)

- Trovare min (max) in una sequenza

## Correttezza

- Con invarianti di ciclo

## Costo

- $n$  candidati ne assumo  $m$
- "Costo":  $O(n \cdot c_c + m \cdot c_a)$
- Dipende da **ordine** dei candidati ( $m$  varia)

## Caso peggiore

Assumo **tutti** gli  $n$  candidati (arrivano in ordine crescente)

Costo  $\Theta(nc_c + nc_a) = \Theta(n c_a)$   
 $(c_a > c_c)$

## Caso migliore?

- Il primo assunto è il miglior candidato

## Analisi probabilistica

- I candidati arrivano in ordine **casuale**:
  - ▶ Assegno un **rango** ad ogni candidato:  
 $pos(i)$  è la posizione del candidato  $i$   
 $pos(i) \in 1 \dots n$  e  $pos(i) \neq pos(j)$  sse  $i \neq j$
  - ▶  $\langle pos(1), pos(2), \dots, pos(n) \rangle$  è una **permutazione** della lista dei candidati  $\langle 1, 2, \dots, n \rangle$
  - ▶ Ognuna delle  $n!$  permutazioni ha **uguale probabilità**
- **Idea** dell'analisi probabilistica:  
 Uso conoscenza su distribuzione degli input per analizzare l'algoritmo

## Variabili casuali indicatrici

- Dato uno spazio dei campioni  $S$  e un evento  $A$ , si definisce la variabile casuale indicatrice:

$$I\{A\} = \begin{cases} 1 & \text{se si verifica } A \\ 0 & \text{se non si verifica } A \end{cases}$$

## Lemma

Per un evento  $A$ , sia  $X_A = I\{A\}$

Allora  $E[X_A] = Pr\{A\}$

## Dim

$\bar{A}$  complemento di  $A$ :

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot Pr\{A\} + 0 \cdot Pr\{\bar{A}\} \text{ (def valore atteso)} \\ &= Pr\{A\} \end{aligned}$$

Valore atteso: media ponderata dei possibili valori di  $I\{A\}$

Metodo per conversione tra **probabilità** e **valori attesi**

## Esempi

### Numero atteso di teste lanciando una moneta una volta

- Moneta imparziale
- Spazio dei campioni:  $S = \{T, C\}$
- $Pr\{T\} = Pr\{C\} = 1/2$
- Variabile casuale indicatrice:  
 $X_T = I\{T\}$   
 $X_T$  conta il numero di teste in un lancio
- Dal lemma:  
 $E[X_T] = Pr\{T\} = 1/2$

## Numeri attesi di teste in $n$ lanci

- Questo numero NON è una probabilità !
- $X$ : variabile casuale per il numero di teste in  $n$  lanci
- Variabile casuale indicatrice
  - $X_i = I\{i\text{-mo lancio è evento } T\}$
  - $X = \sum_{i=1}^n X_i$
  - Numero atteso di teste ( $T$ ):  

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n Pr\{T\} = \sum_{i=1}^n 1/2 = n/2$$
- Linearità valori attesi:  
 valore atteso della somma = somma dei valori attesi

## Analisi del problema dell'assunzione

- Candidati arrivano in ordine **casuale**
- $X$  è una variabile casuale:  
 numero di volte che si assume una nuova persona
- Variabili casuali indicatrici**  $X_1, X_2, \dots, X_n$ :

$$X_i = I\{ \text{candidato } i \text{ è assunto} \} = \begin{cases} 1 & \text{se candidato assunto} \\ 0 & \text{se candidato non assunto} \end{cases}$$

- $X = X_1 + X_2 + \dots + X_n$
- Lemma  $\Rightarrow E[X_i] = Pr\{ \text{candidato } i \text{ è assunto} \}$
- Devo **calcolare**  $Pr\{ \text{candidato } i \text{ è assunto} \}$

## $Pr\{ \text{candidato } i \text{ è assunto} \}$

- Candidato  $i$  assunto sse è migliore dei precedenti  $1, 2, \dots, i-1$
- Candidati in ordine casuale  
⇒ ciascuno dei primi  $i$  candidati ha la stessa probabilità di essere il migliore
- $Pr\{ \text{candidato } i \text{ è migliore fino ad ora} \} = 1/i$

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n 1/i = \ln n + O(1) \end{aligned}$$

- **Serie armonica:**  $\sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$
- **Costo atteso di assunzione:**  $O(c_a \ln n)$   
molto meglio del caso peggiore  $O(n c_a)$

## Algoritmi randomizzati

### E' corretta l'analisi vista?

- Posso non conoscere la distribuzione delle interviste (la permutazione dei candidati)
- Esempio: alcune sequenze di interviste **non capitano mai**.
- Non posso essere sicuro che ognuna delle  $n!$  permutazioni capiti con uguale probabilità

### Cambio lo scenario

- Uso la randomizzazione **dentro l'algoritmo** per **imporre** una distribuzione agli input:
  - ▶ L'agenzia manda in anticipo la lista degli  $n$  candidati
  - ▶ Ogni giorno scelgo un candidato a caso dalla lista
- **Forzo** un ordine casuale

## Algoritmi randomizzati

- Un algoritmo è randomizzato se il suo **comportamento** è determinato in parte da valori prodotti con un **generatore di numeri (pseudo-)casuali**
- **RANDOM(a, b)**  
ritorna un intero  $r$  t.c.  $a \leq r \leq b$  e ciascuno dei  $b - a + 1$  valori possibili di  $r$  è ugualmente probabile
- **RANDOM** è implementato con un generatore **pseudo-casuale**: un metodo *deterministico* che ritorna numeri che
  - ▶ superano test statistici e
  - ▶ "appaiono" casuali

## Problema di assunzione randomizzato

- Non intervista sempre i candidati nell'ordine presentato, ma **permuto casualmente** quest'ordine
- **Randomizzazione** nell'algoritmo non nella distribuzione d'ingresso

### RANDOMIZED-HIRE-ASSISTANT( $n$ )

Permutare casualmente la lista di candidati  
HIRE-ASSISTANT( $n$ )

### Costo atteso di RANDOMIZED-HIRE-ASSISTANT: $O(c_a \ln n)$

**Proof:** Avendo permuto l'array di input si ha una situazione identica all'analisi probabilistica di HIRE-ASSISTANT deterministico

## Differenza sottile

- Il costo di esecuzione non dipende da un particolare input
- Il caso peggiore ha la stessa probabilità degli altri

## Permutazione casuale array

- Permuta **sul posto** i dati
- Voglio **permutazione casuale uniforme**, ogni permutazione dei numeri da 1 a  $n$  (sono  $n!$ ) è ugualmente probabile

RANDOMIZE-IN-PLACE( $A$ )

```
 $n \leftarrow A.length$ 
for  $i \leftarrow 1$  to  $n$ 
    scambia  $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$ 
```

## Tempo

$O(1)$  per ogni iterazione  $\Rightarrow O(n)$  totale

## Quicksort

# Quicksort

## Caratteristiche principali

- Caso peggiore:  $\Theta(n^2)$
- Tempo di esecuzione atteso:  $\Theta(n \lg n)$
- Costanti in  $\Theta(n \lg n)$  piccole
- Ordina sul posto

## Per ordinare $A[p..r]$ :

- **Divide:**  $A[p..r]$ , in  $A[p..q - 1]$  e  $A[q + 1..r]$ , t. c.
  - ▶ ogni elemento in  $A[p..q - 1] \leq A[q]$  e
  - ▶  $A[q] \leq$  ogni elemento in  $A[q + 1..r]$
- **Impera:** Ordina i due sottoarray con chiamate a QUICKSORT
- **Combina:** sottoarray ordinati sul posto

Chiamata iniziale:  $\text{QUICKSORT}(A, 1, n)$

$\text{QUICKSORT}(A, p, r)$

```

1  if  $p < r$ 
2     $q \leftarrow \text{PARTITION}(A, p, r)$ 
3     $\text{QUICKSORT}(A, p, q - 1)$ 
4     $\text{QUICKSORT}(A, q + 1, r)$ 

```

$q$  è il pivot

$\text{PARTITION}(A, p, r)$

```

1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$ 
4    if  $A[j] \leq x$ 
5       $i \leftarrow i + 1$ 
6       $A[i] \leftrightarrow A[j]$ 
7   $A[i + 1] \leftrightarrow A[r]$ 
8  return  $i + 1$ 

```

# Correttezza di PARTITION

## Invariante di ciclo

- ①  $A[r] \leftarrow \text{pivot}$
- ②  $\forall A[x] \in A[p..i] : A[x] \leq \text{pivot}$
- ③  $\forall A[x] \in A[i + 1..j - 1] : A[x] > \text{pivot}$
- ④  $\forall A[x] \in A[j..r - 1] : A[x]$  non è stato esaminato

# Correttezza di PARTITION

## Invariante di ciclo

- **Inizializzazione:**  
 $A[r]$  è il pivot,  $A[p..i]$  e  $A[i + 1..j - 1]$  sono vuoti
- **Conservazione:**  
 Se  $A[j] \leq \text{pivot}$ , scambia  $A[j]$  e  $A[i + 1]$  e  $i++$  e  $j++$   
 Se  $A[j] > \text{pivot}$  solo  $j++$
- **Conclusione:** Ciclo termina con  $j = r$

$A[p..i] < \text{pivot}$  ;  $A[i + 1..r - 1] > \text{pivot}$  ;  $A[r] = \text{pivot}$

Linea 7 sposta il pivot nella posizione corretta  $A[i + 1] \leftrightarrow A[r]$

# Analisi di QUICKSORT

## Tempi di esecuzione

- **Tempo** di PARTITION:  
 $\Theta(n)$
- **Tempo** di QUICKSORT  
**dipende** dal partizionamento dei sottoarray:
  - ▶ Se **bilanciati**  $\Rightarrow$  veloce come mergesort
  - ▶ Se **sbilanciati**  $\Rightarrow$  lento come insertion sort
  - ▶ Caso medio analogo a mergesort, ma costanti più piccole

## Intuizione caso peggiore

- Sottoarray completamente sbilanciati (0 in uno e  $n - 1$  nell'altro)
- $$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$
- Come insertion sort
  - Quando capita?
    - ▶ Array **ordinato** (insertion sort:  $O(n)$ )

## Caso migliore

- Sottoarray sempre bilanciati  $\Rightarrow$
- Ognuno  $\leq n/2$  elementi

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

## Partizionamento con proporzionalità costante

- Suppongo PARTITION divida sempre in 9-a-1:

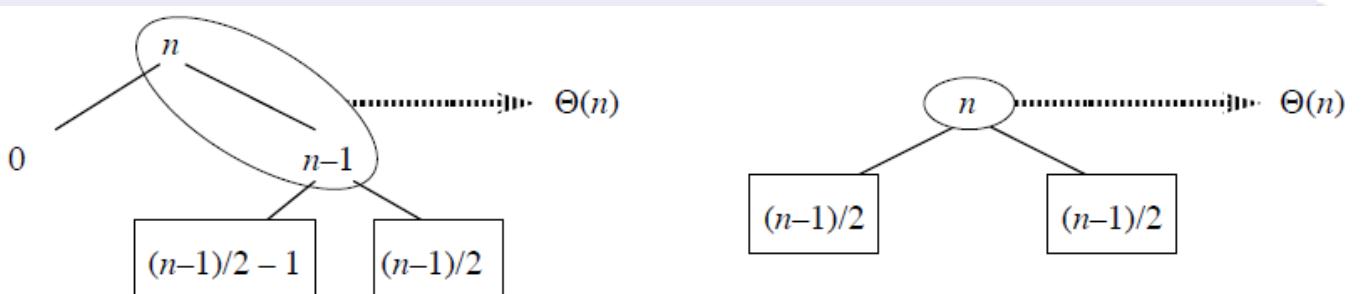
$$\begin{aligned} T(n) &= T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$

- Simile a  $T(n) = T(n/3) + T(2n/3) + O(n)$
- $\log_{10} n$  livelli pieni  
 $\log_{10/9} n$  livelli non vuoti
- $\Omega(\log_{10} n)$  e  $O(\log_{10/9} n)$
- Altezza albero di ricorsione:  $\Theta(\lg n)$
- Intuizione: il tempo di esecuzione medio di Quicksort è molto più vicino al caso migliore che al peggiore

## Intuizione caso medio

Una volta sono fortunato una volta no...

- Alterno tagli buoni e cattivi



- Ogni "nodo" a sx ha  $\Theta(n) + \Theta(n - 1) = \Theta(n)$
- A dx  $\Theta(n)$
- Entrambe  $O(n \lg n)$  anche se costante a sx > costante a dx

# Quicksort randomizzato

- Ipotesi **caso medio**: tutte le permutazioni di ingresso sono ugualmente probabili
  - ▶ Non è sempre vero!!
- **Randomizzo QUICKSORT**
- Permuto casualmente l'input?
- No! Scelgo un **elemento a caso** come pivot

RANDOMIZED-PARTITION( $A, p, r$ )

- 1  $i \leftarrow \text{RANDOM}(p, r)$
- 2 scambia  $A[r] \leftrightarrow A[i]$
- 3 **return** PARTITION( $A, p, r$ )

RANDOMIZED-QUICKSORT( $A, p, r$ )

- 1 **if**  $p < r$
- 2      $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3     RANDOMIZED-QUICKSORT( $A, p, q - 1$ )
- 4     RANDOMIZED-QUICKSORT( $A, q + 1, r$ )

- Randomizzazione **impedisce** a specifici input di causare sempre il **caso peggiore**
- Esempio:  
array ordinato è un caso peggiore per QUICKSORT,  
non per RANDOMIZED-QUICKSORT!

# Analisi di quicksort

## Analizziamo:

- Tempo di esecuzione del caso peggiore di QUICKSORT e RANDOMIZED-QUICKSORT (stesso)
- Tempo atteso di RANDOMIZED-QUICKSORT

## Caso peggiore

Ad ogni livello peggior taglio

Ricorrenza per caso peggiore

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n)$$

**Ipotesi:**  $T(n) \leq cn^2$  (vediamo  $O$ )

$$T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + c_1 \cdot n$$

$$= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + c_1 \cdot n$$

---


$$\max \text{ con } q = 0 \text{ o } q = n - 1 \text{ (scelgo } q = 0\text{)}$$

$$\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$$

$\Rightarrow$

$$T(n) \leq cn^2 - c(2n - 1) + c_1 \cdot n \leq cn^2$$

$$\text{se } c(2n - 1) \geq c_1 \cdot n \Rightarrow \text{c t.c. } c(2n - 1) > c_1 \cdot n$$

$\Rightarrow$  caso peggiore è  $O(n^2)$  E' anche  $\Omega(n^2)$

$\Rightarrow$  caso peggiore  $\Theta(n^2)$

## Caso medio

### In realtà tempo di esecuzione atteso

- Costo dominante: PARTITION
- Ogni volta PARTITION rimuove il pivot  
⇒ PARTITION chiamato al più  $n$  volte
- Costo PARTITION : **costante** + iterazioni in ciclo for (= numero di confronti)
- $X =$  numero **totale** di confronti in tutte le chiamate a PARTITION
- Costo totale  $O(n + X)$
- Calcoliamo un limite asintotico per  $X$

## Quicksort

QUICKSORT( $A, p, r$ )

- 1 **if**  $p < r$
- 2        $q \leftarrow \text{PARTITION}(A, p, r)$
- 3       QUICKSORT( $A, p, q - 1$ )
- 4       QUICKSORT( $A, q + 1, r$ )

PARTITION( $A, p, r$ )

- 1  $x \leftarrow A[r]$
- 2  $i \leftarrow p - 1$
- 3 **for**  $j \leftarrow p$  **to**  $r - 1$
- 4     **if**  $A[j] \leq x$
- 5        $i \leftarrow i + 1$
- 6        $A[i] \leftrightarrow A[j]$
- 7  $A[i + 1] \leftrightarrow A[r]$
- 8 **return**  $i + 1$

## Calcoliamo un limite al numero totale di confronti $X$

- Rinomino elementi di  $A$ :  $z_1, z_2, \dots, z_n$ , t.c.  $z_i \leq z_j$  se  $i < j$
- $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$  elementi compresi tra  $z_i$  e  $z_j$ , inclusi
- $z_i$  e  $z_j$  confrontati al massimo una volta
  - ▶ elementi confrontati solo col pivot
  - ▶ pivot non presente in chiamate successive
- $X_{ij} = I\{z_i \text{ confrontato con } z_j\}$  (in qualunque momento)

$$\bullet X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr\{z_i \text{ e' confrontato con } z_j\}$$

## $Pr\{z_i \text{ confrontato con } z_j\}$

- Numeri **in partizioni diverse mai** confrontati
  - ▶ Se  $z_i < \text{pivot} < z_j \Rightarrow z_i$  e  $z_j$  mai confrontati
- Se  $z_i$  o  $z_j$  scelto per primo in  $Z_{ij}$  allora  $z_i$  o  $z_j$  confrontato con tutti gli altri elementi di  $Z_{ij}$ 
  - ▶  $\Rightarrow z_i$  e  $z_j$  confrontati SSE il **primo** pivot in  $Z_{ij}$  è  $z_i$  o  $z_j$
- Pivot scelti casualmente e indipendentemente  $|Z_{ij}| = j - i + 1$ 
  - $\Rightarrow$  probabilità  $z_i$  o  $z_j$  scelto per primo in  $Z_{ij}$  è  $\frac{1}{j-i+1}$

$$\begin{aligned} Pr\{z_i \text{ e' confrontato con } z_j\} &= Pr\{z_i \text{ o } z_j \text{ primo pivot scelto da } Z_{ij}\} \\ &= Pr\{z_i \text{ primo pivot scelto da } Z_{ij}\} \\ &\quad + Pr\{z_j \text{ primo pivot scelto da } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} \end{aligned}$$

## Sostituiamo in $E[X]$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr\{z_i \text{ e' confrontato con } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1}$$

Cambiamo variabili ( $k = j - i$ )

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) \\ &= O(n \lg n) \end{aligned}$$

⇒ tempo di esecuzione atteso di RANDOMIZED-QUICKSORT

- Era  $O(n + X)$
- Quindi  $O(n \lg n)$

## Ordinamento in tempo lineare

## Ordinamento per confronti

- Usa **confronto** di elementi per ordinare

## Limiti inferiori per ordinamento

Algoritmo	Tempo di esecuzione caso peggiore	Tempo di esecuzione atteso/ caso medio
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (atteso)
...	...	...

## Nessun ordinamento può essere meglio di $n \lg n$ ?

di sicuro  $\Omega(n)$  per esaminare tutti gli input

## Tempo esecuzione ordinamento

### Albero di decisione

- Albero binario che rappresenta i possibili confronti fatti da:
  - ▶ uno specifico algoritmo di ordinamento
  - ▶ su input di una data dimensione
- Conto solo **confronti**
- **Astrae** da tutto il resto: controllo di flusso, spostamento di dati...

Per ogni algoritmo di ordinamento basato su **confronti**

- Un albero per ogni  $n$
- Albero modella tutti i possibili percorsi di esecuzione

## Lunghezza del cammino più lungo dalla radice alle foglie?

### Caso peggiore

- Dipende dall'algoritmo: Insertion sort:  $\Theta(n^2)$  Merge sort:  $\Theta(n \lg n)$

INSERTION-SORT( $A$ )

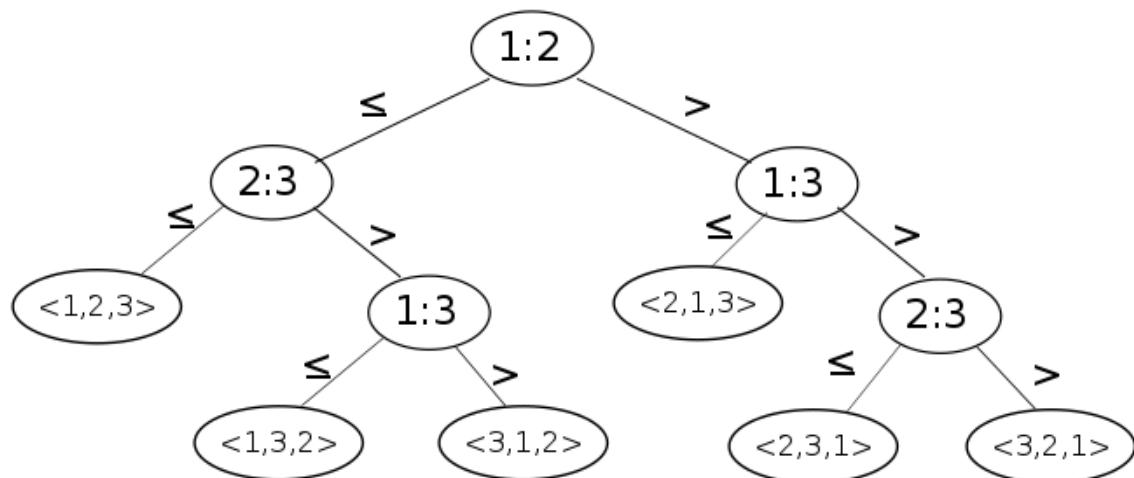
```

1  for  $j \leftarrow 2$  to  $A.length$ 
2       $key \leftarrow A[j]$ 
3       $i \leftarrow j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ 
5           $A[i + 1] \leftarrow A[i]$ 
6           $i \leftarrow i - 1$ 
7       $A[i + 1] \leftarrow key$ 

```

## Insertion sort su 3 elementi

- radice corrisponde a ciclo per  $j = 2$
- altri nodi  $j = 3$
- nodi interni: secondo valore è la chiave
- le foglie sono permutazioni dei valori in ingresso



## Lemma

Ogni albero binario di altezza  $h$  ha al massimo  $2^h$  foglie

$$h = \text{altezza} \quad I_h = \# \text{ di foglie},$$

$$\text{Allora } I_h \leq 2^h$$

## Dim

Induzione su  $h$

- **Base:**  $h = 0$  Albero con un solo nodo (foglia)  $\Rightarrow 2^h = 1$

- **Passo induttivo:**

Supponiamo **vero** per  $h - 1$

Ogni **foglia** dell'albero di altezza  $h - 1$  ha al massimo 2 nuove **foglie**

$$I_h \leq 2 \cdot I_{h-1} = 2 \cdot 2^{h-1} \quad (\text{ipotesi induttiva})$$

$$= 2^h$$

## Teorema

Ogni albero di decisione che ordina  $n$  elementi ha altezza  $\Omega(n \lg n)$

## Dim

- Almeno una foglia per ogni permutazione:  $I \geq n!$

Altrimenti non sarebbe corretto!

- Lemma:

$$2^h \geq I \geq n! \Rightarrow 2^h \geq n!$$

- Ig di entrambi i membri:

$$h \geq \lg(n!)$$

- Approssimazione di Stirling:  $n! > \left(\frac{n}{e}\right)^n$

$$h \geq \lg(n!) \geq \lg\left(\frac{n}{e}\right)^n = n \lg\left(\frac{n}{e}\right)$$

$$= n \lg n - n \lg e = \Omega(n \lg n)$$

- **Nota:**  $\Omega$  perché c'è  $\geq$

## ⇒ Merge sort

E' un **ordinamento per confronto asintoticamente ottimo**

## Ordinamento in tempo lineare

E' possibile ordinare in meno di  $\Omega(n)$ ?

- Ordinamento **non** basato su confronti
- Niente è gratis: abbiamo bisogno di più spazio ...

### Counting sort

- *Ipotesi essenziale :*  
numeri da ordinare sono **numeri naturali**  $\in \{0, 1, \dots, k\}$
- **Input:**  $A[1..n]$ , dove  $A[j] \in \{0, 1, \dots, k\}$  per  $j = 1, 2, \dots, n$
- **Output:**  $B[1..n]$ , ordinati  
 $B$  è già allocato
- **Memoria aggiuntiva:**  $C[0..k]$

COUNTING-SORT( $A, B, k$ )

```

1 sia  $C[0..k]$  un nuovo array
2 for  $i \leftarrow 0$  to  $k$ 
3    $C[i] \leftarrow 0$ 
4 for  $j \leftarrow 1$  to  $A.length$ 
5    $C[A[j]] \leftarrow C[A[j]] + 1$ 
6 //  $C[i]$  ora contiene il numero di elementi uguali a  $i$ 
7 for  $i \leftarrow 1$  to  $k$ 
8    $C[i] \leftarrow C[i] + C[i - 1]$ 
9 //  $C[i]$  ora contiene il numero di elementi minori o uguali a  $i$ 
10 for  $j \leftarrow A.length$  downto 1
11    $B[C[A[j]]] \leftarrow A[j]$ 
12    $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

## Nota

$C$  è riempito a partire dalla posizione 0 (primo valore della chiave)

Esempio:  $A = [2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3]$ 

- $k = 5$
- 1<sup>o</sup> **for** :  $C[0, 0, 0, 0, 0, 0]$
- 2<sup>o</sup> **for** :  $C[2, 0, 2, 3, 0, 1]$
- 3<sup>o</sup> **for** :  $C[2, 2, 4, 7, 7, 8]$
- $\Rightarrow$  posizione di 5 è 8; quelle di 3 sono 7, 6, 5
- 1<sup>a</sup> iterazione ultimo **for**  
 $B[C[A[8]]] = B[C[3]] = B[7] = A[8] = 3_3$   
 $B[-, -, -, -, -, -, 3_3, -]$   
 $C[2, 2, 4, 6, 7, 8]$
- 2<sup>a</sup> iterazione ultimo **for**  
 $B[C[A[7]]] = B[C[0]] = B[2] = A[7] = 0_2$   
 $B[-, 0_2, -, -, -, -, 3_3, -]$   
 $C[1, 2, 4, 6, 7, 8]$

... Esempio:  $A = [2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3]$

- 3<sup>a</sup> iterazione ultimo **for**

$$B[C[A[6]]] = B[C[3]] = B[6] = A[6] = 3_2$$

$$B[-, 0_2, -, -, -, 3_2, 3_3, -]$$

$$C[1, 2, 4, 5, 7, 8]$$

- 4<sup>a</sup> iterazione ultimo **for**

$$B[C[A[5]]] = B[C[2]] = B[4] = A[5] = 2_2$$

$$B[-, 0_2, -, 2_2, -, 3_2, 3_3, -]$$

$$C[1, 2, 3, 5, 7, 8]$$

- 5<sup>a</sup> iterazione ultimo **for**

$$B[C[A[4]]] = B[C[0]] = B[1] = A[4] = 0_1$$

$$B[0_1, 0_2, -, 2_2, -, 3_2, 3_3, -]$$

$$C[0, 2, 3, 5, 7, 8]$$

- ...

- $B[0_1, 0_2, 2_1, 2_2, 3_1, 3_2, 3_3, 5_1]$

## Counting sort è stabile

- Chiavi con lo stesso valore appaiono nello stesso ordine in uscita di quello che avevano in input
- Vero per come funziona l'ultimo ciclo

## Correttezza

Tramite invarianti di ciclo!

## Analisi

- $\Theta(n + k) \Rightarrow \Theta(n)$  se  $k = O(n)$
- Ordinamento in **tempo lineare!!**
- Quanto grande può essere  $k$ ?
  - ▶ OK per valori a 32-bit?  $k = 2^{32} = 4G$  memoria
  - ▶ 16-bit? Forse no
  - ▶ 4-bit? Probabilmente sì
- Counting sort usato in radix sort

## So ordinare su una cifra ...

### Come ordino su $n$ cifre?

- Posso ordinare dalla cifra più significativa, ma...  
dovrei mettere da parte più dati intermedi
- **Esempio:** ordinare i fogli di un libro di 1000 pagine mescolate:  
10 mucchietti per le centinaia,  
poi ognuna ha 10 mucchietti per le decine ecc.

## Radix sort

**Idea** (controintuitiva): ordino le cifre **meno significative** per prime

RADIX-SORT( $A, d$ )

**for**  $i \leftarrow 1$  **to**  $d$

Usa ordinamento **stabile** per ordinare array  $A$  sulla cifra  $i$

## Radix sort

**Idea** (controintuitiva): ordino le cifre **meno significative** per prime

RADIX-SORT( $A, d$ )

**for**  $i \leftarrow 1$  **to**  $d$

Usa ordinamento **stabile** per ordinare array  $A$  sulla cifra  $i$

326  
453  
608  
835  
751  
435  
704  
690

## Radix sort

**Idea** (controintuitiva): Ordino le cifre **meno significative** per prime

RADIX-SORT( $A, d$ )

**for**  $i \leftarrow 1$  **to**  $d$

Usa ordinamento stabile per ordinare array  $A$  sulla cifra  $i$

S	S
326	690
453	751
608	453
835	704
751	→ 835
704	326
690	608

## Radix sort

**Idea** (controintuitiva): Ordino le cifre **meno significative** per prime

RADIX-SORT( $A, d$ )

**for**  $i \leftarrow 1$  **to**  $d$

Usa ordinamento stabile per ordinare array  $A$  sulla cifra  $i$

S	S	S
326	690	704
453	751	608
608	453	326
835	704	835
751	→ 835	→ 435
435	435	751
704	326	453
690	608	690

## Radix sort

**Idea** (controintuitiva): Ordino le cifre **meno significative** per prime

RADIX-SORT( $A, d$ )

**for**  $i \leftarrow 1$  **to**  $d$

Usa ordinamento stabile per ordinare array  $A$  sulla cifra  $i$

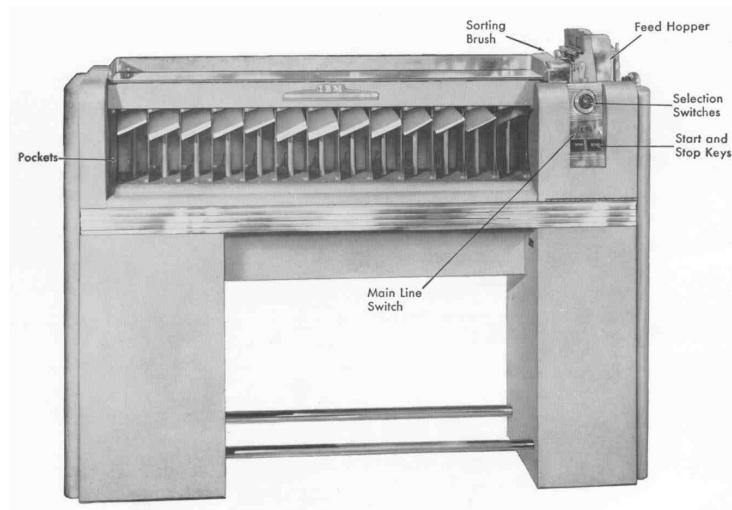
	<b>S</b>	<b>S</b>	<b>S</b>
326	690	704	326
453	751	608	435
608	453	326	453
835	704	835	608
751	→ 835	→ 435	→ 690
435	435	751	704
704	326	453	751
690	608	690	835

## Correttezza

- **Induzione su  $i$  (colonna)**
- Suppongo cifre  $1, 2, \dots, i - 1$  ordinate
  - ▶ Se 2 cifre in posizione  $i$  sono **diverse**  
⇒ ordinamento su posizione  $i$  è corretto  
(posizioni  $1, \dots, i - 1$  irrilevanti)
  - ▶ Se 2 cifre in posizione  $i$  sono **uguali** (es. 23 e 25)  
⇒ numeri sono già nell'ordine giusto (ipotesi induttiva)  
Ordinamento **stabile** sulla cifra  $i$  li lascia nell'ordine giusto
- ⇒ ordinamento **stabile** sulla cifra  $i$  lascia le cifre  $1, \dots, i$  ordinate

## Uso di radix sort

- Usato da IBM negli anni 1930 per ordinare schede perforate (per censimento)
- Macchine ordinatrici di schede lavoravano con una colonna alla volta
- L'uomo "a mano" realizzava radix-sort!



## Analisi di radix-sort

Uso counting sort come ordinamento intermedio:

- $\Theta(n + k)$  per passo (cifre nell'intervallo  $0, \dots, k$ )
- $d$  passi
- $\Rightarrow \Theta(d(n + k))$
- Se  $k = O(n)$   
 $\Rightarrow \Theta(d n)$

## Suddivisione chiavi

- Parole di  $b$  bit divise in "pezzi" di  $r$  bit
- $d = \lceil b/r \rceil$  cifre di  $r$  bit ciascuna
- **Counting sort** con  $k = 2^r - 1$

## Esempio

- Parole di 32 bit, cifre di 8 bit
  - ▶  $b = 32$
  - ▶  $r = 8$
  - ▶  $d = \lceil b/r \rceil = \lceil 32/8 \rceil = 4$
  - ▶  $k = 2^r - 1 = 2^8 - 1 = 255$

## Analisi

- $n$  parole
- Tempo =  $\Theta(\frac{b}{r}(n + 2^r))$

## Scegliere $r$

### Bilanciare $b/r$ e $n + 2^r$

- Scegliendo  $r \approx \lg n$  si ha  
 $\Theta(\frac{b}{r}(n + 2^r)) = \Theta(\frac{b}{\lg n}(n + n)) = \Theta(\frac{b n}{\lg n})$
- Meglio di  $\Theta(n)$
- Se fosse  $r < \lg n$ ,  $\Rightarrow \frac{b}{r} > \frac{b}{\lg n}$ ,  $\Rightarrow n + 2^r$  non migliora e resta a  $\Theta(n)$
- Se fosse  $r > \lg n$ ,  $\Rightarrow n + 2^r$  diventa più grande di  $n$   
 Esempio se  $r = 2 \lg n \Rightarrow 2^r = 2^{2 \lg n} = (2^{\lg n})^2 = n^2$

## Esempio

- Per ordinare  $2^{16}$  numeri a 32 bit  
 usare  $r = \lg 2^{16} = 16$  bit  
 $\lceil b/r \rceil = 2$  passi

## Radix sort **vs** merge sort e quicksort

- 1 milione ( $2^{20}$ ) di interi a 32-bit
- **Radix sort:**  $\lceil \frac{32}{20} \rceil = 2$  passi (su  $n$  dati)
  - ▶ Nota: in radix sort un “passo” è  $2n$ :  
uno per counting sort  
uno per muovere i dati da  $B$  ad  $A$  (counting sort non ordina sul posto)
- **Merge sort/quicksort:**  $lgn = 20$  passi (su  $n$  dati)
- $\Rightarrow 4n$  vs  $20n$

## Limiti inferiori per ordinamento

Algoritmo	Tempo di esecuzione caso peggiore	Tempo di esecuzione atteso/ caso medio
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (atteso)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(k + n))$	$\Theta(d(k + n))$

# Hashing

## Insiemi dinamici

- Operazioni su **dizionari**:  
INSERT, SEARCH, DELETE
- **Tabelle hash**: implementazione efficiente di dizionario
  - ▶ tempo atteso di ricerca:  $O(1)$
  - ▶ tempo di ricerca, caso peggiore:  $\Theta(n)$
- Generalizzazione di **indirizzamento diretto**:
  - ▶ Elemento con chiave  $k$  nella posizione  $k$

## Tabelle ad indirizzamento diretto

- Insieme dinamico
- Universo delle chiavi:  $U = \{0, 1, \dots, m - 1\}$
- Se  $m$  non è troppo grande  $\Rightarrow$ 
  - ▶ tabella ad indirizzamento diretto:  $T[0 \dots m - 1]$
  - ▶ con chiavi **duplicate**: lista collegata

DIRECT-ADDRESS-SEARCH( $T, k$ )

return  $T[k]$

DIRECT-ADDRESS-INSERT( $T, x$ )

$T[x.\text{key}] \leftarrow x$

DIRECT-ADDRESS-DELETE( $T, x$ )

$T[x.\text{key}] \leftarrow \text{NIL}$

**Tempo?**

## Tabelle ad indirizzamento diretto

## Tabelle hash

- Spesso ... tabella con  $|U|$  elementi è troppo grande
- Spesso ...  $|K| \ll |U|$  ( $K$  chiavi memorizzate)
- Usiamo **funzione hash**  
Memorizzo  $k$  in  $T[h(k)]$  (non in  $T[k]$ )
- $h : U \rightarrow \{0, 1, \dots, m - 1\}$
- Funzione hash è **non iniettiva**

Si hanno **collisioni**:

$$h(k) = h(k') \text{ con } k \neq k'$$

- ▶ Può capitare se  $|K| \leq m$
- ▶ Capita sicuramente se  $|K| > m$

- Due metodi per gestire collisioni:
  - ▶ Concatenamento
  - ▶ Indirizzamento aperto

## Funzioni hash

$$h : U \rightarrow \underbrace{\{0, 1, \dots, m - 1\}}_{\text{numero cella}}$$

- $h(k)$  dovrebbe realizzare un **Hash uniforme semplice**:

*Ogni elemento ha la stessa probabilità di andare in ogni cella indipendentemente da dove vanno gli altri elementi*

- In pratica non è possibile
  - ▶ non conosco la distribuzione di probabilità da cui sono prese le chiavi
  - ▶ le chiavi potrebbero non essere prese indipendentemente una dall'altra
- Si usano informazioni sul dominio delle chiavi per creare una buona funzione hash

## Metodo delle divisioni

- $h(k) = k \bmod m$
- $a \bmod m = a - \left\lfloor \frac{a}{m} \right\rfloor \cdot m$
- **Esempio:**  
 $m = 20$  e  $k = 91 \Rightarrow h(k) = 11$
- **Vantaggio:** Veloce
- **Svantaggio:** Evitare alcuni valori di  $m$ :
  - ▶ Evitare potenze intere di 2  
Perché ?  
Se  $m = 2^p \Rightarrow h(k)$  sono le  $p$  cifre meno significative di  $k$
  - ▶ Critico per rappresentazioni di stringhe!
- Buona scelta per  $m$ :  
*numero primo non troppo vicino ad una potenza di 2*

## Chiavi: numeri naturali

- $h(k)$  t.c.  $k \in \mathbb{N}$
- Se  $k \notin \mathbb{N}$ ? Convertire  $k$
- **Stringa** di caratteri?
- Notazione posizionale con opportuna base
  - ▶ Esempio: CLRS
  - ▶ ASCII: C = 67, L = 76, R = 82, S = 83
  - ▶ 7 bit (ASCII di base)
  - ▶  $\Rightarrow$   
 $CLRS \rightarrow (67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0)$   
 $= 141,764,947$

## Risoluzione collisioni con concatenamento

Tutti gli elementi con lo stesso hash sono in una **lista collegata**

### Inserimento

CHAINED-HASH-INSERT( $T, x$ )

inserisce  $x$  in testa alla lista  $T[h(x.key)]$

- Tempo nel caso **peggiore?**  $O(1)$
- Se l'elemento non è già nella lista
- Altrimenti: un'altra ricerca

## Risoluzione collisioni con concatenamento

## Ricerca

CHAINED-HASH-SEARCH( $T, k$ )

**return** ricerca un elemento con chiave  $k$  nella lista  $T[h(k)]$

- **Tempo** ? proporzionale alla **lunghezza** della lista in  $h(k)$

## Cancellazione

CHAINED-HASH-DELETE( $T, x$ )

cancella  $x$  dalla lista  $T[h(x.key)]$

- Tempo nel caso **peggiore** è  $O(1)$   
con liste doppiamente collegate  
(non considera il tempo per trovare l'elemento da cancellare)
- Con liste semplicemente collegate, costa quanto la ricerca

- **Non** è una struttura dati **molto dinamica**

- ▶ si può *raddoppiare / dimezzare* la tabella via via
- ▶ quanto costa?

## Esercizio su hash

- Si consideri una tabella hash con  $m = 13$  e con funzione hash calcolata con il metodo delle divisioni
- Si consideri la seguente successione di chiavi:  
 $< 5, 21, 10, 34, 47, 3, 18, 35, 11, 23 >$ 
  - Indicare la configurazione di una tabella hash a indirizzamento diretto (con concatenamento) dopo l'inserimento di tutte le chiavi precedenti.
  - Si cancellino nell'ordine le seguenti chiavi:  $< 18, 21, 34, 11 >$  e si disegni la configurazione ottenuta dopo aver eseguito tutte le cancellazioni.

## Analisi di hash con concatenamento

- Tempo richiesto per:
  - ▶ Ricerca con successo
  - ▶ Ricerca senza successo
- **Fattore di caricamento**  $\alpha = \frac{n}{m}$ 
  - ▶  $n$  = # di elementi memorizzati nella tabella
  - ▶  $m$  = # di slot nella tabella (# di liste collegate)
  - ▶  $\alpha$ : numero medio di elementi in ogni lista collegata
  - ▶ si può avere  $\alpha < 1, \alpha = 1$ , o  $\alpha > 1$
- Caso **peggiore**: tutte le  $n$  chiavi hanno hash nello stesso slot  
Una singola lista di lunghezza  $n$   
 $\Rightarrow$  tempo:  $\Theta(1) + \Theta(n) = \Theta(n)$
- Caso **medio** dipende dalla funzione hash

## Caso medio di hash con concatenamento

- Ipotesi: **hash uniforme semplice**: per ogni elemento ognuno degli  $m$  è ugualmente probabile come hash
- $n_j = \text{lunghezza lista } T[j] (j = 0, 1, \dots, m - 1)$
- $\Rightarrow n = n_0 + n_1 + \dots + n_{m-1}$
- Valor atteso di  $n_j$ :  $E[n_j] = \alpha = \frac{n}{m}$
- Calcolo la funzione hash in  $O(1)$   
 $\Rightarrow$  tempo per cercare l'elemento con chiave  $k$  dipende dalla lunghezza  $n_{h(k)}$  della lista  $T[h(k)]$

**Due casi:**

- ▶ Ricerca **senza successo**
- ▶ Ricerca **con successo**

## Ricerca senza successo

### Teorema

Una ricerca senza successo richiede un tempo atteso di  $\Theta(1 + \alpha)$

### Dimostrazione

- Hash uniforme semplice  $\Rightarrow$  l'hash di ogni chiave non già nella tabella finisce in ognuno degli  $m$  slot con la stessa probabilità
- Per ricercare **senza successo** ogni chiave  $k$  bisogna cercare **fino alla fine della lista**  $T[h(k)]$
- $T[h(k)]$  ha **lunghezza attesa**  $E[n_{h(k)}] = \alpha$
- $\Rightarrow$  con calcolo della funzione hash:  
 $\Theta(1 + \alpha)$

## Ricerca con successo:

- Diverso da ricerca con successo
- Ogni lista **non ha** la stessa probabilità di essere oggetto delle ricerche
- Questa è proporzionale al numero di elementi nella lista

### Teorema

Una ricerca con successo richiede un tempo atteso  $\Theta(1 + \alpha)$

### Dimostrazione

- Cerco  $x$  (può essere **qualsiasi** degli  $n$  elementi memorizzati)
- Gli  $n_x$  elementi **prima** di  $x$  nella lista  $h(x.key)$  sono stati **inseriti dopo**  $x$  in  $h(x.key)$   
 $\Rightarrow$  Cercando  $x$  esaminò  $n_x + 1$  elementi
- Calcolo numero atteso di elementi inseriti in  $h(x.key)$  dopo  $x$
- $x_i$ :  $i$ -mo elemento inserito nella tabella ( $i = 1, 2, \dots, n$ )
- $k_i = x_i.key$
- Variabile casuale indicatrice  
 $X_{ij} = I\{h(k_i) = h(k_j)\}$
- Hash uniforme semplice  $\Rightarrow Pr\{h(k_i) = h(k_j)\} = \frac{1}{m} \Rightarrow E[X_{ij}] == \frac{1}{m}$

- Numero atteso di elementi esaminati  
(ricerca **con successo**  $\Rightarrow$  valore atteso sugli  $n$  elementi inseriti)

$$\begin{aligned}
 E\left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij}\right)\right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}]\right) = \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) = 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i\right) \\
 &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2}\right) = 1 + \frac{n-1}{2m} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
 \end{aligned}$$

- Tempo totale (con calcolo funzione hash)  
 $\Theta\left(2 + \frac{\alpha}{2} - \frac{\alpha}{2n}\right) = \Theta(1 + \alpha)$
- Se  $n = O(m) \Rightarrow \alpha = O(1) \Rightarrow \Theta(1)$

## Indirizzamento aperto

### Idea

- Memorizza tutte le chiavi nella tabella hash
- Ogni slot contiene una chiave o NIL

### Cercare chiave $k$

- Ispeziona slot  $h(k)$  :
  - Se slot  $h(k)$  contiene la chiave  $k \Rightarrow$  ricerca con successo
  - Se contiene NIL  $\Rightarrow$  ricerca senza successo.
  - Se  $h(k)$  contiene  $k' \neq k$ 
    - Calcola l'indice di qualche altro slot da ispezionare e vai a 1)
- La successione di slot ispezionati deve essere una **permutazione** dei numeri di slot  $<0, 1, \dots, m-1>$ 
  - esamino tutti gli slot se devo
  - non esaminiamo nessuno slot più di una volta

## Funzione hash, con indirizzamento aperto

$$h : U \times \underbrace{\{0, 1, \dots, m-1\}}_{\text{numero ispezione}} \rightarrow \underbrace{\{0, 1, \dots, m-1\}}_{\text{numero cella}}$$

- $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  (**sequenza di ispezione**) deve essere una **permutazione** di  $\langle 0, 1, \dots, m-1 \rangle$
- Esempio:  $h(k, i) = (h'(k) + i) \bmod m$  ( $h'(k) = k \bmod m$ )

HASH-SEARCH( $T, k$ )

```

1   $i \leftarrow 0$ 
2  repeat
3       $j \leftarrow h(k, i)$ 
4      if  $T[j] = k$ 
5          return  $j$ 
6       $i \leftarrow i + 1$ 
7  until  $T[j] = \text{NIL}$  or  $i = m$ 
8  return  $\text{NIL}$ 
```

## Inserimento

- Cerco il valore e lo inserisco nel primo NIL trovato

HASH-INSERT( $T, k$ )

```

1   $i \leftarrow 0$ 
2  repeat
3       $j \leftarrow h(k, i)$ 
4      if  $T[j] = \text{NIL}$ 
5           $T[j] \leftarrow k$ 
6          return  $j$ 
7      else  $i \leftarrow i + 1$ 
8  until  $i = m$ 
9  error "hash table overflow"
```

- Ritorna il numero dello slot che prende la chiave  $k$  o indica un errore se non c'è uno slot vuoto dove mettere  $k$

## Esercizio su hash

- Si consideri una tabella hash con  $m = 13$  e con funzione hash calcolata con il metodo delle divisioni
- Si consideri la seguente successione di chiavi:  
 $< 5, 21, 10, 34, 47, 3, 18, 35, 11, 23 >$ 
  - ① Indicare la configurazione di una tabella hash a indirizzamento aperto (con esplorazione lineare) dopo l'inserimento di tutte le chiavi precedenti.
  - ② Si cancellino nell'ordine le seguenti chiavi:  $< 18, 21, 34, 11 >$  e si disegni la configurazione ottenuta dopo aver eseguito tutte le cancellazioni.

## Cancellazione

- Posso semplicemente mettere NIL nello slot contenente la chiave  $k$  da cancellare?
- No! Potrei interrompere la catena di ricerca per una chiave  $k'$  inserita dopo  $k$
- **Soluzione:** Uso un valore speciale DEL (deleted) invece di NIL quando cancello
- **Ricerca** tratta DEL come se lo slot contenesse una chiave che non corrisponde a quella cercata
- **Inserimento** tratta DEL come se lo slot fosse vuoto per riusarlo (piccola modifica del codice)
- **Svantaggio:** ora il tempo di ricerca non dipende più da  $\alpha$

## Calcolare la sequenza di esplorazioni

### Hash uniforme

- Ciascuna chiave è ugualmente probabile che generi ciascuna delle  $m!$  permutazioni di  $< 0, 1, \dots, m - 1 >$  come sequenza di esplorazione
- Generalizza l'**hash uniforme semplice**
- Difficile implementare un vero hash uniforme  $\Rightarrow$
- Si approssima garantendo che almeno la sequenza di esplorazione sia una **permutazione** di  $< 0, 1, \dots, m - 1 >$
- **Nessuna di queste tecniche può produrre tutte le  $m!$  sequenze di esplorazione**
- Si usano funzioni hash ausiliarie  

$$h'() : U \rightarrow \{0, 1, \dots, m - 1\}$$

## Esplorazione lineare

- $h(k, i) = (h'(k) + c_i) \text{ mod } m$   
Data la chiave  $k$  e il numero di esplorazione  $i$  ( $0 \leq i < m$ )
- L'esplorazione iniziale determina l'intera sequenza
- Si hanno **solo  $m$  possibili sequenze** (non  $m!$ )
- Si ha **clustering** (addensamento) **primario**: lunghe sequenze (run) di slot occupati
- *Run lunghi tendono a diventare più lunghi*: uno slot vuoto preceduto da  $i$  slot pieni verrà riempito successivamente con probabilità  $(i+1)/m$

## Esplorazione quadratica

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \text{ mod } m$   
 $c_1, c_2 \neq 0$  costanti
- Bisogna scegliere  $c_1, c_2$ , e  $m$  per assicurare che si abbia una permutazione completa di  $\langle 0, 1, \dots, m-1 \rangle$
- Ad esempio:  $c_1 = \frac{1}{2}, c_2 = \frac{1}{2}$
- Può esserci **clustering secondario**: se  $h'(k) = h'(k')$  ( $k \neq k'$ ) allora  $k$  e  $k'$  avranno la stessa sequenza di esplorazione

## Doppio hash

- Usa due funzioni hash ausiliarie:
  - ▶  $h_1$  per prima esplorazione
  - ▶  $h_2$  per esplorazioni restanti
- $h(k, i) = (h_1(k) + h_2(k) \cdot i) \text{ mod } m$ .
- Bisogna avere  $h_2(k)$  primo relativo con  $m$  (no fattori comuni) per garantire che la sequenza di esplorazione sia una permutazione completa di  $\langle 0, 1, \dots, m - 1 \rangle$ 
  - ▶ Scegliere  $m$  come una potenza di 2 e  $h_2$  che produca sempre un numero dispari  $> 1$
  - ▶ Scegliere  $m$  primo e:  

$$h_1(k) = k \text{ mod } m$$

$$h_2(k) = 1 + (k \text{ mod } m').$$
 Con  $m'$  più piccolo di  $m$  (es.  $m' = m - 1$ )
- Si hanno  $\Theta(m^2)$  ( $<< m!$ ) diverse sequenze di esplorazione ( $m$  per  $h_1(k)$  e  $m$  per  $h_2(k)$ )
- Comunque migliore di  $m$  (per hash lineare)

## Analisi di indirizzamento aperto

- **Ipotesi**
  - ▶ Tabella non piena  $0 \leq n < m \Rightarrow 0 \leq \alpha < 1$
  - ▶ Supponiamo **hash uniforme**
  - ▶ No cancellazioni

## Ricerca senza successo

Numero atteso di esplorazioni è al più  $\frac{1}{1-\alpha}$

## Inserimento

Numero atteso di esplorazioni è al più  $\frac{1}{1-\alpha}$

## Ricerca con successo

Numero atteso di esplorazioni è al più  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

# Funzioni hash (cont.)

$h : U \rightarrow \underbrace{\{0, 1, \dots, m-1\}}_{\text{numero cella}}$

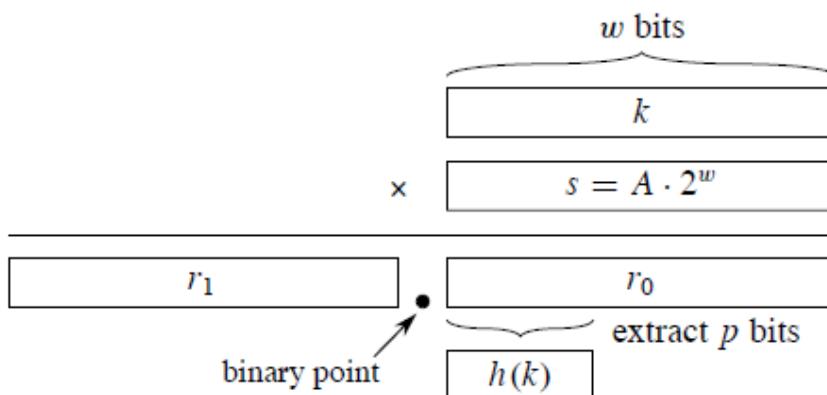
## Metodo delle moltiplicazioni

- Scegliere una costante  $A$ :  $0 < A < 1$   

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$
- $kA \bmod 1 = kA - \lfloor kA \rfloor =$  parte frazionaria di  $kA$
- **Svantaggio:** Più lento del metodo delle divisioni
- **Vantaggio:** Il valore di  $m$  non è critico

## Implementazione (relativamente) facile

- $m = 2^p$  ( $p$  intero)
- $w$ : numero di bit della parola della macchina
- Supponiamo che  $k$  stia in una sola parola ( $k$  richiede  $w$  bit)
- Scelgo  $s$  intero  $0 < s < 2^w$  ( $s$  richiede  $w$  bit)
- $A = \frac{s}{2^w}$  ( $0 < A < 1$ )



$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- $k \cdot s = r_1 2^w + r_0$  (prodotto di due parole di  $w$  bit)
- $r_1$  : parte intera di  $kA$ : ( $r_1 = \lfloor kA \rfloor$ )
- $r_0$  : parte frazionaria di  $kA$ : ( $r_0 = kA \bmod 1 = kA - \lfloor kA \rfloor$ )
- $\lfloor m(kA \bmod 1) \rfloor \Rightarrow$ 
  - ▶ shift a sinistra  $r_0$  di  $p = \lg m$  bit
  - ▶ prendo i  $p$  bit a sinistra del punto binario
- Non serve shiftare prendo i  $p$  bit più significativi di  $r_0$

## Esempio

- $m = 8$  (quindi  $p = 3$ ),  $w = 5$ ,  $k = 21$
- $0 < s < 2^5$  scelgo  $s = 13 \Rightarrow A = \frac{13}{32}$
- Calcolare  $h(k)$  con la formula o con l'implementazione.
- Con la formula:
  - ▶  $kA = 21 \cdot 13/32 = 273/32 = 8 + \frac{17}{32}$
  - ▶  $kA \bmod 1 = 17/32$
  - ▶  $m(kA \bmod 1) = 8 \cdot 17/32 = 17/4 = 4 + \frac{1}{4}$
  - ▶  $\lfloor m(kA \bmod 1) \rfloor = 4$
  - ▶  $\Rightarrow h(k) = 4$
- Con l'implementazione:
  - ▶  $k \cdot s = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17$
  - ▶  $r_1 = 8, r_0 = 17$
  - ▶ Scritto con  $w = 5$  bit,  $r_0 = 10001$
  - ▶ Prendo i  $p = 3$  bit più significativi di  $r_0$
  - ▶ 100 in binario o 4 in decimale
  - ▶  $\Rightarrow h(k) = 4$

## Come scegliere A

- Il metodo delle moltiplicazioni funziona con ogni valore  $0 < A < 1$
- Lavora meglio con alcuni valori rispetto ad altri sulla base della chiave su cui fare l'hash
- Knuth suggerisce di usare  $A \approx \frac{\sqrt{5}-1}{2}$
- Quindi **dato w si sceglie s intero t.c.**  $\frac{s}{2^w} \approx (\sqrt{5} - 1)/2$

## Hash randomizzato

- Conoscendo la funzione hash usata un “avversario sleale” potrebbe inviare tutte le chiavi nello stesso slot
- Potrei randomizzare l’hash come con quicksort randomizzato?
- Scelgo casualmente la funzione *all’inizio* dell’utilizzo di una tabella (non per ogni calcolo della funzione)
- Come per quicksort randomizzato: **nessun input** può provocare sistematicamente il comportamento nel caso peggiore dell’algoritmo

## Hash universale

- $p$  è un numero primo t.c.  $\forall k$  si ha  $0 \leq k \leq p - 1$
- scelgo  $a$  e  $b$  naturali t.c.  $0 \leq a \leq p - 1$  e  $0 \leq b \leq p$
- funzione hash:  $h_{a,b} = ((a \cdot k + b) \bmod p) \bmod m$
- esempio:  $h_{3,4}(8) = 5$
- Scegliendo  $a$  e  $b$  casualmente ho una famiglia con  $p(p - 1)$  funzioni hash diverse per un certo  $m$

## Hash perfetto

- Scenario: insieme di chiavi **statico**
- **Hash perfetto:** complessità è  $O(1)$  nel caso peggiore!
  - ▶ **Idea:** provo con varie funzioni hash fino a trovarne una che non genera collisioni
  - ▶ **Problemi:**
    - ★  $m$  dovrebbe essere troppo grande
    - ★ potrei dover provare tante (troppe) volte
- **Soluzione:** hash a **due livelli** con hash universale in ogni livello
  - ▶ **Primo livello** è come hash con concatenamento
  - ▶ **Secondo livello:** tabelle hash con dimensioni diverse ( $n_i^2$  dove  $n_i$  sono i valori in  $T[i]$ )
  - ▶ Si basa su un **teorema**: *Se memorizzo  $n$  chiavi in una tabella hash con  $m = n^2$  utilizzando una funzione  $h$  da una classe universale di funzioni allora la probabilità che si verifichi una collisione è  $< \frac{1}{2}$*
  - ▶ **Nota:** probabilità  $< \frac{1}{2}$  non certezza, devo provare varie funzioni per ogni  $T[i]$  per non avere collisioni nelle tabelle di secondo livello

# Analisi ammortizzata

## Analisi ammortizzata

- Studia **tempo** per eseguire una **sequenza** di operazioni (**diverse**) su una struttura dati
- Mostra che anche se alcune operazioni sono costose in **media** il costo per operazione è piccolo

Non si fa una media su una distribuzione di input

- Non c'è probabilità
- Considero **costo medio nel caso peggiore** per una sequenza di  $n$  operazioni
- Posso ottenere il costo medio per operazione (**costo ammortizzato**)

# Tabelle dinamiche

- Tabella hash di dimensione  $m$
- $n$  varia
- Se  $\alpha = \frac{n}{m} = 1$   
rialloco con  $m' > m$  e copio gli oggetti
- Se  $\alpha \leq 0.5$   
alloco tabella più piccola e copio gli oggetti
- **Obiettivi**
  - 1 Tempo ammortizzato per operazione  $O(1)$
  - 2 Poco spazio inutilizzato

## TABLE-INSERT

- Inizialmente  $T.num = T.size = 0$

TABLE-INSERT( $T, x$ )

```

if  $T.size = 0$ 
    alloca tabella  $T.table$  con 1 elemento
     $T.size \leftarrow 1$ 
if  $T.num = T.size$ 
    alloca  $NT$  con  $2 \cdot T.size$  elementi
    inserisci gli elementi di  $T.table$  in  $NT$ 
    free  $T.table$ 
     $T.table \leftarrow NT$ 
     $T.size \leftarrow 2 \cdot T.size$ 
inserisci  $x$  in  $T.table$ 
 $T.num \leftarrow T.num + 1$ 

```

## Tempo di esecuzione

- Conto solo inserimento  
Altri costi costanti per ogni chiamata
- Costa ( $O(1)$ ) per ogni inserimento
- Supponiamo di fare solo inserimenti
- $c_i = \text{costo operazione } i\text{-ma}$ 
  - ▶ tabella **non piena**:  $c_i = 1$
  - ▶ tabella **piena**:  $c_i = i$  (copio  $i$  elementi)
  - ▶  $\Rightarrow c_i = O(i)$  in generale
- **Analisi superficiale** (caso peggiore)
  - ▶  $n$  operazioni,  $c_i = O(i)$
  - ▶  $\Rightarrow$  tempo per  $n$  operazioni  $O(n^2)$  (serie aritmetica)
- Analisi non corretta
  - ▶ espando solo se  $i - 1$  è una potenza esatta di 2 (tabella piena)!

## Analisi corretta

- Non si espande sempre:

$$c_i = \begin{cases} i & \text{se } i - 1 \text{ e' una potenza esatta di 2} \\ 1 & \text{altrimenti} \end{cases}$$

$$\text{Costo totale} = \sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \text{ (serie geometrica)}$$

$$= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} = n + 2n - 1$$

$$< n + 2n = 3n$$

- Per **analisi aggregata** il costo ammortizzato per operazione è 3
- Serie geometrica:  $\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}$

# Alberi binari di ricerca

## Alberi di ricerca

- Supportano operazioni dinamiche
- Usati spesso come dizionario
- Operazioni di base in  $O(h)$  ( $h =$  altezza albero)
- Vari tipi di alberi di ricerca:
  - ▶ alberi binari di ricerca (ABR)
  - ▶ alberi Rosso-Neri (ARN)
  - ▶ alberi B

# Alberi Binari di Ricerca

## Alberi binari di ricerca

- Rappresentato con struttura dati collegata (ogni nodo è un oggetto)
  - ▶  $T.root$  punta a radice albero  $T$
  - ▶ Campi:
    - ★ *chiave* (ed eventualmente dati satellite)
    - ★ *left*: punta a figlio sx
    - ★ *right*: punta a figlio dx
    - ★ *p*: punta a padre  $T.root.p = NIL$

## Proprietà albero binario di ricerca

- Se  $y$  è nel sottoalbero sx di  $x$ , allora  $y.key \leq x.key$
- Se  $y$  è nel sottoalbero dx di  $x$ , allora  $y.key \geq x.key$

# Inserimento

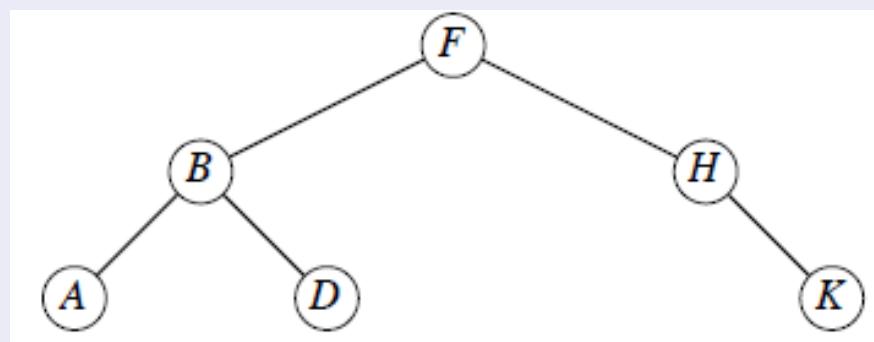
$\text{TREE-INSERT}(T, z)$

```

 $y \leftarrow \text{NIL}$  // Tiene traccia del padre
 $x \leftarrow T.\text{root}$ 
while  $x \neq \text{NIL}$ 
     $y \leftarrow x$ 
    if  $z.\text{key} < x.\text{key}$ 
         $x \leftarrow x.\text{left}$ 
    else  $x \leftarrow x.\text{right}$ 
 $z.p \leftarrow y$ 
if  $y = \text{NIL}$ 
     $T.\text{root} \leftarrow z$  // L'albero  $T$  era vuoto
elseif  $z.\text{key} < y.\text{key}$ 
     $y.\text{left} \leftarrow z$ 
else  $y.\text{right} \leftarrow z$ 

```

# Inserimento



Inserire C

## Tempo?

- $O(h)$

## Quanto vale $h$ ?

- Nel caso migliore  $h = \lg n$
- Nel caso peggiore  $h = n$
- Da cosa dipende?
- Dall'**ordine** di inserimento

## Interrogare ABR

Chiamata: TREE-SEARCH( $\text{root}[T], k$ )

TREE-SEARCH( $x, k$ )

```

if  $x = \text{NIL}$  or  $k = x.\text{key}$ 
    return  $x$ 
if  $k < x.\text{key}$ 
    return TREE-SEARCH( $x.\text{left}, k$ )
else return TREE-SEARCH( $x.\text{right}, k$ )

```

## Tempo ?

$O(h)$

## Versione iterativa

- iterativa più veloce
- ricorsiva più intuitiva

ITERATIVE-TREE-SEARCH( $x, k$ )

```

while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
    if  $k < x.\text{key}$ 
         $x \leftarrow x.\text{left}$ 
    else  $x \leftarrow x.\text{right}$ 
return  $x$ 
```

## Correttezza?

Invariante di ciclo

## Minimo e Massimo

per proprietà ABR

- **Chiave minima** è nel nodo più a sinistra
- **Chiave massima** è nel nodo più a destra

TREE-MINIMUM( $x$ )

```

while  $x.\text{left} \neq \text{NIL}$ 
     $x \leftarrow x.\text{left}$ 
return  $x$ 
```

TREE-MAXIMUM( $x$ )

```

while  $x.\text{right} \neq \text{NIL}$ 
     $x \leftarrow x.\text{right}$ 
return  $x$ 
```

## Tempo ?

- $O(h)$

# Attraversamento simmetrico di un albero (*inorder*)

INORDER-TREE-WALK( $x$ )

```
if  $x \neq \text{NIL}$ 
    INORDER-TREE-WALK( $x.\text{left}$ )
    stampa  $x.\text{key}$ 
    INORDER-TREE-WALK( $x.\text{right}$ )
```

- Anche:
  - ▶ **attraversamento anticipato di un albero** (*preorder*)
  - ▶ **attraversamento posticipato di un albero** (*postorder*)
- Con *inorder*  
elementi sono stampati in ordine monotonamente crescente

## Correttezza

Per induzione dalla proprietà dell'albero binario di ricerca

## Tempo

- *Intuitivamente*  $\Theta(n)$  (albero con  $n$  nodi) perché visita ogni nodo una volta sola
- $T(n) = \Omega(n)$  (visita **tutti** gli  $n$  nodi)
- E' possibile mostrare che  $T(n) = O(n)$

$$T(n) = O(n)$$

- $c$ : costo per sottoalbero vuoto ( $T(0)$ )
- Per  $n > 0$  un nodo ha:  
 $k$  nodi in sottoalbero sx e  
 $n - k - 1$  nodi in sottoalbero dx
- $\Rightarrow T(n) \leq T(k) + T(n - k - 1) + d$   
 $(d = \text{costo per eseguire la funzione})$
- Ipotesi di sostituzione:  $T(n) \leq (c + d)n + c$   
 Per  $n = 0$ :  $(c + d) \cdot 0 + c = c = T(0)$   
 Per  $n > 0$ :

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d = (c + d)n + c \end{aligned}$$

## Successore e predecessore

- Successore di  $x$ :
  - ▶  $y$  t.c.  $y.key$  è la più piccola chiave  $> x.key$
- Se  $x$  ha la chiave più grande
  - ▶  $\Rightarrow$  successore di  $x$  è NIL

## Due casi

- ①  $x$  ha un sottoalbero **destro non vuoto**  
 $\Rightarrow$  Successore di  $x$  è il **minimo** nel **sottoalbero destro** di  $x$
- ②  $x$  ha un sottoalbero **destro vuoto**  
 $\Rightarrow$  Successore ( $y$ ) di  $x$  è  
 antenato più prossimo di  $x$  il cui figlio sinistro è anche antenato di  $x$

TREE-PREDECESSOR è simmetrico di TREE-SUCCESSOR

TREE-SUCCESSOR( $x$ )

```

if  $x.right \neq NIL$ 
    return TREE-MINIMUM( $x.right$ )
 $y \leftarrow x.p$ 
while  $y \neq NIL$  and  $x = y.right$ 
     $x \leftarrow y$ 
     $y \leftarrow y.p$ 
return  $y$ 

```

## Tempo ?

- $O(h)$
- TREE-SUCCESSOR e TREE-PREDECESSOR visitano nodi su un cammino verso il basso o verso l'alto dell'albero

## Trapianto di sottoalberi

- TRAPIANTO( $T, u, v$ )
- sostituisce:
  - ▶ sottoalbero con radice  $u$
  - ▶ con quello con radice  $v$
- aggiustando collegamenti con il padre
- nota: non fa uno **scambio!**

TRAPIANTO( $T, u, v$ )

```

if  $u.p = \text{NIL}$ 
     $T.root \leftarrow v // u$  era la radice di  $T$ 
elseif  $u = u.p.left$ 
     $u.p.left \leftarrow v$ 
else  $u.p.right \leftarrow v$ 
if  $v \neq \text{NIL}$ 
     $v.p \leftarrow u.p$ 
  
```

# Cancellazione di $z$

## Tre casi a seconda del numero di figli

### Caso 1: $z$ non ha figli

- ▶ Cancella  $z$  aggiustando il puntatore del padre

### Caso 2: $z$ ha un figlio

- ▶ Cancella  $z$
- ▶ Il padre ( $z.p$ ) punta al figlio di  $z$  invece che a  $z$

$\text{TREE-DELETE}(T, z)$

```

1  if  $z.left = \text{NIL}$ 
2      TRAPIANTO( $T, z, z.right$ )
3  elseif  $z.right = \text{NIL}$ 
4      TRAPIANTO( $T, z, z.left$ )
5  ...

```

( $\text{TRAPIANTO}(T, u, v)$  sostituisce  $u$  con  $v$ )

## Tre casi a seconda del numero di figli

### Caso 1: $z$ non ha figli

### Caso 2: $z$ ha un figlio

### Caso 3: $z$ ha due figli

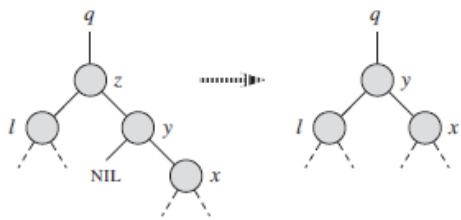
- ▶ Chi è  $y = \text{succ}(z)$ ?
  - ★ **Minimo** del sottoalbero dx
- ▶ Quanti figli ha  $y = \text{succ}(z)$ ?
  - ★ 0 o 1 (altrimenti non sarebbe il successore!)

$\Rightarrow$

- ▶ Cerco  $y = \min(z.right)$
- ▶ **Cancello**  $y$  (Caso 1 o 2)
- ▶ Sostituisco la chiave di  $z$  e dati satellite con quelli di  $y$

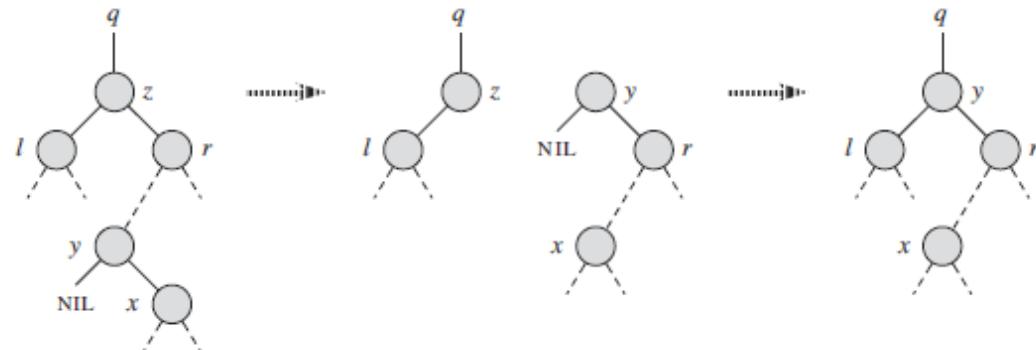
## Cancello $y$ : due sottocasi

A)  $y$  è il figlio di  $z$  (radice del sottoalbero)



B) Non lo è :

lo faccio diventare tale  
poi come A



## Cancellazione nodo $z$

TREE-DELETE( $T, z$ )

```

1  if  $z.left = \text{NIL}$ 
2      TRAPIANTO( $T, z, z.right$ )
3  elseif  $z.right = \text{NIL}$ 
4      TRAPIANTO( $T, z, z.left$ )
5  else  $y \leftarrow \text{TREE-MINIMUM}(z.right)$ 
6      if  $y.p \neq z$ 
7          TRAPIANTO( $T, y, y.right$ )
8           $y.right \leftarrow z.right$ 
9           $y.right.p \leftarrow y$ 
10     TRAPIANTO( $T, z, y$ )
11      $y.left \leftarrow z.left$ 
12      $y.left.p \leftarrow y$ 

```

## Sottocaso A

$y$  è il figlio di  $z$

TREE-DELETE( $T, z$ )

...

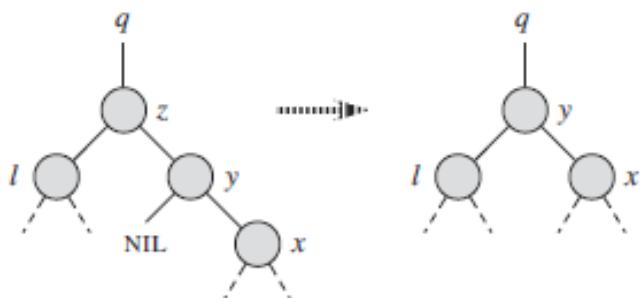
**if**  $y.p \neq z$

...

TRAPIANTO( $T, z, y$ )

$y.left \leftarrow z.left$

$y.left.p \leftarrow y$



## Sottocaso B

$y$  **non** è il figlio di  $z$  (radice del sottoalbero)

TREE-DELETE( $T, z$ )

...

**if**  $y.p \neq z$

TRAPIANTO( $T, y, y.right$ )

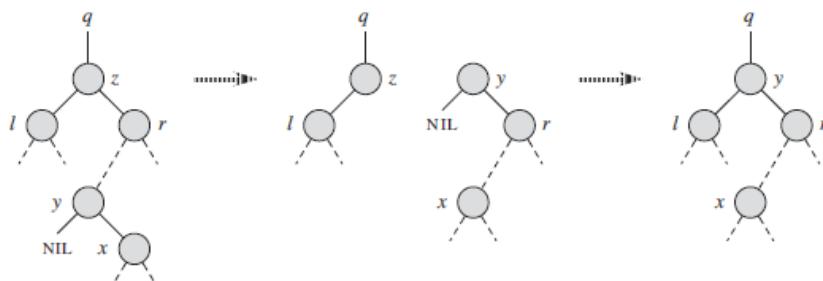
$y.right \leftarrow z.right$

$y.right.p \leftarrow y$

TRAPIANTO( $T, z, y$ )

$y.left \leftarrow z.left$

$y.left.p \leftarrow y$



## Alberi binari di ricerca **costruiti in modo casuale**

- Date  $n$  chiavi inserirle in ordine casuale in un ABR vuoto
- Ciascuna delle  $n!$  permutazioni è ugualmente probabile
- Non consideriamo cancellazioni
- **Altezza attesa:**  $O(\lg n)$
- Non lo dimostriamo!

### Esercizio 12.4.3

- Ogni ABR con  $n$  chiavi **non** è ugualmente probabile
- Provare con  $n = 3$ 
  - ▶ 6 possibili permutazioni dei dati
  - ▶ 5 ABR diversi
  - ▶ 4 alberi appaiono solo una volta
  - ▶ 1 albero appare due volte

## Ordinamento tramite ABR

### Esercizio 12.3-3

- Usare TREE-INSERT e INORDER-TREE-WALK per ordinare un insieme di numeri
- Posso ordinare  $n$  numeri:
  - ▶ costruendo prima un ABR inserendo questi numeri
  - ▶ stampando i numeri con un attraversamento simmetrico
- Quali sono i tempi di esecuzione nel caso peggiore e nel caso migliore?
- E' un ordinamento stabile? (vedi dopo)
- Come posso fare a randomizzare questo algoritmo?

# Alberi binari di ricerca con chiavi uguali

Le chiavi uguali sono un problema per l'implementazione di ABR

## Problema 12-1

- Quali sono le prestazioni asintotiche inserendo  $n$  chiavi identiche in un ABR inizialmente vuoto?
- E' possibile modificare TREE-INSERT verificando se  $z.key = x.key$  prima di  $z.key < x.key$  e se  $z.key = y.key$  prima di  $z.key < y.key$ . Quando valgono le uguaglianze:
  - ① mantenere una lista concatenata con chiavi uguali
  - ② mantenere un flag booleano  $b[x]$  nel nodo  $x$  e porre  $x$  a  $x.left$  o  $x.right$  a seconda del valore del flag (che si alterna ad ogni visita di  $x$  con valore uguale)
  - ③ Scegliere casualmente se assegnare  $x.left$  o  $x.right$  a  $x$
- Per ogni strategia trovare le prestazioni asintotiche inserendo  $n$  chiavi identiche in un ABR inizialmente vuoto
- Cosa comportano queste strategie per la ricerca dei valori?

# Inserimento

$\text{TREE-INSERT}(T, z)$

```

 $y \leftarrow \text{NIL}$  // Tiene traccia del padre
 $x \leftarrow T.\text{root}$ 
while  $x \neq \text{NIL}$ 
   $y \leftarrow x$ 
  if  $z.\text{key} < x.\text{key}$ 
     $x \leftarrow x.\text{left}$ 
  else  $x \leftarrow x.\text{right}$ 
 $z.p \leftarrow y$ 
if  $y = \text{NIL}$ 
   $T.\text{root} \leftarrow z$  // L'albero  $T$  era vuoto
elseif  $z.\text{key} < y.\text{key}$ 
   $y.\text{left} \leftarrow z$ 
else  $y.\text{right} \leftarrow z$ 

```

# Alberi rosso-neri

## Alberi rosso-neri

- ABR, ogni nodo  $x$  ha un attributo booleano  $x.color$
- $x.color$ : **rosso** o **nero**
- Eredita gli altri attributi di ABR
- **Foglie vuote (NIL) e nere**
  - ▶ Una sentinella,  $T.NIL$  per tutte le foglie di  $T$
  - ▶  $T.NIL.color$ : NERO
  - ▶ Padre della radice:  $T.NIL$
  - ▶ Non interessa key in  $T.NIL$

## Proprietà alberi rosso-neri

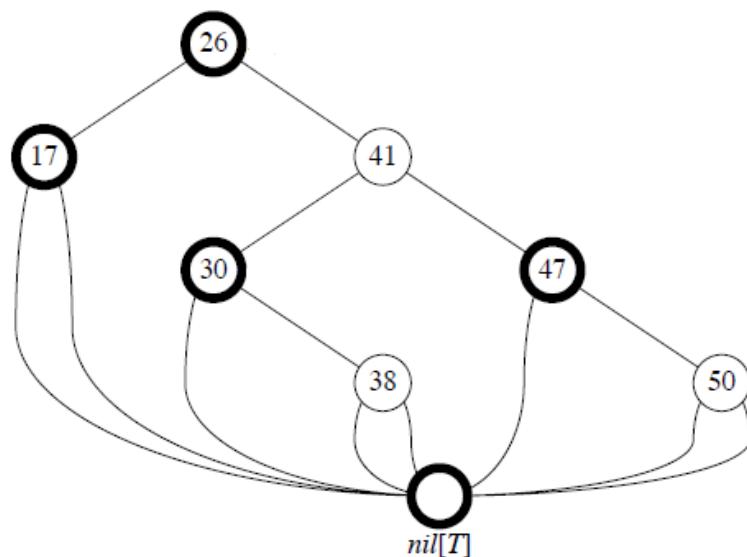
P1 Ogni nodo è **rosso** o **nero**

P2 La **radice** è **nera**

P3 Ogni **foglia** ( $T.\text{NIL}$ ) è **nera**

P4 Se un nodo è **rosso**, allora entrambi i suoi figli sono **neri**  
 (No due rossi consecutivi in un cammino semplice da radice a foglia)

P5 **Tutti** i cammini da ogni nodo alle foglie contengono lo **stesso numero** di nodi **neri**



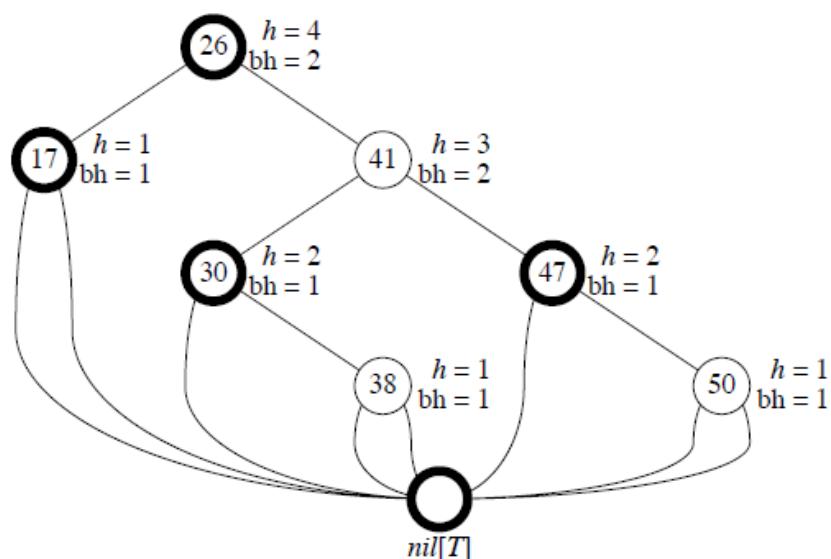
Nodi con bordo neretto indicano nodi neri

## Altezza di un albero RN, definizioni

- Dato il nodo  $x$ 
  - ▶ **Altezza** del nodo  $h(x)$ :  
numero di archi nel cammino più lungo fino ad una foglia
  - ▶ **Altezza-nera** del nodo  $bh(x)$ :  
numero di **nodi neri** (incluso  $T.\text{NIL}$ ) nel cammino da  $x$  alla foglia (escluso  $x$ )
  - ▶  $P5 \Rightarrow bh$  ben definita

## Proprietà 5

- Tutti i cammini dal nodo alle foglie contengono lo stesso numero di nodi neri



Ogni nodo con altezza  $h$  ha altezza-nero  $\geq h/2$

### Dimostrazione

- P4  $\Rightarrow$   
al **massimo**  $h/2$  nodi nel cammino dal nodo alla foglia sono rossi
- $\Rightarrow$  **almeno**  $h/2$  sono neri

### Proprietà 4

- Se un nodo è rosso entrambi i suoi figli sono neri

Sottoalbero con radice in nodo  $x$  contiene  
**almeno**  $2^{bh(x)} - 1$  nodi interni

**Dimostrazione** : induzione sull'altezza di  $x$   
 $h = h(x)$  e  $bh = bh(x)$

• **Passo base:**

$h(x) = 0 \Rightarrow x$  foglia  $\Rightarrow bh(x) = 0$   
 $0$  nodi interni ( $2^0 - 1 = 0$ )  $\Rightarrow$  OK

• **Passo induttivo:**

Ogni figlio di  $x$  ha  $h' = h - 1$  e

$$bh' = \begin{cases} bh & \text{se figlio rosso} \\ bh - 1 & \text{se figlio nero} \end{cases}$$

Ipotesi induttiva: ogni figlio ha almeno  $2^{bh-1} - 1$  nodi interni

$\Rightarrow$  sottoalbero con radice in  $x$  contiene almeno

$$2 \cdot (2^{bh-1} - 1) + 1 = 2^{bh} - 1 \text{ nodi interni}$$

## Lemma 13.1

L'altezza **massima** di un albero rosso-nero con  $n$  nodi interni è  $2 \lg(n+1)$

Da fatti precedenti:

$$n \geq 2^{bh} - 1 \geq 2^{h/2} - 1$$

$$n + 1 \geq 2^{h/2}$$

$$\lg(n+1) \geq h/2$$

$$\Rightarrow h \leq 2\lg(n+1)$$

## Operazioni su alberi rosso-neri

### Operazioni che non modificano ABR

- MIN
- MAX
- SUCCESSOR
- PREDECESSOR
- SEARCH

Si eseguono in tempo  $O(h)$

**Impiegano  $O(\lg n)$  con alberi rosso-neri**

# Inserimento e cancellazione non facili

## Inserimento

Come coloro il nuovo nodo?

- **rosso?** Può violare P4  
(neri entrambi i figli di un rosso)
- **nero?** Può violare P5  
(stesso numero di nodi in ogni cammino)

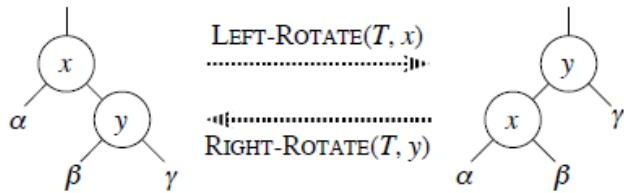
## Cancellazione

Se si **cancella** che colore aveva il nodo rimosso?

- **rosso?** OK:
  - ▶ non abbiamo cambiato altezza-nero (bh)
  - ▶ non abbiamo due nodi rossi a fila
  - ▶ non violiamo P2 (se il nodo rimosso è rosso, non è la radice)
- **nero?** Posso violare
  - ▶ P2 se il nodo rimosso era la radice e suo figlio (nuova radice) era rosso
  - ▶ P4 (due rossi a fila)
  - ▶ P5 (cammino nero)

## Rotazioni

- **Ristrutturazione** dell'albero usata per mantenere ARN **bilanciati**
- Cambio solo i puntatori e non cambio proprietà ABR
- Si ha **rotazione sinistra** e **destra**



- Mantiene ordinamento delle chiavi
- Dopo la rotazione l'albero è **più bilanciato**

## Tempo ?

- $O(1)$  LEFT-ROTATE e RIGHT-ROTATE:  
modifica un numero costante di puntatori

$\text{LEFT-ROTATE}(T, x)$

```

 $y \leftarrow x.\text{right}$  // Imposta  $y$ 
 $x.\text{right} \leftarrow y.\text{left}$  // Sposta sottoalbero sx di  $y$  in dx di  $x$ 
if  $y.\text{left} \neq T.\text{NIL}$ 
     $y.\text{left}.p \leftarrow x$ 
 $y.p \leftarrow x.p$  // Collega il padre di  $x$  a  $y$ 
if  $x.p = T.\text{NIL}$ 
     $T.\text{root} \leftarrow y$ 
elseif  $x = x.p.\text{left}$ 
     $x.p.\text{left} \leftarrow y$ 
else  $x.p.\text{right} \leftarrow y$ 
     $y.\text{left} \leftarrow x$  // Pone  $x$  a sx di  $y$ 
     $x.p \leftarrow y$ 

```

- Prima dell'esecuzione  $x.\text{right} \neq T.\text{NIL}$
- padre della radice è  $T.\text{NIL}$

**RB-INSERT( $T, z$ )**

```

1   $y \leftarrow T.\text{NIL}$ 
2   $x \leftarrow T.\text{root}$ 
3  while  $x \neq T.\text{NIL}$ 
4       $y \leftarrow x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x \leftarrow x.\text{left}$ 
7      else  $x \leftarrow x.\text{right}$ 
8   $z.p \leftarrow y$ 
9  if  $y = T.\text{NIL}$ 
10      $T.\text{root} \leftarrow z$ 
11  elseif  $z.\text{key} < y.\text{key}$ 
12       $y.\text{left} \leftarrow z$ 
13  else  $y.\text{right} \leftarrow z$ 
14   $z.\text{left} \leftarrow T.\text{NIL}$ 
15   $z.\text{right} \leftarrow T.\text{NIL}$ 
16   $z.\text{color} \leftarrow \text{RED}$ 
17  RB-INSERT-FIXUP( $T, z$ )

```

**Inserimento**

- Al termine di RB-INSERT  $z$  è rosso
- Si potrebbe violare una proprietà RB:
- Quale?
  - P1 OK
  - P2 Violazione se  $z$  è la radice
  - P3 OK
  - P4 Violazione se  $z.p$  è rosso (sia  $z$  che  $z.p$  rossi)
  - P5 OK
- Rimuove la violazione con RB-INSERT-FIXUP

## RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color = RED$ 
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16   $T.root.color \leftarrow BLACK$ 
```

### Invariante di ciclo

All'inizio di ogni iterazione

- a)  $z$  è rosso
- b) Al più una violazione rosso-nero:
  - ▶ P2:  $z$  è una radice rossa
  - 
  - ▶ P4:  $z$  e  $z.p$  entrambi rossi
- c) Se  $z.p$  è la radice allora  $z.p$  è nero (non ne parliamo)

### Inizializzazione

- a)  $z$  settato a rosso
- b) se  $z.p$  era rosso violo P4, altrimenti no
- Il resto dell'albero immutato, ho aggiunto un rosso  $\Rightarrow$  altezze-nere OK

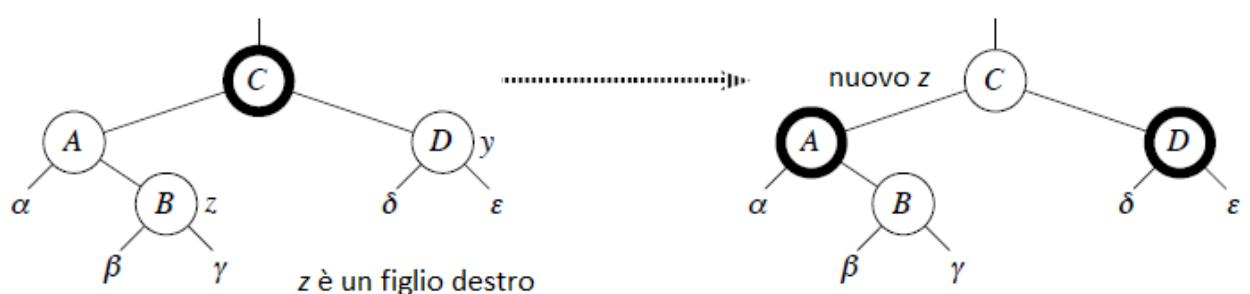
## Terminazione

- Ciclo termina se  $z.p$  è nero  $\Rightarrow$  P4 OK
- P2 potrebbe essere violata, sistemata con ultima linea

## Conservazione

- Termina sicuramente quando  $z$  è la radice ( $z.p$  è  $T.\text{NIL}$  nero)
- All'inizio del while **unica violazione: P4**  
(entrambi i figli di un nodo rosso sono neri)
- RB-INSERT-FIXUP sistema P4
  - ▶ 6 casi
  - ▶ 3 casi ( $z.p$  è un figlio sx) simmetrici con gli altri ( $z.p$  è un figlio dx)
  - ▶  $y$  è lo *zio* di  $z$  (fratello di  $z.p$ )

## Caso 1: $y$ (zio di $z$ ) rosso



- $z.p.p$  è nero,
- Pongo  $z.p \leftarrow BLACK$  e  $y \leftarrow BLACK \Rightarrow$  P4 OK,
- Possibili problemi con p5  $\Rightarrow$  pongo  $z.p.p \leftarrow RED$
- Ora P5 OK (tutti i cammini hanno lo stesso numero di neri)
- Vado in  $z.p.p$

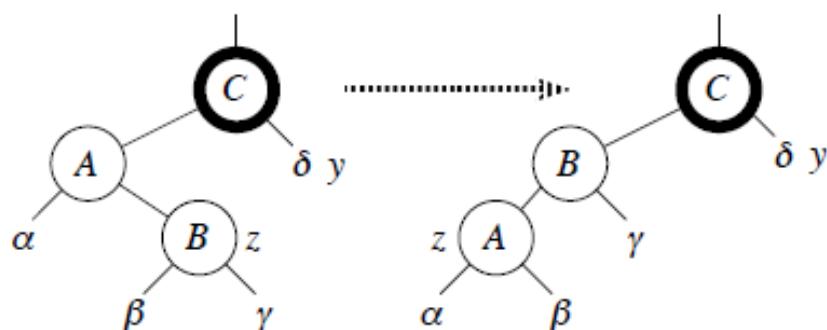
RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color = RED$ 
2      if  $z.p = z.p.p.left$ 
3           $y \leftarrow z.p.p.right$  // zio
4          if  $y.color = RED$ 
5               $z.p.color \leftarrow BLACK$ 
6               $y.color \leftarrow BLACK$ 
7               $z.p.p.color \leftarrow RED$ 
8               $z \leftarrow z.p.p$ 
9      else
10
11
12
13
14
15      else (come 3-14 scambiando "right" e "left")
16       $T.root.color \leftarrow BLACK$ 

```

Caso 2:  $y$  nero,  $z$  figlio dx



- $z.p.p$  è ancora **nero**
- Eseguo **ruotazione sinistra** intorno a  $z.p \Rightarrow$   
 $z$  è un figlio sx, caso 3

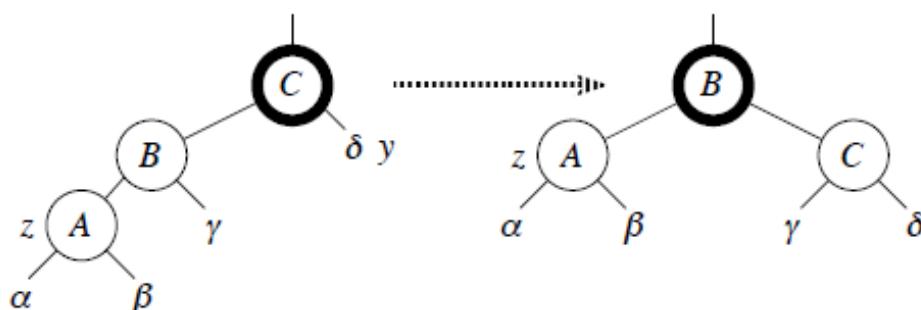
RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color = RED$ 
2      if  $z.p = z.p.p.left$ 
3           $y \leftarrow z.p.p.right$ 
4          if  $y.color = RED$ 
5               $z.p.color \leftarrow BLACK$ 
6               $y.color \leftarrow BLACK$ 
7               $z.p.p.color \leftarrow RED$ 
8               $z \leftarrow z.p.p$ 
9          else      if  $z = z.p.right$ 
10              $z \leftarrow z.p$ 
11             LEFT-ROTATE( $T, z$ )
12
13
14
15     else (come 3-14 scambiando "right" e "left")
16      $T.root.color \leftarrow BLACK$ 

```

Caso 3:  $y$  nero,  $z$  figlio sx



- Pongo  $z.p \leftarrow$  nero;  $z.p.p \leftarrow$  rosso
- Ruoto a **destra** su  $z.p.p$
- $\Rightarrow$  OK P4
- $z.p$  è nero  $\Rightarrow$  STOP

## RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color = RED$ 
2      if  $z.p = z.p.p.left$ 
3           $y \leftarrow z.p.p.right$ 
4          if  $y.color = RED$ 
5               $z.p.color \leftarrow BLACK$ 
6               $y.color \leftarrow BLACK$ 
7               $z.p.p.color \leftarrow RED$ 
8               $z \leftarrow z.p.p$ 
9          else      if  $z = z.p.right$ 
10              $z \leftarrow z.p$ 
11             LEFT-ROTATE( $T, z$ )
12              $z.p.color \leftarrow BLACK$ 
13              $z.p.p.color \leftarrow RED$ 
14             RIGHT-ROTATE( $T, z.p.p$ )
15     else (come 3-14 scambiando "right" e "left")
16      $T.root.color \leftarrow BLACK$ 

```

## Tempo?

- RB-INSERT:  $O(\lg n)$  fino a RB-INSERT-FIXUP
- In RB-INSERT-FIXUP:
  - ▶ Ogni iterazione:  $O(1)$
  - ▶ Ogni iterazione o è l'ultima o sale di 2 livelli
  - ▶ Si hanno  $O(\lg n)$  livelli  $\Rightarrow O(\lg n)$
  - ▶ Al massimo 2 rotazioni in totale
- Inserimento in albero RN:  $O(\lg n)$

## Cancellazione

Non la trattiamo

# Strutture dati aumentate

## Progettare algoritmi

- Raramente progetto una struttura dati da zero
- Spesso prendo una struttura dati nota e **aggiungo** informazioni
- La struttura dati può permettere **nuove operazioni**
- Gestire correttamente la nuova informazione **senza perdita di efficienza**

## Statistiche d'ordine dinamiche

Albero RN, più :

- OS-SELECT( $x, i$ ): ritorna il puntatore al nodo che contiene l' **$i$ -ma chiave più piccola** del sotto-albero con radice in  $x$
- OS-RANK( $T, x$ ): ritorna la posizione (**rango**) di  $x$  nell'ordine lineare determinato da un attraversamento **inorder** di  $T$  (indicato con  $rank(x, T)$ )

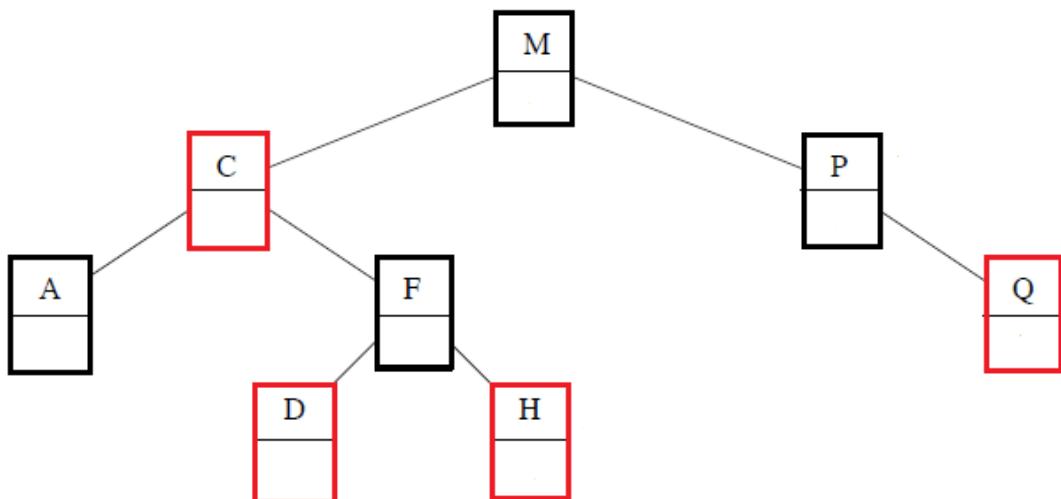
## Implementazione semplice?

- Eseguo un attraversamento in-order e conto quanti nodi trovo
  - ▶ Per  $\text{OS-SELECT}(x, i)$  mi fermo dopo  $i$  nodi
  - ▶ Per  $\text{OS-RANK}(T, x)$  mi fermo quando trovo  $x$
- Costo?  $O(n)$

## Aumento RN

- Aggiungo  $x.size = \# \text{ nodi}$  nel sotto-albero con radice in  $x$ 
  - ▶ Incluso  $x$
  - ▶ Foglie (sentinelle) escluse
- Def  $T.\text{NIL}.size = 0 \Rightarrow$   
 $x.size = x.left.size + x.right.size + 1$
- Struttura dati **aumentata**:  
 $< x.key, x.p, x.left, x.right, x.color, x.size >$

## Esempio



## Chiavi duplicate

- Rango non definito per chiavi uguali
- Lo definisco rispetto ad attraversamento inorder  
 (Se cambio  $C = D \Rightarrow \text{rank}(C, T) = 2$  e  $\text{rank}(D, T) = 3$ )

## OS-SELECT

- Ritorna il puntatore al nodo con l' $i$ -ma chiave più piccola del sotto-albero con radice in  $x$
- Chiamata iniziale:  $\text{OS-SELECT}(T.\text{root}, i)$

$\text{OS-SELECT}(x, i)$

```

1   $r \leftarrow x.\text{left.size} + 1$ 
2  if  $i = r$ 
3      return  $x$ 
4  elseif  $i < r$ 
5      return  $\text{OS-SELECT}(x.\text{left}, i)$ 
6  else return  $\text{OS-SELECT}(x.\text{right}, i - r)$ 

```

Esempio:  $\text{OS-SELECT}(T.\text{root}, 5)$

## Correttezza

- Ricorsivo in coda  
 $\Rightarrow$  posso vederlo come iterativo  
 $\Rightarrow$  invariante di ciclo
- Prima di ogni iterazione l' $i$ -mo valore si trova nel sottoalbero con radice in  $x$
- $r = \text{rango di } x \text{ nel sottoalbero con radice } x$ 
  - ▶ Se  $i = r \Rightarrow$  OK ritorna  $x$
  - ▶ Se  $i < r \Rightarrow$   $i$ -mo elemento più piccolo è nel sottoalbero sx
  - ▶ Se  $i > r \Rightarrow$   $i$ -mo elemento più piccolo è nel sottoalbero dx  
tolgo gli  $r$  elementi nel sottoalbero di  $x$   
che **precedono** quelli nel sottoalbero destro di  $x$

## Costo

- Ogni chiamata ricorsiva scende di un livello
- L'albero RN ha  $O(\lg n)$  livelli  $\Rightarrow O(\lg n)$  chiamate
- **Tempo**  $O(\lg n)$

## OS-RANK( $T, x$ )

- Ritorna il rango di  $x$  nell'albero  $T$

### OS-RANK( $T, x$ )

```

1   $r \leftarrow x.left.size + 1$ 
2   $y \leftarrow x$ 
3  while  $y \neq T.root$ 
4      if  $y = y.p.right$ 
5           $r \leftarrow r + y.p.left.size + 1$ 
6       $y \leftarrow y.p$ 
7  return  $r$ 

```

## Correttezza

### Invariante di ciclo

- All'inizio di ogni iterazione del ciclo  
 $r = rank(x.key, y)$

### Inizializzazione

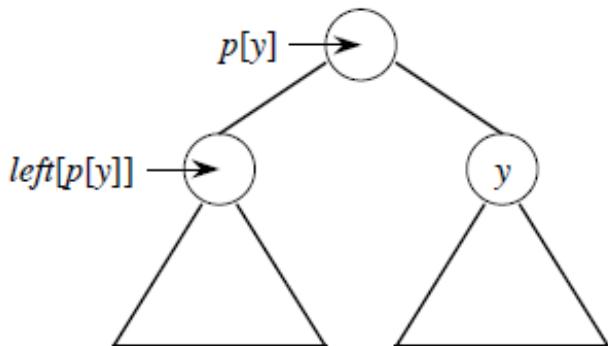
- $r = rank(x.key, x)$  e  $y = x$

### Terminazione

- Ciclo termina quando  $y = T.root$
- Sottoalbero con radice in  $y$  è l'intero albero
- $r = rank(x.key, T)$   
**(proprietà utile)**

## Conservazione

- se  $r = \text{rank}(x.\text{key}, y)$  all'inizio dell'iterazione,  
allora  $r = \text{rank}(x.\text{key}, y.p)$  alla fine dell'iterazione:
  - ▶ se  $y$  è un **figlio sx**, sottoalbero del fratello ha nodi che seguono  $x$   
 $\Rightarrow r$  non cambia
  - ▶ se  $y$  è un **figlio dx**, tutti i nodi nel sottoalbero del fratello  
 precedono tutti i nodi nel sottoalbero di  $y$   
 $\Rightarrow r \leftarrow r + y.p.\text{left.size} + 1$



## Analisi

- $y$  sale di un livello ad ogni iterazione
- $O(\lg n)$

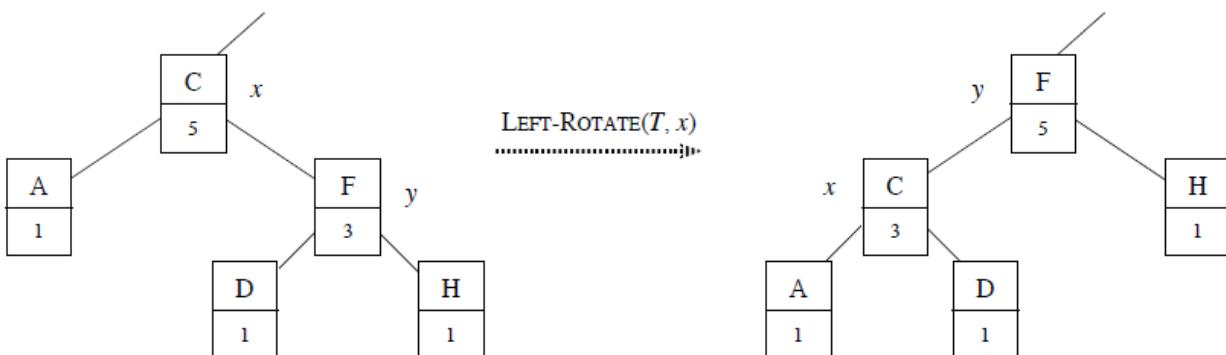
## Gestione attributo *size*

- Gestire  $x.\text{size}$  in **inserimento** e **cancellazione**
- Gestione **efficiente** per non doverli ricalcolare tutti ( $\Omega(n)$ )
  - ▶ Come farei in  $\Omega(n)$ ?
  - ▶ Devo risalire sull'albero e aggiornare tutti i size.  
 Come?
  - ▶ Attraversamento differito
  - ▶ Scrivere algoritmo
- Ci riusciamo senza aumentare costo inserimento e cancellazione  
 $(O(\lg n))$ ?

# Inserimento

## Visita per inserimento

- Il nuovo nodo sarà un **descendente** di ciascun nodo visitato (solo di questi nodi)
- Incrementa *size* di ogni nodo **visitato**
- In **fixup**:
  - Sale nell'albero
  - Cambia i colori  $O(\lg n)$  volte
  - Esegue  $\leq 2$  rotazioni
- Cambiamento colore non influenza le dimensioni dei sottoalberi
- Rotazioni si!**
  - Nuove dimensioni sulla base delle vecchie e delle dimensioni dei figli



## Rotazioni

- Aggiungo in fondo a LEFT-ROTATE:
  - $y.size \leftarrow x.size$  (sottoalbero totale **non cambia**)
  - $x.size \leftarrow x.left.size + x.right.size + 1$
  - Altri nodi **non cambiano**
- Analogamente per rotazione **destra**
- Aggiorno  $x.size$  in  $O(1)$  per rotazione
- Aggiornamento campi durante fixup:  $O(1)$
- $\Rightarrow O(\lg n)$  per **inserire**

## Cancellazione

- In 2 passi (no dettagli):
  - ① Divide un nodo  $y$
  - ② Fixup
    - ★ Percorso  $y \rightarrow \text{root}$ , decrementa  $\text{size}$  in ogni nodo trovato ( $O(\lg n)$ )
    - ★ Solo cambiamenti di colore e ( $\leq 3$ ) rotazioni
    - ★  $O(1)$  per aggiornamento campi  $\text{size}$
- $O(\lg n)$  per cancellare

## Come aumentare una struttura dati

- ① Scegliere una **struttura dati sottostante**
- ② Determinare **informazioni aggiuntive** da gestire
- ③ **Verificare** che si possano gestire le informazioni aggiuntive per le **operazioni esistenti** sulla struttura dati con analoghi tempi di esecuzione
- ④ Sviluppare **nuove operazioni**

Non devo eseguire i passi in quest'ordine preciso!

## Per statistiche d'ordine?

- ① Scegliere una **struttura dati sottostante**  
⇒ Alberi RN
- ② Determinare **informazioni aggiuntive** da gestire  
⇒  $x.size$
- ③ **Verificare** che si possono gestire le informazioni aggiuntive per le **operazioni esistenti** sulla struttura dati  
⇒ Mantenere *size* durante inserimenti e cancellazioni
- ④ Sviluppare **nuove operazioni**  
⇒ OS-SELECT e OS-RANK

Alberi RN sono particolarmente adatti ad essere aumentati

## Teorema

Se si **aumenta** un albero RN con un **campo**  $f$ , dove  $x.f$  dipende solo da informazioni in

$x$ ,  $x.left$ , e  $x.right$  (includendo  $x.left.f$  e  $x.right.f$ )

**Allora** si possono mantenere i valori di  $f$  in tutti i nodi in  $O(\lg n)$

## Dimostrazione

*Idea:*

La modifica di  $x.f$  (dipende solo da  $x$  e dai suoi figli) si propaga solo agli antenati di  $x$  ( $x.p, x.p.p, \dots, root$ ).

⇒  $O(\lg n)$  aggiornamenti a costo  $O(1)$  ciascuno

## Inserimento

- Non sempre posso aggiornare  $f$  scendendo verso il basso... posso farlo risalendo fino alla radice
- **Fixup:** solo cambiamenti di **colore** (non modificano  $f$ ) e **rotazioni**
  - ▶ **Rotazione** influenza  $f$  di  $\leq 3$  nodi ( $x, y, \text{padre}$ )  
Ricalcolo ciascuna  $f$  in  $O(1)$
  - ▶ Dopo, se necessario, propago i cambiamenti di  $f$  a salire sull'albero ( $O(\lg n)$ )
  - ▶ Totale  $O(\lg n)$  per rotazione
  - ▶ In ogni fixup al massimo 2 rotazioni
- $O(\lg n)$  per aggiornare  $f$  durante il fixup

## Cancellazione

- Analogo: divido un nodo e risalgo per aggiornare  $f$
- Fixup ha  $\leq 3$  rotazioni
- $O(\lg n)$  per rotazione  $\Rightarrow O(\lg n)$  per aggiornare  $f$  durante il fixup  
**(teorema)**

## Alberi di intervalli

Gestiscono un insieme di intervalli, per esempio intervalli temporali

## Operazioni

- INTERVAL-INSERT( $T, x$ )
  - ▶  $x.int$  contiene un intervallo
- INTERVAL-DELETE( $T, x$ )
- INTERVAL-SEARCH( $T, i$ )
  - ▶ ritorna il puntatore ad un nodo  $x$  in  $T$  t.c.  $x.int$  interseca l'intervalle  $i$
  - ▶ **Ogni** intervallo in  $T$  che si sovrappone è OK
  - ▶ Ritorna puntatore a  $T.NIL$  se non c'è nodo che si sovrappone a  $i$  in  $T$

L'intervalle  $i$  ha **attributi**  $i.low$  e  $i.high$

## Sovrapposizione di intervalli

- Gli intervalli  $i$  e  $j$  **si sovrappongono** sse:

$$i.\text{low} \leq j.\text{high}$$

AND

$$j.\text{low} \leq i.\text{high}.$$

- **Alternativamente**

$i$  e  $j$  **non si sovrappongono** sse:

$$i.\text{low} > j.\text{high}$$

OR

$$j.\text{low} > i.\text{high}$$

## Aumentare strutture dati

- Approccio diviso in 4 parti

- ① Scegliere una **struttura dati sottostante**
- ② Determinare **informazioni aggiuntive** da gestire
- ③ **Verificare** che si possono gestire le informazioni aggiuntive per le **operazioni esistenti** sulla struttura dati
- ④ Sviluppare **nuove operazioni**

### 1) Struttura dati di base

- Alberi RN
- Ogni nodo  $x$  contiene l'intervallo  $x.\text{int}$
- **Chiave**  $x.\text{key}$ : **estremo inferiore** dell'intervallo ( $x.\text{int}.low$ )
- Attraversamento *inorder* elenca ordinatamente gli intervalli in funzione dell'estremo inferiore

## 2) Informazioni aggiuntive

- Ogni nodo  $x$  contiene

$x.\max$  = **massimo** tra tutti gli estremi dx degli intervalli memorizzati nel sottoalbero con radice  $x$

$$x.\max = \max \begin{cases} x.\text{int}.high \\ x.\text{left}.\max \\ x.\text{right}.\max \end{cases}$$

- Può essere  $x.\text{left}.\max > x.\text{right}.\max$  ?

Certo!

La posizione nell'albero è determinata solo dall'estremo **inferiore non da quello superiore**

## 3) Gestione delle informazioni

- Facile

$x.\max$  dipende solo da:

- ▶ informazione in  $x$ :  $x.\text{int}.high$
- ▶ informazione in  $x.\text{left}$ :  $x.\text{left}.\max$
- ▶ informazione in  $x.\text{right}$ :  $x.\text{right}.\max$

- Applico il teorema

- Potrei aggiornare  $\max$  scendendo durante l'inserimento in tempo  $O(1)$  per ogni rotazione

## 4) Sviluppare nuove operazioni

INTERVAL-SEARCH( $T, i$ )

```

1  $x \leftarrow T.root$ 
2 while  $x \neq T.NIL$  e  $i$  non si sovrappone a  $x.int$ 
3   if  $x.left \neq T.NIL$  e  $x.left.max \geq i.low$ 
4      $x \leftarrow x.left$ 
5   else  $x \leftarrow x.right$ 
6 return  $x$ 
```

### Tempo

$O(\lg n)$

## Correttezza

Idea: devo controllare solo 1 dei 2 figli del nodo

### Teorema

La procedura INTERVAL-SEARCH( $T, i$ ) restituisce un nodo il cui intervallo si sovrappone a  $i$

OPPURE

restituisce  $T.NIL$  se in  $T$  non ci sono nodi il cui intervallo si sovrappone a  $i$

### In altri termini...

- Se la ricerca va a **destra**, allora:
  - ▶ C'è una sovrapposizione nel sottoalbero dx, OPPURE
  - ▶ Non c'è sorapposizione in nessuno dei due sottoalberi
- Se la ricerca va a **sinistra**, allora:
  - ▶ C'è una sovrapposizione nel sottoalbero sx, OPPURE
  - ▶ Non c'è sorapposizione in nessuno dei due sottoalberi

# Dimostrazione

## Se la ricerca va a destra

- Se c'è una sovrapposizione nel sottoalbero dx, OK
- Se non c'è sovrapposizione a dx, mostro che non c'è sovrapposizione a sx (non sbaglio strada)
- Siamo andati a destra perché
  - ▶  $x.left = T.NIL \Rightarrow$  no sovrapposizione a sx
  - OPPURE
  - ▶  $x.left.max < i.low \Rightarrow$  no sovrapposizione a sx

## Se la ricerca va a sinistra

- Se c'è una sovrapposizione nel sottoalbero sx, OK!
- Se non c'è sovrapposizione a sx, mostro che non c'è sovrapposizione a dx

## Vado a sinistra, e non c'è sovrapposizione a sx ( $i.low \leq x.left.max$ )

- Mostro che non c'è sovrapposizione a destra
- Siamo andati a sinistra perché :
  $i.low \leq x.left.max = j.high$  per qualche  $j$  nel sottoalbero sx
- Non c'è sovrapposizione a sinistra ( $i$  e  $j$  non si sovrappongono)
  - ▶ Se  $i.low > j.high$  oppure  $j.low > i.high$
- Siccome  $i.low \leq j.high$ , dobbiamo avere  $j.low > i.high$
- Sia  $k$  un **qualsiasi** intervallo nel sottoalbero dx
- Gli **estremi inferiori** sono le **chiavi**  $\Rightarrow$ 

$$\underbrace{j.low}_{sx} \leq \underbrace{k.low}_{dx}$$
- $\Rightarrow i.high < j.low \leq k.low \Rightarrow i.high < k.low$
- $\Rightarrow i$  e  $k$  non si sovrappongono

**(teorema)**

# Programmazione dinamica

## Programmazione dinamica

- **Divide-et-impera**

- ▶ scomponere il problema in sottoproblemi **indipendenti**
- ▶ risolvere i sottoproblemi
- ▶ li ricombina

- **Programmazione dinamica**

- ▶ sottoproblemi **non indipendenti**
  - ★ condividono dei sottoproblemi
- ▶ quando risolvo un **sottoproblema comune**,  
salvo la soluzione in una **tavella**, per riutilizzarla dopo
- ▶ “programmazione” si riferisce all’uso di una **tecnica tabulare**

- Usata spesso per problemi di **ottimizzazione**

- ▶ problemi di **massimizzazione, minimizzazione**
- ▶ un problema potrebbe avere più soluzioni ottime
  - ★ voglio trovare **una** soluzione con **il** valore ottimo

## Bellman - Eye of the Hurricane: An Autobiography (1984)

I spent the Fall quarter (of 1950) at RAND. ... Where did the name, dynamic programming, come from?

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word **research**.

I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term **mathematical**.

The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation.

What title, what name, could I choose?

...

## Bellman - Eye of the Hurricane: An Autobiography (1984)

...

In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons.

I decided therefore to use the word **programming**. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely **dynamic**, in the classical physical sense.

It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible.

Thus, I thought **dynamic programming** was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

## Passi principali

Per sviluppare un algoritmo di programmazione dinamica

- ① Caratterizzare la struttura di una soluzione ottima
- ② Definire ricorsivamente il valore di una soluzione ottima del problema
- ③ Calcolare una soluzione ottima in modo bottom-up
  - ▶ da sottoproblemi semplici a quelli più difficili, fino al problema originario
- ④ Costruire una soluzione ottima del problema richiesto

Problema: taglio delle aste

## Problema: taglio delle aste

- Prezzo di un'asta di acciaio dipende dalla sua lunghezza
- Un'asta lunga  $n$  può essere tagliata in pezzi più corti
- Trovare il taglio **ottimo** che **massimizza** il ricavo della vendita
  - ▶ potrei anche venderla intera

## Tabella dei prezzi

lunghezza $i$	1	2	3	4	5	6	7	8	9	10
prezzo $p_i$	1	5	8	9	10	17	17	20	24	30

lunghezza i	1	2	3	4
prezzo p <sub>i</sub>	1	5	8	9

## Possibili tagli ( $n = 4$ )

- 8 possibili modi di taglio

[4] (9)

[1, 3] (9)

[1, 1, 2] (7)

[1, 1, 1, 1] (4)

[2, 2] (10)

[1, 2, 1] (7)

[3, 1] (9)

[2, 1, 1] (7)

- $2^{n-1}$  modi per tagliare un'asta lunga  $n$

▶  $n - 1$  punti di taglio

▶  $2^{n-1}$  combinazioni di taglio - non taglio

▶ come un numero binario con  $n - 1$  cifre

- Taglio ottimo: [2, 2]

- Ricavo massimo ( $r_n$ ):  $r_4 = 10$

## Sottostruttura ottima

Esprimo  $r_n$  in funzione di  $r_i$  ( $i < n$ )

- ∀ taglio  $i$  ho:  $r_n = r_i + r_{n-i}$

▶ se non taglio:  $r_n = p_n$  (prezzo asta intera)

▶ ⇒

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

- **Ottimo** è la somma dei ricavi **ottimi** delle 2 **semiaste**

- Ottimo **incorpora** i 2 ottimi delle soluzioni dei 2 sottoproblemi

- Dim (per assurdo):

▶ Supponiamo che  $r_i$  (o  $r_{n-i}$ ) **non** sia l'ottimo del sottoproblema  
 ▶ sostituisco a  $r_i$  una soluzione ottima  $r'_i > r_i$  ho  $r'_n > r_n$   
 ▶ ⇒  $r_n$  non sarebbe un ottimo

- Un problema ha una **sottostruttura ottima** se  
 la sua soluzione incorpora le soluzioni ottime dei suoi sottoproblemi  
 (che posso risolvere indipendentemente)

## Altra formulazione

$r_n$  dipende dall'ottimo di **un solo** sottoproblema:

- $r_n = \text{prezzo del primo pezzo} + \text{taglio ottimo della restante asta}:$   

$$r_n = p_i + r_{n-i}$$
- vale anche se non taglio l'asta ( $i = n$ ):  

$$r_n = p_n + r_0 \text{ con } r_0 = 0$$
- Quindi:  $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$

## Passi principali

Per sviluppare un algoritmo di programmazione dinamica

- ① Caratterizzare la struttura di una soluzione ottima
- ② **Definire ricorsivamente il valore di una soluzione ottima del problema**
- ③ Calcolare una soluzione ottima in modo bottom-up
  - ▶ dai sottoproblemi più semplici a quelli più difficili, fino al problema originario
- ④ Costruire una soluzione ottima del problema richiesto

## Algoritmo ricorsivo

- Ottenuto da:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

- dove

$p$ : tabella prezzi

$n$ : lunghezza asta

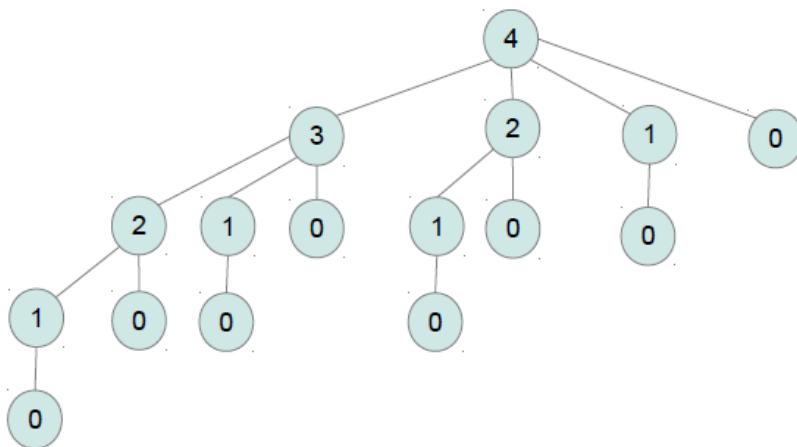
CUT-ROD( $p, n$ )

```

if  $n = 0$ 
    return 0
for  $i \leftarrow 1$  to  $n$ 
     $q \leftarrow -\infty$ 
     $q \leftarrow \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
return  $q$ 
```

## Costo

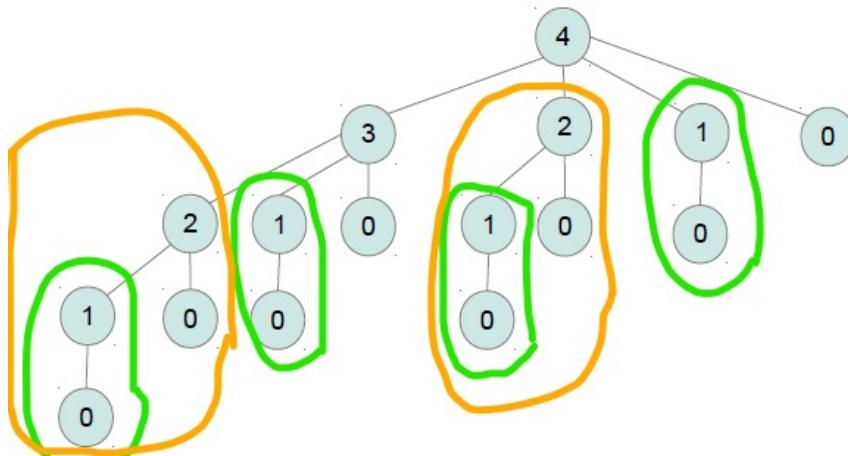
- $T(n) = \text{costante} + \sum_{j=0}^{n-1} T(j)$
- Si può mostrare che  $T(n) = \Theta(2^n)$
- In pratica considero tutti i possibili modi di taglio dell'asta ( $2^{n-1}$ )
- Lo stesso problema è risolto più e più volte!!



Albero di ricorsione (nel nodo indica  $n$ )

## Costo

- $T(n) = \text{costante} + \sum_{j=0}^{n-1} T(j)$
- Si può mostrare che  $T(n) = \Theta(2^n)$
- In pratica considero tutti i possibili modi di taglio dell'asta ( $2^{n-1}$ )
- Lo stesso problema è risolto più e più volte!!



Albero di ricorsione (nel nodo indico  $n$ )

## Algoritmo di programmazione dinamica

- Usando memoria aggiuntiva miglioro il tempo di esecuzione
  - ▶ diventa polinomiale!!
  - ▶ aumento la complessità spaziale riducendo quella temporale
- Idea:
  - ▶ memorizzo il risultato dei sottoproblemi appena calcolati
  - ▶ se trovo di nuovo un sottoproblema considero il risultato memorizzato
- Quindi:
  - ▶ risolvo ogni sottoproblema **una volta sola**
  - ▶ il costo diventa **polinomiale** se:
    - ★ il numero di problemi *distinti* da risolvere è polinomiale e
    - ★ la risoluzione dei singoli problemi richiede tempo polinomiale
- Due tecniche per implementare la programmazione dinamica:
  - ▶ top-down
  - ▶ bottom-up

## Metodo top-down

- risolvo il **problema di dimensione**  $n$ , e ricorsivamente risolvo **sottoproblemi** più piccoli
- memorizzo in una tabella i risultati già calcolati
- **prima** di lanciare la **ricorsione** sul problema più piccolo, **controllo** nella tabella se non ho già calcolato la soluzione

## Metodo bottom-up

- parto dai problemi più piccoli e li risolvo in ordine crescente di dimensione
- quando arrivo al problema di dimensione  $i$ , ho già risolto tutti i problemi di dimensioni  $< i$

## Stesso tempo di esecuzione asintotico

- **Top-down** a volte non esamina tutti i possibili sottoproblemi
- **Bottom-up** ha costi inferiori per le chiamate ricorsive (le costanti)

## Memoization (annotazione)

MEMOIZED-CUT-ROD( $p, n$ )

```
Sia  $r[0 .. n]$  un nuovo array
for  $i \leftarrow 0$  to  $n$ 
     $r[i] \leftarrow -\infty$ 
return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX( $p, n, r$ )

```
if  $r[n] \geq 0$ 
    return  $r[n]$ 
if  $n = 0$ 
     $q \leftarrow 0$ 
else  $q \leftarrow -\infty$ 
    for  $i \leftarrow 1$  to  $n$ 
         $q \leftarrow \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
     $r[n] \leftarrow q$ 
return  $q$ 
```

## Algoritmo ricorsivo (visto prima)

- Differente rispetto a programmazione dinamica!

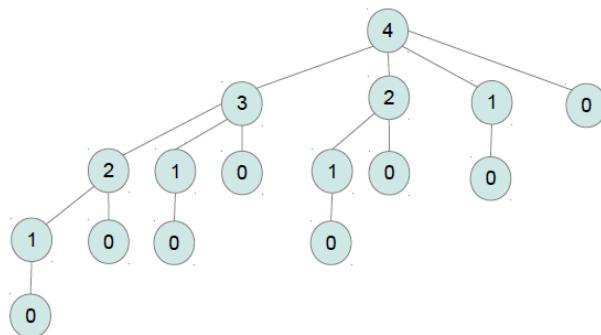
$\text{CUT-ROD}(p, n)$

```

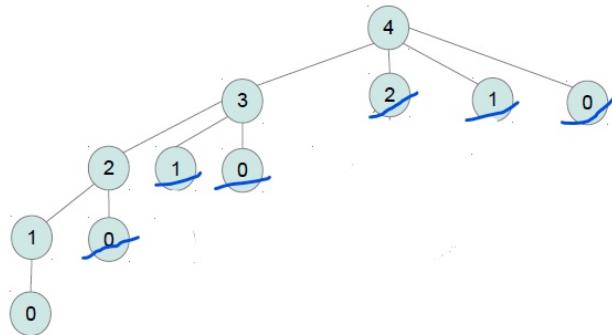
if  $n = 0$ 
    return 0
for  $i \leftarrow 1$  to  $n$ 
     $q \leftarrow \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
return  $q$ 

```

- Algoritmo ricorsivo



- Memoization



- $2^n >> n^2$  quando  $n$  cresce, ma  $2^4 = 4^2 \dots$

## Passi principali

Per sviluppare un algoritmo di programmazione dinamica

- ① Caratterizzare la struttura di una soluzione ottima
- ② Definire ricorsivamente il valore di una soluzione ottima del problema
- ③ **Calcolare una soluzione ottima in modo bottom-up**
  - ▶ dai sottoproblemi più semplici a quelli più difficili, fino al problema originario
- ④ Costruire una soluzione ottima del problema richiesto

## Calcolo bottom-up di soluzione ottima

### Metodo **bottom-up**

- parto dai problemi più piccoli e li risolvo in ordine crescente di dimensione
- quando arrivo al problema di dimensione  $i$ , ho già risolto tutti i problemi di dimensioni  $< i$

BOTTOM-UP-CUT-ROD( $p, n$ )

```

Sia  $r[0..n]$  un nuovo array
 $r[0] \leftarrow 0$ 
for  $j \leftarrow 1$  to  $n$ 
     $q \leftarrow -\infty$ 
    for  $i \leftarrow 1$  to  $j$ 
         $q \leftarrow \max(q, p[i] + r[j-i])$ 
     $r[j] \leftarrow q$ 
return  $r[n]$ 
```

## Complessità

- MEMOIZED-CUT-ROD:  $T(n) = \Theta(n^2)$ 
  - ▶ ogni sottoproblema è risolto una volta sola nel ciclo  $n$  iterazioni per risolvere un problema di dimensione  $n$
  - ▶ in tutto  $n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n+1)}{2}$  iterazioni
- BOTTOM-UP-CUT-ROD:  $T(n) = \Theta(n^2)$ 
  - ▶ 2 cicli annidati
  - ▶ ancora  $n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n+1)}{2}$  iterazioni
- Top-down e bottom-up hanno lo stesso tempo di esecuzione asintotico

## Passi principali

Per sviluppare un algoritmo di programmazione dinamica

- ① Caratterizzare la struttura di una soluzione ottima
- ② Definire ricorsivamente il valore di una soluzione ottima del problema
- ③ Calcolare una soluzione ottima in modo bottom-up
  - ▶ dai sottoproblemi più semplici a quelli più difficili, fino al problema originario
- ④ **Costruire una soluzione ottima del problema richiesto**

## Ricostruire una soluzione

- Algoritmi CUT-ROD restituiscono il **massimo ricavo**, non il **modo** di taglio per ottenerlo
- uso array  $s[ ]$  in cui tengo traccia del modo di effettuare il taglio:
  - ▶  $s[j]$ : lunghezza del primo pezzo nel taglio ottimale di un'asta lunga  $j$

EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```

 $r[0..n]$  ed  $s[0..n]$  due nuovi array
 $r[0] \leftarrow 0$ 
for  $j \leftarrow 1$  to  $n$ 
     $q \leftarrow -\infty$ 
    for  $i \leftarrow 1$  to  $j$ 
        if  $q < p[i] + r[j-i]$ 
             $q \leftarrow p[i] + r[j-i]$ 
             $s[j] \leftarrow i$ 
     $r[j] \leftarrow q$ 
return ( $r, s$ )

```

PRINT-CUT-ROD-SOLUTION( $p, n$ )

```

( $r, s$ )  $\leftarrow$  EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )
while  $n > 0$ 
    print  $s[n]$ 
     $n \leftarrow n - s[n]$ 

```

## Esempio di esecuzione

Eseguire

- PRINT-CUT-ROD-SOLUTION( $p, 7$ )

lunghezza $i$	1	2	3	4	5	6	7	8	9	10
prezzo $p_i$	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$p_i$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

# Longest Common Subsequence

(sottosequenza comune più lunga )

## Problema

Date 2 sequenze,  $X = \langle x_1, \dots, x_m \rangle$  e  $Y = \langle y_1, \dots, y_n \rangle$   
trovare una sottosequenza comune ad  $X$  e  $Y$  di lunghezza massima

## Sottosequenza

- Sottosequenza di una sequenza:  
sequenza a cui tolgo 0 o più elementi
- - ▶  $X = \langle x_1, x_2, \dots, x_m \rangle$
  - ▶  $Z = \langle z_1, z_2, \dots, z_k \rangle$  è una **sottosequenza** di  $X$  se  
 $\exists$  una sequenza di indici  $I = \langle i_1, i_2, \dots, i_k \rangle$  t.c.  
 $i_j > i_h$  se  $j > h$     $\forall j, h = 1, 2, \dots, k$  e  
 $x_{i_j} = z_j$     $\forall j = 1, 2, \dots, k$
- $\langle b, c, f \rangle$  sottosequenza di  $\langle a, b, c, d, e, f, g \rangle$  con  $I = \langle 2, 3, 6 \rangle$

## Esempi sottosequenze

- calibro
  - ▶ Sottosequenze: a, li, ab, cio
  - ▶ **Non** sono sottosequenze: ba, oa, ...

## Esempi LCS

- panca  $\leftrightarrow$  campa  
 $\rightarrow pa$ , ma anche aa
- angelo  $\leftrightarrow$  demone  
 $\rightarrow ne$
- CDADDCC  $\leftrightarrow$  BABADBC  
 $\rightarrow ADC$

## Algoritmo forza-bruta

- Per ogni sotto-seguenza di  $X$ 
  - ▶ controllare se è una sotto-seguenza di  $Y$

## Tempo

- $2^m$  sottosequenze di  $X$  da controllare
- $\Theta(n)$  per controllare ogni sottosequenza:
  - ▶ scorrere  $Y$  per la prima lettera, poi per la seconda e così via
- $\Theta(n \cdot 2^m)$

## Programmazione dinamica: passi principali

- ① Caratterizzare la struttura di una soluzione ottima
- ② Definire ricorsivamente il valore di una soluzione ottima del problema
- ③ Calcolare una soluzione ottima in modo bottom-up
  - ▶ dai sottoproblemi più semplici a quelli più difficili, fino al problema originario
- ④ Costruire una soluzione ottima del problema richiesto

# Passo 1: Sotto-struttura ottima (sottosequenza comune)

## Notazione

- $X_i$  = prefisso di  $X$  con  $i$  elementi:  $\langle x_1, \dots, x_i \rangle$
- $Y_j$  = prefisso di  $Y$  con  $j$  elementi:  $\langle y_1, \dots, y_j \rangle$
- **Esempio:**  
se  $X = \langle A, B, C, D \rangle$  si ha  
 $X_1 = \langle A \rangle$  e  $X_3 = \langle A, B, C \rangle$

## Teorema

Date le sequenze  $X = \langle x_1, \dots, x_m \rangle$  e  $Y = \langle y_1, \dots, y_n \rangle$   
sia  $Z = \langle z_1, \dots, z_k \rangle$  una qualsiasi LCS di  $X$  e  $Y$

- ① Se  $x_m = y_n$ , allora  $z_k = x_m = y_n$  e  $Z_{k-1}$  è una LCS di  $X_{m-1}$  e  $Y_{n-1}$
- ② Se  $x_m \neq y_n$ , allora  $z_k \neq x_m$  implica che  $Z$  è una LCS di  $X_{m-1}$  e  $Y$
- ③ Se  $x_m \neq y_n$ , allora  $z_k \neq y_n$  implica che  $Z$  è una LCS di  $X$  e  $Y_{n-1}$

## Dimostrazione 1/3

- ① Se  $x_m = y_n$  allora  $z_k = x_m = y_n$  e  $Z_{k-1}$  è LCS di  $X_{m-1}$  e  $Y_{n-1}$
- ② Mostro che se  $x_m = y_n$  allora  $z_k = x_m = y_n$ 
  - ▶ Suppongo  $z_k \neq x_m$
  - ▶ **Costruisco**  $Z' = \langle z_1, \dots, z_k, x_m \rangle$ 
    - ★ E' una CS di  $X$  e  $Y$  e ha lunghezza  $k + 1 \Rightarrow$
    - ★  $Z'$  è una CS più lunga di  $Z \Rightarrow$
    - ★ **contraddice** ipotesi che  $Z$  sia una LCS
- ③ Mostro che  $Z_{k-1}$  è una **LCS** di  $X_{m-1}$  e  $Y_{n-1}$ 
  - ▶ E' chiaramente una **CS**
  - ▶ Mostro che è **LCS**
    - ★ **supponiamo** che esista una CS  $W$  di  $X_{m-1}$  e  $Y_{n-1}$  che sia più lunga di  $Z_{k-1}$  (lunghezza di  $W \geq k$ )
    - ★ Costruisco  $W'$  appendendo  $x_m$  a  $W$
    - ★  $W'$  è una CS di  $X$  e  $Y$ , ha lunghezza  $\geq k + 1 \Rightarrow$  contraddice il fatto che  $Z$  sia una LCS di  $X$  e  $Y$

## Dimostrazione 2-3

① visto

- ② Se  $x_m \neq y_n$ , allora  $z_k \neq x_m$  implica che  $Z$  è una LCS di  $X_{m-1}$  e  $Y$
- ③ Se  $x_m \neq y_n$ , allora  $z_k \neq y_n$  implica che  $Z$  è una LCS di  $X$  e  $Y_{n-1}$

- ② Se  $z_k \neq x_m$ , allora  $Z$  è una **CS** di  $X_{m-1}$  e  $Y$ . Mostro che è **LCS**

**Suppongo**  $\exists$  una sottosequenza  $W$  di  $X_{m-1}$  e  $Y$  con lunghezza  $> k$

Allora  $W$  è una sottosequenza comune di  $X$  e  $Y \Rightarrow$

contraddice il fatto che  $Z$  sia una LCS

- ③ Simmetrico di 2

**Quindi**

una LCS di due sequenze contiene al suo interno (come prefisso) una LCS di prefissi delle sequenze:

**sottostruttura ottima!**

## Passo 2: Formulazione ricorsiva

- Esapro uno o due sottoproblemi per trovare LCS di  $X$  e  $Y$

► Se  $x_m = y_n$  devo **risolvere un sottoproblema**

- ★ trovare una LCS di  $X_{m-1}$  e  $Y_{n-1}$   
accordo  $x_m = y_n$  a questa LCS e ottengo una LCS di  $X$  e  $Y$

► Se  $x_m \neq y_n$  devo **risolvere due sottoproblemi**:

- ★ trovare una LCS di  $X_{m-1}$  e  $Y$
- ★ trovare una LCS di  $X$  e  $Y_{n-1}$

La più lunga di queste due LCS è una LCS di  $X$  e  $Y$

- Soluzione ricorsiva di LCS con una ricorrenza:

- ▶  $c[i, j] = \text{lunghezza di LCS di } X_i \text{ e } Y_j$
- ▶ voglio  $c[m, n]$

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \text{ (X o Y e' vuota)} \\ c[i - 1, j - 1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(c[i, j - 1], c[i - 1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

- Se  $x_i \neq y_j$  devo risolvere **solo due** sottoproblemi
  - ▶ LCS di  $X_i$  e  $Y_{j-1}$
  - ▶ LCS di  $X_{i-1}$  e  $Y_j$

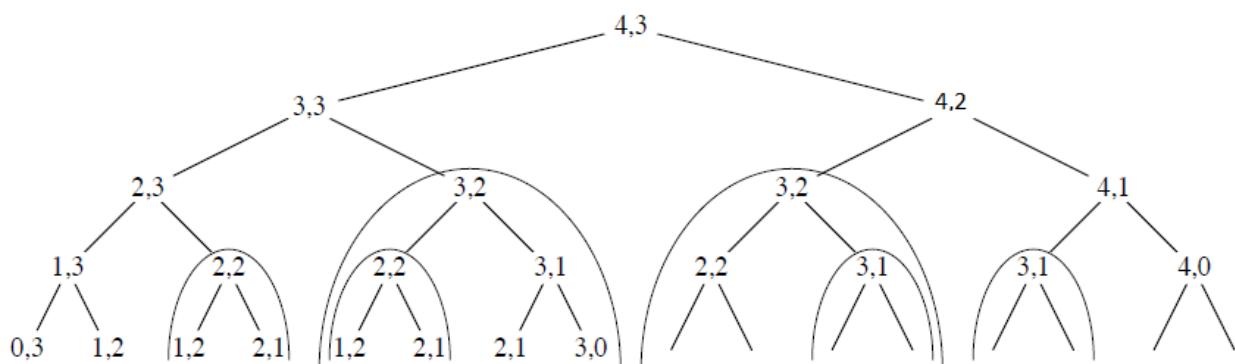
- Per taglio delle aste li consideravo tutti

- Per il teorema non abbiamo escluso soluzioni!

Posso scrivere un algoritmo ricorsivo basato su questa formulazione

### Esempio: *LCS(casa, cut)*

- ad ogni livello si ri-chiama a sx con  $i - 1$ , a dx con  $j - 1$
- dovrei arrivare a 0, 0 in ogni cammino



### Costo

- Altezza:  $n + m$  (qui  $4 + 3 = 7$ )
- Tempo è potenzialmente **esponenziale**
- Molti sottoproblemi ripetuti
- Invece di ricalcolare memorizzo in una tabella (memoization)

## Passo 3: Calcolare la lunghezza della soluzione ottima

LCS-LENGTH( $X, Y$ )

```

 $m \leftarrow X.length$ 
 $n \leftarrow Y.length$ 
Siano  $b[1..m, 1..n]$  e  $c[0..m, 0..n]$  due nuove tabelle
for  $i \leftarrow 1$  to  $m$ 
     $c[i, 0] \leftarrow 0$ 
for  $j \leftarrow 0$  to  $n$ 
     $c[0, j] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $m$ 
    for  $j \leftarrow 1$  to  $n$ 
        if  $x_i = y_j$ 
             $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
             $b[i, j] \leftarrow "\nwarrow"$ 
        elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
             $c[i, j] \leftarrow c[i - 1, j]$ 
             $b[i, j] \leftarrow "\uparrow"$ 
        else  $c[i, j] \leftarrow c[i, j - 1]$ 
             $b[i, j] \leftarrow "\leftarrow"$ 
return  $c$  e  $b$ 
```

## Passo 3: Calcolare la lunghezza della soluzione ottima

### Tabelle ausiliari

- $c[i, j]$  contiene la lunghezza ottima di LCS per  $X_i$  e  $Y_j$
- $b[i, j]$  indica la strada seguita (il sottoproblema) per risolvere LCS di  $X_i$  e  $Y_j$
- Se  $b[i, j] = \nwarrow$ , ho esteso LCS di un carattere ( $x_i = y_j$ )
- LCS contiene  $x_i$  (e  $y_j$ ) in cui  $b[i, j] = \nwarrow$

Esempio: ABCBTA  $\leftrightarrow$  ABRACA

	A	B	R	A	C	A
0	0	0	0	0	0	0
A 0	1	1	1	1	1	1
B 0	1	2	2	2	2	2
C 0						
B 0						
T 0						
A 0						

Esempio: ABCBTA  $\leftrightarrow$  ABRACA

j	0	1	2	3	4	5	6
i	y	A	B	R	A	C	A
x	0	0	0	0	0	0	0
0	x	0	0	0	0	0	0
1	A	0	↖1	←1	←1	↖1	←1
2	B	0	↑1	↖2	←2	←2	←2
3	C	0	↑1	↑2	↑2	↖3	←3
4	B	0	↑1	↖2	↑2	↑2	↑3
5	T	0	↑1	↑2	↑2	↑2	↑3
6	A	0	↖1	↑2	↑2	↖3	↑3

## Passo 4: Costruire soluzione ottima

- Chiamata: PRINT-LCS( $b, X, m, n$ )

PRINT-LCS( $b, X, i, j$ )

```

if  $i = 0$  or  $j = 0$ 
    return
if  $b[i,j] = " \nwarrow "$ 
    PRINT-LCS( $b, X, i - 1, j - 1$ )
    print  $x_i$ 
elseif  $b[i,j] = " \uparrow "$ 
    PRINT-LCS( $b, X, i - 1, j$ )
else PRINT-LCS( $b, X, i, j - 1$ )

```

## Distanza di editing

### Edit Distance ( $X \rightarrow Y$ )

- Trasformare la stringa  $X$  nella stringa  $Y$
- Posso usare solo operazioni elementari:
  - ▶ **Inserimento** di un carattere
  - ▶ **Cancellazione** di un carattere
  - ▶ **Sostituzione** di un carattere
  - ▶ ...
- Definiamo (come per LCS):
  $X_i = \langle x_1, x_2 \dots x_i \rangle$  e  $Y_j = \langle y_1, y_2 \dots y_j \rangle$
- **Sottoproblemi**: trova sequenza minima di operazioni che converta  $X_i$  in  $Y_j$  ( $0 \leq i \leq m$  e  $0 \leq j \leq n$ )
- **Minimizzo** le operazioni:
 il loro numero è la distanza tra stringhe
- Esempio: *Carta* → *Parla* è a distanza 2

## Idea generale

- $c[i, j]$  è il costo di una soluzione ottima al problema  $X_i \rightarrow Y_j$
- Suppongo di **conoscere** l'ultima operazione usata per  $X_i \rightarrow Y_j$
- Sei possibilità
- Posso calcolare  $c[i, j]$  in funzione di valori precedenti

## Se l'ultima operazione era una **copia**

- Doveva essere  $x_i = y_j$
- Resta il sottoproblema  $X_{i-1} \rightarrow Y_{j-1}$
- **Sottostruttura ottima:**  
Una soluzione ottima al problema  $X_i \rightarrow Y_j$   
**deve includere** una soluzione ottima a  $X_{i-1} \rightarrow Y_{j-1}$
- Quindi, supponendo che l'ultima operazione fosse una copia:

$$c[i, j] = c[i - 1, j - 1] + \text{costo(copia)}$$

## Se era una **sostituzione**

- Doveva essere  $x_i \neq y_j$
- Sottostruttura ottima (come copia)
  - ▶ **Sottoproblema**  $X_{i-1} \rightarrow Y_{j-1}$ .
- Supponendo che l'ultima operazione fosse una sostituzione:

$$c[i, j] = c[i - 1, j - 1] + \text{costo(sostituzione)}$$

## Se era uno **scambio**

- Doveva essere  $x_i = y_{j-1}$  e  $x_{i-1} = y_j$   
(con  $i, j \geq 2$ )
- **Sottoproblema**  $X_{i-2} \rightarrow Y_{j-2}$  e
- Supponendo che l'ultima operazione fosse uno scambio:

$$c[i, j] = c[i - 2, j - 2] + \text{costo(scambio)}$$

## Se era una cancellazione

- Non ho restrizioni su  $X$  e  $Y$
- Cancellazione: rimozione di un carattere da  $X_i$  lasciando  $Y_j$  invariato
- **Sottoproblema**  $X_{i-1} \rightarrow Y_j$
- Supponendo che l'ultima operazione fosse una cancellazione:

$$c[i, j] = c[i - 1, j] + \text{costo(cancellazione)}$$

## Se era una inserimento

- Non ho restrizioni su  $X$  e  $Y$
- Inserimento: rimozione di un carattere da  $Y_j$  lasciando  $X_i$  invariato
- **Sottoproblema**  $X_i \rightarrow Y_{j-1}$
- Supponendo che l'ultima operazione fosse un inserimento si ha

$$c[i, j] = c[i, j - 1] + \text{costo(inserimento)}$$

## Casi base

- Se  $i = 0$  o  $j = 0$
- $X_0$  e  $Y_0$  sono stringhe vuote
- Converte la **stringa vuota** in  $Y_j$  con  $j$  **inserimenti**
  - ▶  $c[0, j] = j \cdot \text{costo(inserimento)}$
- Converte  $X_i$  in  $Y_0$  con  $i$  **cancellazioni**
  - ▶  $c[i, 0] = i \cdot \text{costo(cancellazione)}$
- Con  $i = j = 0$  ho  $c[0, 0] = 0$ 
  - ▶ Non c'è costo per convertire la stringa vuota nella stringa vuota

## Formulazione ricorsiva

$c[i, j]$  per  $i, j > 0$  applicando le formule precedenti

$$c[i, j] = \min \begin{cases} c[i - 1, j - 1] + \text{cost}(copy) & \text{se } x[i] = y[j] \\ c[i - 1, j - 1] + \text{cost}(replace) & \text{se } x[i] \neq y[j] \\ c[i - 2, j - 2] + \text{cost}(twiddle) & \text{se } i, j \geq 2, \\ & x[i] = y[j - 1], \\ & \text{e } x[i - 1] = y[j] \\ c[i - 1, j] + \text{cost}(delete) & \text{sempre,} \\ c[i, j - 1] + \text{cost}(insert) & \text{sempre.} \end{cases}$$

## Algoritmo

- Come LCS lo pseudocodice riempie la tabella “per righe”
- Funzionerebbe anche “per colonne”
- $op[i, j]$  memorizza l’operazione usata

EDIT-DISTANCE( $x, y$ )

```

 $m \leftarrow x.length$ 
 $n \leftarrow y.length$ 
Siano  $c[0..m, 0..n]$  e  $op[0..m, 0..n]$  due nuove tabelle
for  $i \leftarrow 0$  to  $m$ 
     $c[i, 0] \leftarrow i \cdot \text{cost}(delete)$ 
     $op[i, 0] \leftarrow \text{DELETE}$ 
for  $j \leftarrow 0$  to  $n$ 
     $c[0, j] \leftarrow j \cdot \text{cost}(insert)$ 
     $op[0, j] \leftarrow \text{INSERT}$ 
...

```

```

...
for  $i \leftarrow 1$  to  $m$ 
  for  $j \leftarrow 1$  to  $n$ 
     $c[i,j] \leftarrow \infty$ 
    if  $x_i = y_j$ 
       $c[i,j] \leftarrow c[i-1,j-1] + cost(copy)$ 
       $op[i,j] \leftarrow COPY$ 
    if  $x_i \neq y_j$  and  $c[i-1,j-1] + cost(replace) < c[i,j]$ 
       $c[i,j] \leftarrow c[i-1,j-1] + cost(replace)$ 
       $op[i,j] \leftarrow REPLACE(by y_j)$ 
    if  $i \geq 2$  and  $j \geq 2$  and  $x_i = y_{j-1}$  and
       $x_{i-1} = y_j$  and  $c[i-2,j-2] + cost(twiddle) < c[i,j]$ 
       $c[i,j] \leftarrow c[i-2,j-2] + cost(twiddle)$ 
       $op[i,j] \leftarrow TWIDDLE$ 
    if  $c[i-1,j] + cost(delete) < c[i,j]$ 
       $c[i,j] \leftarrow c[i-1,j] + cost(delete)$ 
       $op[i,j] \leftarrow DELETE$ 
    if  $c[i,j-1] + cost(insert) < c[i,j]$ 
       $c[i,j] \leftarrow c[i,j-1] + cost(insert)$ 
       $op[i,j] \leftarrow INSERT(y_j)$ 
return  $c$  and  $op$ 

```

## Analisi

- **Tempo**  $\Theta(m \cdot n)$
- **Spazio**  $\Theta(m \cdot n)$

## Ricostruire la sequenza ottima di operazioni

- Si usa la tabella  $op$  restituita da EDIT-DISTANCE
- Chiama OP-SEQUENCE( $op, m, n$ )

OP-SEQUENCE( $op, i, j$ )

```

if  $i = 0$  and  $j = 0$ 
    return
if  $op[i, j] = COPY$  or  $op[i, j] = REPLACE$ 
     $i' \leftarrow i - 1$ 
     $j' \leftarrow j - 1$ 
elseif  $op[i, j] = TWIDDLE$ 
     $i' \leftarrow i - 2$ 
     $j' \leftarrow j - 2$ 
elseif  $op[i, j] = DELETE$ 
     $i' \leftarrow i - 1$ 
     $j' \leftarrow j$ 
else // Deve essere  $op[i, j] = INSERT$ 
     $i' \leftarrow i$ 
     $j' \leftarrow j - 1$ 
OP-SEQUENCE( $op, i', j'$ )
PRINT  $op[i, j]$ 
```

## Esempio: ALABARDA e BANDANA

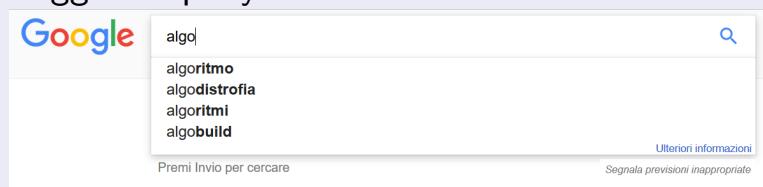
/	/	B	A	N	D	A	N	A
/	0	1	2	3	4	5	6	7
A	1	1	1	2	3	4	5	6
L	2	2	2	2	3	4	5	6
A	3	3	2	3	3	3	4	5
B	4	3	3	3	4	4	4	5
A	5	4	3	4	4	4	5	4
R	6	5	4	4	5	5	5	5
D	7	6	5	5	4	5	6	6
A	8	7	6	6	5	4	5	6

# Utilizzi di Edit Distance

## Correzione ortografica

- Due usi:

- ▶ Correggere documenti con parole scritte in modo errato
- ▶ Suggerire query all'utente



- Due approcci principali:

- ▶ Parole isolate
  - ★ Controllo ogni parola indipendentemente dalle altre
  - ★ Non trovo errori di parole scambiate (es: *pino* ↔ *lino*)
- ▶ Context-sensitive
  - ★ Guardo il contesto della frase (es: Ho tagliano un lino)

## Google Books Ngram Viewer

Graph these comma-separated phrases:   case-insensitive  
 between  and  from the corpus  with smoothing of



## Correzione di documenti

- Necessario per documenti "letti" con OCR
  - ▶ Considero errori specifici
    - ★ esiste modem in documenti del XIX secolo?
    - ★ Confondo *m* con *n*
  - ▶ Può usare conoscenza specifica del dominio
    - ★ Es., OCR può confondere O e D
- Anche documenti elettronici (ed pagine Web) hanno errori
  - ▶ Si confondono O e I (adiacenti in una tastiera QWERTY)
- Spesso non modifichiamo il documento, ma ottimizzo il match query-documenti memorizzati

## Correzione di parole isolate

- Ipotesi: esiste un lessico con lo spelling corretto delle parole
- Due approcci:
  - ▶ Usa lessico standard
    - ★ Webster's English Dictionary
    - ★ Un lessico specifico per documenti in domini specifici
  - ▶ Usa le parole della collezione
    - ★ Tutte le parole nel Web
    - ★ Tutti i nomi, acronimi ecc.
    - ★ (Inclusi gli errori !?!)

- Dato lessico  $L$  e la stringa  $Q$ , trovare le parole in  $L$  più vicine a  $Q$
- Cosa vuol dire "più vicino"?
- Alternative:
  - ▶ Edit distance (già vista)
  - ▶ Weighted edit distance
  - ▶ Intersezione di n-gram

## Weighted edit distance

- Edit Distance dove il costo di una operazione dipende dai caratteri coinvolti non solo dal tipo di operazione
  - ▶ Per gestire errori di OCR o di battitura
    - ★ Esempio:  $m$  confuso più spesso con  $n$  che con  $q$
    - ★ Sostituendo  $m$  con  $n$  dovrei avere edit distance inferiore
- Serve una matrice di pesi  $W$  che dipende dall'applicazione

## Modificare Edit distance tenendo conto di $W$

- ...

## Usare edit distance

- Posso confrontare  $Q$  con tutte le parole in  $L$ ?
- Troppo costoso
- Una possibilità :
  - ▶ Da  $Q$  **elenco** tutte le sequenze di caratteri con edit distance inferiore ad una soglia (es. 2)
  - ▶ Interseco questo insieme con  $L$
  - ▶ Mostro i termini trovati all'utente

## Edit distance con tutto il dizionario?

- Come riduco l'insieme dei termini candidati nel dizionario?
- Una possibilità : intersezione di n-gram

## Intersezione di n-gram

- Enumero tutti gli n-gram in  $Q$  e in  $L$
- Uso un indice di n-gram per trovare tutti i termini in  $L$  che contengono **qualunque** n-gram di  $Q$
- O cerco tutti i termini che contengono **abbastanza** n-gram di  $Q$

## Esempio con 3-gram

- $L = \text{november}$ 
  - ▶ 3-gram: nov, ove, vem, emb, mbe, ber
- $Q = \text{december}$ 
  - ▶ 3-gram: dec, ece, cem, emb, mbe, ber
- Tre 3-gram di sovrapposizione (su 6 in ogni parola)
- Come avere una misura di sovrapposizione normalizzata?

## Coefficiente di Jaccard

- Dati gli insiemi  $X$  e  $Y$   
il coefficiente di Jaccard è :

$$JC = \frac{|X \cap Y|}{|X \cup Y|}$$

- $X$  e  $Y$  non devono avere la stessa dimensione
- $0 \leq JC \leq 1$ 
  - ▶ 1 se  $X$  e  $Y$  hanno gli stessi elementi
  - ▶ 0 se sono disgiunti
- Nel nostro caso  $X$  e  $Y$  sono insiemi di n-gram
- Per trovare i termini considero una soglia
  - ▶ Es., se  $JC > 0.8$ , corrispondono

# Algoritmi elementari per grafi

## Grafo orientato (diretto)

- $G = (V, E)$ :  
 $V$  è un insieme finito ed  $E$  è una relazione binaria in  $V$
- $V$ : **insieme dei vertici** di  $G$
- $E$ : **insieme degli archi** di  $G$
- Arco: insieme  $\{u, v\}$  dove  $u, v \in V$  e  $u \neq v$
- Indico l'arco con  $(u, v)$  anziché con  $\{u, v\}$
- **Grado** di un vertice: numero di archi che incidono nel vertice

## Grafo non orientato (indiretto)

- $G = (V, E)$  in cui  $E$  è composto da coppie di vertici *non ordinate* anziché da coppie *ordinate*

## Cammini

- **Cammino** di lunghezza  $k$  dal vertice  $u$  a  $u'$  in  $G = (V, E)$ : sequenza  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  di vertici t.c.  $u = v_0$ ,  $u' = v_k$  e  $(v_{i-1}, v_i) \in E$  per  $i = 2, \dots, k$   
**Lunghezza** del cammino: numero di archi nel cammino
- In  $G$  orientato un cammino  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  forma un **ciclo** se  $v_0 = v_k$  e il cammino contiene almeno un arco  
 Un grafo senza cicli è **aciclico**
- $G$  non orientato è **connesso** se ogni coppia di vertici è collegata attraverso un cammino
- **Componenti connesse** di  $G$ : classi di equivalenza dei vertici secondo la relazione “è raggiungibile da”
- Un grafo orientato è **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dall'altro

## Rappresentazione grafi

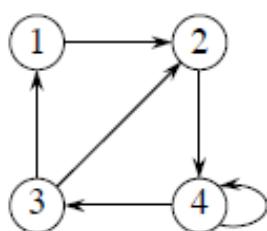
- $G = (V, E)$  può essere **diretto** o **indiretto**
- Due rappresentazioni comuni:
  - 1 Liste di adiacenza
  - 2 Matrice di adiacenza
- **Tempo** spesso espresso considerando sia  $|V|$  che  $|E|$
- Anche **spazio**
- In notazione asintotica si omette la cardinalità  
 Esempio:  $O(V + E)$

## Liste di adiacenza

- Vettore  $Adj$  di  $|V|$  liste, una per nodo
- $Adj[u]$  contiene tutti i nodi  $v$  t.c.  $(u, v) \in E$   
(per grafi indiretti considero sia  $(u, v)$  che  $(v, u)$ )
- **Pesi** degli archi ( $w : E \rightarrow R$ ) si memorizzano nelle liste  
(in generale attributi)

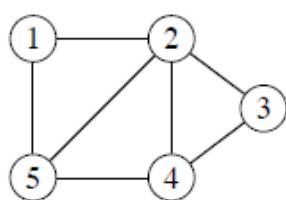
## Esempi

- Grafo diretto



<i>Adj</i>	
1	→ 2 /
2	→ 4 /
3	→ 1 → 2 /
4	→ 4 → 3 /

- Grafo indiretto



<i>Adj</i>	
1	→ 2 → 5 /
2	→ 1 → 5 → 4 → 3 /
3	→ 2 → 4 /
4	→ 2 → 5 → 3 /
5	→ 4 → 1 → 2 /

## Liste di adiacenza

- Vettore  $Adj$  di  $|V|$  liste, una per nodo
- $Adj[u]$  contiene tutti i nodi  $v$  t.c.  $(u, v) \in E$   
(per grafi indiretti considero sia  $(u, v)$  che  $(v, u)$ )
- **Pesi** degli archi ( $w : E \rightarrow R$ ) si memorizzano nelle liste  
(in generale attributi)

## Spazio

- $\Theta(V + E)$

## Tempo

- **elencare tutti** i nodi adiacenti ad  $u$  :  
 $\Theta(u.degree)$
- **determinare** se  $(u, v) \in E$  :  
 $O(u.degree)$

## Matrice di adiacenza

- Matrice  $|V| \times |V| A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

## Esempi visti prima

	1	2	3	4	5	
1	0	1	0	0	1	
2	1	0	1	1	1	
3	0	1	0	1	0	
4	0	1	1	0	1	
5	1	1	0	1	0	

	1	2	3	4	
1	0	1	0	0	
2	0	0	0	1	
3	1	1	0	0	
4	0	0	1	1	

## Matrice di adiacenza

- Matrice  $|V| \times |V| A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

## Spazio

- $\Theta(V^2)$

## Tempo

- **elencare tutti** i vertici adiacenti a  $u$ :  
 $\Theta(V)$
- **determinare** se  $(u, v) \in E$  :  
 $\Theta(1)$
- Uno meglio e uno peggio rispetto a liste di adiacenza...
- Per grafi pesati memorizzo pesi invece di bit

## Visita in Ampiezza (Breadth-First search)

- Scopre vertici raggiungibili da nodo sorgente  $s \in V$
- **Input:** Grafo  $G = (V, E)$  e nodo sorgente  $s \in V$
- **Output:**  $\forall v \in V$ 
  - ▶  $v.d$  = distanza (numero minimo di archi) da  $s$  a  $v$
  - ▶  $v.\pi = u$  t.c.  $(u, v)$  è l'ultimo arco nel cammino minimo  $s \rightsquigarrow v$ 
    - ★  $u (= v.\pi)$ : **predecessore** di  $v$
    - ★ l'insieme di archi  $\{(v.\pi, v) : v \neq s\}$  **forma un albero**
  - ▶ Vedremo poi una generalizzazione con pesi sugli archi

## Coda FIFO

- ENQUEUE inserisce in coda FIFO
- DEQUEUE rimuove da coda FIFO
- Come si potrebbe implementare?
- Quanto costano ENQUEUE e DEQUEUE?

$\text{BFS}(G, s)$

```

for ogni vertice  $u \in G.V - \{s\}$ 
     $u.\text{color} \leftarrow \text{WHITE}$ 
     $u.d \leftarrow \infty$ 
     $u.\pi \leftarrow \text{NIL}$ 
     $s.\text{color} \leftarrow \text{GRAY}$ 
     $s.d \leftarrow 0$ 
     $s.\pi \leftarrow \text{NIL}$ 
     $Q \leftarrow \emptyset$ 
ENQUEUE( $Q, s$ )
while  $Q \neq \emptyset$ 
     $u \leftarrow \text{DEQUEUE}(Q)$ 
    for ogni  $v \in G.\text{Adj}[u]$ 
        if  $v.\text{color} = \text{WHITE}$ 
             $v.\text{color} \leftarrow \text{GRAY}$ 
             $v.d \leftarrow u.d + 1$ 
             $v.\pi \leftarrow u$ 
            ENQUEUE( $Q, v$ )
     $u.\text{color} \leftarrow \text{BLACK}$ 

```

## Idea

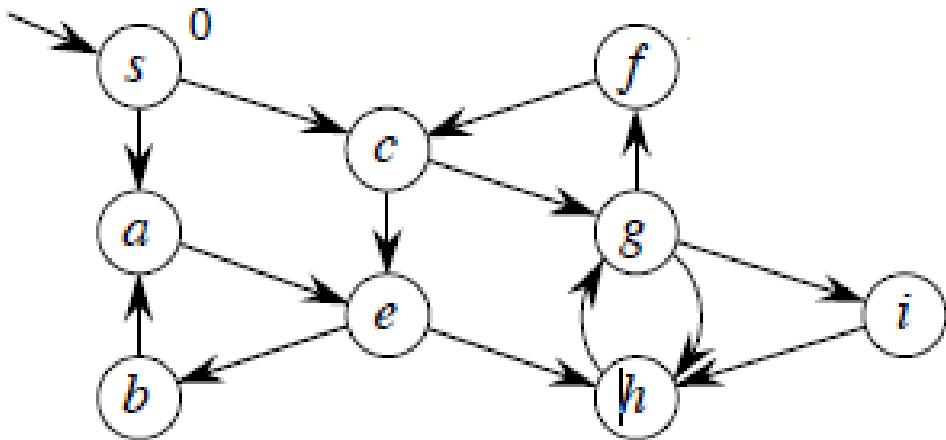
- Manda un'onda da  $s$ 
  - ▶ Prima colpisce tutti i nodi a **distanza 1** da  $s$
  - ▶ Poi colpisce tutti i nodi a **distanza 2** da  $s$
  - ▶ ...
- Memorizza il fronte dell'onda nella coda  $Q$   
 $v \in Q$  sse l'onda ha colpito  $v$ , ma non è ancora uscita da  $v$

## Colore nodi

- Mentre BFS progredisce ogni nodo ha un colore:
  - ▶  $\text{WHITE} = \text{non scoperto}$
  - ▶  $\text{GRAY} = \text{scoperto, ma non terminato}$
  - ▶  $\text{BLACK} = \text{terminato}$

## Esempio

Grafo diretto



## Esercizio

Si consideri il seguente grafo orientato costituito da 7 vertici e da 7 archi (il grafo è rappresentato tramite liste di adiacenza)

A → B

C → A, E, F

D → C

E → D

F → G

- Si disegni il grafo
- Si rappresenti tramite matrice di adiacenza
- Si esegua un attraversamento BFS partendo dal nodo A e indicando sui nodi la distanza (tramite lista di adiacenza)
- Si esegua un attraversamento BFS partendo dal nodo D e indicando sui nodi la distanza (tramite lista di adiacenza)

## Esercizio

$\text{BFS}(G, s)$

```

for ogni vertice  $u \in G.V - \{s\}$ 
     $u.\text{color} \leftarrow \text{WHITE}$ 
     $u.d \leftarrow \infty$ 
     $u.\pi \leftarrow \text{NIL}$ 
 $s.\text{color} \leftarrow \text{GRAY}$ 
 $s.d \leftarrow 0$ 
 $s.\pi \leftarrow \text{NIL}$ 
 $Q \leftarrow \emptyset$ 
ENQUEUE( $Q, s$ )
while  $Q \neq \emptyset$ 
     $u \leftarrow \text{DEQUEUE}(Q)$ 
    for ogni  $v \in G.\text{Adj}[u]$ 
        if  $v.\text{color} = \text{WHITE}$ 
             $v.\text{color} \leftarrow \text{GRAY}$ 
             $v.d \leftarrow u.d + 1$ 
             $v.\pi \leftarrow u$ 
            ENQUEUE( $Q, v$ )
     $u.\text{color} \leftarrow \text{BLACK}$ 

```

## Valori di $d$ in $Q$

- Si può mostrare che  $Q$  contiene nodi con valori  $d$  nel seguente modo:

$$i, i, i, \dots, i, i+1, i+1, \dots, i+1$$

- ▶ Solo 1 o 2 valori.
- ▶ Se ci sono 2 valori questi differiscono di 1 e quelli minori sono prima.
- Ogni nodo prende un valore  $d$  finito almeno una volta  $\rightarrow$  i valori assegnati ai nodi crescono monotonicamente nel tempo
- La **dimostrazione di correttezza è complessa** ...

## Complessità

- **Tempo** = ???
  - ▶ Ciclo sui vertici  $\Rightarrow$  **Tempo** =  $\Theta(V)$ ?
  - ▶ Doppio ciclo sui vertici  $\Rightarrow$  **Tempo** =  $\Theta(V^2)$ ?
- BFS può non raggiungere tutti i nodi  $\Rightarrow O$  e non  $\Theta$
- **Tempo** =  $O(V + E)$
- Si può usare aggregazione:
  - ▶ ENQUEUE e DEQUEUE richiedono  $O(1)$
  - ▶  $O(V)$  perché ogni nodo è **messo nella coda** al massimo una volta
  - ▶  $O(E)$  perché ogni nodo è **tolto dalla coda** al massimo una volta e si esamina  $(u, v)$  solo quando  $u$  è tolto dalla coda
  - ▶ Ogni arco è analizzato
    - ★ al massimo **una volta se diretto**
    - ★ al massimo **due volte se non diretto**

## Correttezza

- La dimostrazione di correttezza è complessa

## Visita in profondità (Depth-First search)

- **Input:**  $G = (V, E)$  (diretto o indiretto), nessun nodo sorgente
- **Output** per ogni nodo  $v$ :
  - ▶  $v.d$  = tempo di **scoperta**
  - ▶  $v.f$  = tempo di **fine**
  - ▶  $v.\pi$
- $v.d$  e  $v.f$  utili in seguito
- Esplora sistematicamente ogni arco
  - ▶ Ri-inizia da diversi nodi se necessario
  - ▶ Appena scopre un arco esplora da questo
  - ▶ *BFS* invece mette il nodo scoperto in coda FIFO

## Colore nodi

- Ogni nodo ha un colore che cambia:
  - ▶ *WHITE* = **non scoperto**
  - ▶ *GRAY* = scoperto, ma **non terminato**
  - ▶ *BLACK* = **terminato**

## Tempi di scoperta e di terminazione

- Interi unici:  $1 \leq v.d < v.f \leq 2|V|$
- $\forall v: v.d < v.f$

# DFS

$\text{DFS}(G)$

```

for ogni vertice  $u \in G.V$ 
     $u.\text{color} \leftarrow \text{WHITE}$ 
     $u.\pi \leftarrow \text{NIL}$ 
     $time \leftarrow 0$ 
for ogni vertice  $u \in G.V$ 
    if  $u.\text{color} = \text{WHITE}$ 
         $\text{DFS-VISIT}(G, u)$ 
    
```

# DFS-Visit

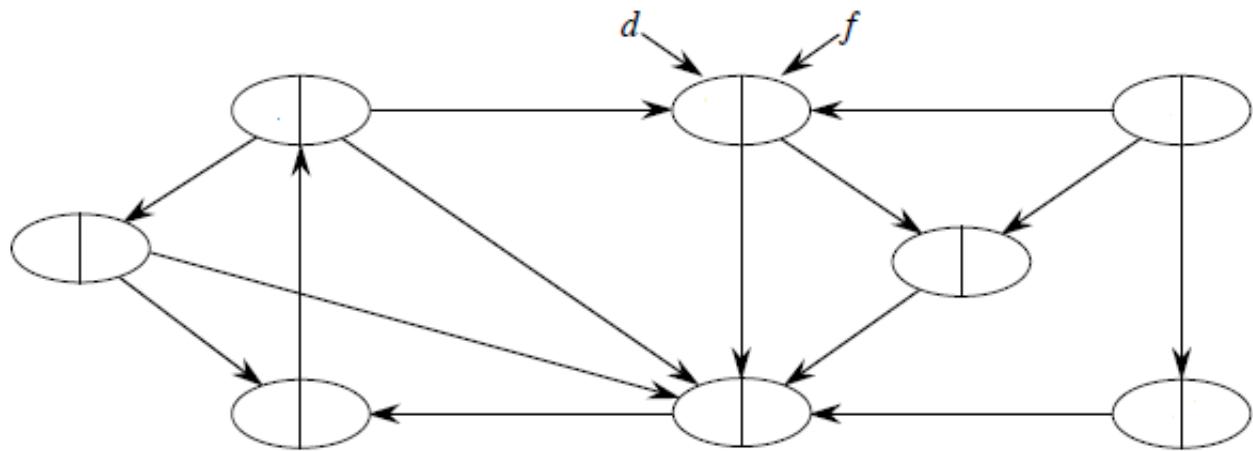
$\text{DFS-VISIT}(G, u)$

```

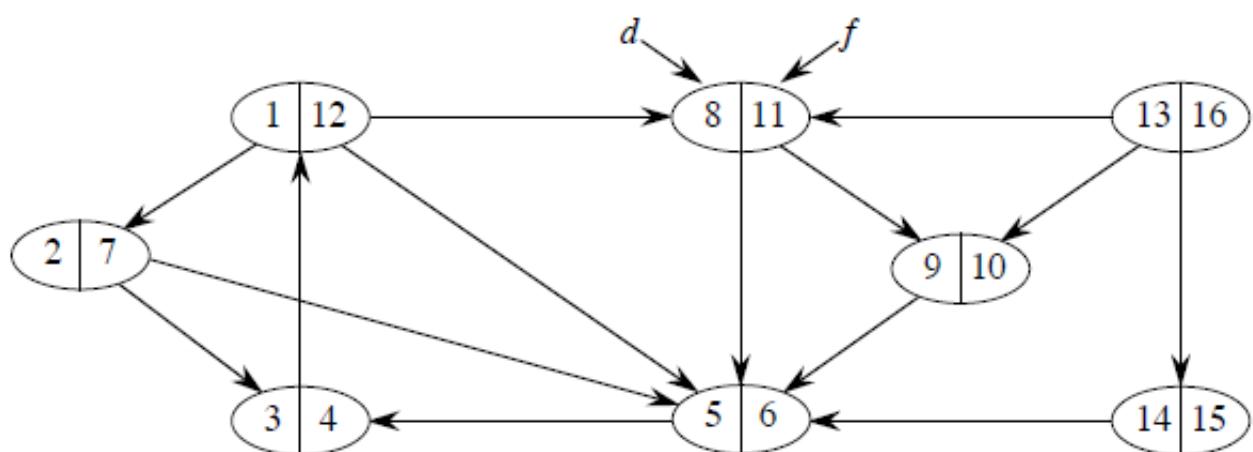
 $time \leftarrow time + 1$ 
 $u.d \leftarrow time$ 
 $u.\text{color} \leftarrow \text{GRAY}$ 
for ogni  $v \in G.\text{Adj}[u]$ 
    if  $v.\text{color} = \text{WHITE}$ 
         $v.\pi \leftarrow u$ 
         $\text{DFS-VISIT}(G, v)$ 
 $u.\text{color} \leftarrow \text{BLACK}$ 
 $time \leftarrow time + 1$ 
 $u.f \leftarrow time$ 
    
```

**Esempio**

- In ogni nodo si indica ( $v.d|v.f$ )

**Esempio**

- In ogni nodo si indica ( $v.d|v.f$ )



# Esempio

## Tempo

- $\Theta(V + E)$ 
  - ▶ Analisi simile a *BFS*
  - ▶  $\Theta$ , **non solo**  $O$ , perché esplora ogni nodo e arco

## Foresta depth-first (DF)

- Quando scopro vertice  $v$  durante l'ispezione di  $Adj[u]$  ho  $v.\pi = u$
- Il sotto-grafo dei predecessori forma una **foresta depth-first** contenente almeno un albero DF
- Ogni albero contiene archi  $(u, v)$  t.c.  
 $u$  è **gray** e  
 $v$  è **bianco**  
quando esploro  $(u, v)$

## Teorema delle parentesi (no dim)

In una **visita in profondità** di  $G(V, E)$   $\forall$  coppia di vertici  $u, v$  è soddisfatta **una sola** delle seguenti condizioni:

- ① Gli intervalli  $[u.d, u.f]$  e  $[v.d, v.f]$  sono **completamente disgiunti**:  
 $u.d < u.f < v.d < v.f$  oppure  $v.d < v.f < u.d < u.f$   
e **inoltre** ne  $u$  ne  $v$  sono **un discendente** dell'altro in un albero DF
- ② L'intervallo di  $v$  è **completamente contenuto** in quello di  $u$   
 $u.d < v.d < v.f < u.f$  e  
 $v$  è un discendente di  $u$  in un albero DF
- ③ L'intervallo di  $u$  è **completamente contenuto** in quello di  $v$   
 $v.d < u.d < u.f < v.f$  e  
 $u$  è un discendente di  $v$  in un albero DF

**Non può succedere** che  $u.d < v.d < u.f < v.f$

Come le parentesi:

- OK: () [] ([] [()])
- Non OK: ([] [()]

## Corollario (annidamento degli intervalli dei discendenti (no dim))

$v$  è un **discendente** di  $u$  nella foresta DF sse  $u.d < v.d < v.f < u.f$

## Teorema del cammino bianco (no dimostrazione)

In una foresta DF di un grafo  $G(V, E)$

$v$  è un **discendente** di  $u$

sse al tempo  $u.d$ , c'è un cammino  $u \rightsquigarrow v$  in  $G$  fatto di soli nodi bianchi (ancora da scoprire)

(ad eccezione di  $u$  che è stato appena colorato di grigio)

## Classificazione di archi

- Durante la visita in profondità classifico gli archi del grafo:
  - ▶ **albero (T, Tree)**: archi in foresta DF trovati esplorando  $(u, v)$
  - ▶ **indietro (B, Back)**:  $(u, v)$  t.c.  $u$  è un **discendente** di  $v$
  - ▶ **avanti (F, Forward)**:  $(u, v)$  t.c.  $v$  è un **discendente di**  $u$ , ma non è  $T$
  - ▶ **trasversale (C, Cross)**: ogni altro arco
    - ★ tra nodi dello stesso albero DF
    - 
    - ★ di diversi alberi DF

## Classificare l'arco $(u, v)$ durante la visita

- Se  $v.\text{color} = \text{WHITE}$   $\Rightarrow T$
- Se  $v.\text{color} = \text{GRAY}$   $\Rightarrow B$
- Se  $v.\text{color} = \text{BLACK}$   $\Rightarrow F$  o  $C$ 
  - ▶ Se  $u.d < v.d$   $\Rightarrow F$   $v$  è un discendente di  $u$  (t. parentesi)
  - ▶ Se  $u.d > v.d$   $\Rightarrow C$

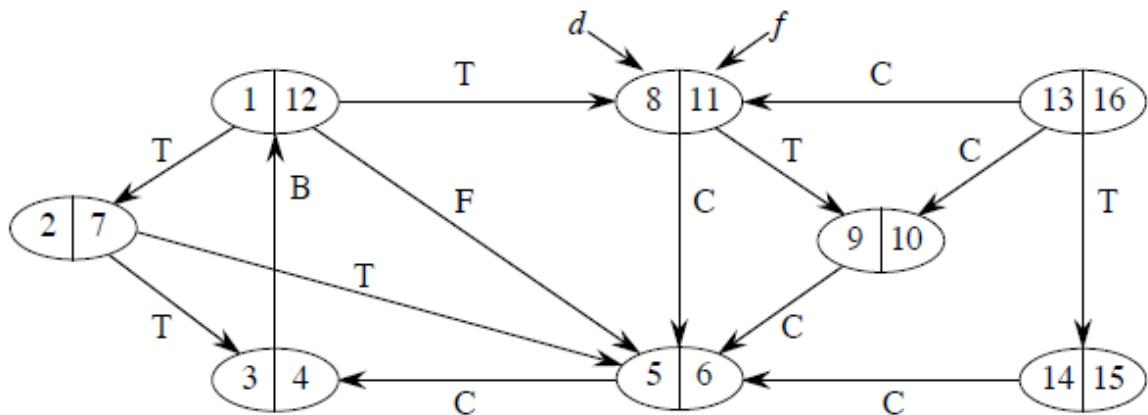
## Teorema (no dim)

Con DFS di un grafo non diretto ho solo archi  $T$  e  $B$

No archi  $F$  o  $C$

## Esempio completo

- In ogni nodo si indica ( $v.d|v.f$ )
- Archi etichettati ( $T, B, F, C$ )

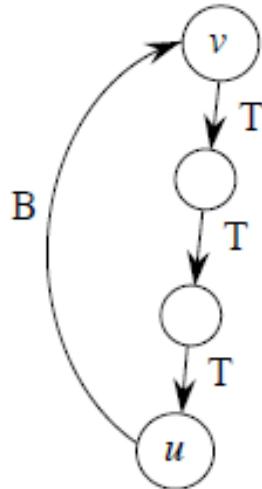


## Esempio

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	1	0	0
E	0	0	1	0	0	0	0	0
F	0	0	0	0	1	0	0	0
G	0	0	0	1	0	1	0	1
H	0	0	0	0	1	0	0	0

**Lemma**

Un grafo diretto  $G$  è aciclico sse  
una visita DFS di  $G$  non genera archi back

**Lemma**

Un grafo diretto  $G$  è aciclico sse  
una visita DFS di  $G$  non genera archi back

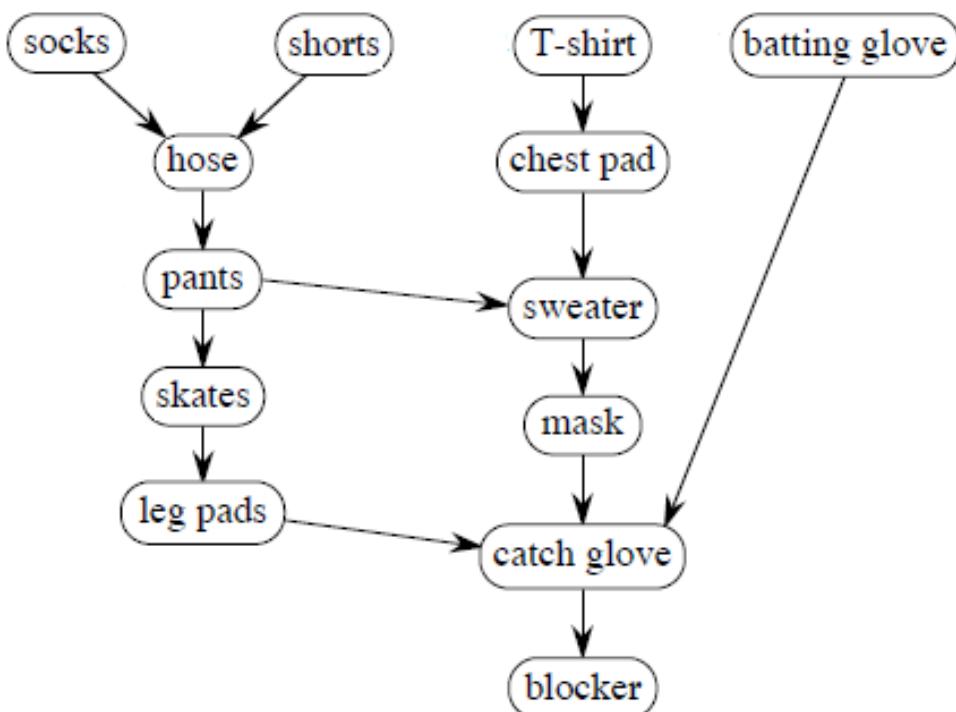
**Dimostrazione**

- Se c'è un arco back  $\Rightarrow$  ciclo
  - ▶ Supponiamo  $\exists$  un arco back  $(u, v)$
  - ▶ Allora  $v$  è un antenato di  $u$  nella foresta DF
  - ▶ Quindi c' è un percorso  $v \rightsquigarrow u$ , insieme a  $u \rightarrow v$  c' è il ciclo:  
 $v \rightsquigarrow u \rightarrow v$
- Se c'è un ciclo  $\Rightarrow$  arco back
  - ▶ Supponiamo che  $G$  contenga il ciclo  $c$
  - ▶ Sia  $v$  il primo nodo scoperto in  $c$  e sia  $(u, v)$  l'arco precedente in  $c$
  - ▶ Al tempo  $v.d$ , i vertici di  $c$  formano un cammino bianco  $v \rightsquigarrow u$  (poichè  $v$  è il primo vertice scoperto in  $c$ )
  - ▶ Da teorema del cammino bianco  $u$  è discendente in  $v$  nella foresta DF
  - ▶ Quindi,  $(u, v)$  è un arco back.

## Ordinamento topologico

- Un **ordinamento topologico** di un grafo aciclico o DAG (Directed Acyclic Graph)  $G = (V, E)$  è un
  - ▶ **ordinamento lineare** di tutti i suoi vertici t.c.
  - ▶ se  $G$  contiene un arco  $(u, v)$  allora  $u$  **appare prima di**  $v$  nell'ordinamento (da qualche parte)
  - ▶  $\neq$  ordinamento di numeri
- Per grafi ciclici **non è possibile** effettuare ordinamento lineare
- Utile per gestire oggetti che hanno un **ordine parziale**:
  - ▶  $a > b$  e  $b > c \Rightarrow a > c$
  - ▶ Potrei avere  $a$  e  $b$  t.c. non si sa se  $a > b$  o  $b > a$  (esempio, so che  $a > c$  e  $b > c$ )
- Si può sempre avere un ordinamento **totale** ( $a > b$  o  $b > a \forall a \neq b$ ) da un ordinamento **parziale**

### Esempio: vestire un portiere di pattinaggio su ghiaccio



socks (calzini) - shorts (pantaloni) - hose (calze) - pants (pantaloni) - skates (pattini) - leg pads (paragambe) - t-shirt - chest pad (sternale) - sweater (maglione) - mask (maschera) - batting glove (mano dx) - catch glove (mano sx) - blocker (mano dx)

# Ordinamento topologico: algoritmo

TOPOLOGICAL-SORT( $G$ )

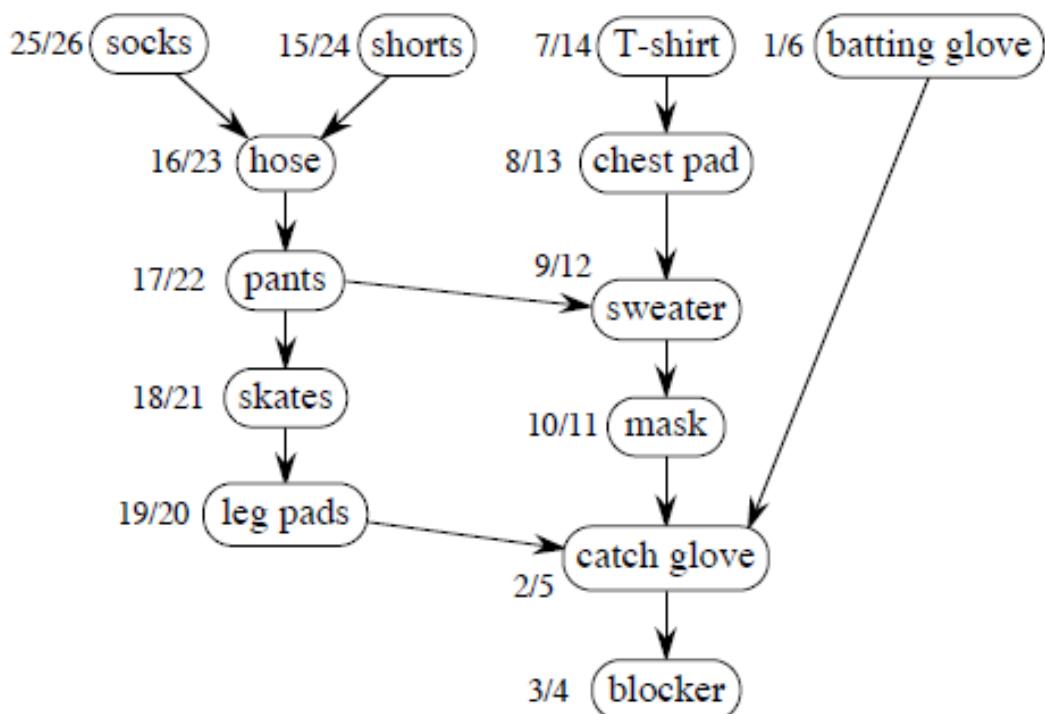
- 1 Chiama DFS( $G$ ) per calcolare i tempi di completamento  $v.f$  per ogni vertice  $v$
- 2 Completata l'ispezione del vertice, inserisce il vertice in testa a una lista concatenata
- 3 **return** la lista concatenata di vertici

- 3 equivale a:  
"Emetti i vertici in ordine di tempo di terminazione decrescente"
- Non serve ordinare per tempo di terminazione

## Tempo

$\Theta(V + E)$

## Esempio: vestire un portiere di pattinaggio su ghiaccio



## Esempio

- 26 socks (calzini)
- 24 shorts (pantaloni)
- 23 hose (calze)
- 22 pants (pantaloni)
- 21 skates (pattini)
- 20 leg pads (paragambe)
- 14 t-shirt
- 13 chest pad (sternale)
- 12 sweater (maglione)
- 11 mask (maschera)
- 6 batting glove (guanto per la mano dx)
- 5 catch glove (per la mano sinistra)
- 4 blocker (per la mano destra)

## Correttezza

Mostriamo che se  $(u, v) \in E$ , allora  $v.f < u.f$

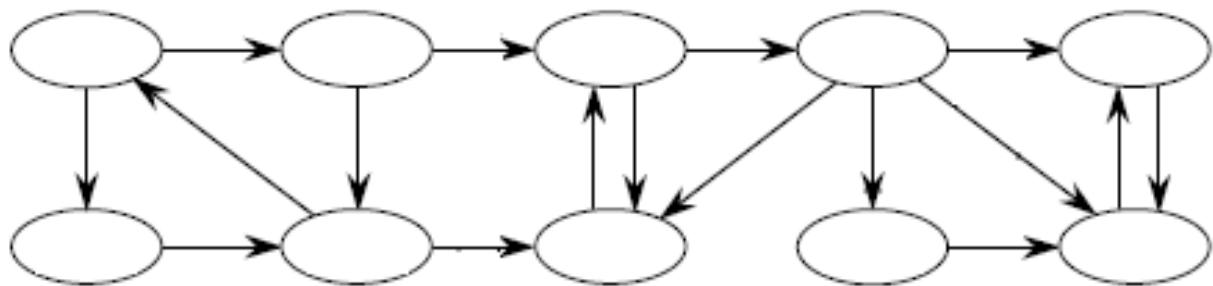
Quando **esploro**  $(u, v)$ , che colori hanno  $u$  e  $v$ ?

- $u$  è **grigio**
- Anche  $v$  è **grigio**?
  - ▶ No, altrimenti  $v$  sarebbe un antenato di  $u$ 
    - ★  $\Rightarrow (u, v)$  è un arco back
    - ★  $\Rightarrow$  contraddizione del lemma precedente (DAG non ha archi back)
- E'  $v$  **bianco**?
  - ▶ Allora diventa un discendente di  $u$   
Dal teorema delle parentesi si ha  $u.d < v.d < v.f < u.f$   
 $\Rightarrow v.f < u.f$
- E'  $v$  **nero**?
  - ▶ Allora  $v$  è già terminato  
Esploriamo  $(u, v)$  quindi  $u$  non ancora terminato  
 $\Rightarrow v.f < u.f$

## Componenti fortemente connesse

- Dato il grafo diretto  $G = (V, E)$
- Una componente fortemente connessa (Strongly Connected Component SCC) di  $G$  è un **insieme massimale di vertici**  $C \subseteq V$  t.c. per ogni  $u, v \in C$ , esistono entrambi i cammini  $u \rightsquigarrow v$  e  $v \rightsquigarrow u$

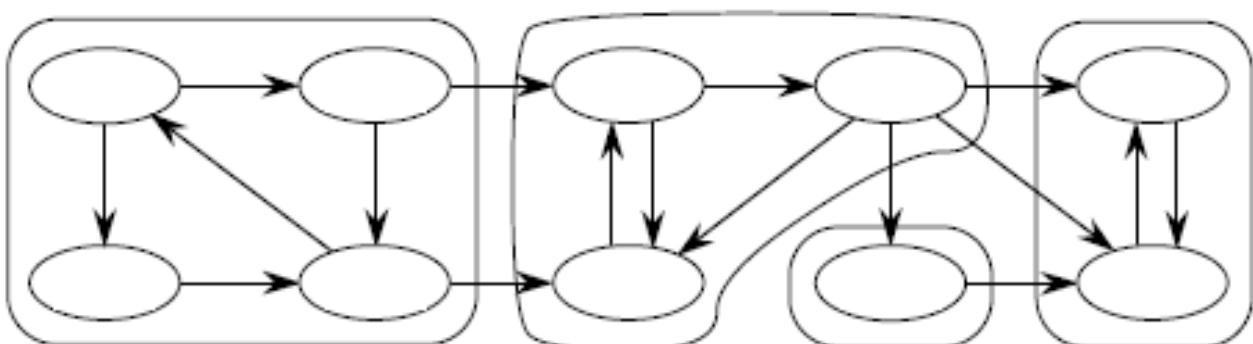
## Esempio



## Componenti fortemente connesse

- Dato il grafo diretto  $G = (V, E)$
- Una componente fortemente connessa (Strongly Connected Component SCC)  $A$  di  $G$  è un **insieme massimale di vertici**  $C \subseteq V$  t.c. per ogni  $u, v \in C$ , esistono entrambi i cammini  $u \rightsquigarrow v$  e  $v \rightsquigarrow u$

## Esempio



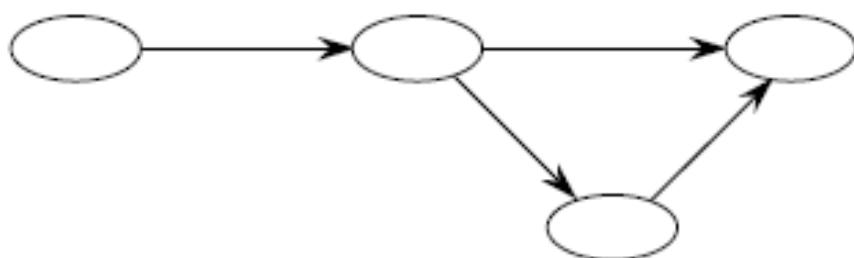
## $G^T = \text{trasposta di } G$

- $G^T = (V, E^T)$ 
  - ▶  $E^T = \{(u, v) : (v, u) \in E\}$
- $G^T$  è  $G$  con tutti gli archi invertiti
- Creazione di  $G^T$ 
  - ▶ semplice con matrice di adiacenza
  - ▶ in tempo  $\Theta(V + E)$  con liste di adiacenza
- $G$  e  $G^T$  hanno le **stesse SCC**
  - ▶  $u$  e  $v$  sono raggiungibili reciprocamente in  $G$  sse lo sono in  $G^T$

## Grafo delle componenti

- $G^{SCC} = (V^{SCC}, E^{SCC})$ 
  - ▶  $V^{SCC}$  ha un vertice per ogni SCC in  $G$
  - ▶  $E^{SCC}$  ha un arco se c'è un arco fra le corrispondenti SCC in  $G$

## Per esempio di prima?



**Lemma 22.13:**  $G^{SCC}$  è un DAG

Formalmente:

- siano  $C$  e  $C'$  due SCC distinte in  $G$   
 siano  $u, v \in C$ ,  $u', v' \in C'$ , e  
 supponiamo che ci sia un cammino  $u \rightsquigarrow u'$  in  $G$
- **allora** non ci può anche essere un cammino  $v' \rightsquigarrow v$  in  $G$

**Dimostrazione**

- Supponiamo che ci sia un cammino  $v' \rightsquigarrow v$  in  $G$
- allora ci sono cammini  
 $u \rightsquigarrow u' \rightsquigarrow v'$  e  
 $v' \rightsquigarrow v \rightsquigarrow u$  in  $G$
- $u$  e  $v'$  sono raggiungibili l'uno dall'altro e quindi non sono in  $SCC$  separate (contraddizione ipotesi)

**Componenti fortemente connesse: algoritmo****STRONGLY-CONNECTED-COMPONENTS( $G$ )**

- 1 Chiama  $DFS(G)$  per calcolare  $v.f$  per ogni vertice  $v$
- 2 Calcola  $G^T$
- 3 Chiama  $DFS(G^T)$ , nel ciclo in  $DFS$  considera i vertici in ordine decrescente rispetto ai tempi  $u.f$  (calcolati nella riga 1)
- 4 Genera l'output dei vertici di ciascun albero della foresta DF prodotta alla riga 3 come una singola componente fortemente connessa

**Tempo**

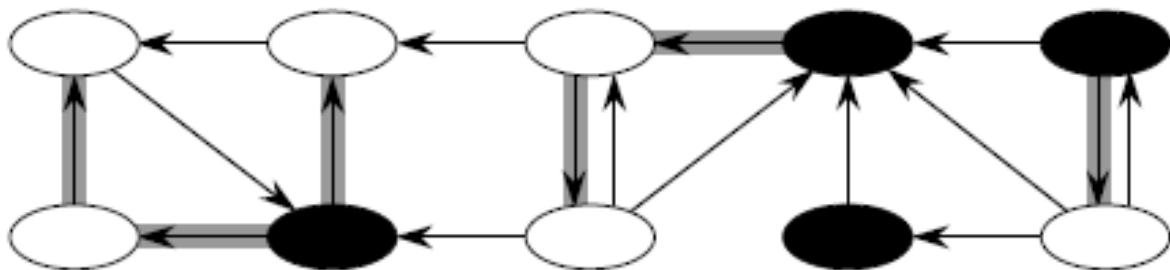
$$\Theta(V + E)$$

## Esempio di prima

① Fare DFS

②  $G^T$

③ DFS



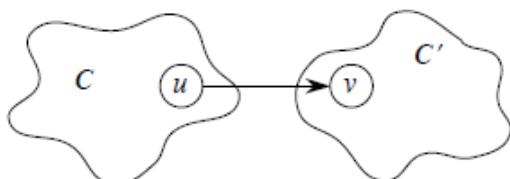
Grafo di  $G^T$  quindi archi girati rispetto a prima (radici foresta DF annerite)

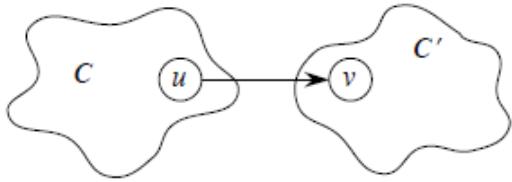
### Notazione per lemma

- $u.d$  e  $u.f$  si riferiscono al primo DFS
- Estendo notazione per  $d$  e  $f$  ad insiemi di vertici  $U \subseteq V$ :
  - $d(U) = \min_{u \in U} \{u.d\}$  (primo tempo di scoperta)
  - $f(U) = \max_{u \in U} \{u.f\}$  (ultimo tempo di fine)

### Lemma 22.14

- Siano  $C$  e  $C'$  due SCC distinte in  $G = (V, E)$  e
- supponiamo esista un arco  $(u, v) \in E$  t.c.  $u \in C$  e  $v \in C'$
- **allora**  $f(C) > f(C')$



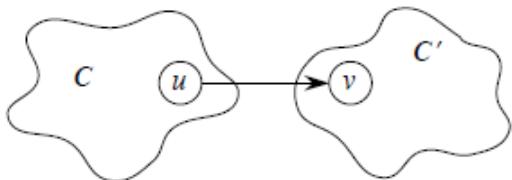


## Dimostrazione 1/2

Due casi a seconda del primo tra  $d(C)$  e  $d(C')$

- Se  $d(C) < d(C')$

- ▶  $x$ : **primo** vertice scoperto in  $C$
- ▶ al tempo  $x.d$ , tutti i vertici in  $C$  e  $C'$  sono bianchi
- ▶ cammini di vertici bianchi da  $x$  a tutti i vertici in  $C$  e  $C'$
- ▶ teorema **cammino bianco**: tutti i vertici in  $C$  e  $C'$  sono discendenti di  $x$  nell'albero DF
- ▶ teorema delle **parentesi**:  $x.f = f(C) > f(C')$



## Dimostrazione 2/2

- Se  $d(C) > d(C')$

- ▶  $y$ : primo vertice scoperto in  $C'$
- ▶ al tempo  $y.d$ 
  - ★ tutti i vertici in  $C'$  sono bianchi e c'è un cammino bianco da  $y$  ad ogni vertice in  $C'$
  - ★  $\Rightarrow$  tutti i vertici in  $C'$  sono discendenti di  $y$
  - ★  $\Rightarrow y.f = f(C')$
- ▶ al tempo  $y.d$ 
  - ★ tutti i vertici in  $C$  sono bianchi ( $d(C) > d(C')$ )
  - ★  $G^{SCC}$  è un DAG (lemma) c'è un arco  $(u, v) \Rightarrow \nexists$  arco da  $C'$  a  $C$
  - ★ Nessun vertice in  $C$  è raggiungibile da  $y$
  - ★ Al tempo  $y.f$  tutti i vertici in  $C$  sono ancora bianchi
  - ★  $\Rightarrow \forall w \in C, w.f > y.f = f(C')$
- ▶  $\Rightarrow f(C) > f(C')$

## Lemma 22.14 (appena dimostrato)

- Siano  $C$  e  $C'$  due SCC distinte in  $G = (V, E)$  e
- supponiamo esista un arco  $(u, v) \in E$  t.c.  $u \in C$  e  $v \in C'$
- **allora**  $f(C) > f(C')$

## Corollario 22.15

- Siano  $C$  e  $C'$  due SCC distinte in  $G = (V, E)$  e
- **supponiamo** che ci sia un arco  $(u, v) \in E^T$ , dove  $u \in C$  e  $v \in C'$
- **allora**  $f(C) < f(C')$

## Dimostrazione

$$(u, v) \in E^T \Rightarrow (v, u) \in E$$

Le SCC di  $G$  e  $G^T$  sono le stesse,  $f(C') > f(C)$

## Corollario 22.15 (appena dimostrato)

- Siano  $C$  e  $C'$  due SCC distinte in  $G = (V, E)$  e
- **supponiamo** che ci sia un arco  $(u, v) \in E^T$ , dove  $u \in C$  e  $v \in C'$
- **allora**  $f(C) < f(C')$

## Corollario

- Siano  $C$  e  $C'$  due SCC distinte in  $G = (V, E)$ , e
- **supponiamo** che  $f(C) > f(C')$
- **allora** non ci può essere un arco tra  $C$  e  $C'$  in  $G^T$

## Dimostrazione

Supponiamo per assurdo che esista l'arco  
allora si avrebbe  $f(C) < f(C')$  (Corollario 22.15)  
contraddicendo l'ipotesi  $f(C) > f(C')$

## Intuizione su correttezza di SCC

- DFS su  $G^T$  inizia con SCC  $C$  t.c.  $f(C)$  è massimo
  - ▶ Inizia da  $x \in C$  e visita tutti i vertici in  $C$
  - ▶ Siccome  $f(C) > f(C')$  da corollario  $\forall C' \neq C$   
 $\Rightarrow$  no archi da  $C$  a  $C'$  in  $G^T$
  - ▶  $\Rightarrow$  DFS visita solo vertici in  $C$
- Vertice successivo:  $y \in C'$  t.c.  $f(C')$  è massimo in tutte le SCC  $\neq C$ 
  - ▶ DFS visita tutti i vertici in  $C'$   
i soli archi che escono da  $C'$  vanno in  $C$  già visitato
- ...
- Si continua
  - ▶ Quando scelgo un vertice posso raggiungere solo:
    - ★ vertici nella sua SCC
    - ★ vertici in SCC **già visitati** nel secondo DFS

## Strutture dati per insiemi disgiunti

# Strutture dati per insiemi disgiunti

- Noto come: **union-find**
  - Gestisce una collezione  $S = \{S_1, \dots, S_k\}$  di **insiemi disgiunti dinamici**
  - Ogni insieme è identificato da un **rappresentante** (un membro dell'insieme)
- Non importa qual è il rappresentante, ma è **sempre lo stesso**

# Strutture dati per insiemi disgiunti

## Operazioni

- **MAKE-SET(x)**  
crea un nuovo insieme  $S_i = \{x\}$  e aggiunge  $S_i$  a  $S$
- **UNION(x, Y)**  
se  $x \in S_x$ ,  $y \in S_y$ ,  $\Rightarrow S \leftarrow S - S_x - S_y \cup \{S_x \cup S_y\}$ 
  - ▶ **Rappresentante** è qualunque elemento di  $S_x \cup S_y$  (es. uno dei rappresentanti di  $S_x$  o  $S_y$ )
  - ▶ **Distrugge**  $S_x$  e  $S_y$  (insiemi disgiunti)
- **FIND-SET(x)**  
ritorna il rappresentante dell'insieme che contiene  $x$

# Applicazione

## Componenti connesse

In un grafo  $G = (V, E)$ , i vertici  $u$  e  $v$  sono nella stessa componente连通的 sse c'è un cammino tra loro

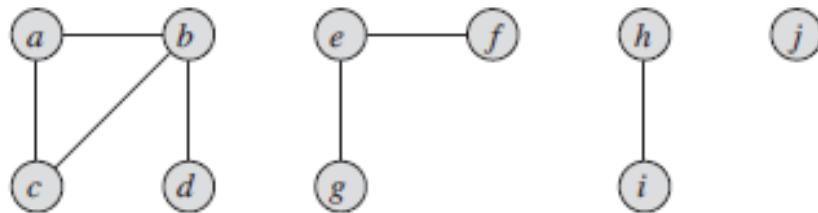
- Le componenti connesse partizionano i vertici in classi di equivalenza
- Un grafo non orientato è **connesso** se ogni coppia di vertici è collegata tramite un cammino
- Un grafo orientato può essere **fortemente connesso**
- In ogni caso: classi di equivalenza secondo la relazione "è raggiungibile da"

## Algoritmo

CONNECTED-COMPONENTS( $G$ )

```

for ogni vertice  $v \in G.V$ 
    MAKE-SET( $v$ )
for ogni arco  $(u, v) \in G.E$ 
    if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
        UNION( $u, v$ )
    
```



Arco	Insiemi disgiunti									
MAKE-SET	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

## Rappresentazione con lista concatenata

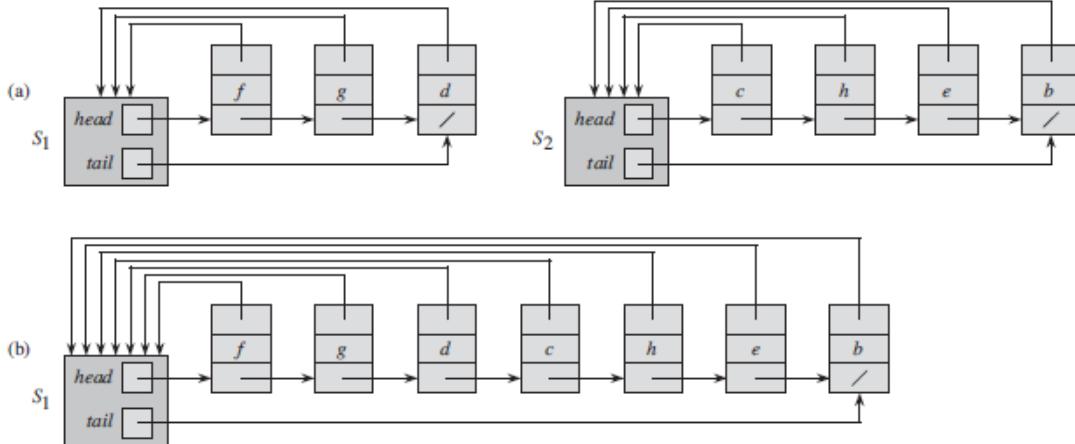
- Ogni insieme è una singola lista concatenata
- Ogni nodo nella lista ha campi per:
  - ▶ l'elemento dell'insieme
  - ▶ puntatore al rappresentante
  - ▶ puntatore al successore
- La lista ha testa (puntatore al rappresentante) e coda

## Operazioni

MAKE-SET: crea una lista con un solo elemento

FIND-SET: restituisce il puntatore al rappresentante

UNION: è quello che costa. Ci sono un paio di modi per implementarlo  
(ne vediamo uno)



UNION( $x, Y$ ): accoda la lista di  $x$  alla fine della lista di  $y$   
 Usa il puntatore alla coda di  $y$  per trovare la fine

- Aggiornare il puntatore al rappresentante per ogni nodo nella lista di  $x$
- Accoda una lista lunga ad una corta può prendere abbastanza tempo

## Euristica dell'unione pesata

- accodare sempre la lista corta alla lunga
- una singola UNION può sempre richiedere tempo  $\Omega(n)$ 
  - ▶ entrambi gli insiemi hanno  $n/2$  membri

## Teorema

Tempo con unione pesata per una sequenza di  $m$  operazioni su  $n$  elementi  
 $O(m + n \lg n)$

# Componenti connesse

## In immagini

- Permettono di trovare "oggetti" in immagini
- Definite introducendo alcune nozioni topologiche di base
  - ▶ *Adiacenza di pixel* gli 8 pixel adiacenti al pixel  $P$  sono chiamati 8NN (*eight nearest neighbors*)  
i più vicini (0,2,4,6) sono chiamati 4NN (*four nearest neighbors*)

5	6	7
4	P	0
3	2	1

- ▶ Un *percorso* è una sequenza di pixels  $P_1, P_2, \dots, P_n$  t.c.:  
 $P_{k-1}$  ( $k > 1$ ) è un vicino di  $P_k$  e  $P_{k+1}$  ( $k < n$ ) è un vicino di  $P_k$   
 Si può avere un "percorso-4" (con 4NN) o un "percorso-8" (con 8NN)

# Componenti connesse )

## In immagini

- *Connessione* Un insieme di pixel  $S$  è connesso se, per ogni coppia di pixel  $A$  e  $B \in S$ , esiste un percorso il cui primo elemento è  $A$  e l'ultimo è  $B$ , e se tutti i pixel nel percorso  $\in S$
- Una *componente连通* è un insieme massimale  $S$  di pixel neri connessi con un "percorso-8" (o un "percorso-4")

SCATTI EFFETTUATI DAL 1 DICEMBRE AL 1 FEBBRAIO		SCATTI EFFETTUATI DAL 1 DICEMBRE AL 1 FEBBRAIO	
LETTURA AL 1 FEBBRAIO	529	LETTURA AL 1 FEBBRAIO	529
LETTURA AL 1 DICEMBRE	348	LETTURA AL 1 DICEMBRE	348
TOTALE SCATTI EFFETTUATI	181	TOTALE SCATTI EFFETTUATI	181

- Se consideriamo i **pixel neri** come **nodi** del grafo e gli **archi** collegano **pixel adiacenti** allora è equivalente a:
  - ▶ Le componenti connesse di un grafo sono le classi di equivalenza dei nodi secondo la relazione "è raggiungibile da"

# Calcolo CC in immagini

## Algoritmo iterativo

- Basato su due scansioni per righe dell'immagine
  - ▶ Nella prima scansione assegno etichette provvisorie a pixel neri
    - ★ Un pixel nero (un run di pixel neri) può essere connesso ad una componente nella riga precedente dell'immagine e in tal caso ne prende l'etichetta
    - ★ Se il pixel nero è isolato, prende una nuova etichetta
  - ▶ Nella seconda scansione sotto-componenti con etichette diverse sono fuse insieme (es. per gestire 'U')
    - ★ Usando operazioni UNION-FIND
- Serve una matrice per memorizzare le etichette dei pixel

# Calcolo CC in immagini

## Algoritmo ricorsivo

- Non memorizza le etichette dei pixel e trova le componenti "cancellando" i loro pixel dall'immagine in ingresso
- Implementazione diretta della definizione di componente connessa: quando si trova un pixel nero questo è cancellato e si chiama la stessa funzione per tutti i pixel connessi neri
- I pixel non sono etichettati ⇒ non è agevole assegnare un pixel alla sua componente connessa
- L'unica informazione sulla componente connessa è la posizione del suo bounding box  
Questa informazione è comunque utile in molte applicazioni

# Progetto Laboratorio su CC

- Webcam collegata a Raspberry PI
- Elaborazione di immagini tramite libreria PIL



credits: Cantiani Camilla & Canti Edoardo

```
from PIL import Image
import numpy as np
import os
...
def main():
    os.system('fswebcam -r 640x280 --no-banner --save img.jpg' )
    img = Image.open('img.jpg')
    thresh = 100
    fn = lambda x : 255 if x > thresh else 0
    r = img.convert('L').point(fn, mode='1')
    matrix = FindConnectedComponents(r)
    BoundingBox(img, matrix)
    plt.axis('off')
    plt.show()
```

```

def FindConnectedComponents(matrix):
    pixels = matrix.load()
    labelMatrix= np.zeros((matrix.size[0],matrix.size[1]))
    label = 0
    invalid_label = 0
    for i in range(matrix.size[0]):
        for j in range(matrix.size[1]-1) :
            if pixels[i,j] == 0 :
                if (labelMatrix[i][j-1]==0 and
                    labelMatrix[i-1][j] ==0 and
                    labelMatrix[i-1][j-1] ==0 and
                    labelMatrix[i-1][j+1] ==0 ) :
                    label+=1
                    labelMatrix[i][j]= label
            else:
                if labelMatrix[i][j-1] != 0 :
                    labelMatrix[i][j] = labelMatrix[i][j-1]
            ...

```

```

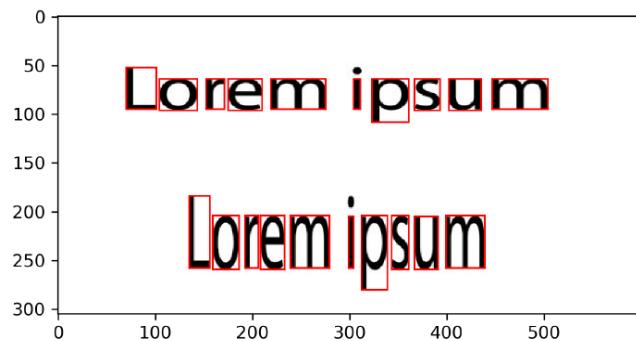
for i in range(matrix.size[0]-1):
    for j in range(matrix.size[1]) :
        if  pixels[i,j] == 0:
            if  pixels[i+1,j-1] == 0 and \
                labelMatrix[i+1][j-1] != labelMatrix[i][j]:
                target1 = labelMatrix[i+1][j-1]
                target2 = labelMatrix[i][j]

                if target1 < target2 and target1 != 0:
                    labelMatrix[labelMatrix == target2] = target1
                    invalid_label+=1
                if target1 > target2 and target2 != 0:
                    labelMatrix[labelMatrix == target1] = target2
                    invalid_label+=1

print("number of connected components:", label-invalid_label)

```

# Output

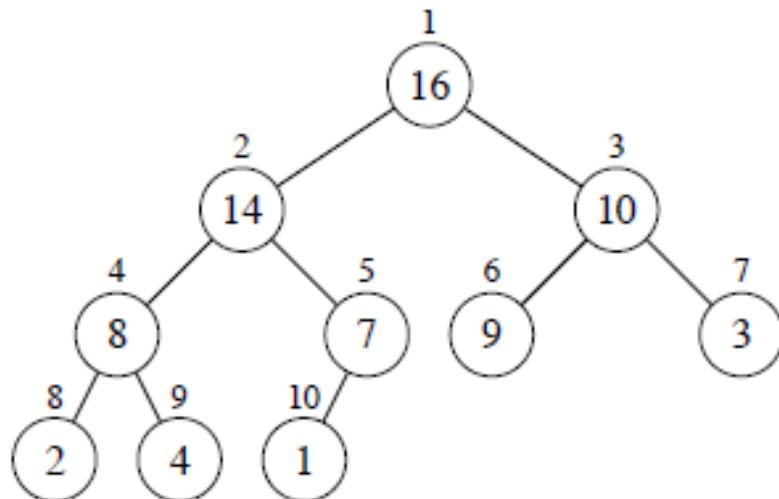


# Heap

## Heap

- Albero binario **quasi completo**

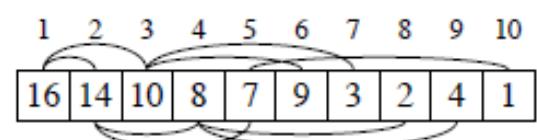
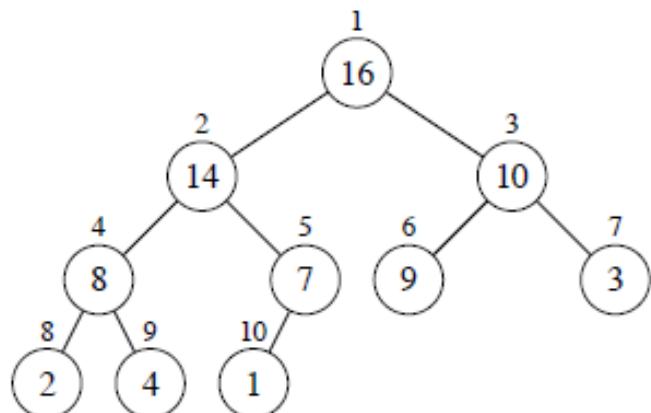
- Altezza di un nodo = # di archi nel cammino semplice più lungo dal nodo ad una foglia
- Altezza di heap = altezza della radice =  $\Theta(\lg n)$



- Heap **può** essere memorizzato in un array  $A$

- Radice dell'albero:  $A[1]$
- Padre di  $A[i] \rightarrow A[\lfloor i/2 \rfloor]$
- Figlio sx di  $A[i] \rightarrow A[2i]$
- Figlio dx di  $A[i] \rightarrow A[2i + 1]$

- Due tipi: **max-heap** e **min-heap**



## Proprietà heap

- **max-heap:**  
 $\forall$  nodo  $i$ , eccetto la radice  
 $A[\text{PARENT}(i)] \geq A[i]$
- **min-heap:**  
 $\forall$  nodo  $i$ , eccetto la radice  
 $A[\text{PARENT}(i)] \leq A[i]$
- Per induzione e transitività di  $\geq$   
 proprietà max-heap garantisce che  
 il **massimo** di un max-heap è la **radice**
- Analogamente per min-heap
- HEAPSORT usa max-heap

## MAX-HEAPIFY conserva proprietà max-heap

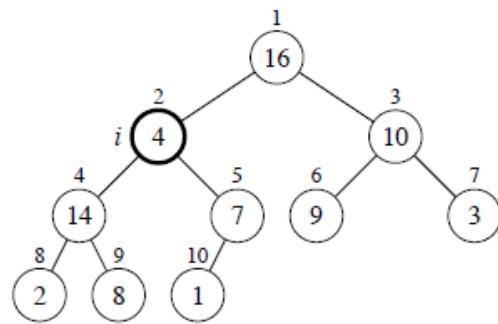
- Prima di MAX-HEAPIFY,  $A[i]$  può essere più piccolo dei suoi figli
- I sottoalberi sx e dx di  $i$  sono max-heap
- Dopo MAX-HEAPIFY, il sottoalbero con radice  $i$  è un max-heap

MAX-HEAPIFY( $A, i$ )

```

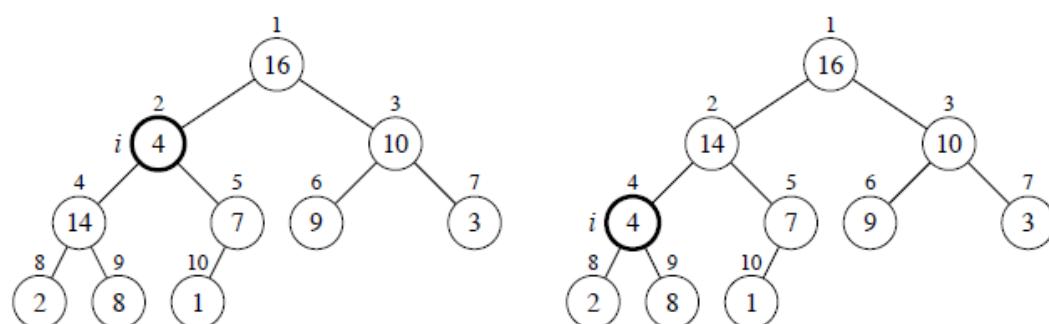
 $l \leftarrow \text{LEFT}(i)$ 
 $r \leftarrow \text{RIGHT}(i)$ 
if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
     $\text{massimo} \leftarrow l$ 
else  $\text{massimo} \leftarrow i$ 
if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{massimo}]$ 
     $\text{massimo} \leftarrow r$ 
if  $\text{massimo} \neq i$ 
    scambia  $A[i]$  con  $A[\text{massimo}]$ 
    MAX-HEAPIFY( $A, \text{massimo}$ )
  
```

## Esempio



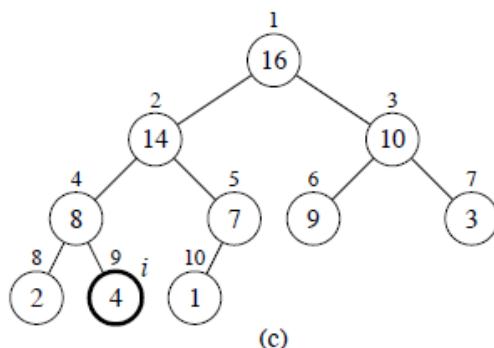
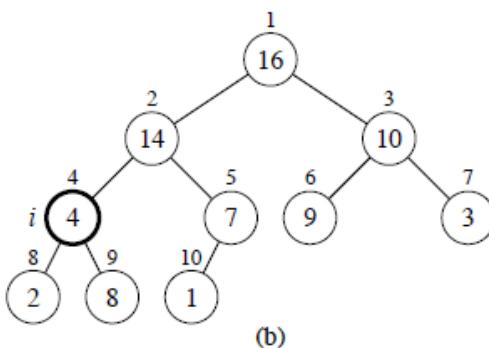
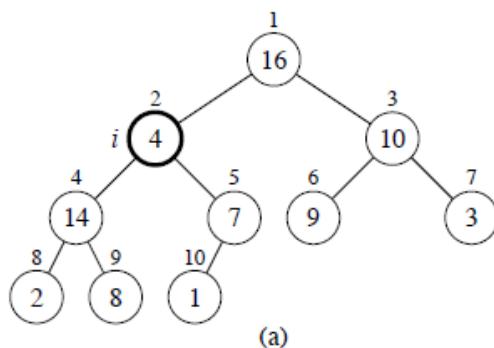
- Nodo 2 viola inizialmente la proprietà max-heap

## Esempio



- Nodo 2 viola inizialmente la proprietà max-heap

## Esempio



- Nodo 2 viola inizialmente la proprietà max-heap

## Analisi

### Tempo

- $O(\lg n)$
- libro: analisi formale con la ricorrenza
- intuitivamente
  - ▶ Heap è un albero binario quasi completo
  - ▶  $O(\lg n)$  livelli; ad ogni livello
    - ★ confronto 3 elementi e al max ne scambio 2

### Correttezza

- Con **invariante di ciclo**
- Non lo vediamo...

## Costruire un heap

Dato array non ordinato produce max-heap bottom-up

BUILD-MAX-HEAP( $A$ )

```
 $A.heap-size \leftarrow A.length$ 
for  $i \leftarrow \lfloor A.length/2 \rfloor$  downto 1
    MAX-HEAPIFY( $A, i$ )
```

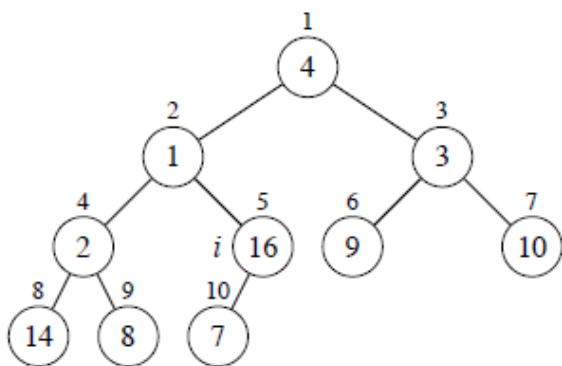
### Nota

da  $\lfloor A.length/2 \rfloor$  in poi sono foglie, quindi max-heap!

## Esempio

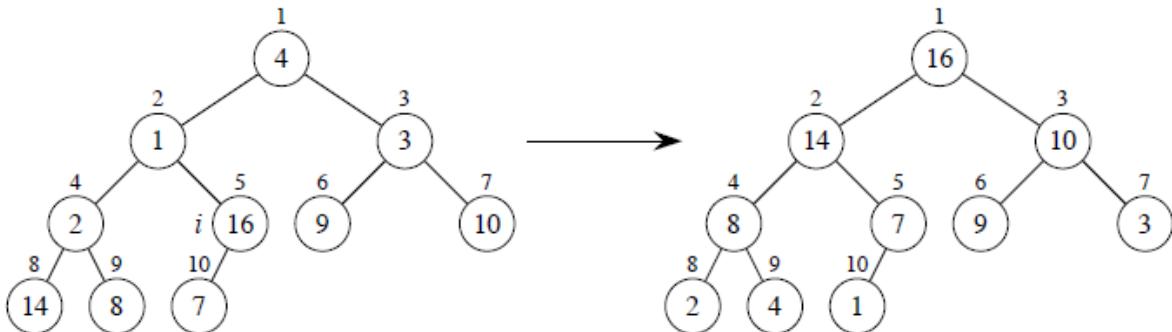
Costruire un max-heap da:

$A$	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7



- $i$  inizia da 5
- MAX-HEAPIFY applicato a sottoalberi con radici 16, 2, 3, 1, 4

## Esempio



Implementazione di coda con priorità tramite heap

## Code con priorità

- Gestiscono un **insieme dinamico  $S$**
- Ogni elemento nell'insieme ha una **chiave**
- Operazioni con Max-priority:
  - ▶  $\text{INSERT}(S, x)$ : inserisce l'elemento  $x$  in  $S$
  - ▶  $\text{MAXIMUM}(S)$ : restituisce l'elemento di  $S$  con massima chiave
  - ▶  $\text{EXTRACT-MAX}(S)$ : restituisce e rimuove l'elemento di  $S$  con massima chiave
  - ▶  $\text{INCREASE-KEY}(S, x, k)$ : incrementa il valore dell'elemento  $x$  a  $k$  ( $k \geq$  valore corrente della chiave di  $x$ )

## Esempio di applicazione di max-priority

- Schedulazione job

# Implementazione di coda con priorità tramite heap

## max-priority queue implementata con max-heap

- con Heap **compromesso** tra
  - ▶ **coda**: rapido inserimento, lenta estrazione  
inserisco subito, ma la devo scorrere (o ordinare) per trovare max
  - ▶ **coda ordinata** lento inserimento, rapida estrazione  
la devo scorrere per inserire, ma prendo subito l'elemento buono
- Con heap **entrambe** le operazioni prendono tempo  $O(\lg n)$
- Min-priority queue implementata con min-heap

## Trovare il massimo

E' semplice: è la radice

`HEAP-MAXIMUM( $A$ )`

`return  $A[1]$`

## Tempo

$\Theta(1)$

## Estrarre il massimo

HEAP-EXTRACT-MAX( $A$ )

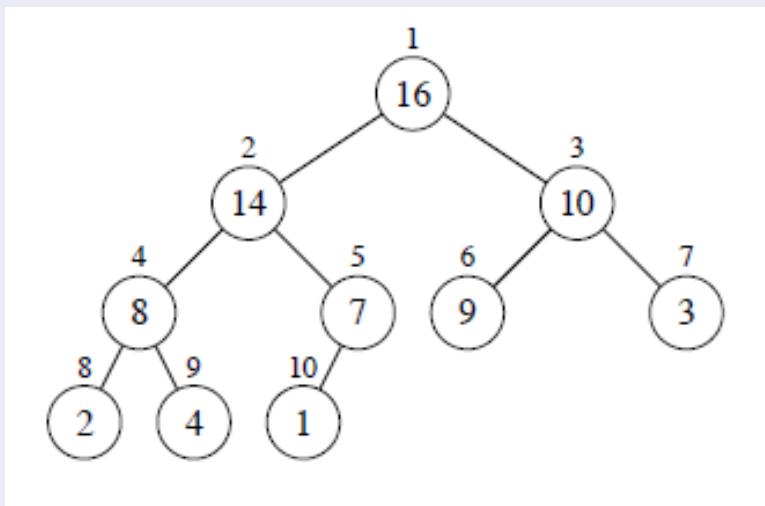
```

if  $A.\text{heap-size} < 1$ 
    error "underflow dell'heap"
 $\max \leftarrow A[1]$ 
 $A[1] \leftarrow A[A.\text{heap-size}]$ 
 $A.\text{heap-size} \leftarrow A.\text{heap-size} - 1$ 
MAX-HEAPIFY( $A, 1$ )
return  $\max$ 
```

### Analisi

- Assegnamenti di tempo costante più tempo per MAX-HEAPIFY
- Tempo:**  $O(\lg n)$

### Esempio: eseguire HEAP-EXTRACT-MAX



- Prende 16 da nodo 1
- Muove 1 dal nodo 10 al nodo 1
- Cancello nodo 10
- MAX-HEAPIFY dalla radice per preservare le proprietà max-heap

# Incrementare valore chiave

## Input

Insieme  $S$ , elemento  $i$ , nuovo valore per  $key$

HEAP-INCREASE-KEY( $A, i, key$ )

```

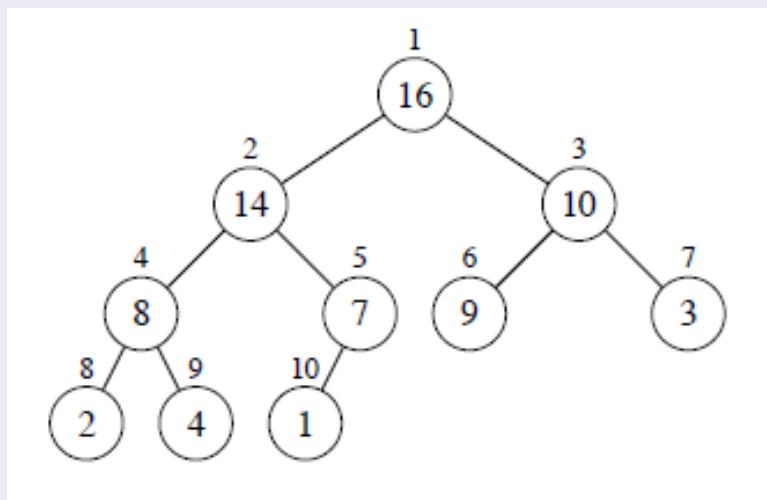
if  $key < A[i]$ 
    error "nuova chiave piu' piccola della corrente"
 $A[i] \leftarrow key$ 
while  $i > 1$  and  $A[Parent(i)] < A[i]$ 
    scambia  $A[i] \leftrightarrow A[Parent(i)]$ 
     $i \leftarrow Parent(i)$ 

```

## Analisi

- Cammino verso l'alto da nodo  $i$  è lungo  $O(\lg n)$
- **Tempo:**  $O(\lg n)$

## Esempio: incrementare key di 9 a 15



- Scambia le chiavi dei nodi 4 e 9
- poi dei nodi 2 e 4

# Inserimento in heap

MAX-HEAP-INSERT( $A$ ,  $key$ )

$A.heap-size \leftarrow A.heap-size + 1$

$A[A.heap-size] \leftarrow -\infty$

HEAP-INCREASE-KEY( $A$ ,  $A.heap-size$ ,  $key$ )

- La sentinella serve per non far fallire il primo if in HEAP-INCREASE-KEY

## Analisi

- Assegnamenti con tempo costante + tempo per HEAP-INCREASE-KEY
- **Tempo:**  $O(\lg n)$

# Alberi di connessione minimi

# Alberi di connessione minimi

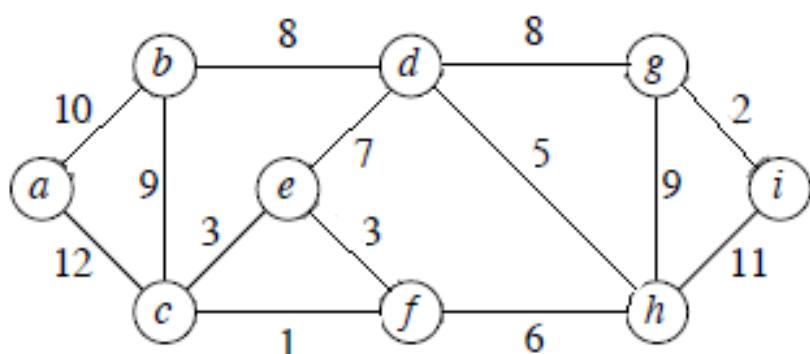
## Una città ha un insieme di case e un insieme di strade

- La strada  $(u, v)$  collega la casa  $u$  con la casa  $v$
- $w(u, v)$  costo per passare una linea ADSL su strada  $(u, v)$
- **Obiettivo:** collegare tutte le case a Internet
  - ① da ogni casa raggiungo ogni casa (ognuno è **connesso**) e
  - ② costo totale di collegamento è **minimo**

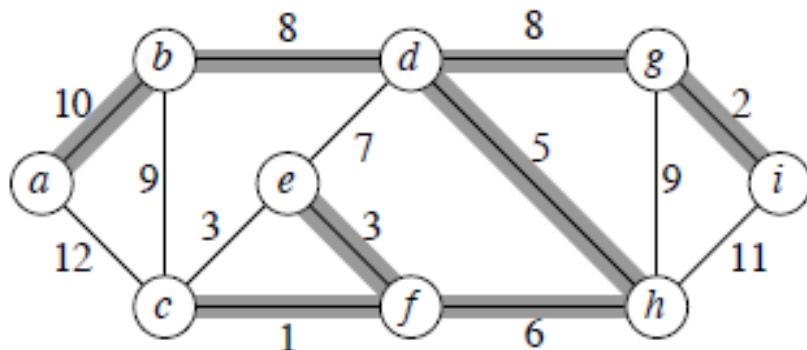
## Modellato con un grafo non diretto $G = (V, E)$

- **Peso**  $w(u, v) \quad \forall (u, v) \in E$
- Trovare  $T \subseteq E$  t.c.
  - ①  $T$  connette tutti i vertici (albero di connessione)
  - ②  $w(T) = \sum_{(u,v) \in T} w(u, v)$  (minimo)
- Minimum Spanning Tree: MST

## Esempio di grafo



# Esempio di grafo (archi in MST sono ombreggiati)



## Proprietà di MST

- Quanti archi?  $|V| - 1$
- Non ha cicli (è un albero ...)
- Può non essere unico
  - ▶ nell'esempio se  $(e, f) \leftrightarrow (c, e)$   
altro MST con **stesso** peso

## Costruire la soluzione

- $A$  è un insieme di archi
- Inizialmente  $A = \emptyset$
- Aggiungo archi ad  $A$  conservando un **invariante di ciclo**:  $A$  è un sottoinsieme di qualche MST
  - ▶ Se  $A$  è un sottoinsieme di qualche MST  
 $(u, v)$  è **sicuro** per  $A$  sse  
 $A \cup \{(u, v)\}$  è un sottoinsieme di qualche MST
  - ▶ Aggiungo solo archi sicuri

## Algoritmo generico MST

GENERIC-MST( $G, w$ )

- 1  $A \leftarrow \emptyset$
- 2 **while**  $A$  non forma un albero di connessione
- 3     trova un arco  $(u, v)$  che e' **sicuro** per  $A$
- 4      $A \leftarrow A \cup \{(u, v)\}$
- 5 **return**  $A$

### Correttezza con invarianti di ciclo

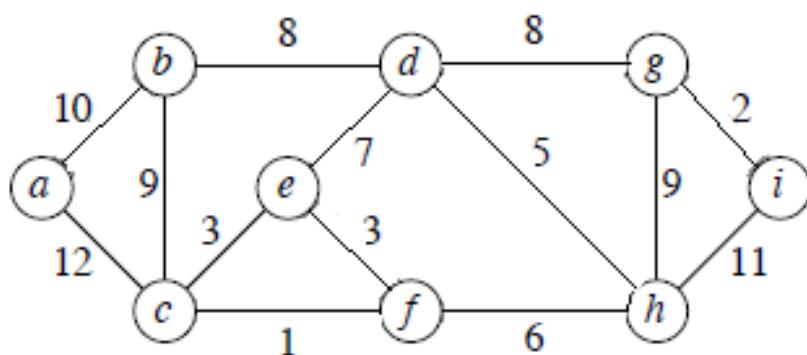
- **Inizializzazione:** Insieme vuoto è sottoinsieme di MST
- **Conservazione:** Aggiungo solo archi sicuri  $\Rightarrow$   
 $A$  rimane un sottoinsieme di qualche MST
- **Terminazione:** Tutti gli archi aggiunti ad  $A$  sono in un MST  
 $\Rightarrow A$  è un albero di copertura che è anche un MST

### Trovare un arco sicuro

- Esempio: arco  $(c, f)$  ha il peso minore  
 E' sicuro per  $A = \emptyset$ ?

**Intuitivamente:**

- ▶  $S \subset V$  un qualunque insieme di vertici t.c.  $c \in S$  e  $f \notin S$  ( $f \in V - S$ )
- ▶ Ogni MST deve avere almeno un arco che connette  $S$  con  $V - S$
- ▶ Scegliamo l'arco con peso minimo (in questo caso  $(c, f)$ )



## Definizioni

$G = (V, E)$  un grafo non orientato

$S \subset V$  e  $A \subseteq E$

- **taglio**  $(S, V - S)$ :  
partizione di  $V$  in insiemi disgiunti  $S$  e  $V - S$
- Un arco  $(u, v) \in E$  **attraversa** il taglio  $(S, V - S)$  se  $u \in S$  e  $v \in V - S$  (o viceversa)
- Un taglio **rispetta** un insieme di archi  $A$  sse nessun arco in  $A$  attraversa il taglio
- Un arco è **leggero** per un taglio se ha peso **minimo** tra tutti gli archi che attraversano il taglio

## Teorema

Sia  $A$  un sottoinsieme di qualche MST,

$(S, V - S)$  sia un taglio che rispetta  $A$  e

$(u, v)$  sia un arco leggero che attraversa  $(S, V - S)$

**Allora**  $(u, v)$  è sicuro per  $A$

## Dimostrazione 1/3

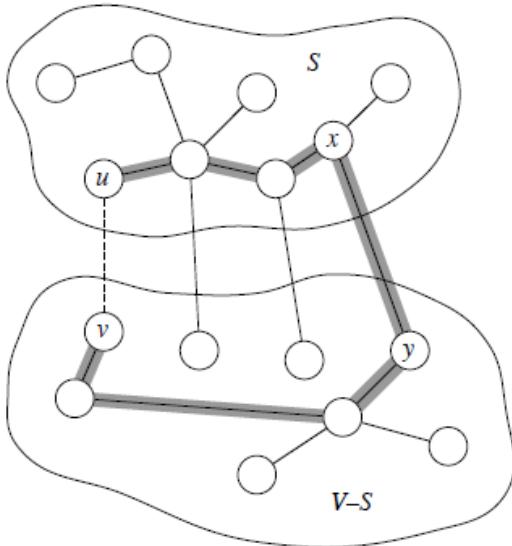
Sia  $T$  un MST che contiene  $A$

- Se  $T$  contiene  $(u, v)$ : **OK**  $(u, v)$  è sicuro per  $A$
- altrimenti costruisco un altro MST  $T'$  che include  $A \cup \{(u, v)\}$

▶ ...

## Dimostrazione 2/3

- altrimenti costruisco un altro MST  $T'$  che include  $A \cup \{(u, v)\}$ 
  - ▶  $T$  è MST  $\Rightarrow$  ha un unico cammino  $p$  tra  $u$  e  $v$
  - ▶  $p$  attraversa il taglio  $(S, V - S)$  almeno una volta in  $(x, y)$
  - ▶  $(u, v)$  è leggero  $\Rightarrow w(u, v) \leq w(x, y)$
  - ▶ ...



In figura

- Archi (tranne  $(u, v)$ ) sono in  $T$
- $A$  è qualche sottoinsieme di archi di  $T$
- Archi ombreggiati: cammino  $p$

## Dimostrazione 3/3

- altrimenti costruisco un altro MST  $T'$  che include  $A \cup \{(u, v)\}$ 
  - ▶ ...
  - ▶ Il taglio rispetta  $A \Rightarrow (x, y) \notin A$
  - ▶ Costruisco  $T' = T - \{(x, y)\} \cup \{(u, v)\}$
  - ▶  $w(T') = w(T) - w(x, y) + w(u, v)$
  - ▶  $\leq w(T)$  perché  $w(u, v) \leq w(x, y)$
  - ▶  $T'$  è un albero di copertura,  $w(T') \leq w(T)$ , e  $T$  è un MST
  - ▶  $\Rightarrow T'$  è MST
  - ▶ Quindi  $(u, v)$  è **sicuro** per  $A$  :
    - ★  $A \subseteq T$  e  $(x, y) \notin A \Rightarrow A \subseteq T'$
    - ★  $A \cup \{(u, v)\} \subseteq T'$
    - ★  $T'$  è un MST  $\Rightarrow (u, v)$  è sicuro per  $A$

## GENERIC-MST( $G, w$ )

```

1  $A \leftarrow \emptyset$ 
2 while  $A$  non forma un albero di connessione
3     trova un arco  $(u, v)$  che e' sicuro per  $A$ 
4      $A \leftarrow A \cup \{(u, v)\}$ 
5 return  $A$ 
```

### In GENERIC-MST:

- $G_A = (V, A)$  è una foresta che contiene componenti connesse
  - ▶ Inizialmente ogni componente è un singolo vertice
- Ogni arco sicuro fonde due componenti
- MST ha  $|V| - 1$  archi
  - ⇒ *for* itera  $|V| - 1$  volte
- Alla fine  $|V| - 1$  archi sicuri ⇒ una sola componente

### Corollario identifico $S$

Se  $C = (V_C, E_C)$  è una componente连通的 nella foresta  $G_A = (V, A)$  e  $(u, v)$  è un arco leggero che connette  $C$  ad un'altra componente in  $G_A$  (i.e.  $(u, v)$  è un arco leggero che attraversa il taglio  $(V_C, V - V_C)$ )  
**allora**  $(u, v)$  è sicuro per  $A$

### Dimostrazione

Porre  $S = V_C$  nel teorema

Questo porta all'algoritmo di **Kruskal**

## Strutture dati per insiemi disgiunti

- **union-find**
- Gestisce una collezione  $S = \{S_1, \dots, S_k\}$  di insiemi disgiunti dinamici
- Ogni insieme è identificato da un rappresentante (un elemento dell'insieme)
  - ▶ Non importa quale, ma sempre lo stesso

## Operazioni

- **MAKE-SET( $x$ )**: crea un nuovo insieme  $S_i = \{x\}$  e aggiunge  $S_i$  a  $S$
- **UNION( $x, y$ )**: se  $x \in S_x$ ,  $y \in S_y$ , allora  
 $S \leftarrow S - S_x - S_y \cup \{S_x \cup S_y\}$ 
  - ▶ distrugge  $S_x$  e  $S_y$  (insiemi devono essere disgiunti)
  - ▶ il rappresentante di  $S$  è qualunque elemento di  $S_x \cup S_y$
- **FIND-SET( $x$ )**: ritorna il rappresentante dell'insieme che contiene  $x$

**MST-KRUSKAL( $G, w$ )**

```

 $A \leftarrow \emptyset$ 
for ogni vertice  $v \in G.V$ 
  MAKE-SET( $v$ )
ordina gli archi di  $G.E$  in senso non decrescente rispetto al peso  $w$ 
for ogni arco  $(u, v) \in G.E$ , preso in ordine di peso non decrescente
  if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
     $A \leftarrow A \cup \{(u, v)\}$ 
    UNION( $u, v$ )
return  $A$ 

```

- Potrei aggiungere un test per fermarsi dopo  $|V| - 1$  UNION

## Analisi

Initializzare  $A$  :  $O(1)$   
 Primo ciclo for :  $|V|$  volte MAKE-SET  
 Ordinare  $E$  :  $O(E \lg E)$   
 Secondo ciclo for :  $O(E)$  volte (FIND-SET e UNION)

- Dipende da implementazione union-find!

## Richiami

### Algoritmo generico

GENERIC-MST( $G, w$ )

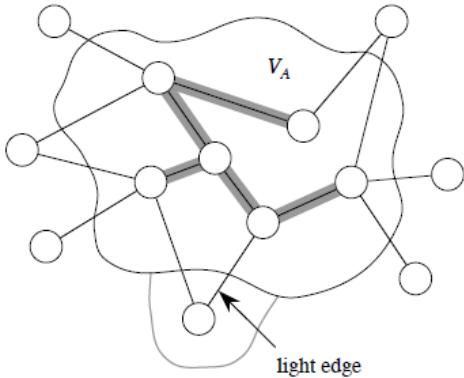
- 1  $A \leftarrow \emptyset$
- 2 **while**  $A$  non forma un albero di connessione
- 3     trova un arco  $(u, v)$  che è **sicuro** per  $A$
- 4      $A \leftarrow A \cup \{(u, v)\}$
- 5 **return**  $A$

### Teorema

Sia  $A$  un sottoinsieme di qualche MST,  
 $(S, V - S)$  sia un taglio che rispetta  $A$  e  
 $(u, v)$  sia un arco leggero che attraversa  $(S, V - S)$   
**Allora**  $(u, v)$  è sicuro per  $A$

## Algoritmo di Prim

- Caso particolare di GENERIC-MST
- $A$  è sempre un albero
- Inizia da “radice” arbitraria  $r$
- Ad ogni passo, trova un **arco leggero** che attraversa il **taglio**  $(V_A, V - V_A)$ , dove  $V_A = \{ \text{vertici toccati da archi di } A \}$
- Lo aggiunge ad  $A$



Archi di  $A$  ombreggiati

$\text{MST-PRIM}(G, w, r)$

```

 $Q \leftarrow \emptyset$ 
for ogni  $u \in G.V$ 
     $u.\text{key} \leftarrow \infty$ 
     $u.\pi \leftarrow \text{NIL}$ 
    INSERT( $Q, u$ )
DECREASE-KEY( $Q, r, 0$ ) //  $r.\text{key} \leftarrow 0$ 
while  $Q \neq \emptyset$ 
     $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
    for ogni  $v \in G.\text{Adj}[u]$ 
        if  $v \in Q$  and  $w(u, v) < v.\text{key}$ 
             $v.\pi \leftarrow u$ 
            DECREASE-KEY( $Q, v, w(u, v)$ ) //  $v.\text{key} \leftarrow w(u, v)$ 

```

## Come trovare arco leggero rapidamente?

Nella **coda a priorità**  $Q$ :

- Ogni elemento è un vertice in  $V - V_A$  (ancora da aggiungere a  $V_A$ )
- $v.key$  è il peso minimo di ogni arco  $(u, v)$  con  $u \in V_A$
- EXTRACT-MIN restituisce  $v$  t.c. esiste  $u \in V_A$  e  $(u, v)$  è un arco leggero che attraversa  $(V_A, V - V_A)$
- $v.key = \infty$  se  $v$  non è adiacente a nessun vertice in  $V_A$

Gli archi di  $A$  formano un albero con radice  $r$ :

- Per ogni vertice  $v.\pi = \text{padre di } v$   
 $v.\pi = NIL$  se  $v = r$

## Invariante di ciclo

- Prima di ogni iterazione del while:
  - ①  $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$
  - ② I vertici già inseriti in MST sono quelli in  $V - Q$
  - ③ Per ogni vertice  $v \in Q$   
 se  $v.\pi \neq NIL$  allora  $v.key (< \infty)$  è il peso di un arco **leggero**  $(v, v.\pi)$  che collega  $v$  a qualche vertice che si trova già in  $A$
- Alla fine,  $V_A = V$   
 $\Rightarrow Q = \emptyset$   
 e MST è  $A = \{(v, v.\pi) : v \in V - \{r\}\}$

## Code con priorità (richiamo...)

- Gestiscono insieme dinamico  $S$
- Ogni elemento nell'insieme ha una **chiave**
- Operazioni con Min-priority:
  - ▶  $\text{INSERT}(S, x)$ : inserisce l'elemento  $x$  in  $S$
  - ▶  $\text{MINIMUM}(S)$ : restituisce l'elemento di  $S$  con chiave minima
  - ▶  $\text{EXTRACT-MIN}(S)$ : rimuove e restituisce l'elemento di  $S$  con chiave minima
  - ▶  $\text{DECREASE-KEY}(S, x, k)$ : decrementa il valore dell'elemento  $x$  a  $k$   
( $k \leq$  valore corrente della chiave di  $x$ )

## Analisi

Dipende da come viene impletata la coda a priorità :

- Se  $Q$  è un heap binario
  - ▶  $\text{INSERT } O(\lg n)$
  - ▶  $\text{EXTRACT-MIN } O(\lg n)$
  - ▶  $\text{DECREASE-KEY } O(\lg n)$
- **Analisi:**

Inizializzare $Q$ e primo ciclo for:	$O(V \lg V)$
Decrementare chiave di $r$ :	$O(\lg V)$
Ciclo while :	$ V $ chiamate a EXTRACT-MIN $\Rightarrow O(V \lg V)$ $\leq  E $ chiamate a DECREASE-KEY $\Rightarrow O(E \lg V)$
Totale:	$O(E \lg V)$
- PRIM è asintoticamente uguale a KRUSKAL:  $O(E \lg V)$

# Cammini minimi da sorgente unica

## Cammini minimi da sorgente unica

Trova la **strada più corta** tra due nodi

- **Input:**

- ▶ Grafo diretto  $G = (V, E)$
- ▶ Funzione peso  $w : E \rightarrow \mathbb{R}$

- **Peso** del cammino  $p = < v_0, v_1, \dots, v_k >$ :

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

= somma dei pesi degli archi nel cammino  $p$

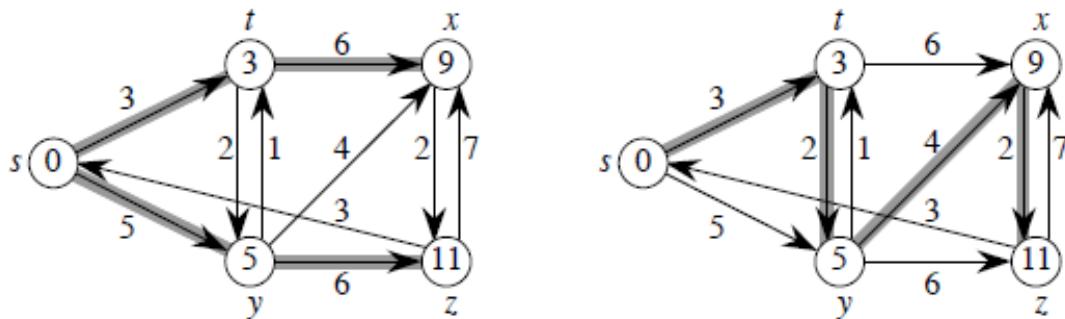
- Peso del cammino più corto (**shortest-path**) da  $u$  a  $v$ :

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{se esiste un cammino } u \rightsquigarrow v \\ \infty & \text{altrimenti} \end{cases}$$

- Il **cammino più corto** da  $u$  a  $v$  è ogni cammino t.c.  $w(p) = \delta(u, v)$

## Esempio

Cammini più corti da  $s$  (valori di  $\delta$  nei vertici. Archi ombreggiati indicano cammini minimi.)



- Lo shortest path **può essere non unico**
- I cammini minimi sono organizzati come un albero
- **Pesi**: qualunque misura che:
  - ▶ si accumula linearmente lungo il cammino,
  - ▶ deve essere minimizzata
- **Esempi**: tempo, costo

## Generalizzazione di ricerca in ampiezza a grafi pesati

### Varianti

- **Singola-sorgente**: Trova cammini più corti da un vertice sorgente  $s \in V$  ad ogni vertice  $v \in V$
- **Singola-destinazione**: Trova cammini più corti verso un dato vertice destinazione
- **Singola-coppia**: Trova il cammino più corto da  $u$  a  $v$ . di risolvere single-source
- **All-pairs**: Trova lo shortest path da  $u$  a  $v$  per tutte le coppie  $u, v \in V$

## Archi con pesi negativi

- OK, se **nessun ciclo con peso totale negativo è raggiungibile** dalla sorgente
  - ▶ Con un ciclo con peso negativo ci si muove dentro e:  $w(s, v) = -\infty$  per tutti i  $v$  nel ciclo
- OK se il ciclo con pesi negativi non è raggiungibile dalla sorgente o se archi con pesi negativi non generano cicli con peso negativo
- Alcuni algoritmi funzionano solo se non ci sono archi con pesi negativi nel grafo

## Sottostruttura ottima

### Sottostruttura ottima Shortest-Path (SP)

- **Lemma:** Ogni sotto-cammino di uno SP è uno SP
- **Proof:** Cut-and-paste
- **Supponiamo** che cammino  $p$  ( $u \xrightarrow{p_{ux}} x \xrightarrow{p_{xy}} y \xrightarrow{p_{yv}} v$ ) sia SP da  $u$  a  $v$ :  
 $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$
- **Supponiamo** che esista uno SP  $x \xrightarrow{p'_{xy}} y$  con  $w(p'_{xy}) < w(p_{xy})$
- **Costruiamo**  $p'$ :  $u \xrightarrow{p_{ux}} x \xrightarrow{p'_{xy}} y \xrightarrow{p_{yv}} v$
- Allora
 
$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) \end{aligned}$$
- Quindi  $p$  non era SP! (**CVD lemma**)

## Shortest path *non possono contenere cicli*

- Già escluso cicli con **pesi negativi**
- **Pesi positivi**: si può avere un cammino più corto saltando il ciclo

## Output di algoritmi per cammino minimo a singola sorgente

Per ogni vertice  $v \in V$ :

- $v.d = \delta(s, v)$ 
  - ▶ Inizialmente,  $v.d = \infty$
  - ▶ Diminuisce mentre gli algoritmi avanzano  
conserva sempre  $v.d \geq \delta(s, v)$
  - ▶  $v.d$  è una **stima di shortest-path**
- $v.\pi =$  predecessore di  $v$  in uno SP da  $s$
- Se non ha predecessore:  $v.\pi = NIL$ 
  - ▶  $\pi$  induce un albero: shortest-path tree

## Inizializzazione

Tutti gli algoritmi shortest-path iniziano con INIT-SINGLE-SOURCE

INIT-SINGLE-SOURCE( $V, s$ )

```
for ogni vertice  $v \in G.V$ 
     $v.d \leftarrow \infty$ 
     $v.\pi \leftarrow NIL$ 
     $s.d \leftarrow 0$ 
```

## Rilassamento

- Migliora la stima SP per  $v$  passando da  $u$  prendendo  $(u, v)$ ?
- **Rilassamento** arco  $(u, v)$ :
  - ▶ **verificare se**, passando per  $u$  è possibile **migliorare la stima** di cammino minimo per  $v$  precedentemente trovata
  - ▶ **in caso positivo modificare**  $v.d$  e  $v.\pi$ .

$\text{RELAX}(u, v, w)$

```

if  $v.d > u.d + w(u, v)$ 
   $v.d \leftarrow u.d + w(u, v)$ 
   $v.\pi \leftarrow u$ 

```



## Algoritmi single-source shortest-paths

(che vedremo)

- chiamare INIT-SINGLE-SOURCE
- **rilassare archi**
- Algoritmi **differiscono** per l'ordine e quante volte viene rilassato ciascun arco

BELLMAN-FORD( $G, w, s$ )

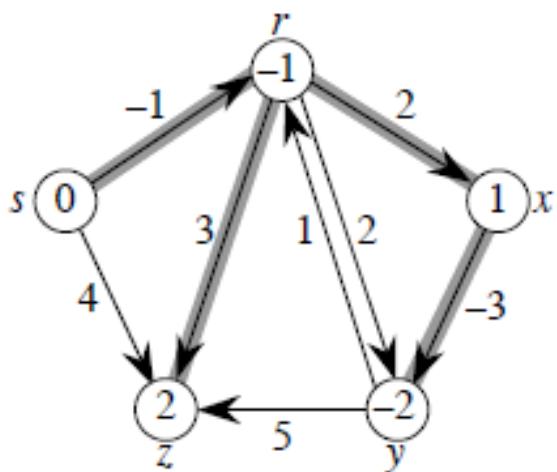
```

INIT-SINGLE-SOURCE( $G.V, s$ )
for  $i \leftarrow 1$  to  $|G.V| - 1$ 
    for ogni arco  $(u, v) \in G.E$ 
        Relax( $u, v, w$ )
    for ogni arco  $(u, v) \in G.E$ 
        if  $v.d > u.d + w(u, v)$ 
            return FALSE
return TRUE

```

- Il primo ciclo for rilassa tutti gli archi  $|V| - 1$  volte
- Il resto controlla che non ci siano cicli negativi (TRUE o FAISE)
  - ▶ possono esserci archi negativi
- Quanto velocemente converge dipendono dall'ordine del rilassamento
- Convergenza garantita dopo  $|V| - 1$  passi in assenza di cicli negativi
- **Tempo:**  $\Theta(V \cdot E)$

Esempio:



# Algoritmo di Dijkstra

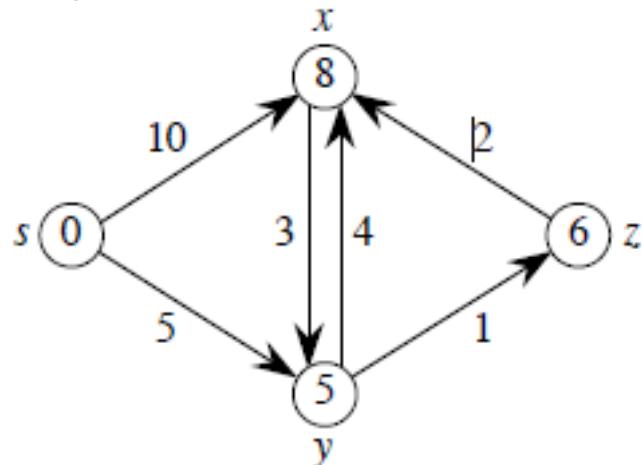
- **Non** si devono avere **archi con peso negativo**
- Una **versione pesata** di **ricerca in ampiezza**.
  - ▶ Invece di una coda FIFO usa una **coda con priorità**
  - ▶ Le **chiavi sono i pesi shortest-path** ( $v.d$ )
- Si hanno **due insiemi di vertici**:
  - ▶  $S$  = vertici dei quali è stato determinato il peso finale shortest-path
  - ▶  $Q$  = coda con priorità =  $V - S$

**DIJKSTRA**( $G, w, s$ )

```

INIT-SINGLE-SOURCE( $G.V, s$ )
 $S \leftarrow \emptyset$ 
 $Q \leftarrow G.V$ 
while  $Q \neq \emptyset$ 
     $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
     $S \leftarrow S \cup \{u\}$ 
    for ogni nodo  $v \in u.Adj$ 
        RELAX( $u, v, w$ )
  
```

- In questo caso RELAX( $u, v, w$ ) aggiorna anche la coda (DECREASE-KEY)
- **Costo:** Come Prim dipende da implementazione coda di priorità
  - ▶ Se heap binario, ciascuna operazione prende tempo  $O(\lg V) \Rightarrow O(E \lg V)$  (analisi aggregata)
  - ▶ Se heap di Fibonacci:  $O(V \lg V + E)$

**Esempio:**Ordine di aggiunta a  $S$ :  $s, y, z, x$