CheatSheet: Algoritmi e Strutture Dati di Alberto Montresor

Simone Alghisi, Emanuele Beozzo, Samuele Bortolotti

Dicembre 25, 2019

Contents

Algorithms

1 Lo Pseudocodice

Per descrivere gli algoritmi, ovvero delle procedure composte da operazioni elementari e che devono risolvere un problema computazionale in un tempo finito, dobbiamo ricorrere ad un linguaggio pressoché formale, in quanto il linguaggio naturale può generare delle ambiguità.

1.1 Notazione in pseudocodice

- $a = b$: assegnamento del valore di b alla variabile a;
- $a \leftrightarrow b$: scambio di valori all'interno delle variabili;
- $T[]A = newT[1..n]$: creazione di un array del tipo dato T;
- $T[][]A = new\ T[1..n][1..n]$: creazione di una matrice di tipo dato T;
- **int, float, boolean, char**: tipi di dato che utilizzeremo nella nostra pseudocodifica;
- $==, \neq, \geq, \leq, >, <$: operatori di confronto;
- $+, -, \cdot, /, \lfloor x \rfloor, \lceil x \rceil, \log, x^2$: classici operatori matematici;
- *iif(condizione,* v_1*,* v_2*)*: operatore ternario;
- **if** *condizione* **then** istruzione: operatore di selezione;
- **if** *condizione* **then** $istruzione_1$ **else** $istruzione_2$
- **while** *condizione* **do** istruzione: ciclo while;
- **foreach** *elemento* $\in$ *insieme* **do** istruzione
- **for** $i = 1$ **to** n **do** istruzione: ciclo for;
- **for** $i = n$ **downto** 1 **do** istruzione: ciclo for al contrario;
- **return**
- *% commento*
- $CLASS$ *variabile* = new $CLASS$: istanziazione di una classe;
- *r.campo* = 10: accesso ad un campo di una struttura;
- **delete** variabile: per la cancellazione;
- $r =$ **nil**: assegnazione ad una variabile di tipo puntatore il valore di nil

2 Sottovettore di Somma Massima: Algoritmo di Kadane

Problema: Dato un vettore di interi A[1...n], restituire il sottovettore A[i...j] (con i, j $<$ n e i $\leq$ j) di somma massimale, ovvero la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.
Soluzione (con Programmazione Dinamica): **Algoritmo di Kadane**

Algorithm 1 Kadane Algorithm

```
1 int maxSumKadane(int[] A, int n)
2         int maxSoFar = 0
3         int maxHere = 0
4         for i = 1 to n do
5                 maxHere = max(maxHere + A[i], 0)
6                 maxSoFar = max(maxSoFar, maxHere)
7         return maxSoFar
```

3 Valutazione Algoritmi

3.1 Calcolo del Minimo

Problema: Dato un vettore di interi S[1...n] trovare il minimo
Soluzione:

Algorithm 2 Ricerca Minimo

```
1 int min(int[] S, int n)
2         int min = S[1]
3         for i=2 to n do
4                 if S[i] < min then
5                         min = S[i]
6         return min
```

Nella versione seguente viene ritornata una coppia ¡int, int¿ che contiene minimo e la sua posizione:

Algorithm 3 Ricerca Minimo ritornando gli indici

```
1 (int, int) min(int[] S, int n)
2         int min = S[1]
3         int posMin = 1;
4         for i=2 to n do
5                 if S[i] < min then
6                         min = S[i]
7                         minPos = i
8         return (min, minPos)
```

3.2 Ricerca Binaria

Problema: Dato un vettore ordinato di interi S[1...n] e un intero v, controllare se nel vettore è presente un elemento A[i] = v (con $0 \leq i \leq n$) e in quel caso restituirne l'indice.
Soluzione (Ricerca Binaria): Analizzo l'elemento centrale del sottovettore (indice m)

- S[m]=v allora ho trovato il valore cercato;
- S[m]<v allora vuol dire che dovrò cercare il valore nella metà di destra;
- S[m]>v allora vuol dire che dovrò cercare il valore nella metà di sinistra

Algorithm 4 Ricerca Binaria

```
1  int binarySearch(int[] S, int v, int start, int end)
2          if start < end then
3                  return 0
4          else
5                  int m = ⌊(start+end)/2⌋
6                  if S[m] == v then
7                          return m
8                  else if S[m] < v then
9                          return binarySearch(S, v, m+1, end)
10                 else
11                         return binarySearch(S, v, start, m−1)
```

4 Analisi degli Algoritmi

4.1 Tempo di Calcolo

- Ogni istruzione richiede un tempo costante per essere eseguita;
- Questa costante è potenzialmente diversa per ogni istruzione;
- Ogni istruzione viene eseguita un certo numero (#) di volte che dipende da codice a codice

Funzione min()

Algorithm 5 Ricerca Minimo + calcolo del costo

```
1  int min(int[] S, int n)                  % Costo # Volte
2          int min = S[1]                   % c1      1
3          for i = 2 to n do                % c2      n
4                  if S[i] < min then       % c3      n−1
5                          min = S[i]       % c4      n−1
6          return min                       % c5      1
```

$T(n) = c1 + c2n + c3(n-1) + c4(n-1) + c5 = (c2 + c3 + c4)n + (c1 + c5 - c3 - c4) = an + b$

4.2 Complessità Algoritmi vs Complessità Problemi: Moltiplicazione di Gauss

Problema: Moltiplicazione numeri complessi

- (a + bi)(c + di) = [ac - bd] + [ad + bc]i
- Input: a, b, c, d
- Output: ac - bd, ad + bc

Se consideriamo un modello di calcolo dove la moltiplicazione costa 1 mentre le addizioni e le sottrazioni costano 0.01 otteniamo che il costo dell'algoritmo risulta essere 2*(2 + 0.01) = 4.02
Usiamo la tecnica della **moltiplicazione di Gauss**:

- A1 = a*c
- A2 = b*d
- m = (a + b)*(c + d)
- A3 = m - A1 - A2 = (a + b)*(c + d) - (a*c) - (b*d) = ~~ac~~ + ad + bc + ~~bd~~ - ~~ac~~ -~~bd~~ = ad + bc
- A4 = A1 - A2 = ac - bd

Notiamo che A3 e A4 sono esattamente i termini che ci aspettiamo vengano restituiti dalla moltiplicazione dei numeri complessi e che, in questo caso, il costo delle operazioni all'interno del nostro sistema risulta essere 3.05.

4.3 Notazione Asintotica e Proprietà

Definzione $\mathcal{O}(g(n))$:
Sia g(n) una funzione di costo; indichiamo con $\mathcal{O}(g(n))$ un insieme di funzioni f(n) tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cg(n) \ , \forall n \geq m$$

Definizione $\Omega(g(n))$:
Sia g(n) una funzione di costo; indichiamo con $\Omega(g(n))$ un insieme di funzioni f(n) tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n) \ , \forall n \geq m$$

Regola Generale:

- **Espressioni Polinomiali**
  $a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0, a_k > 0 \implies f(n) = \Theta(n^k)$
- **Casi Particolari**
  La complessità di $f(n) = 5$ e di $f(n) = 5 + \sin(n)$ è $\Theta(1)$
  (P.S. la stessa cosa vale se vi è una limitazione al numero di dati che è necessario fornire in input)
- ... altri casi trattati nella parte di teoria

4.4 Complessità

$\forall r < s, h < k, a < b$
$\mathcal{O}(1) \subset \mathcal{O}(\log^r n) \subset \mathcal{O}(\log^s n) \subset \mathcal{O}(n^h) \subset \mathcal{O}(n^h \log^r n) \subset \mathcal{O}(n^h \log^s n) \subset \mathcal{O}(n^k) \subset \mathcal{O}(a^n) \subset \mathcal{O}(b^n)$

5 Equazioni di Ricorrenza

5.1 Metodo dell'Albero di Ricorsione, o per Livelli

Srotoliamo la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli di ricorsione.
Primo Esempio

$$T(n) = \begin{cases} 4T(n/2) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

T(n) = n + 4T(n/2) = n + 4(4T(n/4) + n) = n + 4n/2 + 16T($n/2^2$) = n + 2n + 16n/4 + 64T(n/8)
= ...
= n + 2n + 4n + ... + 2^{k-1} + $4^k T(n/2^k)$ se poniamo $2^k = n \implies k = \log n$
= n + 2n + 4n + ... + $2^{\log n - 1} n$ + $4^{\log n} T(1)$ = n $\sum_{j=0}^{\log n - 1} 2^j + 4^{\log n}$
= n $\frac{2^{\log n - 1}}{2-1}$ + $4^{\log n} = n(n-1) + n^2 = 2n^2 - n = \Theta(n^2)$

Secondo Esempio

$$T(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Proviamo a visualizzare l'albero delle chiamate:

Livello	Dim.	Costo chiam.	N. chiamate	Costo livello
0	n	n^3	1	n^3
1	$n/2$	$(n/2)^3$	4	$4(n/2)^3$
2	$(n/4)$	$(n/4)^3$	16	$16(n/4)^3$
...	...	...	...	...
i	$n/2^i$	$(n/2^i)^3$	4^i	$4^i(n/2^i)^3$
...	. ..	...	...	...
$l-1$	$n/2^{l-1}$	$(n/2^{l-1})^3$	4^{l-1}	$4^{l-1}(n/2^{l-1})^3$
$l = \log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Dunque sappiamo che al livello $\log n$ il costo sarà pari a $4^{\log n}$, duqnue ci basta fare la sommatoria fino a $\log n - 1$ e poi aggiungere il restante:
T(n) = $\sum_{i=0}^{\log n - 1} 4^i \cdot n^3/2^{3i} + 4^{\log n}$

possiamo portare fuori i termini costanti dalla sommatoria

$= n^3 \sum_{i=0}^{\log n - 1} \frac{4^i}{2^{3i}} + 4^{\log n} = n^3 \sum_{i=0}^{\log n - 1} \frac{2^{2i}}{2^{3i}} + 2^{2\log n} = n^3 \sum_{i=0}^{\log n - 1} (\frac{1}{2})^i + n^2$

estendiamo la sommatoria fino a ∞

$\leq n^3 \sum_{i=0}^{\infty} (\frac{1}{2})^i + n^2 = n^3 \cdot \frac{1}{1-\frac{1}{2}} + n^2 = 2n^3 + n^2$

Abbiamo dimostrato che T(n) = $\mathcal{O}(n^3)$: non possiamo affermare che sia $\Theta(n) = n^3$ perchè abbiamo dimostrato solo un limite superiore.
Tuttavia basta notare che l'equazione originale possiede un termine del tipo an^3 (con a $\in \mathbb{N}$) $\implies$ T(n) = $\Omega(n^3)$
Visto che abbiamo dimostrato anche il limite inferiore $\implies$ T(n) = $\Theta(n^3)$

5.2 Metodo di Sostituzione, o per Tentativi

Metodo in cui si cerca di indovinare una soluzione, in base alla propria esperienza, e si dimostra la correttezza di questa soluzione tramite induzione

Primo Esempio

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Limite Superiore
Tentativo: T(n) = $\mathcal{O}$(n)
$\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$

Caso Base: Dimostriamo la disequazione per T(1)
$T(1) = 1 \overset{?}{\leq} 1 \cdot c \iff c \geq 1$

Ipotesi Induttiva: $\forall k < n : T(k) \leq ck$
Passo di Induzione: Dimostriamo la disequazione per T(n)
$T(n) = T(\lfloor n/2 \rfloor) + n \leq c(\lfloor n/2 \rfloor) + n \leq cn/2 + n \overset{?}{\leq} cn \iff c/2 + 1 \leq c$
$\iff c \geq 2$
Abbiamo provato che T(n) $\leq$ cn e un valore di c che rispetta entrambe le disequazioni è ad esempio c = 2.
Questo vale per n = 1, e per tutti i valori di n seguenti; quindi m = 1.
Abbiamo provato che T(n) = $\mathcal{O}$(n), ora dobbiamo occuparci del **Limite Inferiore**

Tentativo: T(n) = Ω(n)
$\exists c > 0, \exists m \geq 0 : T(n) \geq cn, \forall n \geq m$

Caso Base: Dimostriamo la disequazione per T(1)
$T(1) = 1 \overset{?}{\geq} 1 \cdot c \iff c \leq 1$

Ipotesi Induttiva: $\forall k < n : T(k) \geq ck$
Passo di Induzione: Dimostriamo la disequazione per T(n)
T(n) = T($\lfloor n/2 \rfloor$) + n $\geq c(\lfloor n/2 \rfloor) + n \geq cn/2$ - 1 + n

sottraggo 1 in modo che sia sicuramente più piccolo del limite inferiore

$cn/2 - 1 + n \overset{?}{\geq} cn \iff c/2 - 1/n + 1 \geq c \iff c \leq 2 - 2/n$

Mettendo assieme le due condizioni si ha che nel caso base (per $n \leq 1$) $c \leq 1$ e nel passo induttivo (per $n > 1$) $c \leq 2 - 2/n$ (quindi nel caso peggiore, cioè per $n = 2$, si ha $c \leq 1$): un valore c che soddisfa entrambe le disequazioni per un $n \geq 1$ è $c = 1$. Questo vale ovviamente per $n = 1$ e quindi per $m = 1$.
Notare che era possibile dimostrare che $T(n) = \Omega(n)$ come fatto nel caso del Metodo per Livelli, cioè notando che nell'equazione fosse presente un termine del tipo an^1 (con $a \in \mathbb{N}$).
Visto che abbiamo dimostrato sia il Limite Inferiore che il Limite Superiore possiamo dire che $T(n) = \Theta(n)$.

Secondo Esempio (Difficoltà Matematica)

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Tentativo: $T(n) = \mathcal{O}(n)$
$\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$

Ipotesi Induttiva: $\forall k < n : T(k) \leq ck$
Passo di Induzione: Dimostriamo la disequazione per $T(n)$
$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \leq c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 1$
$\leq 2c(n/2) + 1 \overset{?}{\leq} cn \iff \cancel{cn} + 1 \leq \cancel{cn} \iff 1 \leq 0$ NO!

Pur non sembrando, il tentativo è corretto, tuttavia non riusciamo a dimostrarlo per via di un termine di ordine inferiore, usiamo un ipotesi induttiva più stretta:
$\exists b > 0, \forall k < n : T(k) \leq ck - b$
$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \leq c(\lfloor n/2 \rfloor) - b + c(\lceil n/2 \rceil) - b + 1$
$\leq 2c(n/2) - 2b + 1 = cn - 2b + 1 \overset{?}{\leq} cn - b \iff b \geq 1$

Caso Base: Dimostriamo la disequazione per $T(1)$
$T(1) = 1 \overset{?}{\leq} 1 \cdot c - b \iff c \geq b + 1$

$\implies$ una coppia di valori che può soddisfare le disequazioni è $b = 1$ e $c = 2$; questo vale da $n = 1$ in poi e quindi per $m = 1$.

Terzo Esempio (Problemi con i Casi Base)

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Tentativo: $T(n) = \mathcal{O}(n \log n)$
$\exists c > 0, \exists m \geq 0 : T(n) \leq c(n \log n), \forall n \geq m$

Ipotesi Induttiva: $\forall k < \text{n} : \text{T(k)} \leq \text{c}(k \log k)$
Passo di Induzione: Dimostriamo la disequazione per T(n)
$\text{T(n)} = 2\text{T}(\lfloor n/2 \rfloor) + \text{n} \leq 2(c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor) + \text{n}$
$\leq 2(c(n/2) \log (n/2)) + \text{n} = 2cn(\log n \text{ - } \log 2) + n$
$= cn(\log n \text{ - } 1) + n \overset{?}{\leq} cn \log n \iff c(\log n - 1) + 1 \leq c \log n$
$\iff \text{c} \geq 1$

Caso Base: Dimostriamo la disequazione per T(1)
$\text{T}(1) = 1 \overset{?}{\leq} c \cdot \log 1 \iff 1 \leq 0$ NO!
...ma, non è un problema: il valore di m lo possiamo scegliere noi.
Proviamo gli altri casi:
$\text{T}(2) = 2\text{T}(\lfloor 2/2 \rfloor) + 2 = 2\text{T}(1) + 2 = 4 \overset{?}{\leq} c \cdot 2 \log 2 \iff \text{c} \geq 2$
$\text{T}(3) = 2\text{T}(\lfloor 3/2 \rfloor) + 3 = 2\text{T}(1) + 3 = 5 \overset{?}{\leq} c \cdot 3 \log 3 \iff \text{c} \geq \frac{5}{3 \log 3}$
$\text{T}(4) = 2\text{T}(\lfloor 4/2 \rfloor) + 4 = 2\text{T}(2) + 4$
...ma a questo punto mi posso fermare perchè ho trovato un termine noto
A questo punto non resta che scegliere il valore maggiore per cui vale c, in questo caso $c \geq 2$
Quindi abbiamo dimostrato che vale per $\text{n} = 2$ in poi e una coppia è $\text{n} = 2$, $\text{c} = 2$ (e quindi $\text{m} = 2$).

5.3 Metodo dell'Esperto, o delle Ricorrenze Comuni

Siano a e b due costanti intere tali che $a \geq 1$ e $b \geq 2$ e, c, β costanti reali tali che $c > 0$ e $\beta \geq 0$.
Sia T(n) data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$$

Posto $\alpha = \log a / \log b = \log_b a$, allora:

$$T(n) = \begin{cases} \Theta(n^{\alpha}) & \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \alpha = \beta \\ \Theta(n^{\beta}) & \alpha < \beta \end{cases}$$

Esempio:
$T(n) = 4T(n/16) + n^2 \implies a = 4,\ b = 16$
$\beta = 2$
$\alpha = \log b / \log a = \log 16 / \log 4 = \log_{16} 4 = 1/2$
Visto che $\beta > \alpha \implies T(n) = \Theta(n^{\beta}) = \Theta(n^2)$

5.4 Ricorrenze Lineari con Partizione Bilanciata (Estesa)

Sia $a \geq 1$, $b > 1$, $f(n)$ asintoticamente positiva, e sia

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ d & n \leq 1 \end{cases}$$

Siano dati 3 casi:

(1)	$\exists \epsilon > 0 : f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) \implies T(n) = \Theta(n^{\log_b a})$
(2)	$f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(f(n) \log n)$
(3)	$\exists \epsilon > 0 : f(n) = \Omega(n^{log_b a + \epsilon}) \wedge$ $\exists c : 0 < c < 1, \exists m > 0 : \implies T(n) = \Theta(f(n))$ $af(n/b) \leq cf(n), \forall n \geq m$

Esempio 1
$T(n) = 9T(n/3) + n \implies a = 9, b = 3, \log_b a = 2$
$f(n) = n = \mathcal{O}(n^{\log_b a - \epsilon}) = \mathcal{O}(n^{2-\epsilon})$, con $\epsilon < 1$
Caso (1) $\implies T(n) = \Theta(n^2)$

Esempio 2
$T(n) = T(2n/3) + 1 \implies a = 1, b = 2/3, \log_b a = 0$
$f(n) = n^0 = \Theta(n^{\log_b a}) = \Theta(n^0)$
Caso (2) $\implies T(n) = \Theta(\log n)$

Esempio 3
$T(n) = 3T(n/4) + n \log n \implies a = 3, b = 4, \log_b a \approx 0.79$
$f(n) = n \log n = \Omega(n^{\log_4 3 + \epsilon})$, con $\epsilon < 1$ - $\log_4 3 \approx 0.208$

Caso(3) $\implies$ Dobbiamo dimostrare che:
$\exists c \leq 1, \exists m > 0 : af(n/b) \leq cf(n), \forall n \geq m$
$af(n/b) = 3n/4 \log n/4 = 3n/4 \log n - 3n/4 \log 4 \leq 3n/4 \log n \overset{?}{\leq} cn \log n$
L'ultima disequazione è soddisfatta da $c = 3/4$ e $m = 1$

5.5 Ricorrenze Lineari di Ordine Costante

Siano a_1, a_2, ..., a_h costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geq 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:

$$T(n) = \begin{cases} \sum\limits_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

Posto $a = \sum\limits_{1 \leq i \leq h} a_i$, allora:

- $T(n)$ è $\Theta(n^{\beta+1})$, se $a = 1$,
- $T(n)$ è $\Theta(a^n n^\beta)$, se $a \geq 2$,

Esempio 1
$T(n) = T(n-10) + n^2 \implies a = 1, \beta = 2$
Caso (1) poichè $a = 1$, il costo è polinomiale $\implies T(n) = \Theta(n^{\beta+1}) = \Theta(n^3)$

Esempio 2
$T(n) = T(n-2) + T(n-1) + 1 \implies a = 2, \beta = 0$
Caso (2) poichè $a = 2$, il costo è esponenziale $\implies T(n) = \Theta(a^n n^\beta) = \Theta(2^n)$

6 Algoritmi di Ordinamento

Problema: Data una sequenza A = a1, a2 , ... an di n valori, restituire una sequenza B = b1 , b2 , ... bn che sia una permutazione di A e tale per cui b1 $\leq$ b2 $\leq$ ... $\leq$ bn.

6.1 Selection Sort

Algorithm 6 Selection Sort

```
1  SelectionSort(ITEM[] A, int n)
2          for i = 1 to n - 1 do
3          int min_temp = min_from_to(A, i, n)
4          A[min_temp] ↔ A[i]
5
6  int min_from_to(ITEM[] A, int inizio, int fine)
7          int i_min = inizio
8          for indice = inizio + 1 to fine do
9          if S[i_min] > S[indice] then
10         i_min = indice
11         return i_min
```

Costo computazionale: La complessità nel caso medio, pessimo ed ottimo è uguale e vale $\mathcal{O}(n^2)$

6.2 Insertion Sort

Algorithm 7 Insertion Sort

```
1  insertionSort(ITEM[] A, int n)
2          for i = 2 to n do
3          % Salvo l'elemento che dovro' shiftare
4          ITEM temp = A[i]
5          % Mi salvo la sua posizione
6          int j = i
7          % Scorro il mazzo a ritroso trovando la sua posizione
8          while j > 1 and temp < A[j-1] do
9                  A[j] = A[j-1]
10                 j = j - 1
11         A[j] = temp
```

Costo computazionale: La complessità nel caso medio e pessimo e vale $\mathcal{O}(n^2)$, mentre per il caso ottimo la complessità è invece $\mathcal{O}(n)$

6.3 Merge Sort

Algorithm 8 Merge Sort

```
1  mergeSort(ITEM[] A, int n)
2          mergeSort_rec(A, 1, n)
3
4  mergeSort_rec(ITEM[] A, int first, int last)
5          if first ≤ last do
6          int mid = (first + last)/2
7          mergeSort_rec(A, first, mid)
8          mergeSort_rec(A, mid + 1, last)
9          merge(A, first, mid, last)
10
11 merge(ITEM[] A, int first, int mid, int last)
12         int i, j, k, h
13         % vettore di appoggio
14         ITEM[] B = new ITEM[1..last]
15         i = first
16         j = mid + 1
17         k = first
18         % riempimento vettore di appoggio in maniera ordinata
19         while i ≤ mid and j ≤ last do
20         if A[i] < A[j] then
21                 B[k] = A[i]
22                 i = i + 1
23         else
24                 B[k] = A[j]
25                 j = j + 1
26         k = k + 1
27         j = last
28         for h = mid downto i do
29                 A[j] = A[h]
30                 j = j − 1
31         for j = first to k − 1 do
32                 A[j] = B[j]
```

Costo computazionale: La complessità di merge è di $\mathcal{O}(n)$, mentre la complessità di merge sort vale in tutti i casi $\mathcal{O}(n \log n)$.

6.4 Heap Sort

Riferimento: Vedi il capitolo riguardante "Strutture dati speciali - Heap".

6.5 Quick Sort

Riferimento: Vedi il capitolo riguardante "Divide-et-Impera".

7 Strutture dati base

Nelle seguenti sottosezioni saranno elencate le specifiche (dati e operazioni) delle strutture dati base che saranno usate nei capitoli successivi.

7.1 Insiemi

Algorithm 9 Specifica SET

```
% Restituisce la cardinalita' dell'insieme
int = size()
% Restituisce true se x e' contenuto nell'insieme
boolean contains(ITEM x)
% Inserisce x nell'insieme, se non gia' presente
insert(ITEM x)
% Rimuove x dall'insieme, se presente
remove(ITEM x)
% Restituisce un nuovo insieme che e' l'unione di A e B
SET union(SET A, SET B)
% Restituisce un nuovo insieme che e' l'intersezione di A e B
SET intersection(SET A, SET B)
% Restituisce un nuovo insieme che e' la differenza di A e B
SET difference(SET A, SET B)
```

7.2 Dizionari

Algorithm 10 Specifica dizionari

```
% Restituisce il valore associato alla chiave k se presente, nil altrimenti
ITEM lookup(ITEM k)
% Associa il valore v alla chiave k
insert(ITEM k, ITEM v)
% Rimuove l'associazione della chiave k
remove(ITEM k)
```

L'implementazione più conosciuta è l'HashTable.

7.3 Stack

Algorithm 11 Specifica STACK

```
% Restituisce true se la pila e' vuota
boolean isEmpty()
% Inserisce v in cima alla pila
push(ITEM v)
% Estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM pop()
% Legge l'elemento in cima alla pila
ITEM top()
```

7.4 Queue

Algorithm 12 Specifica QUEUE

```
% Restituisce true se la coda e' vuota
boolean isEmpty()
% Inserisce v in fondo alla coda
enqueue(ITEM v)
% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue()
% Legge l'elemento in testa alla coda
ITEM top()
```

8 Alberi

Descrizione: Un albero radicato consiste in un insieme di nodi e un insieme di archi orientati che connettono coppie di nodi. Un nodo è definito come radice e, a differenza degli altri nodi, non ha archi entranti e esiste un percorso unico tra la radice e ogni nodo.

8.1 Alberi binari

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come figlio sinistro e figlio destro.
Specifica: Tutte le operazioni possibili sugli alberi binari sono le seguenti.

Algorithm 13 Specifica albero binario

```
% Costruisce un nuovo nodo, contenente v, senza figli o genitori
Tree(ITEM v)
% Legge il valore memorizzato nel nodo
ITEM read()
% Modifica il valore memorizzato nel nodo
write(ITEM v)
% Restituisce il padre, oppure nil se questo nodo e' radice
TREE parent()
% Restituisce il figlio sinistro (destro) di questo nodo o restituisce nil se
    assente
TREE left()
TREE right()
% Inserisce il sottoalbero radicato in t come figlio sinistro (destro) di questo
    nodo
insertLeft(TREE t)
insertRight(TREE t)
% Distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo
deleteLeft()
deleteRight()
```

Memorizzazione: Ogni nodo dell'albero è memorizzato come sottoalbero. Per ogni nodo è necessario memorizzare 3 campi: parent, che è una reference al nodo padre, left che è una reference al nodo figlio sinistro e right che è una reference al figlio destro. **Implementazione**: Implementazione delle operazioni sugli alberi di ricerca, usando la specifica definita sopra.

Algorithm 14 Implentazione alberi binari

```
1  TREE
2          ITEM V
3          TREE parent
4          TREE left
5          TREE right
6
7  Tree(Item v)
8          TREE t = new TREE
9          t.parent = nil
10         t.left = t.right = nil
11         t.value = v
12         return t
13
14 insertLeft(TREE T)
15         if left == nil then
16                 T.parent = this
17                 left = T
18
19 insertRight(TREE T)
20         if right == nil then
21                 T.parent = this
22                 right = T
23
24 deleteLeft()
25         if left ≠ nil then
26                 left.deleteLeft()
27                 left.deleteRight()
28                 left = nil
29
30 deleteRight()
31         if right ≠ nil then
32         right.deleteLeft()
33         right.deleteRight()
34         right = nil
```

8.1.1 Visite alberi binari

Problema: Dato un albero T, visitare tutti i nodi della struttura. **Soluzione**: Esistono principalmente due metodologie per visitare un albero:

- DFS (Depth-First search - visita in profondità): per visitare un albero si visita ricorsivamente ognuno dei suoi sottoalberi. Tre varianti: pre/post/in order. Richiede uno stack.

Algorithm 15 Visita DFS albero binario

```
1  preorder_dfs(TREE albero)
2          if albero ≠ nil then
3                  print albero.read()
4                  preorder_dfs(albero.left())
5                  preorder_dfs(albero.right())
6
7  inorder_dfs(TREE albero)
8          if albero ≠ nil then
9                  inorder_dfs(albero.left())
10                 print albero.read()
11                 inorder_dfs(albero.right())
12
13 postorder_dfs(TREE albero)
14         if albero ≠ nil then
15                 postorder_dfs(albero.left())
16                 postorder_dfs(albero.right())
17                 print albero.read()
```

- BFS(Breadth-First search - visita in ampiezza): ogni livello dell'albero viene visitato, uno dopo l'altro partendo dalla radice. Richiede una coda.

Algorithm 16 Visita BFS albero binario

```
1  bfs(TREE albero)
2          QUEUE q = new QUEUE
3          q.enqueue(albero)
4          while not q.empty() do
5          TREE tmp = q.dequeue()
6          print tmp.read()
7          if tmp.left() ≠ nil then
8                  q.enqueue(tmp.left())
9          if tmp.right() ≠ nil then
10                 q.enqueue(tmp.right())
```

8.1.2 Esempi visite alberi binari con DFS

Esempi: Due algoritmi sugli alberi binari che fanno uno delle BFS.

Algorithm 17 Esempi DFS alberi binari

```
% conta i nodi di un albero binario
count_nodi(TREE albero)
        if albero == nil then
                return 0
        else
                return 1 + count_nodi(albero.left())+count_nodi(albero.right())

% stampa delle espressioni memorizzate in una struttura ad albero binario
stampa_exp(TREE albero)
        if albero.left() == nil and albero.right() == nil then
                print albero.read()
        else
                print "("
                stampa_exp(albero.left())
                print albero.read()
                stampa_exp(albero.right())
                print ")"
```

8.1.3 Esempi visite alberi binari con BFS

Un albero binario di Natale è un albero binario tale per cui vale la seguente regola: tutti i livelli hanno la stessa brillantezza.
La brillantezza di un livello è pari alla somma della brillantezza dei nodi appartenenti a tale livello.
Un albero binario nil è un albero binario di Natale, ma triste.

Algorithm 18 Esempi BFS alberi binari

```
boolean alberoBinNatale(TREE T)
        if T == nil then
                return true
        else
                QUEUE q = new QUEUE
                q.enqueue(T)
                int brillantezzaAlbero = T.brillace
                int currentBrillance = 0
                int nodilivello = 1
                while not q.isEmpty() do
                        TREE u = q.dequeue()
                        currentBrillance = currentBrillance + u.brillace
                        nodilivello = nodilivello - 1
                        if u.left() ≠ nil then
                                q.enqueue(u.left())
                        if u.right() ≠ nil then
                                q.enqueue(u.right())
                        if nodilivello == 0 then
                                if currentBrillance ≠ brillantezzaAlbero then
                                        return false
                                currentBrillance = 0
                                nodilivello = q.size()
        return true
```

La soluzione proposta conosce il numero di nodi che fanno parte di un livello, in quanto, nel momento in cui un livello termina, in coda sono presenti tutti i nodi del livello successivo.
Il primo livello ovviamente ha dimensione 1, in quanto è un albero binario radicato. Ovviamente partendo da questo è semplice sommare la brillantezza dei livelli sapendo che, al termine dello stesso, gli elementi del prossimo livello sono dati dagli n elmenti in coda a tal momento, dove $n = q.size()$, dove q è l'instanza della coda.

8.2 Alberi generici

Un albero generico è un albero con un numero di figli non fissato.
Specifica: Tutte le operazioni possibili sugli alberi generici sono le seguenti.

Algorithm 19 Specifica albero generico

```
1 % Costruisce un nuovo nodo, contenente v, senza figli o genitori
2 Tree(ITEM v)
3 % Legge il valore memorizzato nel nodo
4 ITEM read()
5 % Modifica il valore memorizzato nel nodo
6 write(ITEM v)
7 % Restituisce il padre, oppure nil se questo nodo e' radice
8 TREE parent()
9 % Restituisce il primo figlio, oppure nil se questo nodo e' una foglia
10 TREE leftmostChild()
11 % Restituisce il prossimo fratello, oppure nil se assente
12 TREE rightSibling()
13 % Inserisce il sottoalbero t come primo nodo di questo nodo
14 insertChild(TREE t)
15 % Inserisce il sottoalbero t come prossimo fratello di questo nodo
16 insertSibling(TREE t)
17 % Distruggi l'albero radicato identificato dal primo figlio
18 deleteChild()
19 % Distruggi l'albero radicato identificato dal prossimo fratello
20 deleteSibling()
```

Memorizzazione: Esistono 3 diverse modalità per memorizzare un albero generico: vettore dei figli, primo figlio - prossimo fratello, vettore dei padri.

8.2.1 Visite alberi generici

Problema: Dato un albero generico T, visitare tutti i nodi della struttura. **Soluzione**: Esistono principalmente due metodologie per visitare un albero:

- DFS

Algorithm 20 Visita DFS albero generico

```
1  preorder_generic_dfs(TREE albero)
2          if albero ≠ nil then
3                  print albero.read()
4                  TREE tmp = albero.leftmostChild()
5                  while(tmp ≠ nil)
6                          preorder_generic_dfs(tmp)
7                          tmp = tmp.rightSibling()
8
9  postorder_generic_dfs(TREE albero)
10         if albero ≠ nil then
11                 TREE tmp = albero.leftmostChild()
12                 while(tmp ≠ nil)
13                         postorder_generic_dfs(tmp)
14                         tmp = tmp.rightSibling()
15                 print albero.read()
```

- BFS

Algorithm 21 Visita BFS albero generico

```
1  generic_bfs(TREE albero)
2          QUEUE q = new QUEUE
3          q.enqueue(albero)
4          while not q.empty() do
5                  TREE tmp = q.dequeue()
6                  print tmp.read()
7                  tmp = tmp.leftmostChild()
8                  while tmp ≠ nil do
9                          q.enqueue(tmp)
10                         tmp = tmp.rightSibling()
```

8.3 Alberi Binari di Ricerca (ABR)

Descrizione: Un albero T è un albero binario che rispetta le seguenti proprietà: le chiavi contenute nei nodi del sottoalbero sinistro di un nodo generico u sono minori di u.key, mentre le chiavi contenute nei nodi del sottoalbero destro sono maggiori di u.key.L'idea degli ABR è quella di portare la ricerca binaria negli alberi. Un ABR è un dizionario in cui ogni nodo u contiene delle associazioni chiave-valore salvate attraverso una coppia (u.key, u.value).
Specifica: La struttura di un ARB e tutte le operazioni possibili sono le seguenti.

Algorithm 22 Specifica ABR

```
1  TREE
2          TREE parent
3          TREE left
4          TREE right
5          ITEM key
6          ITEM value
7
8  TREE lookupNode(TREE T, ITEM k)
9  TREE insertNode(TREE T, ITEM k, ITEM v)
10 TREE removeNode(TREE T, Item k)
11 TREE successorNode(TREE t)
12 TREE predecessorNode(TREE t)
13 TREE min()
14 TREE max()
```

Implementazione: Implementazione di un ABR usando la specifica precedente.

Algorithm 23 Specifica ABR: lookup

```
1  % lookup versione ricorsiva
2  TREE lookupNode(TREE T, ITEM k)
3          if T == nil or T.key == k then
4                  return T
5          else
6                  return lookupNode(iif(k < T.key, T.left, T.right), k)
7
8  % loopup versione iterativa
9  TREE lookupNode(TREE T, ITEM k)
10         TREE u = T
11         while u ≠ nil and u.key ≠ k do
12                 if k < u.key then
13                         u = u.left                    % Sotto-albero di
                               sinistra
14                 else
15                         u = u.right                   % Sotto-albero di destra
16         return u
```

Algorithm 24 Specifica ABR: min e max

```
1  TREE min(TREE albero)
2          TREE u = albero
3          if u == nil then
4                  retun u
5          while u.left ≠ nil do
6                  u = u.left
7          return u
8
9  TREE max(TREE albero)
10         TREE u = albero
11         if u == nil then
12                 retun u
13         while u.right ≠ nil do
14                 u = u.right
15         return u
```

Algorithm 25 Specifica ABR: successore e predecessore

```
1  TREE successore(TREE albero)
2          if albero == nil then
3                  return albero
4          if albero.right ≠ nil then
5                  return min(albero.right)
6          else
7          % risalgo la catena di padri
8                  TREE parent = albero.parent
9                  while parent ≠ nil and parent.left ≠ albero do
10                         parent = parent.parent
11                         albero = albero.parent
12         return parent
13
14 TREE predecessore(TREE albero)
15         if albero == nil then
16                 return albero
17         if albero.left ≠ nil then
18                 return max(albero.left)
19         else
20                 % risalgo la catena di padri
21                 TREE parent = albero.parent
22                 while parent ≠ nil and parent.right ≠ albero do
23                         parent = parent.parent
24                         albero = albero.parent
25         return parent
```

Algorithm 26 Specifica ABR: insert e link

```
1  TREE insert(TREE albero, ITEM key, ITEM value)
2          if albero == nil then
3                  albero = new TREE(key, value)
4          % trovo la posizione del nodo da inserire
5          TREE parent = albero
6          TREE t = nil
7          while t ≠ nil and t.key ≠ key do
8                  parent = t
9                  if key < t.key
10                 t = t.left
11                 else
12                         t = t.right
13         % fermato avro' dunque padre e la posizione in cui inserire il figlio
14         if t ≠ nil then
15                 % significa che devo solo aggiornare il valore
16                 t.value = value
17         else
18                 % il nodo da inserire e' nuovo nell'albero, in questo caso va
                       dunque inserito un nuovo nodo
19                 TREE nuovo = new TREE(k, value)
20                 link(parent, nuovo, k)
21         return albero
22
23 % la procedura link e' importante solamente per l'inserimento del nuovo nodo, in
       quanto in base alla chiave stabilisce dove il nodo deve essere posizionato
24 link(TREE albero, TREE inserire, ITEM k)
25         if k < albero.key then
26                 %  inserimento a sinistra
27                 albero.left = inserire
28         else
29                 % significa che il nodo da inserire e' un nodo la cui chiave e'
                       maggiore di quella del padre
30                 albero.right = inserire
31         if inserire ≠ nil then
32                 inserire.parent = albero
```

Algorithm 27 Specifica ABR: cancellazione

```
1  TREE cancellazione(TREE albero, ITEM key)
2          % devo individuare il nodo da eliminare, mi basta quello
3          TREE u = albero
4          while u ≠ nil and u.key ≠ key do
5                  if key < u.key then
6                          u = u.left
7                  else
8                          u = u.right
9          % controllo se il nodo da rimuovere ha veri e propri figli
10         if u.left == u.right == nil then
11                 % caso albero formato da un unico nodo da rimuovere
12                 if u == albero then
13                         return nil
14                 else
15                         link(u.parent, nil, u.key)
16                         delete u
17         % caso invece un solo figlio
18         if u.left == nil then
19                 if u == albero then
20                         u = u.left
21                         u.parent = nil
22                         return u
23                 else
24                         u = u.left
25                         u.parent = (u.parent).parent
26         else if u.right == nil then
27                 if u == albero then
28                         u = u.right
29                         u.parent = nil
30                         return u
31                 else
32                         u = u.right
33                         u.parent = (u.parent).parent
34         else
35                 % entrambi i figli
36                 TREE s = successore(u)
37                 link(s.parent, s.right, s.key)
38                 % sostituzione dei nodi successore e u
39                 u.key = s.key
40                 u.value = s.value
41                 delete s
```

8.4 Alberi Red-Black

Descrizione: Sono dei particolari alberi binari di ricerca che hanno la caratteristica di essere bilanciati e di mantenere il bilanciamento tra l'altezza dele foglie anche in seguito ad inserimenti e cancellazioni. Le principali caratteristiche sono: ogni nodo è colorato di rosso o di nero, le chiavi vengono salvate solo nei nodi interni dell'albero e le foglie sono dei nodi speciali Nil. Per essere un albero red-black un ABR deve rispettare dei vincoli:

- La radice deve essere nera
- Tutte le foglie sono nere
- Entrambi i figli di un nodo rosso sono neri
- Tutti i cammini semplici da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri

Implementazione: Di seguito una parte di una possibile implementazione degli alberi red-black.

Algorithm 28 Specifica Albero Red-Black

```
1  TREE
2          TREE parent
3          TREE left
4          TREE right
5          int color
6          ITEM key
7          ITEM value
8
9  % le funzioni di rotazione vengono usate per mantenere le proprita' dell'albero
       RB
10 TREE rotateLeft(TREE x)
11         % prelevo right e left
12         TREE y = x.right
13         TREE p = x.parent
14         % assegno B a figlio destro di x
15         x.right = y.left
16         if y.left ≠ nil then
17                 y.left.parent = x
18         % x diventa il figlio sinistro di y
19         y.left = x
20         x.parent = y
21         % y diventa figlio di p
22         y.parent = p
23         if p ≠ nil then
24                 if p.left == x then
25                         p.left = y
26                 else
27                         p.right = y
28         return y
29
30 % insert e balanceInsert
31         ...
32
33 % cancellazione
34         ...
```

9 Grafi

Descrizione: Un grafo orientato (rispettivamente non orientato) è una coppia G=(V,E) con V insieme di nodi (o vertici) ed E un insieme di coppie ordinate (rispettivamente non ordinate) di nodi dette archi. La complessità ottimale degli algoritmi sui grafi è espressa in termini sia di n (numero di nodi) che di m (numero di archi) e vale solitamente O(n + m). **Specifica**: Tutte le operazioni possibili sui grafi sono le seguenti.

Algorithm 29 Specifica grafo

```
1 % Crea un nuovo grafo
2 Graph()
3 % Restituisce l'insieme di tutti i nodi
4 SET V()
5 % Restituisce il numero di nodi
6 int size()
7 % Restituisce l'insieme dei nodi adiacenti a u
8 SET adj(NODE u)
9  % Aggiunge il nodo u al grafo
10 insertNode(NODE u)
11 % Aggiunge l'arco (u, v) al grafo
12 insertEdge(NODE u, NODE v)
13 % Rimuove il nodo u dal grafo
14 deleteNode(NODE u)
15 % Rimuove l'arco (u, v) dal grafo
16 deleteEdge(NODE u, NODE v)
```

Memorizzazione: Esistono due principali metodologie per memorizzare un grafo: liste di adiacenza e matrici di adiacenza. La matrice di adiacenza consiste nel memorizzare una matrice $n \cdot n$ in cui viene segnata ad 1 la cella avente come indici gli estremi degli archi presenti. Mentre la lista di adiacenza condiste nell'avere una lista degli n nodi e per ciascun elemento avere un'altra lista contenente i nodi adiacenti al nodo stesso. Le matrici occupano più spazio e permettono più velocemente di sapere se due nodi sono adiacenti, ma richiedono più tempo per visitare tutti gli archi. Con queste tecniche è possibile memorizzare anche grafi pesati i cui archi hanno pesi differenti.

Dettagli sull'implementazione Se non diversamente specificato, nel seguito:

- Assumeremo che l'implementazione sia basata su vettori di adiacenza, statici o dinamici
- Assumeremo che la classe $NODE$ sia equivalente a int (quindi l'accesso alle informazioni avrà costo $\mathcal{O}(1)$)
- Assumeremo che le operazioni per aggiungere nodi e archi abbiano costo $\mathcal{O}(1)$
- Assumeremo che dopo l'inizializzazione, il grafo sia statico

9.1 Visite grafi

Problema: Dato un grafo G=(V,E) e un vertice $v \in V$, visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da v. **Soluzione**: Esistono principalmente due metodologie per visitare un grafo:

- BFS(Breadth-First search - visita in ampiezza): visita i nodi per livelli partendo dalla radice, poi visitando i nodi a distanza 1, poi a distanze 2, ecc fino a visitare tutti i nodi raggiungibili dalla sorgente.

Algorithm 30 Visita BFS

```
1  bfs(GRAPH G, NODE r)
2          QUEUE Q = QUEUE
3          Q.enqueue(r)
4          boolean[] visited = new boolean[1...G.size()]
5          foreach u ∈ G.V() − {r} do
6                  visited[u] = false
7          visited[r] = true
8          while not Q.isEmpty() do
9                  Node u = Q.dequeue()
10                 % visita il nodo u
11                 foreach v : G.adj(u) do
12                         % visita l'arco (u,v)
13                         if not visited[v] then
14                                 visited[v] = true
15                                 Q.enqueue(v)
```

- DFS (Depth-First search - visita in profondità): visita ricorsiva, cioè ogni nodo viene visitato ricorsivamente, visitando anche in modo ricorsivo i suoi nodi adiacenti. Con questo tipo di visita riusciamo a visitare un intero grafo, non solo i nodi raggiungibili da una sorgente.

Algorithm 31 Visita DFS

```
1  % versione ricorsiva con stack implicito
2  dfs(GRAPH G, Node u, boolean[] visited)
3          visited[u] = true
4          % visita il nodo u (pre-order)
5          foreach v ∈ G.adj(u) do
6                  if not visited[v] then
7                          % visita l'arco (u,v)
8                          dfs(G, v, visited)
9          % visita il nodo u (post-order)
10
11
12 % versione iterativa con stack esplicito
13 dfs(GRAPH G, Node r)
14         STACK S = STACK
15         S.push(r)
16         boolean[] visited = new boolean[1...G.size()]
17         foreach u ∈ G.V() do
18                 visited[u] = false
19         while not S.isEmpty() do
20                 Node u = S.pop()
21                 if not visited[u] then
22                         % visita il nodo u (pre-order)
23                         visited[v] = true
24                         foreach v ∈ G.adj(u) do
25                                 % visita l'arco (u,v)
26                                 S.push(v)
```

9.2 Cammini più brevi: BFS

Problema: Dato un grafo G e un nodo r, calcolare il cammino più breve da r a tutti gli altri nodi. Le distanze sono misurate come il numero di archi attraversati. **Soluzione**: Algoritmo per il calcolo del numero di Erdos.

Algorithm 32 Erdos

```
1  erdos(GRAPH G, NODE r, int[] erdos, NODE[] parent)
2          QUEUE Q = QUEUE
3          Q.enqueue(r)
4          foreach u ∈ G.V()−{r} do
5                  erdos[u] = +∞
6          erdos[r] = 0
7          parent[r] = nil % permette di costruire l'albero di copertura
8          while not Q.isEmpty() do
9                  NODE u = Q.dequeue()
10                 foreach v ∈ G.adj(u) do
11                         if erdos[v] == +∞ then          % Se il nodo v non e'
                               stato scoperto
12                                 erdos[v] = erdos[u] + 1
13                                 parent[v] = u
14                                 Q.enqueue(v)
```

Estensione: Dopo aver calcolato Erdos aver inizializzato il vettore dei parent è possibile stampare l'albero di copertura usando tale vettore.

Algorithm 33 Albero di copertura BFS

```
1  printPath(NODE r, NODE s, NODE[] parent)
2          if r == s then
3                  print s
4          else if parent[s] == nil then
5                  print "error"
6          else
7                  printPath(r, parent[s], parent)
8                  print s
```

9.3 Componenti connesse: DFS

Problema: Dato un grafo G non orientato verificare se è connesso (cioè ogni suo nodo è raggiungibile da ogni altro nodo) e identificare le sue componenti connesse (cioè sottografi massimali e connessi). **Soluzione**: Un grafo è connesso se, al termine della DFS tutti i nodi sono stati marcati come visitati. Altrimenti la visita deve ricominciare da capo da un nodo non marcato, identificando una nuova componente del grafo. Strutture necessarie: un vettore *id* che contiene gli identificatori delle componenti, in cui *id*[u] è il numero che identifica la componente connessa a cui appartiene il nodo u. Se l'id di un nodo è uguale a 0, vuol dire che quel nodo deve essere ancora visitato.

Algorithm 34 Componenti connesse (cc) con ricorsione

```
1  int[] cc(GRAPH G)
2          int[] id = new int[1...G.size()]
3          foreach u ∈ G.V() do
4                  id[u] = 0
5          int counter = 0
6          foreach u ∈ G.V() do
7                  if id[u] == 0 then
8                          counter = counter +1
9                          ccdfs(G, counter, u, id)
10         return id
11
12 % visita ricorsiva degli adiacenti seguendo lo schema della DFS
13 ccdfs(GRAPH G, int counter, Node u, int[] id)
14         id[u] = counter
15         foreach v ∈ G.adj(u) do
16                 if id[v] == 0 then
17                         ccdfs(G, counter, v, id)
```

9.4 Grafo non orientato aciclico: DFS

Problema: Dato un grafo G non orientato verificare se è aciclico, cioè se non contiene cicli (con lunghezza maggiore di 2). Restituire true se G contiene un cicli, false altrimenti. **Soluzione**: La soluzione si basa sempre su una DFS.

Algorithm 35 Grafo non orientato aciclico (hasCycle)

```
1  boolean hasCycle(GRAPH G)
2          boolean[] visited = new boolean[1...G.size()]
3          foreach u ∈ G.V() do
4                  visited[u] = false
5          foreach u ∈ G.V() do
6                  if not visited[u] then
7                          if hasCycleRec(G, u, nil, visited) then
8                                  return true
9          return false
10
11 boolean hasCycleRec(GRAPH G, Node u, Node p, boolean[] visited)
12         visited[u] = true
13         foreach v ∈ G.adj(u)−{p} do
14                 if visited[v] then
15                         return true
16                 else if hasCycleRec(G, v, u, visited) then
17                         return true
18         return false
```

9.4.1 Albero di copertura DFS

Problema: Dato un grafo G orientato creare un albero di copertura T e classificare gli archi presenti nel grafo. **Soluzione**: Ogni volta che si esamina un arco da un nodo marcato ad uno non marcato,chiamiamo questo arco come arco dell'albero. Gli archi (u,v) non inclusi nell'albero (cioè non archi dell'albero) sono classificati in 3 categorie:

- Archi in avanti: se u è un antenato di v in T
- Archi all'indietro: se u è un discente di v in T
- Archi di attraversamento: tutti gli archi restanti che non sono in avanti o all'indietro.

Per risolvere il problema di classificazione degli archi usiamo sempre una visita DFS.

Algorithm 36 Albero di copertura

```
1  % time: contatore, dt: discovery time (tempo di scoperta del nodo), ft: finish
       time (tempo di fine visita del nodo e dei suoi archi)
2  dfs-schema(GRAPH G, Node u, int& time, int[] dt, int[] ft)
3          % visita il nodo u (pre-order)
4          time = time + 1
5          dt[u] = time
6          foreach v ∈ G.adj(u) do
7                  % visita l'arco (u,v) (qualsiasi)
8                  if dt[v] == 0 then
9                          % visita l'arco (u,v) (albero)
10                         dfs-schema(G, v, time, dt, ft)
11                 else if dt[u] > dt[v] and ft[v] == 0 then
12                         %visita l'arco (u,v) (indietro)
13                 else if dt[u] < dt[v] and ft[v] ≠ 0 then
14                         %visita l'arco (u,v) (avanti)
15                 else
16                         % visita l'arco (u,v) (attraversamento)
17         % visita il nodo u (post-order)
18         time = time + 1
19         ft[u] = time
```

9.5 Grafo orientato aciclico(DAG): DFS

Problema: Dato un grafo G orientato verificare se è aciclico, cioè se non contiene cicli (con lunghezza maggiore uguale a 2). Restituire true se G contiene un cicli, false altrimenti. **Soluzione**: La soluzione usata per i grafi non orientati non è sempre funzionante. Per risolvere il problema dobbiamo usare il concetto dell'albero di copertura DFS e i vari tipi di archi. Esiste un teorema che afferma che un grafo orientato è aciclico $\iff$ non esistono archi all'indietro nel grafo.

Algorithm 37 Grafo orientato aciclico (hasCycle)

```
boolean hasCycle(GRAPH G, Node u, int &time, int[] dt, int[] ft)
        time = time + 1
        dt[u] = time
        foreach v ∈ G.adj(u) do
                if dt[v] == 0 then
                        if hasCycle(G, v, time, dt, ft) then
                                return true
                else if dt[u] > dt[v] and ft[v] == 0 then
                        return true
        time = time + 1
        ft[u] = time
        return false
```

9.6 Ordinamento topologico: DFS

Problema: Dato un grafo G che rispetta le proprietà di DAG, restituire l'ordinamento topologico di tale grafo. L'ordinamento topologico è un ordinamento lineare tale che se (u,v) ∈ E, allora u appare prima di v nell'ordinamento. **Soluzione**: Usare una DFS nella quale l'operazione principale consiste nell'aggiungere il nodo in testa ad una lista in post-ordine. Il risultato sarà l'ordinamento topologico.

Algorithm 38 Ordinamento topologico

```
STACK topSort(GRAPH G)
        STACK S = STACK
        boolean[] visited = boolean[1...G.size()]
        foreach u ∈ G.V() do
                visited[u] = false
        foreach u ∈ G.V() do
                if not visited[u] then
                        ts-dfs(G, u, visited, S)
        return S

ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
        visited[u] = true
        foreach v ∈ G.adj(u) do
                if not visited[v] then
                        ts-dfs(G, v, visited, S)
        S.push(u)
```

9.7 Componenti fortemente connesse (SCC): DFS

Problema: Dato un grafo G orientato verificare se è fortemente connesso (cioè ogni suo nodo è raggiungibile da ogni altro nodo) e identificare le sue componenti connesse (cioè sottografi orientati massimali e connessi). **Soluzione**: Non è possibile usare lo stesso algoritmo che identifica le componenti fortemente connesse perchè il risultato dipende dal nodo di partenza. Una delle possibili soluzioni è usare l'algoritmo di Kosaraju che consiste nel:

- Effettuare una DFS del grafo per calcolare l'ordinamento topologico
- Calcolare il grafo trasposto G^T di G
- Effettuare una DFS per il calcolo delle cc sul grafo trasposto e usando come ordine di visita l'ordinamento topologico.

Algorithm 39 Calcolo delle componenti fortemente connesse

```
1  int[] scc(GRAPH G)
2          STACK S = topSort(G)
3          % Prima visita DFS per calcolare il top sort
4          GT = transpose(G)                    % Grafo trasposto
5          return cc(GT, S)                     % CC del grafo trasporto
6
7  % calcolo del grafo trasposto
8  GRAPH transpose(GRAPH G)
9           GT = Graph()
10         foreach u ∈ G.V() do
11                 GT.insertNode(u)
12         foreach u ∈ G.V() do
13                 foreach v ∈ G.adj(u) do
14                         GT.insertEdge(v,u)
15         return GT
16
17
18 int[] cc(GRAPH G, STACK S)
19         int[] id = new int[1...G.size()]
20         foreach u ∈ G.V() do
21                 id[u] = 0
22         int counter = 0
23         while not S.isEmpty() do
24                 u = S.pop()
25                 if id[u] == 0 then
26                         counter = counter +1
27                         ccdfs(G, counter, u, id)
28         return id
29
30 ccdfs(GRAPH G, int counter,NODE u, int[] id)
31         id[u] = counter
32         foreach v ∈ G.adj(u) do
33                 if id[v] == 0 then
34                         ccdfs(G, counter, v, id)
```

Realtà: Nella realtà viene usato l'algoritmo di Tarjan che richiede solo una visita e non due e non richiede la trasposizione del grafo.

10 Hashing

Descrizione: L'hash è la struttura dati più efficiente che implementa il concetto di dizionario: consente di memorizzare insiemi dinamici di coppie <chiave, valore>. Le coppie vengono indicizzate in base alla chiave. La struttura si basa su una funzione di hash che mappa ogni chiave k dell'insieme universo in un intero h(k) e un array che verrà usato per salvare i dati. Quando due o più chiavi hanno lo stesso valore hash, diciamo che è avvenuta una collisione. Esistono due principali metodi di gestione delle collisioni:

- Liste di trabocco: le chiavi con lo stesso valore vengono memorizzate in una lista monodirezionale accessibile a partire dall'array (puntatore alla testa della lista). È fondamentale bilanciare la lunghezza delle liste di trabocco altrimenti si perde tutta l'efficienza della struttura.
- Indirizzamento aperto: tutte le chiavi vengono memorizzate nel vettore stesso, senza strutture aggiuntive. Se lo slot scelto per l'inserimento p già utilizzato, se ne cerca uno "alternativo". Esistono diverse tecniche di ricerca con indirizzamento aperto: ispezione lineare (agglomerazione primaria), ispezione quadratica e doppio hashing.

10.1 Doppio hashing

Descrizione: Usa una funzione del tipo: $H(k,i) = (H_1(k) + i \cdot H_2(k)) \bmod m$, con m lunghezza del vettore e H_1, H_2 funzioni di hash.H_1 fornisce la prima ispezione, H_2 le ispezioni successive. Un doppio hash permette le seguenti operazioni: inserimento, cancellazione (usando uno speciale valore deleted) e ricerca.

Algorithm 40 Hashing Doppio: struttura

```
1 % Definizione struttura
2 HASH
3         ITEM[] K                % Tabella delle chiavi
4         ITEM[] V                % Tabella dei valori
5         int m                   % Dimensione della tabella
```

Algorithm 41 Hashing Doppio: specifica

```
1  HASH Hash(int dim)
2          HASH t = new HASH
3          t.m = dim
4          t.K = new ITEM[0...dim-1]
5          t.V = new ITEM[0...dim-1]
6          for i = 0 to dim-1 do
7                  t.K[i] = nil
8          return t
9
10 int scan(ITEM k, boolean insert)
11         int c = m                  % Prima posizione deleted
12         int i = 0                  % Numero di ispezione
13         int j = H1(k)    % Posizione attuale
14         while K[j] ≠ k and K[j] ≠ nil and i < m do
15                 if K[j] == deleted and c == m then
16                         c = j
17                 j = (j + H2(k)) mod m
18                 i = i + 1
19         if insert and K[j] ≠ k and c < m then
20                 j = c
21         return j
22
23 ITEM lookup(ITEM k)
24         int i = scan(k, false)
25         if K[i] == k then
26                 return V[i]
27         else
28                 return nil
29
30 insert(ITEM k, ITEM v)
31         int i = scan(k, true)
32         if K[i] == nil or K[i] == deleted or K[i] == k then
33                 K[i] = k
34                 V[i] = v
35         else
36                 % Errore: tabella hash piena
37
38 remove(ITEM k)
39         int i = scan(k, false)
40         if K[i] == k then
41                 K[i] = deleted
```

11 Insiemi e dizionari

11.1 Insiemi con vettori booleani

Descrizione: Un insieme è una collezione di m oggetti con valori compresi tra 1..m. Un insieme può essere memorizzato in un vettore booleano di m elementi, con l'indice che rappresenta l'elemento e il contenuto del vettore se l'elemento è presente o meno. I vantaggi sono: implementazione semplice e velocità nel verificare se un elemento appartiene all'insieme, mentre come svantaggi abbiamo che occupiamo O(m) memoria anche per salvare pochi elementi e che alcune operazioni vengono eseguite in O(m).
Implementazione: Possibile implementazione del Set attraverso vettori booleani.

Algorithm 42 Set con vettore booleano: operazioni base

```
% definizione struttura dati SET
SET
        boolean[] V
        int size        % numero di elementi presenti attualmente nel SET
        int dim         % numero di elementi che possono stare in totale nel SET

SET Set(int m)
        SET t = new SET
        t.size = 0
        t.dim = m
        t.V = [false] * m    % inizializzo tutti gli elementi a false
        return t

boolean contains(int x)
        if 1 ≤ x ≤ dim then
                return V[x]
        else
                return false

int size()
        return size

insert(int x)
        if 1 ≤ x ≤ dim then
                if not V[x] then
                        size = size+1
                        v[x] = true

remove(int x)
        if 1 ≤ x ≤ dim then
                if V[x] then
                        size = size-1
                        v[x]= false
```

Algorithm 43 Set con vettore booleano: operazioni base

```
% definizione struttura dati SET
SET union(SET A, SET B)
        SET C = Set(max(A.dim, B.dim))
        for i = 1 to A.dim do
                if A.contains(i) then
                        C.insert(i)
        for i = 1 to B.dim do
                if B.contains(i) then
                        C.insert(i)

SET intersection(SET A, SET B)
        SET C = Set(min(A.dim, B.dim))
        for i = 1 to min(A.dim, B.dim) do
                if A.contains(i) and B.contains(i) then
                        C.insert(i)

SET difference(SET A, SET B)
        SET C = Set(A.dim)
        for i = 1 to A.dim do
                if A.contains(i) and not B.contains(i) then
                        C.insert(i)
```

Implementazioni reali: In Java e C++ sono già implementati i set e sono già presenti nelle librerie standard con in nome BitSet. Esistono anche implementazioni dei set con liste, vettori non ordinati o strutture dati più complesse come alberi e hash table.

11.2 Bloom Filters

Descrizione: Struttura dati che unisce i vantaggi dei BitSet a quelli delle tabelle di Hash. Vantaggi: struttura dati dinamica e bassa occupazione di memoria. Svantaggi: non è possibile effettuare cancellazioni, le risposte ricevute dalla struttura sono probabilistiche e i dati non sono realmente memorizzati nella struttura.
Specifica: Sono presenti due principali operazioni:

- insert(k): inserisce l'elemento k nel bloom filter
- bool contains(k): restituisce false se l'elemento k è sicuramente non presente nell'insieme, restituisce true se l'elemento può essere presente oppure no (falsi positivi).

Per l'implementazione usiamo un vettore booleano A di m bit, inizializzato a false e k funzioni di hash: h_1, h_2, ..., h_k: U $\rightarrow$ [0, m-1]

Algorithm 44 Implementazione Bloom Filter

```
insert(k)
        for i = 1 to k do
                A[hi(k)] = true

boolean contains(k)
        for i = 1 to k do
                if A[hi(k)] == false then
                        return false
        return true
```

12 Strutture dati speciali

12.1 Heap

Descrizione: Struttura dati che associa i vantaggi dell'albero a quelli di un array. Gli alberi binari max-heap (rispettivamente min-heap) sono alberi binari completi tali per cui il valore memorizzato in ogni nodo è maggiore (rispettivamente minore) dei valori memorizzati nei suoi figli. Un albero binario si dice completo se:

- Tutte le foglie hanno profondità h o h-1
- Tutti i nodi a livello h sono accatastati a sinistra (non può mancare il figlio sinistro ed esserci il destro)
- Tutti i nodi interni (non foglia) hanno grado 2, eccetto al più uno
- Se il numero di nodi è n, l'altezza è h = $\lfloor \log n \rfloor$

Gli alberi binari heap non impongono un ordinamento totale, ma parziale.

Algorithm 45 Memorizzazione Heap con Vettore

```
% Vettore di memorizzazione
A[1 ... n]
% Radice
root() = 1
% Padre nodo i
p(i) = floor(i/2)
% Figlio sinistro nodo i
l(i) = 2i
% Figlio destro nodo i
r(i) = 2i + 1
```

Algoritmi sulla struttura: L'algoritmo più famoso sulla struttura Heap è l'algoritmo di ordinamento heapsort().
Codice heapSort: È costituito da due parti: heapBuild() che si occupa di costruire un max-heap a partire da un vettore non ordinato e da maxHeapRestore() che si occupa di ripristinare le proprietà del max-heap ordinando nel mentre l'array.
MaxHeapRestore() ha come parametri un vettore A, un indice i tale per cui gli alberi binari con radici l(i) e r(i) sono max-heap e la dimensione dim del vettore A.

Algorithm 46 maxHeapRestore

```
maxHeapRestore(ITEM[] A, int i, int dim)
        int max = i
        if l(i) ≤ dim and A[l(i)] > A[max] then
                max = l(i)
        if r(i) ≤ dim and A[r(i)] > A[max] then
                max = r(i)
        if i ≠ max then
                A[i] ↔ A[max]
                maxHeapRestore(A, max, dim)
```

HeapBuild() ha come parametri un vettore A e la sua dimensione n.

Algorithm 47 heapBuild

```
heapBuild(ITEM[] A, int n)
        for i = floor(n/2) downto 1 do
                maxHeapRestore(A, i, n)
```

Usando gli algoritmi sopra definiti possiamo costruire una nuova funzione heapSort() che permette di ordinare un array A di dimensione n.

Algorithm 48 heapSort

```
heapSort(ITEM[] A, int n)
heapBuild(A, n)
        for i = n downto 2 do
                A[1] ↔ A[i]
                maxHeapRestore(A, 1, i-1)
```

Complessità computazionale: La chiamata di heapBuild() costa $\Theta(n)$, mentre la chiamata maxHeapRestore() costa $\Theta(log\ i)$ con i elementi nel heap. La maxHeapRestore() viene eseguita per un numero di volte che va da 2 a n e quindi il costo totale è $\Theta(n\ log\ n)$.

12.2 Code a priorità

Descrizione: Le code a priorità sono usa struttura dati astratta simile ad una coda in cui ogni elemento inserito possiede una priorità. Possono esistere min-priority queue in cui l'estrazione avviene per valori crescenti di priorità, mentre nelle max-priority queue l'estrazione avviene per priorità decrescente.
Specifica: interfaccia dei metodi per la gestione della coda a priorità.

Algorithm 49 Min Priority Queue Specifica

```
% Crea una coda con priorita' vuota
void MinPriorityQueue()
% Restituisce true se la coda con priorita' e' vuota
boolean isEmpty()
% Restituisce l'elemento minimo di una coda con priorita (non vuota)
Item min()
% Rimuove e restituisce il minimo da una coda con priorita' (non vuota)
Item deleteMin()
% Inserisce l'elemento x con priorita' p nella coda con priorita' e restituisce
    un oggetto PriorityItem che identifica x all'interno della coda
PriorityItem insert(Item x, int p)
% Diminuisce la priorita' dell'oggetto identificato da y portandola a p
decrease(PriorityItem y, int p)
```

Di seguito una possibile implementazione dei metodi della min-priority queue facente uso del min-heap.

Algorithm 50 Min Priority Queue Implentazione

```
% Definizione di PriorityItem
PriorityItem
        int priority        % Priorita'
        Item value          % Elemento
        int pos             % Posizione nel vettore heap

% Funzione per scambiare due elementi nella coda a priorita'
swap(PriorityItem[] H, int i, int j)
        H[i] ↔ H[j]
        H[i].pos = i
        H[j].pos = j

% Definizione della struttura della coda a priorita'
PriorityQueue
        int capacity            % Numero massimo di elementi nella coda
        int dim                 % Numero attuale di elementi nella coda
```

```
17          PriorityItem[] H    % Vettore heap
18
19  % Inizializzazione della coda a priorita'
20  PriorityQueue priorityQueue(int n)
21          PriorityQueue t = new PriorityQueue
22          t.capacity = n
23          t.dim = 0
24          t.H = new PriorityItem[1...n]
25          return t
26
27  % Inserimento nella coda
28  PriorityItem insert(Item x, int p)
29          precondition: dim < capacity
30
31          dim = dim + 1
32          H[dim] = new PriorityItem()
33          H[dim].value = x
34          H[dim].priority = p
35          H[dim].pos = dim
36          int i = dim
37          while i > 1 and H[i].priority < H[p(i)].priority do
38                  swap(H, i, p(i))
39                  i = p(i)
40          return H[i]
41
42  % Cancellazione dell'elemento con priorita' minore
43  Item deleteMin()
44          precondition: dim > 0
45
46          swap(H, 1, dim)
47          dim = dim - 1
48          minHeapRestore(H, 1, dim)
49          return H[dim + 1].value
50
51  % Restituisce l'elemento con priorita' minore
52  Item min()
53          precondition: dim > 0
54
55          return H[1].value
56
57  % Decrementa la priorita' di un elemento della coda
58  decrease(PriorityItem x, int p)
59          precondition: p < x.priority
60
61          x.priority = p
62          int i = x.pos
63          while i > 1 and H[i].priority < H[p(i)].priority do
64                  swap(H, i, p(i))
65                  i = p(i)
```

Costo computazionale: le operazioni di insert(), deleteMin(), decrease() che modificano l'heap e lo sistemano hanno un costo di $O(\log n)$, mentre min() ha un costo pari a $\Theta(1)$.

12.3 Insiemi disgiunti - Merge Find Set

Motivazione e operazioni fondamentali: Siamo interessati a gestire una collezione $S = S1, S2, ...Sk$ di insiemi dinamici disgiunti tali che:

- $\forall i, j : i \neq j \Rightarrow Si \cap Sj = \emptyset$
- $\bigcup_{i=1}^{k} Si = Sj$, dove $k = |S|$

Le operazioni fondamentali possibili su questa struttura sono:

- Creare n insiemi disgiunti, ognuno composto da un unico elemento
- merge(): unire più insiemi
- find(): identificare l'insieme a cui appartiene un elemento

Rappresentante: Ogni insieme è identificato da un rappresentante univoco. Il rappresentante dell'insieme Si è un qualunque membro di Si. Più operazioni di ricerca del rappresentante svolte su uno stesso insieme devono restituire sempre lo stesso oggetto. Solo in caso di unione con altro insieme, il rappresentante degli insiemi può cambiare.

Specifica MFSET:

Algorithm 51 Specifica MFSET

```
%Crea n componenti {1}, ... ,{n}
MFSET Mfset(int n)
%Restituisce il rappresentante della componente contenente x
int find(int x)
%Unisce le componenti che contengono x e y
merge(int x, int y)
```

Realizzazione: Esistono 2 possibili tecniche per memorizzare gli insiemi disgiunti usando due strutture dati differenti.

12.3.1 Realizzazione basata su insieme di liste

Descrizione: Ogni insieme viene rappresentato da una lista concatenata. Il primo oggetto di una lista è il rappresentante dell'insieme e ogni elemento delle lista contiene: un intero/oggetto, un puntatore all'elemento successivo e un puntatore al rappresentante. **Operazioni**:

- Operazione find(x): viene restituito il rappresentante dell'insieme x. L'operazione di find(x) richiede tempo O(1).
- Operazione di merge(x,y): si "appende" la lista che contiene y alla lista che contiene x, modificando i puntatori ai rappresentanti nella lista "appesa" (per ogni elemento di y devo modificare il rappresentante). Nel caso pessimo, il costo di n operazione è O(n^2) e quindi il coso ammortizzato di una operazione vale O(n).

12.3.2 Realizzazione basata su insieme di alberi (foresta)

Descrizione: Ogni insieme viene rappresentato da un albero. La radice è il rappresentante dell'insieme e ha un puntatore a sé stessa (per indicare che è il quel nodo il rappresentante). Ogni nodo dell'albero contiene: un intero/oggetto, un puntatore al padre. **Operazioni**:

- Operazione find(x): risale la lista dei padri di x fino a trovare la radice e restituisce la radice stessa come rappresentante. Il costo è O(n) nel caso pessimo di albero lineare (in pratica una lista)
- Operazione di merge(x,y): si aggancia l'albero radicato in y all'albero alla radice dell'albero x modificando il puntatore al padre della radice di y. Il costo è O(1) (non consideriamo il costo del find per cercare la radice).

12.3.3 Euristiche per migliorare la realizzazione

Definizione algoritmo euristico: particolare tipo di algoritmo progettato per risolvere il problema più velocemente, qualora i metodi classici siano troppo lenti o per trovare una soluzione approssimata qualora i metodi classi siano troppo lenti. Applicheremo delle euristiche ai metodi find e merge visti finito ad ora per migliorare la complessità delle due realizzazioni.

Euristiche per MFSET:

- Euristica del peso (applicata alle liste): è una strategia per diminuire il costo dell'operazione merge. Consiste nel memorizzare nelle liste l'informazione sulla loro lunghezza e nel momento del merge agganciare la lista più corta a quella più lunga in modo da modificare meno riferimenti al rappresentante. La lunghezza della lista può essere salvata e acceduta in tempo costante. Tramite analisi ammortizzata il costo di n-1 operazioni di merge è $\mathcal{O}(nlogn)$ e il costo delle singole operazioni è $\mathcal{O}(logn)$
- Euristica del rango (applicata agli alberi): ogni nodo mantiene informazioni sul proprio rango, cioè rank[x] del nodo x è il numero di archi del cammino più lungo tra x e una delle foglie che discendono da x (rango = altezza del sottoalbero associato al nodo). Il nostro scopo è usare il rango mantenere bassa l'altezza degli alberi. Se i due alberi hanno rango uguale si aggancia indifferentemente un albero alla radice dell'altro e il rango dell'albero finale cresce di 1. Se gli alberi hanno rango diverso si aggancia l'albero con rango più basso all'albero con rango più alto e l'altezza rimane inalterata. Possiamo dimostrare che (vedi slide) un albero MFSET con n nodi ha altezza inferiore a log n, quindi l'operazione find(x) ha costo $\mathcal{O}(logn)$
- Euristica della compressione dei cammini (applicata agli alberi): ogni volta che viene eseguita la find su un particolare albero, l'albero viene "appiattivo" in modo che le successive find sullo stesso albero siano svolte in O(1) (l'appiattimento viene eseguito solo in caso di modifica dell'albero con la merge). Questo è possibile costruendo un albero in cui tutti i nodi siano abbiano come padre direttamente il rappresentante e l'albero abbia altezza 1.

Applicando entrambe le euristiche sugli alberi il rango non è più l'altezza reale, ma un limite superiore all'altezza del nodo (il rango corretto sarebbe troppo costoso da mantenere). Il costo di m operazioni di find in un insieme di m elementi usando gli alberi con le varie euristiche è $\mathcal{O}(m \cdot \alpha(n))$, con $\alpha(n)$ funzione inversa di Ackermann che cresce molto lentamente. Il costo ammortizzato di una singola operazione è quindi O(1).

Implementazione: Implementazione della struttura MFSET usando gli alberi e le varie euristiche presentante sopra.

Algorithm 52 Implementazione MFSET

```
MFSET
        int[] parent
        int[] rank

MFSET Mfset(int n)
        MFSET t = new MFSET
        t.parent = int[1...n]
        t.rank = int[1...n]
        for i = 1 to n do
                t.parent[i] = i
                t.rank[i] = 0
        return t

merge(int x, int y)
        rx = find(x)
        ry = find(y)
        if rx != ry then
                if rank[rx] > rank[ry] then
                        parent[ry] = rx
                else if rank[ry] > rank[rx] then
                        parent[rx] = ry
                else
                        parent[rx] = ry
                        rank[ry] = rank[ry] + 1

int find(int x)
        if parent[x] != x then
                parent[x] = find(parent[x])
        return parent[x]
```

Complessità: Nella tabella sotto vengono riportati i costi degli algoritmi find e merge degli insieme disgiunti. I costi con l'asterisco (*) sono stati calcolati con l'analisi ammortizzata. I costi di merge non tengo costo del costo necessario ad eseguire le 2 find ma assumono di avere già i rappresentati dei due insiemi coinvolti.

Algoritmo	find()	merge()
Liste	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Alberi	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Liste + Euristica sul peso	$\mathcal{O}(1)$	$\mathcal{O}(logn)$*
Alberi + Euristica sul rango	$\mathcal{O}(logn)$	$\mathcal{O}(1)$
Alberi + Euristica sul rango + Compressione cammini	$\mathcal{O}(1)$*	$\mathcal{O}(1)$

12.3.4 Esempio applicazione MFSET - Componenti connesse dinamiche

Problema: Questa struttura dati può essere trovare le componenti connesse di un grafo non orientato dinamico.

Implementazione:

Algorithm 53 Componenti connesse con MFSET

```
1 MFSET cc(Graph G)
2         %n = numero di nodi del grafo
3         MFSET M = Mfset(G.n)
4         foreach u ∈ G.V() do
5                 foreach v ∈ G.adj(u) do
6                         M.merge(u,v)
7         return M
```

Funzionamento: si inizia con componenti connesse costituite da un unico vertice del grafo. Per ogni arco (u,v) $\in$ E, si esegue merge(u,v) per creare un unico insieme che contiene entrambi i lati dell'arco (visto che appartengono alla stessa componente connessa). Alla fine ogni insieme disgiunto rappresenta una componente connessa. Nodi appartenenti a componente connesse non saranno nello stesso insieme in quanto non ci sono archi che collegano i vari nodi su cui eseguire la merge.

Complessità: O(n) + m operazioni di merge(), che si solito ha costo O(1) costante e la complessità risultante sarebbe (On+m). Questo algoritmo è preferibile rispetto a quello visto fino ad ora quando il grafo che dobbiamo gestire è dinamico (possiamo aggiungere anche su quel grafo) in quando con questa versione basterebbe eseguire una merge() sull'arco aggiunto, mentre con la versione classica bisognerebbe eseguire da capo l'algoritmo.

13 Divide-et-impera

13.1 QuickSort

Problema: Dato un vettore di ITEM A[1...n] e una coppia di indici start e end tali che 1 $\leq$ start $\leq$ end $\leq$ n, ordina il vettore A[i...n] in modo crescente. ITEM è un tipo di dato in cui è definito il confronto tra elementi.
Soluzione: l'algoritmo utilizza la tecnica divide-et-impera e si compone di due funzioni: la funzione pivot() si occupa, dato un sotto vettore, di definire un perno e di spostare a sinistra del perno tutti gli elementi più piccoli del perno stesso e destra gli elementi più grandi, mentre la funzione QuickSort() si occupa di richiamare la funzione in modo ricorsivo su sotto array di dimensione minore.

Algorithm 54 Quicksort

```
1  int pivot(ITEM A[], int start, int end)
2          ITEM P = A[start]
3          int j = start
4          for i = start+1 to end do
5                  if A[i]<p then
6                          j = j+1
7                          A[i] ↔ A[j]
8          A[start] = A[j]
9          A[j] = p
10         return j
11
12 void QuickSort(ITEM A[], int start, int end)
13         if start < end then
14                 int j = pivot(A, start, end)
15                 QuickSort(A, start, j-1)
16                 QuickSort(A, j+1, end)
```

Costo computazionale:

- Pivot: $\Theta(n)$
- QuickSort: si comporta in modo diverso in base all'ordine degli elementi. Nel caso pessimo vale $\Theta(n^2)$, nel caso ottimo $\Theta(n\ log\ n)$, mentre il caso medio è difficile da calcolare.

Specifica e assunzioni: Metodi per la gestione della struttura. Assumiamo di memorizzare nella struttura degli interi e non degli oggetti e prevediamo che esista una associazione intero-oggetto memorizzata esternamente.

13.2 Gap

Problema: Dato un vettore V di interi contenente n $\geq$ 2 elementi e tale che V[1] < V[n], trovare la posizione di un gap nel vettore. In un vettore V contente almeno 2 interi, un gap è un indice i tale che 1 < i $\leq$ n, tale che V[i-1] < V[i]. Le condizioni poste nel problema di avere almeno 2 elementi e che valga V[1] < V[n], garantiscono che ci sia almeno un gap.
Soluzione: implementazione ricorsiva con tecnica divide-et-impera. Uso una funzione wrapper per avere a disposizione più parametri.

Algorithm 55 Gap

```
1  int gap(int[] V, int n)
2          return gapRect(V, 1, n)
3
4
5  int gapRec(int[] V, int i, int j)
6          if j == i+1 then
7                  return j
8          m = floor((i+j)/2)
9          if V[m] < V[j] the
10                 return gapRec(V, m, j)
11         else
12                 return gapRec(V, i, m)
```

14 Regole Matematiche

14.1 Sommatorie

Serie Geometrica Finita: $\forall x \neq 1 : \sum_{j=0}^{k} x^j = \frac{x^{k+1}-1}{x-1}$

Serie Geometrica Infinita Decrescente: $\forall$x, $\mid x \mid < 1 : \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

14.2 Cambiamento di Base

$\log_b n = (\log_b a)(\log_a n) \implies a^{\log_b n} = n^{\log_b a}$

15 Trattare le matrici quadrate come Grafi

Attenzione! In questa sezione si vogliono elencare delle tecniche per trattare una matrice quadrata come grafi.
Tuttavia gli schemi riportati in seguito nelle sottosezioni successive non forniscono delle soluzioni già pronte, ma vanno rielaborate a seconda dell'esercizio (essendo infatti degli schemi guida).

15.1 Costruire un grafo a partire da una matrice

Schema generico della costruzione di un grafo a partire da una matrice quadrata:

Algorithm 56 Schema costruzione Grafo da una matrice

```
1  GRAPH schemaCostruisciGrafo(int[][] A, int n)
2          GRAPH G = new GRAPH
3          % inserimento nodi (tanti quante le celle della matrice)
4          for i = 1 to n · n do
5                  G.insertNode(i)
6          % inserimento archi
7          for riga = 1 to n do
8          for colonna = 1 to n do
9                  % id sequenziale del nodo
10                 int idNodo = colonna + (riga − 1) · n
11                 if riga ≠ 1 then
12                         int idAdiacente = colonna + (riga − 2) · n
13                         G.insertEdge(idNodo, idAdiacente)
14                 if colonna ≠ 1 then
15                         int idAdiacente = (colonna − 1) + (riga − 1) · n
16                         G.insertEdge(idNodo, idAdiacente)
17                 if colonna ≠ n then
18                         int idAdiacente = (colonna + 1) + (riga − 1) · n
19                         G.insertEdge(idNodo, idAdiacente)
20                 if riga ≠ n then
21                         int idAdiacente = colonna + riga · n
22                         G.insertEdge(idNodo, idAdiacente)
23         return G
```

15.2 Visita di una matrice in DFS

Algorithm 57 Generica visita DFS ricorsiva in una matrice quadrata

```
1  dfsRec(int[][] M, int n, int r, int c)
2          if 1 ≤ r < n and 1 ≤ c < n and M[r][c] > 0 then
3          % Visita della cella della matrice
4          M[r][c] = 0      %  Visited
5          % chiamate ricorsive
6          dfsRec(M, n, r − 1, c)
7          dfsRec(M, n, r + 1, c)
8          dfsRec(M, n, r, c − 1)
9          dfsRec(M, n, r, c + 1)
10
11 procedura(int[][] M, int n)
12         for r = 1 to n do
13         for c = 1 to n do
14                 if M[r][c] > 0 then
15                         dfsRec(M, n, r, c)
```

Si osservi che nella precedente visita in DFS la visita del nodo deve avvenire **sempre** nei limiti consentiti dalla matrice ed eventualmente una **condizione**, che nel caso illustrato in precedenza è $M[r][c] > 0$.
Successivamente per non iterare all'infinito è sufficiente settare, non appena il nodo viene visitato un flag visited, questo è possibile farlo con una matrice costruita ad hoc in precedenza di booleani oppure modificando il valore della cella dopo averla visitata, in modo da rendere falsa la condizione di visita. Nel caso precedente è stato scelto: $M[r][c] = 0$ che contraddice $M[r][c] > 0$. La chiamata ricorsiva infine va propagata su tutti e 4 i lati (destra, sinistra, in alto ed in basso), non dobbiamo preoccuparci di sforare gli indici in quanto è già previsto dalla visita in DFS.
La funzione procedura invece è il wrapper che chiamerà ricorsivamente la dfs su tutti i nodi non visitati, ancora una volta, lo statement $M[r][c] > 0$ è solo a scopo illustrativo, la condizione dipenderà esclusivamente dal problema, inoltre se si è più comodi è possibile, per non visitare nuovamente i nodi, utilizzare una matrice di supporto booleana in cui si segna se un nodo è stato o meno visitato. In questo esempio si è deciso di farlo rendendo falsa la condizione di visita non appena il nodo viene visitato.
Un altro esempio di visita DFS è il seguente

Algorithm 58 DFS generica con return di valore

```
% esempio tenere il valore massimo
int procedura(ITEM[][]A, int n)
        int maxsofar = 0
        for r = 1 to n do
        for c = 1 to n do
                if A[r, c] = condizione then
                        maxsofar = max(maxsofar, dfs(A, r, c, n))
        return maxsofar

int dfs(ITEM[][]A, int r, int j, int n)
        if 1 ≤ r ≤ n and 1 ≤ c ≤ n and A[r, c] = condizione then
        A[r, c] = ¬ condizione
        % propagazione con somma di un determinato valore
        return valore + dfs(A, r − 1, c, n) + dfs(A, r, c − 1, n) + dfs(A, r +
                1, c, n) + dfs(A, r, c + 1, n)
        else
        return 0
```

L'esempio precedente riporta una semplice DFS con i principi illustrati in precedenza. Tale BFS ritorna un valore in base ad una certa condizione, la procedura ne ritorna il massimo in tutta la matrice quadrata vista come grafo.

15.3 Visita di una matrice in BFS

Algorithm 59 Schema BFS con Matrice quadra

```
int grid(int[][] M, int n)
        int[] dr = [−1, 0, +1, 0] % Mosse possibili sulle righe
        int[] dc = [0, −1, 0, +1] % Mosse possibili sulle colonne
        int[] distance = new int[1...n][1...n]
        QUEUE Q = QUEUE
        for r = 1 to n do
        for c = 1 to n do
                distance[r][c] = iif(M[r][c] == 1, 0, −1)
                if M[r][c] == 1 then
                        Q.enqueue(<r, c>)
        while not Q.isEmpty() do
        int, int r, c = Q.dequeue()   % Riga, colonna della cella corrente
        for i = 1 to 4 do
                nr = r + dr[i] % Nuova riga
                nc = c + dc[i] % Nuova colonna
                if 1 ≤ nr ≤ n and 1 ≤ nc ≤ n and distance[nr][nc] < 0 then
                        distance[nr][nc] = distance[r][c] + 1
                        if M[nr][nc] == destinazione then
                                return distance[nr][nc]
                        else
                                Q.enqueue(<nr, nc>)
```

L'esempio precedente è stato fatto seguendo il principio di un algoritmo di *Erdos* sulla matrice. Questo schema è liberamente modificabile, si ricordi tuttavia di settare i nodi come visitati, in modo da non inserirli più volte nella coda, in questo esempio è stato utilizzato un vettore delle distanze, ma come prima è possibile modificare le celle vistitate in modo da rendere falsa la condizione di visita oppure utilizzare una matrice di appoggio di booleani che per ogni cella segna se è stata visitata o meno. Si noti infine come è molto importane controllare che le chiamate non abbiano superato gli indici della matrice.

16 Programmazione dinamica e memoization

16.1 Programmazione dinamica

Introduzione Tecnica classica di risoluzione dei problemi che consiste nello spezzare il problema ricorsivamente in sottoproblemi. A differenza della tecnica del divide-et-impera, ogni sotto problema viene risolto solo una volta e la sua soluzione viene memorizzata in una tabella. Nel caso bisognasse risolvere nuovamente il sottoproblema, si ottiene la soluzione facendo un semplice accesso alla tabella senza effettuare di nuovo il calcolo. La tabella ha un costo di lookup pari a $\mathcal{O}(1)$.

16.1.1 Domino

Problema: Il gioco del domino è basato su tessere di dimensione 2×1. Scrivere un algoritmo efficiente che prenda in input un intero n e restituisca il numero di possibili disposizioni di n tessere in un rettangolo $2 \times n$.

Soluzione: Iniziamo a considerare alcuni casi per definire una soluzione. Con n=0 abbiamo solo una disposizione possibile (nessuna tessera), con n=1 abbiamo solo una disposizione possibile (tessera messa in verticale). L'ultima tessera può essere disposta in due modi: se la posiziono in verticale devo risolvere il problema con dimensione n-1, mentre se la metto in orizzontale devo per forza mettere 2 tessere e risolvere il problema di dimensione n-2. In generale la soluzione ha la seguente forma:

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ DP[n-2] + DP[n-1] & n > 1 \end{cases}$$

Notiamo che il risultato è la sequenza di Fibonacci: 1, 1, 2, 3, 5, 8, ... Proponiamo 3 soluzioni al problema:

- domino1(): soluzione ricorsiva che non fa uso della programmazione dinamica e risolve più volte gli stessi sottoproblemi
- domino2(): soluzione con programmazione dinamica che memorizza il risultato in una tabella DP. L'approccio è bottom-up cioè si parte risolvendo i casi base e risalendo si risolvono problemi sempre più grandi. La tabella DP in questo caso memorizza il risultato per ogni sottoproblema con dimensione tra 0 e n.
- domino3(): la soluzione proposta è uguale alla precedente con la differenza che l'uso di memoria è ridotto salvando solo la soluzione degli ultimi 3 sottoproblemi.

Implementazione:

Algorithm 60 Domino/Fibonacci versione $\mathcal{O}(2^n)$

```
int domino1(int n)
        if n ≤ 1 then
                return 1
        else
                return domino1(n − 1) + domino1(n − 2)
```

Algorithm 61 Domino/Fibonacci versione $\mathcal{O}(n^2)$ in tempo e spazio

```
int domino2(int n)
        DP = new int[0...n]
        DP[0] = DP[1] = 1
        for i = 2 to n do
                DP[i] = DP[i − 1] + DP[i − 2]
        return DP[n]
```

Algorithm 62 Domino/Fibonacci versione $\mathcal{O}(n^2)$ in tempo $\mathcal{O}(n)$ e spazio

```
int domino3(int n)
        int DP_0 = 1
        int DP_1 = 1
        int DP_2 = 1
        for i = 2 to n do
                DP_0 = DP_1
                DP_1 = DP_2
                DP_2 = DP_0 + DP_1
        return DP_2
```

Complessità: Per calcolare la complessità dobbiamo usare il modello di costo logaritmico che tiene consto lo spazio necessario per salvare un numero della sequenza di Fibonacci ($\Theta(n)$) e il costo per sommare due numeri consecutivi della sequenza ($\Theta(n)$). Il costo per le varie versioni è il seguente:

Funzione	Complessità (Tempo)	Complessità (Spazio)
domino1()	$\mathcal{O}$(n2^n)	$\mathcal{O}(n^2)$
domino2()	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
domino3()	$\mathcal{O}(n^2)$	$\mathcal{O}$(n)

16.1.2 Hateville

Problema: Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi, il vicino i odia i vicini i-1 e i+1 (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante i ha intenzione di donare una quantità D[i], ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini. Scrivere un algoritmo che restituisca il sottoinsieme di indici S $\subseteq$ {1, ..., n } tale per cui la donazione totale T $= \sum_{i \in S} D[i]$ è massimale.

Soluzione: Definiamo il problema in questi termini:

- Sia HV(i) uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale dalle prime i case di Hateville, numerate 1...n

- HV(n) è la soluzione del problema originale
- Se non accetto la donazione i-esima: HV(i) = HV(i-1)
- Se accetto la donazione i-esima: HV(i) = i ∪ HV(i-2)
- Per decidere se accettare una donazione i: HV(i) = highest(HV(i -1), {i} ∪ HV(i-2))
- Casi base: HV(0) = ∅ e HV(1) = {1}

Mettendo tutto insieme otteniamo:

$$HV(i) = \begin{cases} \emptyset & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), \{i\} \cup HV(i-2)) & i \geq 2 \end{cases}$$

Per riuscire a calcolare l'insieme delle soluzioni dobbiamo prima calcolare il valore della soluzione massimale usando la programmazione dinamica e poi ricostruiamo la soluzione partendo da questo valore. La ricorrenza per trovare il valore massimale con la programmazione dinamica è la seguente:

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ max(DP[i-1], DP[i-2] + D[i]) & i \geq 2 \end{cases}$$

Implementazione:

Algorithm 63 Hateville versione $\mathcal{O}(n)$ senza calcolo della soluzione

```
int hateville(int[] D, int n)
        int[] DP = new int[0...n]
        DP[0] = 0
        DP[1] = D[1+
        for i = 2 to n do
                DP[i] = max(DP[i - 1], DP[i - 2] + D[i])
        return DP[n]
```

Algorithm 64 Soluzione di Hateville versione $\mathcal{O}(n)$

```
SET solution(int[] DP, nit[] D, int i)
        if i == 0 then
                return ∅
        else if i == 1 then
                return {1}
        else if DP[i] == DP[ i - 1]  then
                return solution(DP, D, i  - 1)
        else
                SET sol = solution(DP, D, i - 2)
                sol.insert(i)
                return sol
```

Algorithm 65 Hateville versione $\mathcal{O}(n)$ completa

```
int hateville(int[] D, int n)
        int[] DP = new int[0...n]
        DP[0] = 0
```

```
4         DP[1] = D[1+
5         for i = 2 to n do
6                 DP[i] = max(DP[i - 1], DP[i - 2] + D[i])
7         print DP[n]
8         return solution(DP, D, n)
```

Complessità:

- Complessità solution(): $\Theta(n)$
- Complessità hateville() (sia per tempo che per spazio): $\Theta(n)$

16.1.3 Zaino (Knapsack)

Problema: Dato un insieme di oggetti, ognuno caratterizzato da un peso e un profitto, e uno "zaino" con un limite di capacità, individuare un sottoinsieme di oggetti:

- il cui peso sia inferiore alla capacità dello zaino;
- il valore totale degli oggetti sia massimale, i.e. più alto o uguale al valore di qualunque altro sottoinsieme di oggetti

Siano dati in input:

- Vettore w, dove w[i] è il peso (weight) dell'oggetto i-esimo
- Vettore p, dove p[i] è il profitto (profit) dell'oggetto i-esimo
- La capacità C dello zaino

L'output deve essere un insieme S $\subseteq$ {1, ..., n } tale che:

- Il volume totale deve essere minore o uguale alla capacità
- Il profitto totale deve essere massimizzato

Soluzione: Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo DP[i][c] come il massimo profitto che puòessere ottenuto dai primi i $\leq$ n oggetti contenuti in uno zaino di capacità c $\leq$ C. Il massimo profitto ottenibile per il problema originale è rappresentato da DP[n][C]. Considerando i vari casi (prendere o meno l'ultimo elemento e i vari casi base) otteniamo la seguente formula ricorsiva:

$$DP[i][c] = \begin{cases} 0 & i = 0 \text{ or } c = 0 \\ -\infty & c < 0 \\ max(DP[i-1][c - w[i]] + p[i], DP[i-1][c]) & altrimenti \end{cases}$$

Implementazione:

Algorithm 66 Knapsack versione $\mathcal{O}(nC)$ con programmazione dinamica

```
int knapsack(int[] w, int[] p, int n, int C)
        int[][] DP = new int[0...n][0...n]
        for i = 0 to n do
                DP[i][0] = 0
        for c = 0 to n do
                DP[0][c] = 0
        for i = 1 to n do
                for c = 1 to C do
                        if w[i] ≤ c then
                                DP[i][c] = max(DP[i − 1][c − w[i]] + p[i], DP[i
                                    − 1][c])
                        else
                                        DP[i][c] = DP[i − 1][c]
        return DP[n][C]
```

Complessità: La complessità dell'algoritmo è $\mathcal{O}$(nC), ma se consideriamo che sono necessari k = log C bit per rappresentare C, la complessità diventa pseudo-polinomiale e vale T(n) = $\mathcal{O}$(n2^k). La versione ricorsiva che non fa uso della ricorsione costa $\mathcal{O}(2^n)$. Questo è un problema NP completo.

16.2 Memoization

Introduzione Non tutte le soluzioni dei sottoproblemi possibili sono utili alla soluzione del problema originale e quindi non vanno calcolate e memorizzate. Questa tecnica fonde l'approccio di memorizzazione della programmazione dinamica con l'approccio top-down di divide et impera. Passi per utilizzare la memoization:

- Creare la tabella DP e inizializzarla con un valore speciale (+/- infinito, -1, null) per indicare che un certo sottoproblema non è stato ancora risolto
- Ogni volta che si deve risolvere un sottoproblema, si controlla nella tabella se è già stato risolto precedentemente: in caso affermativo si usa il risultato della tabella, altrimenti si calcola il risultato e lo si memorizza
- Ogni sottoproblema viene calcolato una sola volta e memorizzato come nella versione bottom-up

16.2.1 Zaino (Knapsack)

Problema: Il problema è lo stesso visto per la programmazione dinamica.
Implementazione:

Algorithm 67 Knapsack versione $\mathcal{O}(nC)$ zaino con memoization

```
int knapsack(int[] w, int[] p, int n, int C)
        int[][] DP = new int[1...n][1...n]
        for i = 1 to n do
                for c = 1 to C do
                        DP[i][c] = −1
        return knapsackRec(w, p, n, C, DP )


```

```
int knapsackRec(int[] w, int[] p, int i, int c, int[][] DP)
        if c < 0 then
                return −∞
        else if i == 0 or c == 0 then
                return 0
        else
                if DP[i][c] < 0 then
                        int not_taken = knapsackRec(w, p, i − 1, c, DP)
                        int taken = knapsackRec(w, p, i − 1, c − w[i], DP) + p[i
                            ]
                        DP[i][c] = max(not_taken, taken)
        return DP[i][c]
```

Algorithm 68 Knapsack versione con memoization e dizionario

```
int knapsack(int[] w, int[] p, int n, int C)
        DP = new Hash
        return knapsackRec(w, p, n, C, DP )

int knapsackRec(int[] w, int[] p, int i, int c, Hash DP)
        if c < 0 then
                return −∞
        else if i == 0 or c == 0 then
                return 0
        else
                if (i, c) ∈ DP.keys() then
                        int not_taken = knapsackRec(w, p, i − 1, c, DP)
                        int taken = knapsackRec(w, p, i − 1, c − w[i], DP) + p[i
                            ]
                        DP.insert((i, c), max(not_taken, taken))
        return DP.lookup((i, c))
```

Complessità: La complessità dell'algoritmo è $\mathcal{O}(\text{nC})$, quindi non sembrerebbe esserci nessun vantaggio se non la semplicità di convertire codice ricorsivo in memoization. Utilizzando l'hash al posto di una matrice per DP non è necessario fare l'inizializzazione e il costo di esecuzione diventa $\mathcal{O}($ min $(2^n$, nC)).

16.2.2 Zaino (Knapsack) senza limiti

Problema: Il problema è lo stesso visto nei casi precedenti con la modifica che non si pone limite al numero di volte che un oggetto può essere selezionato. Come modificare dunque la formula ricorsiva in questo caso? Potrebbe venirci in mente una soluzione basata sulla programmazione greedy ma per come è espresso il problema siamo certi del fatto che esista una soluzione basata su programmazione dinamica.

$$DP[i][c] = \begin{cases} 0 & i = 0 \text{ or } c = 0 \\ -\infty & c < 0 \\ max(DP[i-1][c - w[i]] + p[i], DP[i-1][c]) & altrimenti \end{cases}$$

Se rimuoviamo il -1 che troviamo nell'equazione di ricorrenza vista nei casi precedenti, ricadiamo nel caso in cui ci chiediamo se ci conviene prendere lo stesso oggetto più volte, pur avendo una capacità minore, oppure se ci conviene non prendere l'oggetto e passare a un altro.
Al posto che modificare la formula precedente, a cui basterebbe come detto precedentemente rimuovere il -1, proviamo a pensare a una soluzione più efficiente a livello di spazio occupato in memoria.

Dato dunque uno zaino senza limiti di scelta di capacità C e n oggetti caratterizzati da un peso w e da un porfitto p, definiamo $DP[c]$ come il massimo profitto che può essere ottenuto da tali oggetti in uno zaino di capacità $c \leq C$

Se pensiamo al fatto che possiamo prendere un numero arbitrario di volte un qualsiasi oggetto, pur rimanendo vincolati dalla capacità residua dello zaino, questo significa che con capacità restante c, ci converrà prendere l'oggetto con profitto massimo ($max(p[i])$) con $w[i] \leq c$.

$$DP[i][c] = \begin{cases} 0 & c = 0 \\ max_{w[i] \leq c}\{DP[c - w[i]] + p[i]\} & c > 0 \end{cases}$$

Implemetazioni

Algorithm 69 Knapsack zaino senza limiti

```
1  int knapsack(int[] w, int[] p, int n, int C)
2          int[] DP = new int[0...C]
3          for i = 0 to C do
4                  DP[i] = -1
5          knapsackRec(w, p, n, C, DP)
6          return DP[C]
7
8  int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)
9          if c == 0 then
10           return 0
11         if DP[c] < 0 then
12           DP[c] = 0
13           for i = 1 to n do
14             if w[i] ≤ c then
15                   int val = knapsackRec(w, p, n, c - w[i], DP) + p[i]
16                   DP[c] = max(DP[c], val)
17         return DP[c]
```

Qual è la complessità della funzione knapsack()?
La funzione knapsack deve generare un vettore grande C e, nel caso peggiore, dovrà riempire tutti i suoi elementi. Riempire un singolo elemento ha costo $\mathcal{O}(n)$ e quindi il costo complessivo della funzione è $\mathcal{O}(Cn)$.

Se utilizziamo questo tipo di approccio lo spazio occupato in memoria è pari a $\Theta(C)$ ma ricostruire la soluzione diventa più complesso in quanto è necessario l'utilizzo di un altro vettore che ci dica da dove venga il massimo.

$$DP[i][c] = \begin{cases} 0 & c = 0 \\ max_{w[i] \leq c}\{DP[c - w[i]] + p[i]\} & c > 0 \end{cases}$$

Algorithm 70 Knapsack Ricostruzione Soluzione

```
1  LIST knapsack(int[] w, int[] p, int n, int C)
2          int[] DP = new int[0...C]
3          int[] pos = new int[0...C]
4          for i = 0 to C do
5                  DP[i] = -1
6                  pos[i] = -1
7          knapsackRec(w, p, n, C, DP, pos)
8          return solution(w, C, pos)
9
10 int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)
11         if c == 0 then
12           return 0
13         if DP[c] < 0 then
14           DP[c] = 0
15           for i = 1 to n do
16             if w[i] ≤ c then
17                    int val = knapsackRec(w, p, n, c - w[i], DP, pos) + p[i]
18                    if val ≥ DP[c] then
19                      DP[c] = val
20                      pos[c] = i
21         return DP[c]
22
23 LIST solution(int[] w, int c, int[] pos)
24         if c==0 or pos[c] < 0 then
25                 return List()
26         else
27                 LIST L = solution(w, c - w[pos[c]], pos)
28                 L.insert(L.head(), pos[c])
29                 return L
```

Quello che viene restituito dalla funzione solution() è una lista di indici dove i suoi elementi possono comparire più di una volta (multinsieme). Notiamo che se $c = 0$ allora lo zaino è stato riempito perfettamente mentre per $pos[c] < 0$ allora lo zaino non può essere riempito interamente (rimarrà della capacità residua ma nessun oggetto in grado di riempirlo perfettamente).

16.3 Altri esempi di programmazione dinamica

16.3.1 Sottosequenza comune massimale

Definizione: Sottosequenza
Una sequenza P è una sottosequenza di T se P è ottenuto da T rimuovendo uno o più dei suoi elementi.

Definizione: Sottosequenza comune
Una sequenza X è una sottosequenza comune di due sequenze T, U se è sottosequenza sia di T che di U.
Scriveremo $X \in \mathcal{CS}(T, U)$

Definizione: Sottosequenza comune
Una sequenza $X \in \mathcal{CS}(T, U)$ è una sottosequenza comune massimale di due sequenze T, U se non esiste un'altra sottosequenza $Y \in \mathcal{CS}(T, U)$ tale che Y sia più lunga di X ($|Y| > |X|$).
Scriveremo $X \in \mathcal{LCS}(T, U)$

Problema: $\mathcal{LCS}$
Date due sequenze T e U, trovare la più lunga sottosequenza comune tra le due.

Per poter descrivere la soluzione matematica ottima abbiamo ancora bisogno di un'altra definizione:
Prefisso: Data una sequenza T composta da $t_1t_2...t_n$, $T(i)$ denota il prefisso dei primi i caratteri cioè: $T(i) = t_1t_2...t_i$

Analisi dei casi ricorsivi

- **Caso 1:**
  Si considerino due prefissi $T(i)$ e $U(j)$ tali per cui il loro ultimo carattere coincide: $t_i = u_j$.
  es: $T(i) = ALBERTO$, $U(j) = PIERO$.
  Soluzione
  $LCS(T(i), U(j)) = LCS(T(i-1), U(j-1)) \oplus t_i$
  nel caso dell'esempio mostrato sopra:
  $LCS(ALBERTO, PIERO) = LCS(ALBERT, PIER) \oplus \text{O}$

- **Caso 2:**
  Si considerino due prefissi $T(i)$ e $U(j)$ tali per cui il loro ultimo carattere è differente: $t_i \neq u_j$.
  es: $T(i) = ALBERT$, $U(j) = PIER$.
  Soluzione
  $LCS(T(i), U(j)) = longest(LCS(T(i-1), U(j)), LCS(T(i), U(j-1)))$
  nel caso dell'esempio mostrato sopra:
  $LCS(ALBERTO, PIERO) = longest(LCS(ALBER, PIER), LCS(ALBERT, PIE))$

Casi Base
La più lunga sottosequenza tra $T(i)$ e $U(j)$ quando uno dei due è vuoto è $LCS(T(i), U(0)) = \emptyset$

Formula Completa

$$LCS(T(i), U(j)) = \begin{cases} \emptyset & i = 0 \text{ or } j = 0 \\ LCS(T(i-1), U(j-1)) \oplus t_i & i > 0 \text{ and } j > 0 \text{ and } t_i = u_j \\ longest(LCS(T(i-1), U(j)), LCS(T(i), U(j-1))) & i > 0 \text{ and } j > 0 \text{ and } t_i \neq u_j \end{cases}$$

Lunghezza della LCS
Date due sequenze T e U di lunghezza n e m, scrivere una formula ricorsiva $DP[i][j]$ che restituisca la lunghezza della LCS dei prefissi $T(i)$ e $U(j)$.

$$DP[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ DP[i-1][j-1] + 1 & i > 0 \text{ and } j > 0 \text{ and } t_i = u_j \\ max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \text{ and } j > 0 \text{ and } t_i \neq u_j \end{cases}$$

L'informazione relativa alla lunghezza della LCS del problema si troverà in $DP[n][m]$.

Esempio funzionamento soluzione e struttura dati
La tabella seguente mostra la struttura dati che sarà presente in memoria dopo aver eseguito l'algoritmo sulle stringhe ATBCBD e TACCBT. La freccia che punta verso il basso vuol dire che l'elemento è stato ricavato dall'elemento presente sulla stessa colonna della tabella, ma sulla riga precedente, la freccia che punta verso destra che l'elemento proviene dalla stessa riga ma

dalla colonna precedente, mentre la freccia diagonale che l'elemento è stato calcolato a partire dall'elemento sulla riga e sulla colonna precedente.

	j	0	1	2	3	4	5	6
i			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	↓ 0	↘ 1	→ 1	→ 1	→ 1	→ 1
2	A	0	↘ 1	↓ 1	↓ 1	↓ 1	↓ 1	↓ 1
3	C	0	↓ 1	↓ 1	↓ 1	↘ 2	→ 2	→ 2
4	C	0	↓ 1	↓ 1	↓ 1	↘ 2	↓ 2	→ 2
5	B	0	↓ 1	↓ 1	↘ 2	↓ 2	↘ 3	→ 3
6	T	0	↓ 1	↘ 2	↓ 2	↓ 2	↓ 3	↓ 3

Il risultato dell'esempio lo troviamo nella tabella in posizione [6,6] e vale 3. L'insieme delle soluzioni (cioè le lettere appartenenti alla sottosequenza massima) possono essere ricavate seguendo le frecce nella presenti nella struttura dati.

Implementazione ricorrenza

Algorithm 71 Lunghezza LCS

```
1  int lcs(int[] T, int[] U, int n, int m)
2          int[][] DP = new int[0...n][0...m]
3          for i = 0 to n do
4                  for j = 0 to m do
5                          if i = 0 or j = 0 then
6                                  DP[i][j] = 0
7          for i = 1 to n do
8                  for j = 1 to m do
9                          if T[i] == U[j] then
10                                 DP[i][j] = DP[i-1][j-1] + 1
11                         else
12                                 DP[i][j] = max(DP[i-1][j], DP[i][j-1])
13         return DP[n][m]
```

Nel caso in cui si voglia invece restituire la sottosequenza comune è invece necessario usare un algoritmo di questo tipo:

Algorithm 72 Ricostruire LCS

```
1  LIST lcs(int[] T, int[] U, int n, int m)
2          ...
3          return subsequence(DP, T, U, n, m)
4
5  LIST subsequence(int[][] DP, ITEM[] T, ITEM[] U, int i, int j)
6          if i== 0 or j==0 then
7                  return List()
8          if T[i]==U[j] then
9                  S = subsequence(DP, T, U, i-1, j-1)
10                 S.insert(S.head(), T[i])
11                 return S
12         else
13                 if DP[i-1][j] > DP[i][j-1] then
14                         return subsequence(DP, T, U, i-1, j)
15                 else
16                         return subsequence(DP, T, U, i, j-1)
```

La complessità computazionale di subsequence è al massimo $T(n) = \mathcal{O}(m+n)$ che corrisponde al numero massimo di controlli che verranno eseguiti per ricostituire la soluzione, mentre il costo per creare e riempire la matrice è pari a $T(n) = \mathcal{O}(mn)$. In totale il costo di LCS è pari a $T(n) = \mathcal{O}(mn)$.

16.3.2 String matching approssimato

Definizione: Un'occorrenza k-approssimata di P in T, dove:

- P = p1...pm è una stringa detta pattern
- T = t1...tn è una stringa detta testo, con m $\leq$ n,

è una copia di P in T in cui sono ammessi k "errori" (o differenze) tra caratteri di P e caratteri di T, del seguente tipo:

- i corrispondenti caratteri in P, T sono diversi (sostituzione)
- un carattere in P non è incluso in T (inserimento)
- un carattere in T non è incluso in P (cancellazione)

Problema: Trovare un'occorrenza k-approssimata di P in T con K minimo (0 $\leq$ k $\leq$ m).

Sottostruttura ottima e formula ricorsiva: Sia DP[0...m][0...n] una tabella di programmazione dinamica tale che DP[i][j] sia il minimo valore k per cui esiste un'occorrenza k-approssimata di P(i) in T(j) che termina nella posizione j.

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & i = 0 \\ min(DP[i-1][j-1] + d, DP[i-1][j] + 1, DP[i][j-1] + 1) & altrimenti \end{cases}$$

dove d = iif(P[i] = T[j], 0 , 1).

La formula tratta i vari casi: se P[i]=T[j] avanza su entrambi i caratteri, se P[i]≠T[j] avanza aggiungi 1 e trova la soluzione minima tra avanzare sul pattern (inserimento), avanzare sul testo (cancellazione) e avanzare su entrambi i caratteri (sostituzione).

Implementazione:

Algorithm 73 String matching approssimato

```
int stringMatching(ITEM[] P, ITEM[] T, int m, int n)
        int[][] DP = new int[0...m][0...n]
        for j = 0 to n do DP[0][j] = 0  % Caso base: i = 0
        for i = 1 to m do DP[i][0] = i  % Caso base: j = 0
        for i = 1 to m do               % Caso generale
                for j = 1 to n do
                        DP[i][j] = min( DP[i-1][j-1] + iif(P[i] == T[j], 0, 1),
                                        DP[i-1][j] + 1,
                                        DP[i][j-1] + 1)
        % Calcola minimo ultima riga
        int pos = 0
        for j = 1 to n do
                if DP[m][j] < DP[m][pos] then
                        pos = j
        return pos
```

16.3.3 Prodotto di catena di matrici

Problema: Data una sequenza di n matrici A1, A2, A3... An,compatibili due a due al prodotto, vogliamo calcolare il loro prodotto.
Il prodotto di matrici: non è commutativo, è associativo: (A1 · A2) · A3 = A1 · (A2 · A3) e si basa sulla moltiplicazione scalare come operazione elementare. Vogliamo calcolare il prodotto delle n matrici impiegando il più basso numero possibile di moltiplicazioni scalari.

Definizioni: Una parentesizzazione Pi,j del prodotto Ai · Ai+1 · ... · Aj consiste:

- nella matrice Ai, se i = j;
- nel prodotto di due parentesizzazioni (Pi,k · Pk+1,j), altrimenti.

La parentesizzazione che richiede il minor numero di moltiplicazioni scalari per essere completata, fra tutte le parentesizzazioni possibili.
In ogni parentesizzazione ottima esiste un ultimo prodotto cioè esiste un indice k tale che P[i...j] = P[i...k] · P[k+1...j] dove P[i...j] rappresenta una parentesizzazione per il prodotto Ai · Ai+1 · ... · Aj.
Definiamo inoltre c_{i-1} il numero di righe della matrice Ai e c_i il numero di colonne della matrice Ai.

Sottostruttura ottima e formula ricorsiva: Se P[ij] = P[i...k]P[k+1...j] è una parentesizzazione ottima del prodotto A[i...j], allora:

- P[i...k] è parentesizzazione ottima del prodotto A[i...k]
- P[k+1...j] è parentesizzazione ottima del prodotto A[k+1...j]

Non conosciamo quanto vale k, ma possiamo provare per tutti i valori possibili fra i e j-1. Definizione ricorsiva:

$$DP[i][j] = \begin{cases} 0 & i = j \\ min_{i \leq k \leq j}(DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j) & i < j \end{cases}$$

Implementazione:

Algorithm 74 Calcolo prodotto di matrici e stampa parentesizzazione ottima usando matrice last

```
1  %ricostruzione della soluzione (stampa prodotto)
2  printPar(int[][] last , int i, int j)
3          if i == j then
4                  print "A["; print i; print "]"
5          else
6                  print "("; stampaPar(last,i,last[i][j]); print "·";
7                  printPar(last,last[i][j]+1,j); print ")"
8
9  %calcolo prodotto matrici usando last
10 int[][] multiply(matrix[] A, int[][] last, int i, int j)
11         if i == j then
12                 return A[i]
13         else
14                 int[][] X = multiply(A, last, i, last[i][j])
15                 int[][] Y = multiply(A, last, last[i][j]+ 1, j)
16                 %matrix-multiplication calcola il prodotto tra le  matrici X e Y
17                 return matrix-multiplication(X,Y)
```

L'implementazione usa le matrici DP e last di dimensione n · n tali che:

- DP[i][j] contiene il numero di moltiplicazioni scalari necessarie per moltiplicare le matrici A[i...j]
- last [i][j] contiene il valore k dell'ultimo prodotto che minimizza il costo per il sottoproblema

Algorithm 75 Calcolo parentesizzazione ottima

```
%calcolo della soluzione ottima
computePar(int[] c, int n)
        int[][] DP = new int[1...n][1... n]
        int[][] last = new int[1...n][1...n]
        % Fill main diagonal
        for i = 1 to n do
                DP[i][i] = 0
        % h: diagonal index
        % i: row
        % j: column
        for h = 2 to n do
                for i = 1 to n - h + 1 do
                        int j = i + h - 1
                        DP[i][j] = +1
                        % k: last product
                        for k = i to j - 1 do
                                int temp = DP[i][k]+DP[k+1][j]+c[i-1]·c[k]·c[j]
                                if temp < DP[i][j] then
                                        DP[i][j] = temp
                                        last [i][j] = k
        return DP[1][n]
```

Costo computazionale: Il costo di computePar è pari a $\mathcal{O}(n^3)$.

16.3.4 Insieme indipendente di intervalli pesati

Definizioni e input: Siano dati n intervalli distinti [a1,b1[, ..., [an,bn[della retta reale, aperti a destra, dove all'intervallo i è associato un profitto w_i, 1≤i≤n. Due intervalli i e j sono disgiunti se bi *leq* ai oppure bi ≤ aj.

Problema: Trovare un insieme indipendente di peso massimo, ovvero un sottoinsieme di intervalli disgiunti tra loro tale che la somma dei loro profitti sia la più grande possibile.

Soluzione: Per usare la programmazione dinamica, è necessario effettuare una pre-elaborazione: ordinare gli intervalli per estremi finali crescenti cioè b1 ≤ b2 ≤ ... ≤ bn.
Una seconda possibile pre-elaborazione consiste nel pre-calcolare il predecessore pred_i = j di i, dove:

- j < i è il massimo indice tale che bj ≤ ai
- se non esiste tale indice, $\text{pred}_i = 0$.

Definiamo la ricorrenza:

$$DP[i] = \begin{cases} 0 & i = 0 \\ max(DP[i-1], DP[pred_i] + w_i) & i > 0 \end{cases}$$

dove DP[i] contiene il profitto massimo ottenibile con i primi i intervalli.

Implementazione:

Algorithm 76 Insieme indipendente di intervalli pesati

```
1  SET maxinterval(int[] a, int[] b, int[]w, int n)
2          {ordina gli intervalli per estremi di fine crescenti}
3          int[] pred = computePredecessor(a, b, n)
4          int[] DP = new int[0...n]
5          DP[0] = 0
6          for i = 1 to n do
7                  DP[i] = max(DP[i-1], w[i]+DP[pred[i]])
8          i = n
9          SET S = Set()
10         while i > 0 do
11                 if DP[i-1] > w[i] + DP[pred[i]] then
12                         i = i - 1
13                 else
14                         S.insert(i)
15                         i = pred[i]
16         return S
17
18
19 int[] computePredecessor(int[] a, int[] b, int n)
20         int[] pred = new int[0...n]
21         pred[0] = 0
22         for i = 1 to n do
23                 j = i-1
24                 while j > 0 and b[j] > a[i] do
25                         j = j-1
26                 pred[i] = j
27         return pred
```

Costo computazionale: Il costo per calcolare i predecessori mostrato nel codice è pari a $\mathcal{O}(n^2)$, ma si può fare meglio di così ed arrivare ad un costo $\mathcal{O}$(n log n). In totale il costo di maxinterval è pari a $\mathcal{O}$(n log n) in quanto il calcolo dei predecessori vale $\mathcal{O}$(n log n), l'ordinamento vale $\mathcal{O}$(n log n), il riempimento di DP vale $\mathcal{O}(n)$ e la ricostruzione della soluzione costa $\mathcal{O}(n)$.

16.4 Schema risoluzione problema

Fasi risoluzione problema con programmazione dinamica/memoization:

- Caratterizzare la struttura di una soluzione ottima
- Dimostrare che la soluzione gode di sottostruttura ottima
- Definire ricorsivamente il valore di una soluzione ottima
- Calcolare il valore di una soluzione ottima "bottom-up" (programmazione dinamica) / "top-down" (memoization)
- Ricostruzione di una soluzione ottima

17 Scelta della struttura dati

17.1 Il problema dei cammini minimi a sorgente singola

Il problema dei cammini minimi che andremo ad affrontare prevede un pattern comune per una serie di algoritmi diversi tra loro, dove, cambiando solamente la struttura dati di riferimento, è possibile ottenere soluzioni con caratteristiche e complessità differenti.

Introduzione al problema
In input verrà ricevuto:

- Grafo orientato $G = (V, E)$
- Un nodo sorgente s
- Una funzione di peso $w \mid E \to R$

Dato un cammino $p = (v_1, v_2, .., v_k)$ con $k > 1$, il costo del cammino è dato da

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Ci è richiesto di trovare un cammino da s ad u, per ogni nodo $u \in V$, il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da s ad u.

Nota sui pesi degli archi
Algoritmi differenti possono funzionare o meno in presenza di alcune categorie speciali di pesi:

- Positivi / positivi + negativi
- Reali / Interi

Si noti inoltre che il problema per come è stato posto precedentemente accetta pesi negativi, ma tuttavia non è possibile avere dei cicli la cui somma sia negativa, in quanto non esisterebbe un cammino minimo, dato che per diminuire il costo del cammino è sufficiente ripercorrere tali archi.

Soluzione ammissibile
Una soluzione ammissibile può essere descritta da un albero di copertura T radicato in s e da un vettore di distanze d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T

Il teorema di Bellman
Una soluzione ammissibile T è ottima se e solo se:

- $d[v] = d[u] + w(u, v)$ per ogni arco $(u, v) \in T$
- $d[v] \leq d[u] + w(u, v)$ per ogni arco $(u, v) \in E$

Algoritmo generico
In seguito verrà mostrato un pattern generico per la risoluzione del problema dei cammini minimi che, in base alla struttura dati utilizzata e alle opportune modifiche che verranno apportate alle righe contrassegnate da (1), (2), (3) e (4), non solo cambierà la complessità legata all'algoritmo, ma inoltre risulteranno differenti le proprietà dello stesso, il quale si comporterà in maniera differente in base all'input fornito.

Algorithm 77 Algoritmo generico per il problema dei cammini minimi a sorgente singola

```
(int[],int[]) shortestPath (Graph G, Node s)
        int[] d = new int[1... G.n]              %d[u] := la distanza da s a u
        int[] T = new int[1... G.n]              %T[u] := padre di u in T
        boolean[] b = new boolean[1... G.n]      %b[u] := true se u presente in S
        foreach u ∈ G.V() − {s} do
                T[u] = nil
                d[u] = +∞
                b[u] = false
        T[s] = nil
        d[s] = 0
        b[s] = true
        DataStructure S = DataStructure()         %(1)
        S.add(s)
        while not S.isEmpty() do
                int u = S.extract()      %(2)
                b[u] = false
                foreach v ∈ G.adj(u) do
                        if d[u] + G.w(u, v) < d[v] then
                                if not b[v] then
                                        S.add(v)         %(3)
                                        b[v] = true
                                else
                                        %Azione da svolgere nel caso v sia
                                            presente in S (4)
                                T[v] = u
                                d[v] = d[u] + G.w(u, v)
        return(T, d)
```

Algoritmo di Dijkstra, 1959

Algoritmo che fa uso come struttura dati di una coda a priorità basata su vettore, esso funziona bene solamente con i pesi positivi ed ha un costo computazionale di $\mathcal{O}(n^2)$ per via del metodo $deleteMin()$ che ha una compessità di $\mathcal{O}(n)$, richiamato per ogni nodo del grafo.

Algorithm 78 Algoritmo di Edsger W. Dijkstra: cammini minimi a sorgente singola

```
(int[],int[]) shortestPath (Graph G, Node s)
        int[] d = new int[1... G.n]              %d[u] := la distanza da s a u
        int[] T = new int[1... G.n]              %T[u] := padre di u in T
        boolean[] b = new boolean[1... G.n]      %b[u] := true se u presente in S
        foreach u ∈ G.V() − {s} do
                T[u] = nil
                d[u] = +∞
                b[u] = false
        T[s] = nil
        d[s] = 0
        b[s] = true
        PriorityQueue Q = PriorityQueue();       %Basata su un vettore (1)
        Q.insert(s,0)
        S.add(s)
        while not S.isEmpty() do
                int u = Q.deleteMin()    %Deve scorrrere tutti gli elementi del
                    vettore (2)
                b[u] = false
                foreach v ∈ G.adj(u) do
                        if d[u] + G.w(u, v) < d[v] then
                                if not b[v] then
                                        Q.insert(v, d[u] + G.w(u, v)) %Semplice
                                            modifica del valore con costo
                                            costante (3)
```

```
22                                          b[v] = true
23                                  else
24                                          Q.decrease(v, d[u] + G.w(u, v)) %
                                                Semplice modifica del valore con
                                                costo costante (4)
25                                  T[v] = u
26                                  d[v] = d[u] + G.w(u, v)
27          return(T, d)
```

Importante è osservare che:

- Ogni nodo viene estratto una e una sola volta;
- Al momento dell'estrazione la sua distanza è minima.

Queste due proprietà, che sono dimostrabili per induzione, consentono di verificare la correttezza dell'algoritmo di Dijkstra.
Un altra proprietà degna di nota è di essere ottimale per il calcolo di cammini minimi a sorgente singola per grafi densi, ovvero grafi il cui numero di archi m sia $\Omega(n^2)$, in quanto la complessità di questo non dipende dal numero di archi. Un analisi più dettagliata sulla complessità:

Riga	Costo	Ripetizioni
(1)	$\mathcal{O}(n)$	1
(2)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
(3)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
(4)	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Costo totale: $\mathcal{O}(n^2)$

Algoritmo di Johnson, 1977
Variante dell'algoritmo di Dijkstra che fa utilizzo di una coda a priorità basata su heap binario e non più su vettore.

Algorithm 79 Algoritmo di Johnson: cammini minimi a sorgente singola

```
1  (int[],int[]) shortestPath (Graph G, Node s)
2           int[] d = new int[1... G.n]              %d[u] := la distanza da s a u
3           int[] T = new int[1... G.n]              %T[u] := padre di u in T
4           boolean[] b = new boolean[1... G.n]      %b[u] := true se u presente in S
5           foreach u ∈ G.V() − {s} do
6                   T[u] = nil
7                   d[u] = +∞
8                   b[u] = false
9           T[s] = nil
10          d[s] = 0
11          b[s] = true
12          PriorityQueue Q = PriorityQueue();       %Basata su heap binario(1)
13          Q.insert(s,0)
14          S.add(s)
15          while not S.isEmpty() do
16                  int u = Q.deleteMin()    %Di costo logaritmico (2)
17                  b[u] = false
18                  foreach v ∈ G.adj(u) do
19                          if d[u] + G.w(u, v) < d[v] then
20                                  if not b[v] then
21                                          Q.insert(v, d[u] + G.w(u, v)) %Di costo
                                                logaritmico (3)
```

```
22                                          b[v] = true
23                                  else
24                                          Q.decrease(v, d[u] + G.w(u, v)) %Di
                                                costo logaritmico (4)
25                                  T[v] = u
26                                  d[v] = d[u] + G.w(u, v)
27          return(T, d)
```

Utilizzando una coda priorità basata su heap binario, le proprietà dell'algoritmo mostrato confrontate con quello di Dijkstra presentano delle leggere differenze, nonostante entrambi funzionino solamente per archi di peso positivo.
Si ha che l'algoritmo mostrato ha le seguenti caratteristiche:

- Ha una complessità differente: $\mathcal{O}(m \log n)$
- Funziona particolarmente bene per i grafi sparsi, ovvero grafi che presentano un limitato numero di archi. Per grafi densi, la complessità precedente supererebbe quello di Dijkstra, infatti in questo caso si avrebbe: m è $\Omega(n^2)$, dunque $\mathcal{O}(m \log n) \approx \mathcal{O}(n^2 \log n) > \mathcal{O}(n^2)$, mentre per grafi che presentano un numero ridotto di archi tale che m è $\mathcal{O}(n)$, il costo computazionale viene approssimato a $\mathcal{O}(n \log n)$.

Analisi della complessità:

Riga	Costo	Ripetizioni
(1)	$\mathcal{O}(n)$	1
(2)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
(3)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
(4)	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$

Dal quale risulta un costo totale di $\mathcal{O}(m \log n)$

Algoritmo di Fredman-Tarjan, 1987
Algoritmo ancora una volta basato su code a priorità, in questo caso però implementate con heap di Fibonacci, una struttura dati complessa che permette di ammortizzare il costo dell'operazione di decrease, ottenendo dunque un costo computazionale differente rispetto ai precedenti due algoritmi.

Algorithm 80 Algoritmo di Fredman-Tarjan: cammini minimi a sorgente singola

```
1  (int[],int[]) shortestPath (Graph G, Node s)
2           int[] d = new int[1 ... G.n]            %d[u] := la distanza da s a u
3           int[] T = new int[1... G.n]             %T[u] := padre di u in T
4           boolean[] b = new boolean[1... G.n]     %b[u] := true se u presente in S
5           foreach u ∈ G.V() − {s} do
6                   T[u] = nil
7                   d[u] = +∞
8                   b[u] = false
9           T[s] = nil
10          d[s] = 0
11          b[s] = true
12          PriorityQueue Q = PriorityQueue();      %Basato su heap di Fibonacci (1)
13          Q.insert(s,0)
14          S.add(s)
15          while not S.isEmpty() do
16                  int u = Q.deleteMin()   %Di costo logaritmico (2)
17                  b[u] = false
18                  foreach v ∈ G.adj(u) do
19                          if d[u] + G.w(u, v) < d[v] then
```

```
20                                  if not b[v] then
21                                          Q.insert(v, d[u] + G.w(u, v)) %Di costo
                                                logaritmico (3)
22                                          b[v] = true
23                                  else
24                                          Q.decrease(v, d[u] + G.w(u, v)) %Di
                                                costo ammortizzato costante (4)
25                                  T[v] = u
26                                  d[v] = d[u] + G.w(u, v)
27          return(T, d)
```

Questo algoritmo, nonostante presenti una complessità relativamente bassa, soffre di fattori moltiplicativi delle varie operazioni abbastanza alte dovute alla gestione della coda con priorità. Questo infatti lo rende un algoritmo giustificabile e perfetto per grafi densi e di dimensioni molto grandi. Per grafi di piccole dimensioni invece si preferisce fare uso degli algoritmi visti in precedenza.

Analisi della complessità:

Riga	Costo	Ripetizioni
(1)	$\mathcal{O}(n)$	1
(2)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
(3)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
(4)	$\mathcal{O}(1)^{(*)}$	$\mathcal{O}(m)$

Dal quale risulta un costo totale di $\mathcal{O}(m + n \log n)$, dove $^{(*)}$ indica che il costo è derivato da un analisi ammortizzata.

Algoritmo di Bellman-Ford-Moore, 1958

Algoritmo basato su una coda, senza priorità, computazionalmente più pesante rispetto all'algoritmo di Dijkstra, che permette però di gestire i grafi con archi di peso negativo.

Algorithm 81 Algoritmo di Bellman-Ford-Moore: cammini minimi a sorgente singola

```
1  (int[],int[]) shortestPath (Graph G, Node s)
2           int[] d = new int[1... G.n]              %d[u] := la distanza da s a u
3           int[] T = new int[1... G.n]              %T[u] := padre di u in T
4           boolean[] b = new boolean[1... G.n]      %b[u] := true se u presente in S
5           foreach u ∈ G.V() − {s} do
6                   T[u] = nil
7                   d[u] = +∞
8                   b[u] = false
9           T[s] = nil
10          d[s] = 0
11          b[s] = true
12          Queue Q = Queue();          %Coda (1)
13          Q.insert(s,0)
14          Q.enqueue(s)
15          while not S.isEmpty() do
16                  int u = Q.dequeue()      %Costo costante (2)
17                  b[u] = false
18                  foreach v ∈ G.adj(u) do
19                          if d[u] + G.w(u, v) < d[v] then
20                                  if not b[v] then
21                                          Q.enqueue(v) %Costo costante (3)
22                                          b[v] = true
23                                  T[v] = u
24                                  d[v] = d[u] + G.w(u, v)
```

```
25          return(T, d)
```

A differenza degli algoritmi precedenti questo algoritmo funziona per grafi con archi che possiedono un peso negativo, a discapito però di una maggiore complessità.
Per studiare la complessità dell'algoritmo di Bellman-Ford-Moore è necessario parlare di passate; sostanzialmente in questo algoritmo un nodo può essere inserito nuovamente in coda, dopo essere stato già rimosso, a differenza degli algoritmi precedenti i cui nodi venivano estratti una singola volta per poi non essere più accodati; ad ogni estrazione infatti, che chiamiamo passata, vengono estratti i nodi il cui costo descrive i cammini minimi di lunghezza pari al numero di passata (dove la passata zeresima consiste con l'estrazione della sorgente) . Le passate si ripeteranno per un massimo di (n-1), nelle quali vegono ogni volta analizzati dei percorsi migliori che descrivono cammini di lughezza al più n - 1.

Il costo computazionale è dunque:

Riga	Costo	Ripetizioni
(1)	$\mathcal{O}(1)$	1
(2)	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
(3)	$\mathcal{O}(1)$	$\mathcal{O}(nm)$

Dal quale risulta un costo totale di $\mathcal{O}(mn)$, dato dal fatto che ogni nodo può essere inserito ed estratto al massimo n-1 volte.

Cammini minimi su DAG

Algoritmo basato su uno stack che contiene i nodi che fanno parte dell'ordinamento topologico a partire dalla sorgente. Si basa sul fatto che, se sappiamo che il grafo in questione non possiede cicli, non c'è modo di tornare su un nodo già visitato e abbassare il suo costo.

Algorithm 82 Algoritmo per i DAG: cammini minimi a sorgente singola

```
1  (int[],int[]) shortestPath (Graph G, Node s)
2           int[] d = new int[1... G.n]              %d[u] := la distanza da s a u
3           int[] T = new int[1... G.n]              %T[u] := padre di u in T
4           foreach u ∈ G.V() - {s} do
5                   T[u] = nil
6                   d[u] = +∞
7           T[s] = nil
8           d[s] = 0
9           STACK S = topsort(G)
10          while not S.isEmpty() do
11                  int u = S.pop()
12                  foreach v ∈ G.adj(u) do
13                          if d[u] + G.w(u, v) < d[v] then
14                                  T[v] = u
15                                  d[v] = d[u] + G.w(u, v)
16          return(T, d)
```

La complessità dell'algoritmo precedente è dunque di $\mathcal{O}(n+m)$ in quanto è una semplice visita BFS del grafo.

Riassumendo, che algoritmo preferire?

Dijkstra	$O(n^2)$	Pesi positivi, grafi densi
Johnson	$O(m \log n)$	Pesi positivi, grafi sparsi
Fredman-Tarjan	$O(n + n \log n)$	Pesi positivi, grafi densi,dimensioni molto grandi
Bellman-Ford-Moore	$O(mn)$	Pesi negativi
DAG	$O(mn)$	DAG
BFS	$O(mn)$	Senza pesi

La soluzione che fa uso della BFS è sostanzialmente l'algoritmo di Erdos introdotto qualche capitolo fa.

17.2 Il problema dei cammini minimi a sorgente multipla

Possibili soluzioni

Input	Complessità	Approccio
Pesi positivi, grafo denso	$O(n \cdot n^2)$	Applicazione ripetuta dell'algoritmo di Dijkstra
Pesi positivi, grafo sparso	$O(n \cdot (m \log n))$	Applicazione ripetuta dell'algoritmo di Johnson
Pesi negativi	$O(n \cdot nm)$	Applicazione ripetuta di Bellman-Ford-Moore, sconsigliata
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di Floyd e Warshall
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di Johnson per sorgente multipla

Algoritmo di Floyd-Warshall, 1962
Algoritmo basato sulla programmazione dinamica per il calcolo dei cammini minimi a sorgente multipla. Il problema che si pone è sostanzialmente quello precedente con l'unica variante che, invece di mantenere un unica sorgente e trovare tutti i cammini minimi dalla stessa agli altri nodi, si vuole fare questo processo per ogni vertice del grafo.
Si definiscono dei termini utili per la comprensione dell'algoritmo:

Assunzione
Assumiamo che esista un ordinamento fra i nodi del grafo $v_1, v_2, ..., v_n$.

I cammini minimi k-vincolati
Sia k un valore in $0, ..., n$. Diciamo che un cammino p^k_{xy} è un cammino minimo k-vincolato fra x e y se esso ha il costo minimo fra tutti i cammini fra x e y che non passano per nessun vertice in $v_{k+1}, ..., v_n$ (x e y sono esclusi dal vincolo).

Distanza k-vincolata
Denotiamo con $d^k[x][y]$ il costo totale del cammino minimo k-vincolato fra x e y, se esiste.

$$d^k[x][y] = \begin{cases} w(p^k_{xy}) & \text{se esiste } p^k_{xy} \\ +\infty & \text{altrimenti} \end{cases}$$

Matrice dei padri
Oltre a definire la matrice d, calcoliamo una matrice T dove $T[x][y]$ rappresenta il predecessore di y nel cammino più breve da x a y.

Principio di funzionamento
L'algoritmo di Floyd-Warshall è basato sulla definizione di cammino minimo k-vincolato, sostanzialmente per ogni coppia di veritici si osserva il comportamento in seguito all'aggiunta di un nuovo

nodo dal quale i cammini possono passare, l'aggiunta di tale vertice può o meno diminuire la distanza tra i nodi in questione. Si parte infatti con i cammini minimi 0 vincolati, i quali rappresentano tutti i possibili cammini che non passano per alcun vertice, ad eccezione dei nodi adiacenti, ovvero gli archi diretti. Mano a mano che si considerano nuovi nodi i costi possono modificarsi; la soluzione finale si troverà una volta analizzati i cammini n-vincolati, in quanto sono i cammini di costo minimo che possono passare per qualunque vertice del grafo.
Ciò che abbiamo definito fino ad ora può essere riassunto dalla seguente ricorrenza:

$$d^k[x][y] = \begin{cases} w(x,y) & k = 0 \\ min(d^{k-1}[x][y], d^{k-1}[x][k] + d^{k-1}[x][y]) & k > 0 \end{cases}$$

Algorithm 83 Algoritmo di Floyd e Warshall cammini minimi a sorgente multipla

```
1  (int[],int[]) floydWarshall (Graph G)
2          int[][] d = new int[1...n][1...n]
3          int[][] T = new int[1...n][1...n]
4          foreach u, v ∈ G.V() do
5                  d[u][v] = +∞
6                  T[u][v] = nil
7          foreach u ∈ G.V() do
8                  foreach v ∈ G.adj(u) do
9                          d[u][v] = G.w(u, v)
10                         T[u][v] = u
11         for k = 1 to G.n do
12                 foreach u ∈ G.V() do
13                         foreach v ∈ G.V() do
14                                 if d[u][k] + d[k][v] < d[u][v] then
15                                         d[u][v] = d[u][k] + d[k][v]
16                                         T[u][v] = T[k][v]
17         return (T,d)
```

La prima parte dell'algoritmo si occupa della trasformazione del nostro grafo, che abbiamo sempre assunto essere espresso mediante liste di adiacenza, in un grafo rappresentato da una matrice di adiacenza. Una volta fatto questo, si fissa il parametro k (che rapprensenta il vincolo del cammino preso in analisi), e per ogni coppia di nodi si verificano i percorsi migliori facendo le opportune modifiche sul costo.
La complessità totale dell'algoritmo è $\mathcal{O}(n^3)$ dovuta alla presenza di tre cicli for annidati che iterano sui nodi del grafo.

18 Programmazione Greedy

18.1 Introduzione

Nella sezione precedente ci siamo occupati di risolvere alcuni problemi mediante l'utilizzo di programmazione dinamica e, in alcuni di questi casi, abbiamo dovuto controllare tutte le possibili soluzioni ottenute poichè non eravamo certi di dove potesse trovarsi il risultato. In questo capitolo ci occuperemo invece della programmazione greedy e ciò di selezionare una sola tra le scelte possibili, quella che ci sembra la migliore (localmente ottima), dimostrando però che questa scelta si rivela poi essere ottima a livello globale.
ATTENZIONE: Non tutti i problemi hanno una scelta ingorda come soluzione!

18.2 Insieme indipendente massimale di intervalli

Input
Sia $S = \{1, 2, ..., n\}$ un insieme di intervalli della retta reale. Ogni intervallo $[a_i, b_i]$ con $i \in S$, è chiuso a sinistra e aperto a destra.

- a_i : tempo di inizio
- b_i : tempo di fine

Definizione del Problema
Un insieme indipendente massimale è un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra loro.

18.2.1 Dalla programmazione dinamica...

Prima di parlare della risoluzione del problema con la tecnica greedy proviamo a vedere come avremmo affrontato il problema con programmazione dinamica, definendo la sottostruttura ottima, per cercare la cosiddetta scelta ingorda.

Sottostruttura Ottima
Si assuma che gli intervalli siano ordinati per tempo di fine: $b_1 \leq b_2 \leq ... \leq b_n$
Definiamo il sottoproblema $S[i...j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j: $S[i...j] = \{k | b_i \leq a_k < b_k \leq a_j\}$
Aggiungiamo due intervalli fittizi:

- Intervallo 0: $b_0 = -\infty$
- Intervallo $n+1$: $a_{n+1} = +\infty$

Il problema iniziale corrisponde al problema $S[0, n+1]$ (ricordiamo che S definisce gli intervalli che cominciano dopo i - cioè dopo $-\infty$ e prima di j - cioè prima di $+\infty$).

Teorema
Supponiamo che $A[i...j]$ sia una soluzione ottimale di $S[i...j]$ e sia k un intervallo che appartiene a $A[i...j]$; allora

- Il problema $S[i...j]$ viene suddiviso in due sottoproblemi
  - $S[i...k]$: gli intervalli di $S[i...j]$ che finiscono prima di k
  - $S[k...j]$: gli intervalli di $S[i...j]$ che iniziano dopo di k
- $A[i..j]$ contiene le soluzioni ottimali di $S[i...k]$ e $S[k...j]$
  - $A[i...j] \cap S[i...k]$ è la soluzione ottimale di $S[i...k]$
  - $A[i...j] \cap S[k...j]$ è la soluzione ottimale di $S[k...j]$

Dimostrazione (in pillole): Come è possibile immaginare se ad esempio $A[i...j] \cap S[i...k]$ **non fosse la soluzione ottimale** di $S[i...k]$ allora vorrebbe dire che esisterebbe un altro insieme di elementi $A'[i...j]$ t.c. $A'[i...j] \cap S[i...k]$ è la soluzione ottimale di $S[k...j]$ ma ciò contraddirrebbe l'ipotesi iniziale! (cioè se non vale per uno dei due sottointervalli non può essere la soluzione complessiva)

Definizione ricorsiva della soluzione
$A[i...j] = A[i...k] \cup \{k\} \cup A[k...j]$

Definizione Ricorsiva del suo costo
Come si determina k? Devo analizzare tutte le possibilità.
Sia $DP[i][j]$ la dimensione del più grande sottoinsieme $A[i...j] \subseteq S[i...j]$ di intervalli indipendenti

$$DP[i][j] = \begin{cases} 0 & S[i...j] = \emptyset \\ max_{k \in S[i...j]}\{DP[i][k] + DP[k][j] + 1\} & altrimenti \end{cases}$$

18.2.2 ...alla soluzione ingorda

Grazie alla precedente definizione possiamo scrivere un algoritmo basato su programmazione dinamica o memoization e, visto che dobbiamo risolvere tutti i sottoproblemi, il costo totale è pari a $\mathcal{O}(n^3)$.
Nel caso di intervalli pesati abbiamo visto che è possibile sviluppare una soluzione con costo $\mathcal{O}(n \log n)$, ma siamo sicuri che sia necessario analizzare tutti i possibili valori k?

Teorema
Sia $S[i...j]$ un sottoproblema non vuoto e m l'intervallo di $S[i...j]$ con il minor tempo di fine (cioè $m \in S[i...j]$), allora:

- il sottoproblema $S[i...m]$ è vuoto
- m è compreso in qualche soluzione ottima di $S[i...j]$

Dimostrazione 1
Sappiamo che:

- $a_m < b_m$ (Definizione di intervallo)
- $\forall k \in S[i...j] : b_m \leq b_k$ (m ha minor tempo di fine)

Ne segue che: $\forall k \in S[i...j] : a_m < b_k$ (Transitività)
Se nessun intervallo in $S[i...j]$ termina prima di a_m allora $S[i...m] = \emptyset$.

Dimostrazione 2
Sia $A'[i...j]$ una soluzione ottima di $S[i...j]$.
Sia m' l'intervallo con il minor tempo di fine in $A[i...j]$.
Sia $A[i..j] = A'[i...j] - \{m'\} \cup \{m\}$ una nuova soluzione attenuta togliendo m' e aggiungendo m ad $A'[i...j]$.
$A[i...j]$ è una soluzione ottima che contiene m, in quanto ha la stessa dimentsione di $A'[i...j]$ e gli intervalli sono indipendenti. Per intenderci ricadiamo in due casi differenti:

- Se $m = m'$ allora non cambia nulla perchè ho aggiunto e tolto lo stesso elemento
- Se $m \neq m'$ allora ho trovato una soluzione differente ma con la stessa cardinalità della prima

Conseguenze

- Non è più necessario analizzare tutti i possibili valori di k: faccio una scelta ingorda ma sicura, seleziono l'attività m con il minor tempo di fine

- Non è più necessario analizzare due sottoproblemi: elimino tutte le attività che non sono compatibili con la scelta ingorda e dunque mi rimane da risolvere solamente il sottoproblema $S[m...j]$

18.2.3 Algoritmo

Algorithm 84 indipendent Set (Greedy)

```
1  SET indipendentSet(int[] a, int[] b)
2          {ordina a e b in modo che b[1] ≤ b[2] ≤ ... ≤ b[n]}
3          SET S = Set()
4          %inserisco direttamente il primo elemento perche' ordinati per fine
5          S.insert(1)
6          int last = 1
7          for i = 2 to n do
8                  if a[i] ≥ b[last] then
9                  S.insert(i)
10                 last = i
11         return S
```

La complessità dell'algoritmo è $\mathcal{O}(n \log n)$ se l'input non è ordinato oppure $\mathcal{O}(n)$ nel caso in cui sia già ordinato.

18.3 Problema del resto

Input
Un insieme di tagli di monete, memorizzati in un vettore di interi positivi $t[1...n]$ e un intero R rappresentante il resto che dobbiamo restituire.

Definizione del problema
Trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli disponibili, assumendo di avere un numero illimitato di monete per ogni taglio.

19 Ricerca locale

Definizione
Se si conosce una soluzione ammissibile ad un problema di ottimizzazione, si può provare a trovare una soluzione migliore nelle "vicinanze" di quella precendente. Si continua in questo modo fino a quando non si è più in grado di migliorare.
Uno schema possibile per la ricerca locale può essere il seguente:

Algorithm 85 Ricerca locale

```
1  ricercaLocale()
2          Sol = una soluzione ammissibile del problema
3          while ∃S ∈ I(Sol) migliore di Sol do
4                  Sol = S
5          return Sol
```

19.1 Il problema del flusso massimo

Rete di flusso:
Una rete di flusso $G = (V, E, s, t, c)$ è data da:

- un grafo orientato $G = (V, E)$;
- un nodo s ∈ V detto sorgente;
- un nodo t ∈ V detto pozzo;
- una funzione di capacità $c : V \times V \to \mathbb{R}^{\geq 0}$, tale per cui $(u, v) \notin E \to c(u, v) = 0$.

Assunzioni

- Per ogni nodo $v \in V$, esiste un cammino $s \rightsquigarrow v \rightsquigarrow t$ da s a t che passa per v;
- Possiamo ignorare i nodi che non godono di questa proprietà.

Flusso:
Un flusso in G è una funzione $f : V \times V \to \mathbb{R}$ che soddisfa le seguenti proprietà:

- Vincolo sulla capacità: $\forall u, v \in V : f(u, v) \leq c(u, v)$.
  Il flusso non deve eccedere la capacità sull'arco.
- Assimmetria: $\forall u, v \in V, f(u, v) = -f(v, u)$.
- Conservazione del flusso: $\forall u \in V - \{s, t\}, \sum_{v \in V} f(u, v) = 0$
  Per ogni nodo, la somma dei flussi entranti deve essere uguale alla somma dei flussi uscenti.

Valore del flusso:
Il valore di un flusso f è definito come:

$$|f| = \sum_{(s,v) \in E} f(s, v)$$

ovvero come la quantità di flusso uscente da s.

Flusso massimo:
Data una rete G = (V, E, s, t, c), trovare un flusso che abbia valore massimo fra tutti i flussi associabili alla rete.

$$|f^*| = max\{|f|\}$$

Flusso nullo:
Definiamo flusso nullo la funzione $f_0 : V \times V \to \mathbb{R}^{\geq 0}$ tale che $f(u, v) = 0$ per ogni $u, v \in V$.

Somma di flussi:
Per ogni coppia di flussi f_1 e f_2 in G, definiamo il flusso somma $g = f_1 + f_2$ come un flusso tale per cui $g(u, v) = f_1(u, v) + f_2(u, v)$.

Capacità residua:
Definamo capacità residua di un flusso f in una rete $G = (V, E, s, t, c)$ una funzione $c_f : V \times V \to \mathbb{R}^{\geq 0}$ tale che $c_f(u, v) = c(u, v) - f(u, v)$.
Si noti che per la definzione di capactià residua si creano degli archi all'indietro.

Reti residue:
Data una rete di flusso $G = (V, E, s, t, c)$ e un flusso f su G, possiamo costruire una rete residua $G_f = (V, E_f, s, t, c_f)$, tale per cui $(u, v)E_f$ se e solo se $c_f(u, v) > 0$.

Algoritmo, schema generale:

Algorithm 86 schema generale flusso massimo

```
1 int [][] maxFlow(GRAPH G, NODE s, NODE t, int [][] c)
2         f = f_0
3         r = c
4         repeat
5                 g = trova un flusso in r tale che |g| > 0, altrimenti f_0
6                 f = f + g
7                 r = Rete di flusso residua del flusso f in G
8         until g = f_0
9         return f
```

Dimostrazione correttezza

Lemma
Se f è un flusso in G e g è un flusso in G_f, allora $f + g$ è un flusso in G.

Dimostrazione:

- Vincolo sulla capacità:

$$\begin{aligned} g(u,v) &\leq c_f(u,v) \\ f(u,v) + g(u,v) &\leq c_f(u,v) + f(u,v) \\ (f+g)(u,v) &\leq c(u,v) - f(u,v) + f(u,v) \\ (f+g)(u,v) &\leq c(u,v) \end{aligned}$$

- Antisimmetria:

$$\begin{aligned} f(u,v) + g(u,v) &= -f(v,u) - g(v,u) \\ f(u,v) + g(u,v) &= -(f(v,u) + g(v,u)) \\ (f+g)(u,v) &= -(f+g)(v,u) \end{aligned}$$

- Conservazione:

$$\begin{aligned} \sum_{v \in V}(f+g)(u,v) &= \sum_{v \in V}(f(u,v) + g(u,v)) \\ &= \sum_{v \in V} f(u,v) + \sum_{v \in V} g(u,v) \\ &= 0 \end{aligned}$$

Algoritmo di Ford-Fulkerson
Tale algoritmo suggerisce che per trovare un flusso aumentante è necessario trovare un cammino $C = v_0, v_1, ..., v_n$ con sv_0 e $t = v_n$ nella rete residua G_f. Si identifica la capacità del cammino, corrispondente alla minore capacità degli archi incontrati (collo di bottiglia):

$$c_f(C) = min_{i=2...n} c_f(v_{i-1}, v_i)$$

A partire da questo si crea dunque un flusso addizionale g tale che:

- $g(v_{i-1}, v_i) = c_f(C)$;
- $g(v_{i-1}, v_i) = -c_f(C)$ (per antisimmetria)
- $g(x, y) = 0$ per tutte le altre coppie (x, y)

Algorithm 87 schema generale flusso massimo

```
1  int[][] maxFlow(GRAPH G, NODE s, NODE t, int[][] c)
2          int[][] f = new int[][] % flusso parziale
3          int[][] g = new int[][] % flusso da cammino aumentante
4          int[][] r = new int[][] % rete residua
5          foreach u,v ∈ G.V() do
6                  f[u][v] = 0
7                  r[u][v] = c[u][v]
8          repeat
9                  g = flusso associato ad un cammino aumentante in r, oppure f_0
10                 foreach u,v ∈ G.V() do
11                         f[u][v] = f[u][v] + g[u][v]      % f = f + g
12                         r[u][v] = c[u][v] - f[u][v]      % Calcola c_f
13         until g = f_0
14         return f
```

Ford e Fulkerson suggerirono una semplice visita del grafo, in profondità oppure in ampiezza.
Edmonds e Karp suggerirono di utilizzare una visita in ampiezza.
Il costo della visita: $\mathcal{O}(V + E)$

Complessità, limite superiore - Versione Ford-Fulkerson
Se le capacità sono intere, l'algoritmo di Ford-Fulkerson ha complessità $\mathcal{O}((V + E)|f^*|)$ (liste) o $\mathcal{O}(V^2|f^*|)$ (matrice).

- L'algoritmo parte dal flusso nullo e termina quando il valore totale del flusso raggiunge $|f^*|$
- Ogni incremento del flusso aumenta il flusso di almeno un'unità
- Ogni ricerca di un cammino richiede una vistita del gafo, con costo $\mathcal{O}(V + E)$ o $O(V^2)$
- La somma dei flussi e il calcolo della rete residua può essere effettuato in tempo $\mathcal{O}(V + E)$ o $\mathcal{O}(V^2)$.

Complessità, limite superiore - Versione Edmonds e Karp
Se le capacità sono intere, l'algoritmo di Edmonds e Karp ha complessità $\mathcal{O}(VE^2)$ nel caso pessimo.

- Vengono eseguiti $\mathcal{O}(VE)$ aumenti di flusso, ognuno dei quali richiede una visita in ampiezza $\mathcal{O}(V + E)$.
- $\mathcal{O}(VE(V + E)) = \mathcal{O}(VE^2)$

Che complessità scegliere?
Qualora impiegassimo una visita in ampiezza per individuare il cammino aumentante nella rete residua facciamo uso sia dell'algoritmo di Ford-Fulkerson che di quello di Edmonds e Karp.
In questo caso dunque si presentano due limiti superiori per lo stesso algoritmo, da una parte abbiamo $\mathcal{O}((V + E)|f^*|)$ (liste di adiacenza) o $\mathcal{O}(V^2|f^*|)$ (matrice di adiacenza), mentre dall'altra $\mathcal{O}(VE^2)$.
Entrambi sono limiti superiori validi, dunque per determinare la complessità dell'algoritmo è sufficiente scegliere il più basso.
Dunque, utilizzando l'algoritmo della ricerca del flusso massimo di Edmonds e Karp (versione di Ford-Fulkerson) la complessità è data da $\mathcal{O}(min((V+E)|f^*|, VE^2))$ (liste di adiacenza), $\mathcal{O}(min(V^2|f^*|, VE^2))$ (matrice di adiacenza).

19.2 Teoremi e lemmi utili

In questa sottosezione sono mostrati alcuni Lemmi e Teoremi utili per la dimostrazione della complessità e della correttezza dell'algoritmo di Edmonds e Karp, che non verranno dimostrate per brevità.

Lemma - Monotonia
Sia $\delta f(s, v)$ la distanza minima da s a v in una rete residua G_f.
Sia $f' = f + g$ un flusso nella rete iniziale, con g flusso non nullo derivante da un cammino aumentante. Allora $\delta f'(s, v) \geq \delta f(s, v)$.

Lemma - Monotonia
Sia $\delta f(s, v)$ la distanza minima da s a v in una rete residua G_f.
Sia $f' = f + g$ un flusso nella rete iniziale, con g flusso non nullo derivante da un cammino aumentante. Allora $\delta f'(s, v) \geq \delta f(s, v)$.

Lemma - Aumenti di flusso
Il numero totale di aumenti di flusso eseguiti dall'algoritmo di Edmonds e Karp è $\mathcal{O}(VE)$

Taglio
Un taglio (S, T) della rete di flusso $G = (V, E, s, t, c)$ è una partizione di V in due sottoinsiemi disgiunti S, T tali che:

$$S = V - T$$
$$s \in S \wedge t \in T$$

Capacità di un taglio:
La capacità $C(S, T)$ attraverso il taglio (S, T) è pari a:

$$C(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

Flusso di un taglio:
Se f è un flusso in G, il flusso netto $F(S, T)$ attraverso (S, T) è:

$$F(S, T) = \sum_{u \in S, v \in T} f(u, v)$$

Lemma - Valore del flusso di taglio
Dato un flusso f e un taglio (S, T), la quantità di flusso $F(S, T)$ che attraversa il taglio è uguale a $|f|$.

Lemma – Capacità taglio
Il flusso massimo è limitato superiormente dalla capacità del taglio minimo, ovvero il taglio la cui capacità è minore fra tutti i tagli.

Teorema conclusivo

- f è un flusso massimo;
- non esiste nessun cammino aumentante per G;
- esiste un taglio minimo (S, T) tale che $C(S, T) = |f|$.

19.3 Esempio di flusso massimo

Problema - Job Assignment
Input:

- Un insieme J contenente n job;
- Un insieme W contenente m worker;
- Una relazione $R \subseteq J \times W$, tale per cui $(j, w) \in \mathbb{R}$ se e solo se il job j può essere eseguito dal worker w.

Output:
Il più grande sottoinsieme $O \subseteq \mathbb{R}$, tale per cui:

- ogni job venga assegnato al più ad un worker;
- ad ogni worker venga assegnato al più un job.

Soluzione:
Si può intuire che il problema richiami le reti di flusso dato che esso fa riferimento ad un problema di matching, in particolare di trovare il più grande sottoinsieme che soddisfa tali abbinamenti.
Per questi tipi di problemi è sufficiente definire un accurata descrizione del processo risolutivo, senza dover scrivere l'algoritmo per la risoluzione.
Per risolvere il job assignment :
Si costruisce una rete di flusso $G = (V, E, s, p, c)$:

- V è l'insieme di nodi rappresentato da job e worker;
- s è un nodo aggiuntivo, detto supersorgente;
- p è un nodo anch'esso aggiuntivo detto superpozzo;
- E è l'insieme di archi definiti come segue:
  - Si aggiunge un arco tra job e worker se esso può essere eseguito dal corrispondente lavoratore;
  - Si aggiunge un arco dalla supersorgente a tutti i job;
  - Si aggiunge un arco da tutti i worker al superpozzo.
- c (funzione di capacità) è definita come segue:
  - c(s, u) = 1 per la sorgente a tutti i nodi job, dato che ogni job deve essere eseguito una singola volta;
  - c(u, v) = 1 per ogni arco tra job e worker in quanto il lavoro deve essere assegnato al più un worker;
  - c(v, p) = 1 per ogni arco tra worker e superpozzo in quanto ad ogni worker può essere assegnato al più un job.

Il flusso massimo della rete di flusso, definita precedentemente, darà come risultato il massimo sottoinsieme cercato.
Complessità del problema:
il massimo flusso ottenibile in questa rete di flusso è dell'ordine di $\mathcal{O}(V)$, rappresentato dal numero di job presenti. Supponendo che il grafo G sia memorizzato con liste di adiacenza l'algoritmo di Ford-Fulkerson esegue in un tempo pari a $\mathcal{O}((V + E)V) = \mathcal{O}(VE)$ (nota bene: la complessità considerata è la seguente, e non quella definita da Edmonds e Karp in quanto superiore).

20 Esercitazioni

20.1 Esecitazione 1 - 10/10/19

Analisi - Ordinamento Funzioni
Ordinare le seguenti funzioni in accordo alla loro complessità asintotica. Si scriva $f(n) < g(n)$ se $\mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$. Si scriva $f(n) = g(n)$ se $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, ovvero se $f(n) = \Theta(g(n))$.
$f_1(n) = 2^{n+2} = \mathcal{O}(2^n)$
$f_2(n) = \log^2 n = \mathcal{O}(\log^2 n)$
$f_3(n) = \log_n (n \cdot (\sqrt{n})^2) + \frac{1}{n^2} = \log_n n^2 + n^{-2} = 2 + n^{-2} = \mathcal{O}(1)$
$f_4(n) = 3n^{0.5} = 3n^{1/2} = 3\sqrt{n} = \mathcal{O}(\sqrt{n})$
$f_5(n) = 16^{n/4} = 2^{4 \cdot n/4} = 2^n = \mathcal{O}(2^n)$
$f_6(n) = 2\sqrt{n} + 4n^{1/4} + 8n^{1/8} + 16n^{1/16} = \mathcal{O}(\sqrt{n})$
$f_7(n) = \sqrt{(\log n)(\log n)} = \sqrt{\log^2 n} = \log n = \mathcal{O}(\log n)$
$f_8(n) = \frac{n^3}{(n+1)(n+3)} = \frac{n^3}{n^2+4n+3} \approx \frac{n^{\cancel{3}}}{n^{\cancel{2}}} = \mathcal{O}(n)$
$f_9(n) = 2^n = \mathcal{O}(2^n)$

Soluzione
$f_3(n) < f_7(n) < f_2(n) < f_4(n) = f_6(n) < f_8(n) < f_1(n) = f_5(n) = f_9(n)$

Ricorrenza 2T(n/8) + 2T(n/4) + n
Trovare un limite asintotico superiore e un limite asintotico inferiore alla seguente ricorrenza, facendo uso del metodo di sostituzione:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/8) + 2T(n/4) + n & n > 1 \end{cases}$$

Soluzione
Per quanto riguarda il limite asintotico inferiore possiamo affermare che $T(n) = \Omega(n)$ in quanto nell'equazione di ricorrenza compare un termine n di grado 1.
Da questa supposizione potremmo pensare di usare il metodo di sostituzione per verificare se $T(n) = \mathcal{O}(n)$. Al posto che fare tentativi a caso potremmo però vedere la ricorrenza nel modo seguente:
$T(n) = 2T(n/8) + 2T(n/4) + n \leq 2T(n/4) + 2T(n/4) + n = 4T(n/4) + n$
A questo punto è possibile applicare il Teorema dell'Esperto per le ricorrenze comuni:
$a = 4$, $b = 4$, $\beta = 1$, $\alpha = \log_b a = \log_4 4 = 1 \implies \alpha = \beta$
$\implies T(n) = \mathcal{O}(n \log n)$
Dunque possiamo procedere con il tentativo per $T(n) = \mathcal{O}(n)$ visto che $\mathcal{O}(n) \subset \mathcal{O}(n \log n)$
Tentativo $T(n) = \mathcal{O}(n)$
$\exists c > 0, \exists m \geq 0 : \mathrm{T(n)} \leq cn, \forall n \geq \mathrm{m}$

Ipotesi Induttiva: $\forall k < \mathrm{n} : \mathrm{T(k)} \leq \mathrm{ck}$
Passo di Induzione: Dimostriamo la disequazione per T(n)
$T(n) = 2T(n/8) + 2T(n/4) + n = 2c(n/8) + 2c(n/4) + n = cn/4 + cn/2 + n$
$= (\frac{1+2}{4})cn + n \overset{?}{\leq} cn \iff (\frac{3}{4})c + 1 \leq c \iff c \geq 4$

Caso Base: Dimostriamo la disequazione per T(1)
$T(1) = 1 \overset{?}{\leq} c \cdot 1 \iff c \geq 1$

Le due disequazioni sono soddisfatte entrambe per valori di $c \geq 4$
Visto che $T(n) = \Omega(n)$ e $T(n) = \mathcal{O}(n) \implies T(n) = \Theta(n)$

Ricorrenza
Trovare i limiti superiore e inferiori più stretti possibili per la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + 4T(\lfloor n/4 \rfloor) + 15T(\lfloor n/8 \rfloor) + n^2 & n > 8 \\ 1 & n \leq 8 \end{cases}$$

Per quanto riguarda il limite asintotico inferiore possiamo affermare che $T(n) = \Omega(n^2)$ in quanto:
$T(n) = 2T(\lfloor n/2 \rfloor) + 4T(\lfloor n/4 \rfloor) + 15T(\lfloor n/8 \rfloor) + n^2 \geq c_1 n^2 = \Omega(n^2)$

Tentativo $T(n) = \mathcal{O}(n^2)$
$\exists c > 0, \exists m \geq 0 :$ T(n) $\leq cn^2, \forall n \geq$ m

Ipotesi Induttiva: $\forall k <$ n : T(k) $\leq$ ck
Passo di Induzione: Dimostriamo la disequazione per T(n)
$T(n) = 2T(\lfloor n/2 \rfloor) + 4T(\lfloor n/4 \rfloor) + 15T(\lfloor n/8 \rfloor) + n^2$
$= 2c(\lfloor n^2/2^2 \rfloor) + 4c(\lfloor n^2/4^2 \rfloor) + 15c(\lfloor n^2/8^2 \rfloor) + n^2$
$\leq 2c(n^2/4) + 4c(n^2/16) + 15c(n^2/64) + n^2$
$= \frac{cn^2}{2} + \frac{cn^2}{4} + \frac{15cn^2}{64} + n^2 \overset{?}{\leq} cn^2 \iff \frac{c}{2} + \frac{c}{4} + \frac{15c}{64} + 1 \leq c$
$\iff (\frac{32+16+15}{64})c + 1 \leq c \iff (\frac{63}{64})c + 1 \leq c \iff c \geq 64$

Caso Base: Dimostriamo la disequazione per i casi da T(1) a T(8)
$T(1) = 1 \overset{?}{\leq} c \cdot 1 \iff c \geq 1$
$T(2) = 1 \overset{?}{\leq} c \cdot 4 \iff c \geq 1/4$
$T(3) = 1 \overset{?}{\leq} c \cdot 9 \iff c \geq 1/9$
...

Notiamo che man mano che usiamo n più grandi otteniamo c mano a mano più piccoli dunque osserviamo che alla fine otteniamo che tutti questi casi sono validi per $c \geq 1$. Le due disequazioni sono soddisfatte entrambe per valori di $c \geq 64$
Visto che $T(n) = \Omega(n^2)$ e $T(n) = \mathcal{O}(n^2) \implies T(n) = \Theta(n^2)$

Analisi - Algortimo di selezione deterministico
Trovare i limiti superiore e inferiori per la seguente equazione di ricorrenza con il metodo di sostituzione:

$$T(n) = \begin{cases} T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + \frac{11}{5}n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Per quanto riguarda il limite asintotico inferiore possiamo affermare che $T(n) = \Omega(n)$ in quanto:
$T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + \frac{11}{5}n \geq c_1 n = \Omega(n)$

Tentativo $T(n) = \mathcal{O}(n)$

$\exists c > 0, \exists m \geq 0 : T(n) \leq cn^2, \forall n \geq m$

Ipotesi Induttiva: $\forall k <$ n : T(k) $\leq$ ck
Passo di Induzione: Dimostriamo la disequazione per T(n)
$T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + \frac{11}{5}n = c(\lfloor n/5 \rfloor) + c(\lfloor 7n/10 \rfloor) + \frac{11}{5}n$
$\leq c(n/5) + c(7n/10) + \frac{11}{5}n = (\frac{2+7}{10})cn + \frac{11}{5}n = \frac{9}{10}cn + \frac{11}{5}n \overset{?}{\leq} cn$
$\iff \frac{9}{10}c + \frac{11}{5} \leq c \iff c \geq 22$

Caso Base: Dimostriamo la disequazione per T(1)
$T(1) = 1 \overset{?}{\leq} c \cdot 1 \iff c \geq 1$

Un valore di c che soddisfa entrambe le disequazioni è $c \geq 22$

Analisi - MergeSortK
Si supponga di scrivere una variante di MergeSort chiamata MergeSortK che, invece di suddividere l'array da ordinare in 2 parti, lo suddivide in K parti, ri-ordina ognuna di esse applicando ricorsivamente MergeSortK, e le riunifica usando un'opportuna variante MergeK di Merge, che fonde K sottoarray invece di 2. Come cambia, se cambia, la complessità temporale di MergeSortK rispetto a quella di MergeSort?
Ricordiamo che l'equazione di ricorrenza di mergeSort() è la seguente:

$$T(n) = \begin{cases} 2T(n/2) + dn & n > 1 \\ c & n \leq 1 \end{cases}$$

Da questa equazione di ricorrenza si ottiene che la complessità di mergeSort() è $\Theta(n \log n)$
Se immaginiamo che mergeSortK suddivida l'array in K parti vuol dire che dovrà chiamare la funzione mergeSortK un numero K di volte e la complessità di ogni elemento , dunque la complessità di ogni array sarà del tipo n/k. Dunque la sua equazione di ricorrenza sarà del tipo

$$T(n) = \begin{cases} KT(n/K) + dn & n > 1 \\ c & n \leq 1 \end{cases}$$

In questo caso risulta particolarmente comodo usare il Teorema dell'Esperto per casi di $K \geq 2$:
$a = K$, $b = K$, $\beta = 1$, $\alpha = \log_K K = 1$
$\implies \alpha = \beta \implies T(n) = \Theta(n \log n)$
Dunque la complessità non cambia.

20.2 Esecitazione 2 - 29/10/19

Analisi - Ordinamento Funzioni
Si consideri la seguente equazione di ricorrenza:

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

dove m è una costante intera positiva.
Si trovino, tramite il metodo della sostituzione, un limite superiore ed un limite inferiore per T(n). Fare particolare attenzione ai casi base.

Soluzione
Tentativo $T(n) = \mathcal{O}(n)$
$\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$

Ipotesi Induttiva: $\forall k > 0 <$ n : T(k) $\leq$ ck
Passo di Induzione: Dimostriamo la disequazione per T(n)

$T(n) = T(m) + T(n-m) + 1 = 1 + cn - cm + 1 = cn - cm + 2 \overset{?}{\leq} cn \iff c \geq 2/m$
Poichè $m \geq 1$, $c \geq 2$ è un valore accettabile per ogni m.

Caso Base: Dimostriamo la disequazione per T(i) con $1 \leq i \leq m$
$T(i) = 1 \overset{?}{\leq} ci \iff c \geq 1/i$
Visto che $1 \leq i \leq m$ allora la disequazione è soddisfatta per $c \geq 1$.
Un valore di c che soddisfa entrambe le disequazioni è ad esempio $c = 2$.

Abbiamo dimostrato il limite superiore, ora dobbiamo dimostrare il limite inferiore.
Tentativo $T(n) = \Omega(n)$
$\exists c > 0, \exists m \geq 0 : T(n) \geq cn, \forall n \geq m$

Ipotesi Induttiva: $\forall k > 0 <$ n : T(k) $\geq$ ck

$T(n) = T(m) + T(n-m) + 1 = 1 + cn - cm + 1 = cn - cm + 2 \overset{?}{\geq} cn \iff c \leq 2/m$
Caso Base: Dimostriamo la disequazione per T(i) con $1 \leq i \leq m$
$T(i) = 1 \overset{?}{\geq} ci \iff c \leq 1/i$
Il più piccolo valore trovato per le due disequazioni è $\frac{1}{m}$ dunque $c \leq 1/m$
Abbiamo dunque dimostrato che $T(n) = \Omega(n)$
Visto che abbiamo dimostrato i limiti superiori e inferiori $\implies T(n) = \Theta(n)$

SortinoSort
Il professor Sortino ha inventato un nuovo algoritmo di ordinamento. Il vettore di input viene diviso in tre parti, di dimensioni approssimativamente uguali n/3. Dopo di che, vengono ordinati ricorsivamente le prime due parti del vettore (ovvero i primi due terzi), i secondi due terzi, e di nuovo i primi due terzi.

Algorithm 88 SortinoSort

```
1 SortinoSort(int[] A, int i, int j)
2         if(j - i + 1) ≤ 6 then
3                 InsertionSort(A,i,j)
```

```
4       else
5               int s = ceil((j - i + 1)/3)
6               SortinoSort(A, i, i + 2s - 1)
7               SortinoSort(A, i + s, j)
8               SortinoSort(A, i, i + 2s - 1)
```

Soluzione
L'equazione di ricorrenza del SortinoSort è del tipo seguente:

$$T(n) = \begin{cases} 3T(2n/3) & n > 6 \\ d & n \leq 6 \end{cases}$$

Dove d è il costo dell'InsertionSort visto che verrà usato quello per riordinare effettivamente il vettore passato in input e che ha una complessità nel caso medio/peggiore pari a $\mathcal{O}(n^2)$.
A questo punto è possibile applicare il *Master Theorem* per ricavare la forma chiusa dell'equazione di ricorrenza come segue:
$a = 3$, $b = \frac{3}{2}$, $\beta = 0$, , $\alpha = \log_b(a) = \log_{\frac{3}{2}}(3)$.
Possiamo osservare che $\alpha > \beta$, dunque vale che: $T(n) = \Theta(n^{\log_{\frac{3}{2}}(3)})$.
Questa complessità sarà sufficientemente bassa per finire sul libro del nostro amato Montresor? La risposta putroppo è no; il professor Sortino dovrà impegnarsi di più se vuole avere questo grande onore. Si può notare infatti che $\log_{\frac{3}{2}}(3)$ è un valore compreso tra 2 e 3 in quanto: $\frac{3}{2}^2 = \frac{9}{4} < 3$ $\frac{3}{2}^3 = \frac{27}{8} > 3$.
La ricorrenza ricavata è quindi $\Theta(n^2) < \Theta(n^{\log_{\frac{3}{2}}(3)}) < \Theta(n^3)$ che è peggiore di tutti gli algoritmi di ordinamento visti fino ad ora, fatta eccezione per il bogosort.

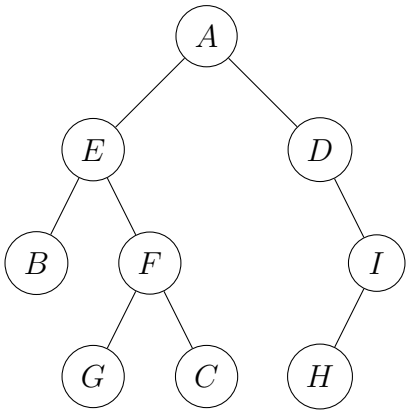
Alberi - Indovina l'albero

Gli ordini di visita di un albero binario di 9 nodi sono i seguenti:

- A, E, B, F, G, C, D, I, H (anticipato)
- B, G, C, F, E, H, I, D, A (posticipato)
- B, E, G, F, C, A, D, H, I (simmetrico).

Si ricostruisca l'albero binario e si illustri brevemente il ragionamento.

Per svolgere l'esercizio sono sufficienti le visite in ordine e le visite in preordine, in particolare grazie alla visita simmetrica siamo in grado di identificare i nodi che stanno nel sottoalbero di sinistra e quello di destra, mentre invece con la visita anticipata riusciamo non solo a determinare la radice ma anche i nodi che vegono visitati appena vengono raggiunti.
Infatti è possibile osservare che il nodo A è la radice dell'albero e nel suo sottoalbero sinistro vi sono: B, E, G, F, C, mentre in quello destro: D, H, I.
Il primo nodo alla sinistra di A è sicuramente E, come si osserva dalla visita anticipata. B è il figlio sinistro di E, mentre invece F è il suo figlio destro, e così via.
É facile dunque ricostruire l'albero che segue:



Alberi – Albero livello–valore

Scrivere un algoritmo che preso in input un albero binario T i cui nodisono associati ad un valore intero T.key, restituisca il numero di nodi dell'albero il cui valore è pari al livello del nodo. Vi ricordo che il livello del nodo è pari al numero di archi che devono essere attraversati per raggiungere il nodo dalla radice. Per cui la radice ha livello 0, i suoi figli hanno livello 1, etc.

La soluzione proposta è la seguente:

Algorithm 89 Albero livello-valore

```
1 int albero_livello_valore(TREE t)
2         return dfs(T, 0)
3
4 int dfs(TREE T, int altezza)
5         if T == nil then
6                 return 0
7         else
8                 int costo = iif(T.key() == altezza, 1, 0)
9                 return costo + dfs(T.right(), altezza + 1) + dfs(T.left(),
                      altezza + 1)
```

Semplicemente è una dfs postordine, in cui viene sommato 1 se il valore chiave del nodo è pari al livello, mentre 0 altrimenti. Il livello viene aumentato di 1 mano a mano che ci si sposta dalla radice.

Alberi – Albero livello–valore
Dato un albero binario contenente interi, scrivere un algoritmo cherestituisca la lunghezza del più lungo cammino monotono crescente radice-discendente, dove:

- il discendente non è necessariamente foglia;
- con lunghezza si intende il numero totale diarchi attraversati;
- con monotona crescente si intende che i valori contenuti nei nodidella sequenza devono essere ordinati in senso crescente da radice a discendente.

La soluzione proposta è la seguente:

Algorithm 90 Percorso cammino-discendente

```
1  int cammino(TREE t)
2          return camminoRadiceDiscendente(t, nil, 0)
3
4  int camminoRadiceDiscendente(TREE t, int precedente, int lun)
5          if t == nil then
6                  return lun
7          else
8          if precedente == nil or precedente ≥ t.value() then
9                  return max(lun, camminoRadiceDiscendente(t.left, t.value, 0),
                       camminoRadiceDiscendente(t.left, t.value, 0))
10         else
11                 return max(camminoRadiceDiscendente(t.left, t.value, lun + 1),
                       camminoRadiceDiscendente(t.left, t.value, lun + 1))
```

Questa soluzione prevede che sia tornata sempre la lunghezza massima del cammino incontrato fino ad ora avendo il riferimento al padre e facendo il controllo sul valore del figlio. Se il figlio è minore del padre allora la lunghezza del cammino corrente incrementa. La soluzione di Montresor invece è molto più elegante:

Algorithm 91 Percorso cammino-discendente Montresor

```
1  int monotone(TREE T)
2          int maxl = 0
3          int maxr = 0
4          if T ≠ nil then
5                  if T.left() ≠ nil and T.left().value > T.value then
6                          maxl = 1 + monotone(T.left())
7                  if T.right() ≠ nil and T.right().value > T.value then
8                          maxr = 1 + monotone(T.right())
9          return max(maxl, maxr)
```

20.3 Esecitazione 3 - 26/11/19

Grafi – Stessa distanza

- In un grafo orientato G, dati due nodi s e v, si dice che:
  - v è raggiungibile dasse esiste un cammino da s a v;
  - a distanza divda s è la lunghezza del più breve cammino da s a v (misurato in numero di archi), oppure $+\infty$ se v non è raggiungibile da s
- Scrivere un algoritmo che prenda in input un grafo orientato $G = (V, E)$ e due nodi s_1, s_2 $\in V$ e che restituisca il numero di nodi in V tali che:
  - siano raggiungibili sia da s_1 che da s_2, e si trovino alla stessa distanza da s_1 e da s_2.
- Discutere la complessità dell'algoritmo proposto.

La soluzione proposta è la seguente:

Algorithm 92 Grafi-Stessa Distanza

```
1  int stessaDistanza(GRAPH G, NODE s, NODE v)
2          int[] erdos_s = new int[1..G.size()]
3          int[] erdos_v = new int[1..G.size()]
4          NODE parent_s = new NODE[1..G.size()]
5          NODE parent_v = new NODE[1..G.size()]
6          erdos(G, s, erdos_s, parent_s)
7          erds(G, v, erdos_v, parent_v)
8          int nodiStessaDistanza = 0
9          foreach u ∈ G.V() do
10         if erdos_s[u] == erdos_v[u] and erdos_s[u] ≠ +∞ then
11                 nodiStessaDistanza = nodiStessaDistanza + 1
12         return nodiStessaDistanza
```

L'algoritmo utilizza due volte la procedura di Erdos, che è possibile trovare nell'apposita sezione Grafi.
Si ricorda che la procedura di Erdos ritorna la distanza minima, intesa come numero di archi attraversati, dal nodo passato a tutti gli altri nodi, salvando il valore di $+\infty$ se il nodo non è raggiungibile dalla sorgente. Dunque è facile trovare la soluzione al problema trovando i nodi abbiano la stessa distanza dalla rispettiva radice e che siano raggiungibli. La soluzione presentata è dunque corretta, per la breve spiegazione riportata in precedenza, e si può notare che la complessità è dell'ordine di $\Theta(n + m)$

Grafi – Grafi bipartiti

- Un grafo non orientato G è **bipartito** se l'insieme dei nodi può essere partizionato in due sotto insiemi disgiunti tali che nessun arco del grafo connette due nodi appartenenti allo stesso sottoinsieme.
- $G = (V, E)$ è **2-colorabile** se è possibile trovare una 2-colorazione di esso, ovvero un assegnamento $c[u] \in C$ per ogni nodo $u \in V$, dove C è un insieme di "colori" di dimensione 2, tale che:

$$(u, v) \in E \Rightarrow c(u) \neq c(v).$$

- Si dimostri che G è bipartito:
  - se e solo se è 2-colorabile
  - se e solo se non contiene circuiti di lunghezza dispari
- Scrivere un algoritmo che prenda in input un grafo bipartito G e restituisca una 2-colorazione di G sull'insieme di colori $C = 0, 1$, espressa come un vettore $c[...n]$. Discuterne la complessità

La prima soluzione proposta è la seguente:

Algorithm 93 Grafi-bipartiti-bfs

```
1  int[] is2colorabile_bfs(GRAPH G, NODE r)
2        int[] colors = new int[1..G.size()]
3        foreach u ∈ G.V()−{r} do
4        colors[u] = − 1
5        colors[r] = 0
6        QUEUE q = new QUEUE
7        q.enqueue(r)
8        while not q.isEmpty() do
9        NODE u = q.enqueue()
10       foreach v ∈ G.adj(u) do
11               if colors[v] == − 1 then
12                       colors[v] = (colors[u] + 1) mod 2
13                       q.enqueue()
14               else if colors[v] == colors[u] then
15                       return nil % errore nessuna colorazione possibile
16       return colors
```

Questo primo approccio richiede una radice del grafo e cerca di assegnare ad ogni nodo un colore mediante una visita in bfs. Si noti che qualora trova un adiacente con colore uguale al nodo corrente restituisce nil in quanto non sono possibili 2-colorazioni. Ovviamente questa soluzione, dato che sfrutta una bfs funziona solamente per grafi connessi, in quanto colora i nodi che sono raggiungibili dalla radice.
Una soluzione che prevede la visita di tutti i nodi di un grafo può essere data dalla seguente soluzione:

Algorithm 94 Grafi-bipartiti-dfs

```
1  boolean is2colorabile_rec_dfs(GRAPH G, NODE u, int[] colors, int color)
2          colors[u] = color
3          foreach v ∈ G.adj(u) do
4          if colors[v] == −1 then
5                  if not is2colorabile_rec_dfs(G, v, colors, (color + 1) mod 2)
                       then
6                          return false
7          else if colors[v] == color then
8                  return false
9          return true
10
11 int[] is2colorabile(GRAPH G)
12         int[] colors = new int[1..G.size()]
13         foreach u ∈ G.V() do
14         colors[u] = − 1
15         foreach u ∈ G.V() do
16         if colors[u] == −1 then
17                 if not is2colorabile_rec_dfs(G, u, colors, 0) then
18                         return nil
19         return colors
```

Il funzionamento della soluzione con dfs è pressoché identico a quella con bfs, essa si avvale di una funzione i*s2colorabile_rec_dfs*, la quale propaga ricosivamente la colorazione tra nodi, qualora essa trovi lo stesso colore in due nodi adiacenti essa ritorna false e dunque l'intero algoritmo ritornerà nil, in quanto non è possibile alcuna 2-colorazione.
Il colore assegnato al primo nodo di ogni componente connessa è 0, tale valore ovviamente è ininfluente ai fini della correttezza dell'algoritmo.
La soluzione dell'algoritmo proposto è $\Theta(n+m)$ in entrambe le versioni, in quanto si sfrutta una semplice visita di un grafo mediante liste di adiacenza.

Grafi – Distanza fra partizioni

- Dato un grafo G e due sottoinsiemi V_1 e V_2 dei suoi vertici, sidefinisce distanza tra V_1 e V_2 la distanza minima per andare da unnodo in V_1 ad un nodo in V_2, misurata in numero di archi.
- Nel caso V_1 e V_2 non siano disgiunti, allora la distanza è 0.
- Scrivere un algoritmo *mindist(Graph G, Set* V_1*, Set* V_2*)* che restituisce la distanza minima fra V_1 e V_2.
- Discutere complessità e correttezza, assumendo che l'implementazione degli insiemi sia tale che il costo di verificare l'appartenenza di un elemento all'insieme abbia costo $\mathcal{O}(1)$.
- Nota: è facile scrivere un algoritmo $\mathcal{O}(nm)$; esistono tuttavia algoritmi di complessità $\mathcal{O}(n^2)$ (con matrice di adiacenza) e $\mathcal{O}(n+m)$.

Algorithm 95 Distanza fra partizioni

```
1  int distanzaPartizioni(GRAPH G, SET V1, SET V2)
2          QUEUE q = new QUEUE
3          int[] distanze = new int[1..G.size()]
4          foreach u ∈ G.V() do
5          if V1.contains(u) then
6                  q.enqueue(u)
7                  distanze[u] = 0
8          if V2.contains(u) then
9                  return 0
10         else
11                 distanze[u] = +∞
12         while not q.isEmpty() do
13         NODE u = q.dequeue()
14         foreach v ∈ G.adj(u) do
15         if distanze[v] == +∞ then
16                 distanze[v] = distanze[u] + 1
17         if V2.contains(v) then
18                 return distanze[v]
19         q.enqueue(v)
20         return +∞
```

Lo pseudocodice precedente prima di tutto inizializza un vettore delle distanze dai vertici del set V_1 al set V_2, mettendo le distanze di tutti i vettori di V_1 la distanza 0 altrimenti $+\infty$. Qualora un vettore del primo set appartenga al secondo allora la distanza tra le due partizioni è 0.
Successivamente si esegue una visita bfs in cui si sfrutta il principio che sta dietro all'algoritmo dei cammini minimi di Erdos, ovvero assegno ad ogni elemento la distanza del padre dal set $V_1 + 1$, il primo nodo che appartiene a V_2 trovato durante la visita è quello con distanza minima, dunque viene tornato il numero di archi attraversati.
Qualora questo non funzionasse allora le due partizioni appartengono a due componenti connesse distinte per cui la distanza è: $+\infty$.
La complessità della soluzione proposta è pari a $\Theta(n + m)$ dato dal costo della visita bfs.

Grafi – Connetti il grafo

- Progettare un algoritmo efficiente che dato un grafo non orientato, restituisca il numero minimo di archi da aggiungere per renderlo connesso.
- Progettare un algoritmo efficiente che dato un grafo non orientato, aggiunga il numero minimo di archi necessari a renderlo connesso.

Soluzione al primo problema:

Algorithm 96 Numero archi per connettere il grafo

```
1  int n_cc(GRAPH G)
2          int[] ids = new int[1..G.V()]
3          foreach u ∈ a G.V() do
4          ids[u] = 0
5          int counter = 0
6          foreach u ∈ a G.V() do
7          if ids[u] == 0 then
8                  counter = counter + 1
9                  ccdfs(G, u, ids, counter)
10         return counter
11
12 ccdfs(GRAPH G, NODE u, int[] ids, int counter)
13         ids[u] = counter
14         foreach v ∈ G.adj(u) do
15         if ids[v] == 0 then
16         ccdfs(G, v, ids, counter)
17
18 int nArchiMinimi(GRAPH G)
19         if G.size() == 0 then
20         return 0
21         else
22         return n_cc(G) - 1
```

É infatti semplice vedere come il numero di archi necessari per connettere un grafo sconnesso è pari al numero di componenti connesse presenti meno 1. La correttezza è infatti immediata dato che per connettere 2 componenti connesse tra loro è necessario un arco, dunque per connettere n componenti connesse tra loro sono necessari $n-1$ archi. La complessità di tale algoritmo è pari a $\Theta(n+m)$ data dalla visita in dfs. Soluzione al problema del collegamento delle componenti connesse.

Algorithm 97 Connetti il grafo

```
1  connettiGrafo(GRAPH G)
2          int[] ids = new int[1..G.V()]
3          int[] rappresentatives = new int[1..G.size()]
4          foreach u ∈ a G.V() do
5          ids[u] = 0
6          int counter = 0
7          foreach u ∈ a G.V() do
8          if ids[u] == 0 then
9                  counter = counter + 1
10                 % salvo un rappresentante
11                 rappresentative[counter] = u
12                 ccdfs(G, u, ids, counter)
13         % connetto il grafo
14         for i = 1 to counter - 1 do
15                 G.insertEdge(rappresentative[i], rappresentative[i + 1])
16
17 ccdfs(GRAPH G, NODE u, int[] ids, int counter)
18         ids[u] = counter
19         foreach v ∈ G.adj(u) do
20         if ids[v] == 0 then
21         ccdfs(G, v, ids, counter)
```

In questa soluzione, nel momento in cui viene identificata una nuova componente connessa viene salvato il primo nodo visitato in un vettore di rappresentativi, nel quale dato l'indice della componente connessa torna il rappresentante.
Al termine dell'identificazione delle componenti connesse, si esegue un ciclo per inserire i nodi che connetteranno le componenti connesse connettendo i nodi identificati dai rappresentativi.

20.4 Esecitazione 4 - 05/12/19

Chi manca?

Sia dato un vettore ordinato $A[1...n]$ contenente n elementi interi distinti appartenenti all'intervallo $1...n+1$. Si scriva un algoritmo, basato sulla ricerca binaria, per individuare in tempo $\mathcal{O}(\log(n))$ l'unico intero dell'intervallo $1...n+1$ che non compare in A.

Soluzione proposta:

Algorithm 98 Chi manca

```
1  int chiManca(int[] A, int n)
2          if A[n] == n then
3                  return n + 1
4          else
5                  return chiMancaRec(A, 1, n)
6
7  int chiMancaRec(int[] A, int in, int fin)
8          if in == fin then
9                  return in
10         else
11         int mid = (in + fin)/2
12         if A[mid] == mid then
13                 return chiMancaRec(A, mid + 1, fin)
14         else
15                 return chiMancaRec(A, in, mid)
```

Il principio di funzionamento dell'algoritmo è il seguente:
In quanto il vettore è composto dai valori da $1..n+1$ ma è di soli n elementi ordinati occorre vedere se è presente un gap, ovvero se l'elemento analizzato è pari alla posizione, se fosse così allora l'elemento da trovare è nella parte di destra, in quanto la parte di sinistra non contiene gap, altrimenti la ricerca va fatta a destra, ma non rimuovendo l'elemento nella chiamata ricorsiva (ancora non sappiamo se è quell'elemento il gap che cerchiamo). Continuando questa procedura arriveremo ad un singolo elemento, ovvero quello che ha generato il gap.
Tuttavia ci sono altri casi da considerare, in particolare se il vettore A contenesse gli elementi $1..n$ allora il gap non sarebbe presente, andando a neutralizzare di fatto l'algoritmo. Possiamo però gestire questo caso semplicemente in una funzione wrapper che controlla se l'ultimo elemento è pari alla dimensione, in questo caso infatti nel vettore A non sono presenti gap e quindi l'elemento mancante è $n+1$.
La soluzione proposta ha soluzione $\mathcal{O}(\log(n))$

Punto fisso

Progettare un algoritmo che, preso un vettore ordinato A di interi distinti, determini se esiste un indice i tale che $A[i] = i$ in tempo $\mathcal{O}(\log(n))$.

La soluzione proposta è la seguente:

Algorithm 99 Punto fisso

```
boolean puntoFisso(int[] A, int n)
        return puntoFissoRec(A, 1, n)

boolean puntoFissoRec(int[] A, int inizio, int fine)
        if inizio > fine then
        return false
        else
        int mid = (inizio + fine)/2
        if A[mid] == mid then
                return true
        else
                if A[mid] < mid then
                        return puntoFissoRec(A, mid + 1, fine)
                else
                        return puntoFissoRec(A, inizio, mid - 1)
```

Il punto fisso per come è definito è semplicemente la ricerca di un elemento che sia uguale alla sua posizione, ovviamente, in quanto il vettore è ordinato, so che se l'elemento è minore dell'indice della metà del vettore allora sicuramente si trova nella parte sinistra, altrimenti nella parte di destra.
La soluzione proposta ha soluzione $\mathcal{O}(\log(n))$

Vettori uni-modulari

Un vettore di interi A è detto unimodulare se ha tutti valori distinti ed esiste un indice h tale che $A[1] > A[2] > ... > A[h-1] > A[h]$ e $A[h] < A[h+1] < A[h+2] < ... < A[n]$, dove n è la dimensione del vettore.
Progettare un algoritmo $\mathcal{O}(\log(n))$ che dato un vettore unimodulare restituisce il valore minimo del vettore.

Soluzione proposta:

Algorithm 100 Vettore unimodulare

```
int minVettoreUniModulare(int[] A, int n)
        return minVettoreUniModulareRec(A, 1, n)

int minVettoreUniModulareRec(int[] A, int inizio, int fine)
        if inizio == fine then
        return A[i]
        else
        if inizio == fine + 1 then
                return min(A[inizio], A[fine])
        else
                int mid = (inizio + fine)/2
                if A[mid - 1] > A[mid] and A[mid] < A[mid + 1] then
                        return A[mid]
                else
                        if A[mid] < A[mid + 1] then
                                return minVettoreUniModulareRec(A, inizio, mid -
                                    1)
                        else
                                return minVettoreUniModulareRec(A, mid + 1, fine
                                    )
```

L'algoritmo precedente si comporta come segue:
Caso base: se è presente un singolo elemento allora l'elemento è il minimo del vettore unimodulare, se invece rimangono presenti due elementi allora ritorna il minimo dei due.
Caso ricorsivo: determino l'elemento centrale, se l'elemento rispecchia la definizione di minimo in

tale vettore allora l'elemento è stato trovato, altrimenti in base ad un semplice confronto determino se proseguire nella ricorsione nella parte sinistra o destra.
La complessità della soluzione proposta è $\mathcal{O}(\log(n))$

Samarcanda
Nel gioco di Samarcanda, ogni giocatore è figlio di una nobile famiglia della Serenissima, il cui compito è di partire da Venezia con una certadotazione di denari, arrivare nelle ricche città orientali, acquistare le merci preziose al prezzo più conveniente e tornare alla propria città perrivenderle. Dato un vettore P di n interi in cui $P[i]$ è il prezzo di una certa merce al giorno i, trovare la coppia di giornate *(x, y)* con $x < y$ per cui risultamassimo il valore $P[y] - P[x]$. Calcolare la complessità e dimostrare la correttezza.

Soluzione proposta:

Algorithm 101 Samarcanda

```
1  int samarcanda(int[ ] A,int i,int j)
2          if i ≥ j then
3          return 0
4          int m = (i+j) / 2
5          int maxLeft = samarcanda(A, i, m)
6          int maxRight = samarcanda(A, m + 1, j)
7          int ml = min(A, i, m)
8          int mr = max(A, m + 1, j)
9          int maxCross = max(0, mr − ml)
10         return max(maxLeft, maxRight, maxCross)
```

Struttura della soluzione:

- Si divide il vettore a metà
- Si applica ricorsivamente la soluzione nelle due metà, ottenendo la migliore soluzione nella metà destra e nella metà sinistra
- Da questo approccio, non vengono considerate le soluzioni in cui si acquista nella metà sinistra del vettore, si vende nella metà destra
- Si cerca quindi il minimo a sinistra e il massimo a destra e si applica ricorsivamente la soluzione
- Caso base: un elemento solo, che ovviamente da origine a zero come massimo guadagno

Questa soluzione è in complessità di $\mathcal{O}(n\log(n))$ che deriva dalla seguente equazione di ricorrenza:

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

Dal *Master Theorem* segue che:
$a = 2$, $b = 2$, $\alpha = \log_b(a) = \log_2(2) = 1$, $\beta = 1$.
In quanto $\beta = \alpha$ allora $T(n) = \Theta(n^\alpha \log(n)) = \Theta(n\log(n))$
Si noti che esiste anche una soluzione in $\mathcal{O}(n)$ che sfrutta la programmazione dinamica, in particolare l'algoritmo di Kadane. Tuttavia in quanto l'esercitazione è sulla tecnica di programmazione *Divide et impera* non è stata inclusa.

20.5 Esecitazione 5 - 19/12/19

Per fare un albero (binario di ricerca) ci vuole...
Dato un vettore V di n interi ordinati e distinti, scrivere una procedurache costruisce un albero binario di ricerca di altezza minima.Discuterne la correttezza e la complessità

Soluzione proposta:

Algorithm 102 Costruzione ABR a partire da vettore ordinato

```
1  Tree buildTree(int[] A, int n)
2          return buildTreeRec(A, 1, n)
3
4  Tree buildTreeRec(int[] A, int inizio, int fine)
5          if inzio > fine then
6          return nil
7          else
8          int meta = (inizio + fine)/2
9          Tree albero = new Tree(A[meta])
10         albero.insertLeft(buildTreeRec(A, inzio, meta - 1))
11         albero.insertRight(buildTreeRec(A, meta + 1, fine))
12         return albero
```

Il funzionamento dell'algoritmo proposto in precedenza è molto semplice, in quanto il vettore è ordinato per costruire l'albero di altezza minima è encessario che abbia alla sinistra e alla sua destra lo stesso numero di nodi o che differiscano di al massimo 1 (Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v deve ≤ 1). Si noti che prendendo sempre il valore mediano i due sottovettori ottenuti sono composti da un numero di elementi che differisce l'uno dall'altro, al più di 1. La parte di sinistra contiene valori minori del valore centrale, mentre la parte di destra valori maggiori, tali parti andranno a comporre rispettivamente il sottoalbero sinistro e destro dell'albero con radice il valore centrale. L'algoritmo così descritto è dunque corretto ed ha complessità data dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 2T(n/2) & n > 0 \\ 1 & n \leq 0 \end{cases}$$

Dal *Master Theorem* sappiamo che:
$a = 2, b = 2, \beta = 0, \alpha = \log_a(b) = \log_2(2) = 1$
Dunque: $\alpha > \beta$ allora $T(n) = \Theta(n^\alpha) = \Theta(n)$

Good vs bad guys
Fra ogni coppia di wrestler professionisti può esserci una rivalità oppureno. Per ragioni di marketing, è una buona idea dividere i wrestler professionisti in due gruppi, "buoni" e "cattivi", e farli combattere fra di loro. Supponete di avere in input un insieme di rivalità, rappresentate comeun vettore di coppie *(x, y)*, dove x e y sono identificatori di wrestler compresi fra *1* ed *n*. Scrivere un algoritmo che restituisca *true* se è possibile suddividere i wrestler professionisti in due sottoinsiemi non vuoti ("buoni" e"cattivi"), non necessariamente della stessa dimensione, in modo tale che non ci siano rivalità all'intero dei due gruppi; *false* altrimenti.

Soluzione proposta:

Algorithm 103 2 colorazione di Grafo con i wrestler

```
1  GRAPH rivalitaWrestler(int[] x, int[] y, int n)
2          GRAPH G = new GRAPH
3          for indice = 1 to n do
4          G.insertNode(indice)
5          for indice = 1 to n do
6          G.insertEdge(x[i], y[i])
7          G.insertEdge(y[i], x[i])
8          return G
9
10 boolean 2wrestlercolorabile(GRAPH G)
11         int[] colors = new int[1..G.size()]
12         foreach u ∈ G.V() do
13         colors[u] = - 1
14         foreach u ∈ G.V() do
15         if colors[u] == -1 then
16                 if not 2wrestlercolorabile_rec(G, u, colors, 0) then
17                         return false
18         return true
19
20 boolean 2wrestlercolorabile_rec(GRAPH G, NODE u, int colors[], int color)
21         colors[u] = color
22         foreach v ∈ G.V() do
23         if colors[v] == -1 then
24                 if not 2wrestlercolorabile_rec(G, v, colors, (color + 1) mod 2)
25                         return false
26         else
27                 if colors[v] == color then
28                         return false
29         return true
```

Il problema proposto è un problema di grafo 2-colorabile, infatti è necessario, dopo la costruzione del grafo a partire dai vettori di rivalità, effetuare una visita in dfs con la colorazione del grafo. La spiegazione della colorazione del grafo la si può trovare nel'esercitazione 3 sotto la voce "Grafi bipartiti".

Grafi – Pozzo universale

- Un pozzo universale è un nodo con out-degree uguale a zero e in-degree uguale a $n - 1$.
- Dato un grafo orientato G rappresentato tramite matrice diadiacenza, scrivere un algoritmo che opera in tempo $\Theta(n)$ in grado di determinare se G contiene un pozzo universale.
- È possibile ottenere la stessa complessità con liste di adiacenza?

La soluzione proposta è la seguente:

Algorithm 104 Pozzo universale

```
1  boolean hashPozzo(NODE[][] G, int n)
2          int candidate = 0 % riga candidata
3          % scorro le colonne
4          for indice = 1 to n do
5                  % se trovassi uno 0 allora proseguo, candidato valido, gli altri
                       non lo sono di sicuro perche' non hanno quel in degree che
                      gli serve
6          if G[candidate][indice] == 1 then
7                  % cambio candidato a quello indicato dall'indice
8                  candidate = indice
9                          % finito il for ho la mia riga candiata, la sua colonna
                              deve contenere tuti 1 per essere un effettivo pozzo
                              universale
10         for riga = 1 to n do
11         if riga != candidate then
12                 if A[riga][candidate] == 0 then
13                         return false
14         % deve anche contenere tutti 0 sulla sua riga
15         for colonna = 1 to n do
16                 if A[candidate][colonna] == 1 then
17                 return false
18         return true
```

La soluzione di Montresor è la seguente:

Algorithm 105 Pozzo universale Montresor

```
1  boolean universalSink(NODE[][] G, int n)
2          int i = 1
3          boolean candidate = false
4          while i < n and candidate == false do
5          int j = i + 1
6          while j ≤ and A[i][j] == 0 do
7                  j = j + 1
8          if j > n then
9                  candidate = true
10         else
11                 i = j
12         int rowtot = ∑_{j∈1..n} A[i][j]
13         int coltot = ∑_{j∈1..n} A[j][i]
14         return rowtot = 0 and coltot == n - 1
```

La logica che sta dietro alla soluzione è la seguente:
In quanto in input abbiamo una matrice di adiacenza, allora un pozzo universale è dato dall'indice della riga composta da tutti 0, la cui colonna (sempre allo stesso indice) è composta da tutti 1,

fatta eccezione per la cella alla sua riga.
Osserviamo inoltre che non è possibile che ci siano più pozzi universali, in quanto se ne esistesse un altro allora dovrebbe avere $n-1$ come in-degree per definizione, ma questo è impossibile dato che esiste un altro pozzo universale che ha out-degree pari a 0.
L'obiettivo è quello di trovare un possibile candidato attraverso una visita il più possibile lineare, per farlo scorriamo la prima riga: nel momento in cui troviamo uno 0, significa che la riga corrente è un potenziale pozzo universale, mentre la colonna di tale 0 non è sicuramente un pozzo universale dato che non può avere in-degree pari a $n-1$.
Nel momento in cui troviamo una cella di valore 1 allora tale riga non può essere un pozzo universale, ci spostiamo dunque sulla riga identificata dalla colonna del valore 1 trovato, questo perché le righe corrispondenti alle colonne precedenti sono state scartate, se si è verificato il caso precedente, mentre non possiamo scartare tale riga in quanto può avere in-degree pari a $n-1$ e dunque un potenziale candidato.
Arrivati al termine della visita lineare abbiamo un candidato, dunque verifichiamo se esso è realmente un pozzo universale effettuando, con costo ancora lineare, la somma di tutti i valori sulla sua riga e sulla sua colonna; se out-degree $= 0$ e in-degree $= n-1$ abbiamo trovato il pozzo universale.
Il costo dell'algoritmo è pari a 3 visite di grandezza n, dunque: $\Theta(n)$.

Ricorrenza $4T(\sqrt(n)) + \log^2(n)$

Si ottengano limiti superiori e inferiori per la seguente ricorrenza:

$$T(n) = \begin{cases} 4T(\lfloor\sqrt(n)\rfloor) + \log^2(n) & n > 1 \\ 1 & n = 1 \end{cases}$$

Questa equazione, in quanto presenta una radice è difficile da portare ad una forma chiusa, dobbiamo dunque ricorrere ad una supposizione che ci semplificherà i calcoli. In particolare: Poniamo $n = 2^k$, sostituendo tale valore nella ricorrenza otteniamo:
$T(2^k) = 4T(\sqrt(2^k)) + \log^2(k^2) = 4T(2^{k/2}) + k^2$

Sostituiamo quindi la variable $T(2^k)$ con $S(k)$ e otteniamo tramite *Master Theorem*
$S(k) = 4S(\frac{k}{2}) + k^2$
Vale:
$a = 4, b = 2, \beta = 2, \alpha = \log_b(a) = \log_2(4) = 2$
Segue che: $\alpha = \beta$ dunque $S(k) = \Theta(k^\alpha \log(k)) = \Theta(k^2 \log(k))$
Esprimiamo la ricorrenza in termini di $T(n)$ applicando la sostituzione $k = \log(n)$, derivata dalla precedente sostituzione $n = 2^k$:
$T(n) = \Theta(\log^2(n) \log(\log(n)))$

20.6 Esecitazione 6 - 20/02/20

Ottimizza la somma
Supponete di avere in input un vettore dininteri positivi distinti $V[1...n]$ e un valore W. Scrivere un algoritmo che:

1. restituisca il massimo valore $X = \sum_{i=1}^{n} x[i]V[i]$ tale che $X \leq W$ e ogni $x[i]$ è un intero non negativo

2. stampi il vettore x

Ad esempio, per $V = [18, 3, 21, 9, 12, 24]$ e $W = 17$, una possibile soluzione ottima è $x = [0, 2, 0, 1, 0, 0]$ da cui deriva $X = 15$.
Discutere correttezza e complessità.

Da un'osservazione più attenta del problema ci si accorge che non è altro che una versione modificata di Knapsack (versione Senza Limiti) dove w e p, rispettivamente i vettori che indicano il profitto e il peso di un oggetto, sono entrambi rappresentati da V. Nonostante bastasse richiamare l'algoritmo ricorsivo proposto nella parte di memoization proproniamo una versione iterativa che permette di stampare anche il vettore x:

Algorithm 106 Ottimizza la Somma

```
1  int knapsack(int[] V, int n, int W)
2          int x[] = new int[1...n]
3          DP[] = new int[0...n][0...W]
4          for i = 0 to n do
5                  DP[i][0] = 0
6          for w = 0 to W do
7                  DP[0][w]
8          for i = 1 to n do
9                  x[i] = 0
10         for i = 1 to n do
11                 for w = 1 to W do
12                         if V[i] ≤ w then
13                                 DP[i][w] = max(DP[i][w - V[i]] + V[i], DP[i-1][w
                                       ])
14                         else
15                                 DP[i][w] = DP[i-1][w]
16         solution(DP, n, W, V, x)
17         print("[")
18         for i = 1 to n do
19                 print(x[i])
20                 print("␣")
21         print("]")
22         return DP[n][W]
23
24 void solution(int[][] DP, int n, int W, int[] V, int[] x)
25         if n > 0 and W > 0 then
26                 if DP[n][W] == DP[n-1][W] then
27                         solution(DP, n-1, W, V, x)
28                 else
29                         x[n]++
30                         solution(DP, n, W-1, V, x)
```

20.7 Esecitazione 7 - 02/03/20

Mosse su scacchiera
TODO

Donald Trump
TODO

Palindroma
TODO

21 Laboratori

21.1 Laboratorio 7 - 04/03/20

Zaino

Avete uno zaino di capacità C ed un insieme di N oggetti. Per ognuno di questi oggetti, sapete il peso (Pi) e il valore (Vi). Dovete scegliere quali oggetti mettere nello zaino in modo da ottenere il maggior valore possibile senza superare la capacità dello zaino.

Soluzione proposta:

Algorithm 107 Zaino (ottimizzazione memoria)

```
%Inizializzazione dei vettori soluzione
%Questi vettori indicano ad un particolare elemento la sua soluzione con le relative capacita'. Le soluzioni ottime per capacita' sono ottenute a tale indice
%Usiamo C + 1 per un motivo di coerenza con l'algoritmo usato e per leggibilita' del codice
int DP_precedente[C]
int DP_successivo[C]
%Vettori Peso e Valore dell'oggetto considerato
int P[N]
int V[N]

int knapsack(int[] DP_precedente, int[] DP_successivo, int[] P, int[] V, int C, int N){
        %Inizializzo il vettore dell'elemento precedente, il successivo verra' calcolato in seguito
        for c = 0 to C do
                DP_precedente[c] = 0
        int taken, not_taken
        %Puntatore per lo swap dei due vettori (swap di indirizzi)
        int *swap
        %Costruzione della soluzione
        %Noto: salvo solo la riga dell'elemento i-1 esimo con tutte le possibili capacita' (le righe ancora precedenti non le uso piu')
        for i = 0 to N do
                for c = 0 to C do
                        %Se l'elemento ci sta nello zaino con capacita' considerata
                        if P[i] <= c then
                                taken = DP_precedente[c - P[i]] + V[i]
                                not_taken = DP_precedente[c]
                                DP_successivo[c] = max(taken, not_taken)
                        else
                                DP_successivo[c] = DP_precedente[c]
        %Effettuiamo uno scambio di vettori (scambiamo gli indirizzi che referenziano il primo elemento del vettore)
        %Ora il vettore relativo all'oggetto precedente referenzia l'elemento successivo, mentre il vettore che riguarda l'oggetto successivo referenzia quello precedente che sovrascrivera'
                swap = DP_successivo
                DP_successivo = DP_precedente
                DP_precedente = swap
        %Per via della logica dello scambio la soluzione si trova nell'oggetto precedente (ovvero oggetto i-esimo) con capacita' C considerata
        return DP_precedente[C];
```

L'algoritmo riprende la logica della versione dello zaino (non illimitato) implementato con programmazione dinamica aggiungendo un'ottimizzazione a livello di memoria: non salvo tutta la matrice DP, ma considero solo due righe di tale matrice (salvate tramite array paralleli): la riga dell'elemento i-esimo per tutte le capacità possibili e la riga dell'elemento i-1 per tutte le capacità e conterrà il profitto massimo calcolato fino ad ora. Questa ottimizzazione nasce dal fatto che la formula usata per l'implementazione considera solamente la riga DP[i-1][...] per effettuare i calcoli successivi e nessun altra riga tra le precedenti e le successive.

Sottoinsieme Crescente di Somma Massima(sottocres)
Vi viene dato in input un array di N interi $A_1 ... A_n$. Per ogni elemento potete scegliere se includerlo nell'insieme soluzione, o se ignorarlo. Se un elemento A_i fa parte dell'insieme, tutti gli elementi che lo succedono nell'array e che hanno valore minore di A_i non possono essere inclusi nell'insieme. In altre parole, gli elementi dell'insieme, quando stampati nell'ordine in cui si trovavano nell'array, devono formare una sequenza crescente. Vogliamo massimizzare la somma degli elementi dell'insieme.

Soluzione proposta: il seguente problema può essere risolto mediante Programmazione Dinamica poiché è possibile notare che, provando a cercare la massima sequenza di numeri crescenti, continuiamo a risolvere gli stessi sottoproblemi più e più volte.

Algorithm 108 Sottoinsieme Crescente di Somma Massima

```
1  int sottocres(int n, int[] A)
2          int[] DP = new int[n]
3          DP[1] = A[1]
4          for i = 2 to n do
5                  DP[i] = A[i]
6                  for j = 1 to i do
7                          if A[i] ≥ A[j] and DP[i] < DP[j] + A[i] then
8                                  DP[i] = DP[j] + A[i]
9          int massimo = −∞
10         for i = 0 to n do
11                 massimo = max(massimo, DP[i])
12         return massimo
```

Spiegazione: ad ogni iterazione del ciclo for andiamo ad inizializzare l'elemento $DP[i]$ con $A[i]$; questa operazione viene fatta per due motivi: il primo è che magari l'emento che stiamo analizzando è di per sè già la più grande sottosequenza crescente che possiamo ottenere all'interno del nostro input, il secondo è perchè, non avendo inizializzato il vettore DP, la nostra funzione farebbe degli accessi non leciti. All'interno del primo ciclo si ha un secondo ciclo (questo ci suggerisce che la complessità della nostra funzione è $\mathcal{O}(n^2)$) che serve per controllare nuovamente tutti gli elementi che abbiamo inserito fino ad ora ed inserire con certezza il massimo in quella casella. Il controllo che viene eseguito si occupa di verificare non solo che $A[i] \geq A[j]$ (notare il fatto che sono ritenuti validi anche numeri uguali perchè definito in questi termini il problema) ma anche che $DP[i] < DP[j] + A[i]$ e che quindi, come dicevamo prima, ci sia effettivamente un guadagno rispetto al numero a cui inizializzamo la cella, oppure che, nel caso di più possibilità, consideriamo effettivamente quella che porta il guadagno massimo.

Le pillole della zia (pillole)
La zia Lucilla deve assumere ogni giorno mezza pillola di una certa medicina. Lei inizia il trattamento con un bottiglia che contiene esattamente N pillole.
Durante il primo giorno lei prende una pillola dalla bottiglia, la spezza in due, ne ingerisce una metà e rimette l'altra metà nella bottiglia.
Nei giorni seguenti lei prende un pezzo a caso della bottiglia (potrebbe essere una pillola intera o

una mezza pillola).
Se ha pescato una mezza pillola la ingerisce. Se ha pescato una pillola intera la spezza a metà, rimette una delle due mezze pillole nella bottiglia e ingerisce l'altra mezza pillola. La zia può svuotare la bottiglia in tanti modi diversi. Rappresentiamo la cura come una stringa di $2N$ caratteri, in cui il carattere i-esimo è "I" se la zia ha pescato una pillola intera nel giorno i e "M" se la zia ha invece pescato una mezza pillola. Nel caso in cui la bottiglia originaria contenga 3 pillole intere, le possibili sequenze sono le seguenti:
$IIIMMM$
$IIMIMM$
$IIMMIM$
$IMIIMM$
$IMIMIM$
Il problema vi richiede di scrivere un programma che, dato N, restituisca il numero di possibili sequenze nel trattamento.

Soluzione proposta: il seguente problema può essere risolto mediante Memoization creando una Matrice $DP[N][N]$ dove indichiamo sulle colonne il numero di pastiglie intere mentre sulle righe il numero di pastiglie a metà. Ad esempio con 3 pastiglie il nostro risultato si troverà in posizione $DP[0][3]$. Semplicemente ogni volta ci chiederemo se spezzare la pastiglia (e quindi andare in posizione $DP[i+1][j-1]$) oppure se prendere una metà (e accedere dunque in posizione $DP[i][j-1]$). Ovviamente questa operazione non può essere svolta sempre in maniera duplice (ad esempio partendo con 3 pastiglie intere possiamo soltanto spezzarne una, è indifferente quale) e i casi base sono quando abbiamo solo pastiglie a metà (quindi per $DP[i][0]$), quando si ha soltanto una pastiglia intera ($DP[0][1]$) oppure quando non se ne ha nessuna ($DP[0][0]$).

Algorithm 109 Le pillole della Zia

```
1  long long int pilloleRec(int row, int col, long long int[][] DP)
2          long long int contenuto = DP[row][col]
3          if contenuto ≠ 0 then
4                  return contenuto
5          else
6                  if row == 0 then
7                          DP[row][col] = pilloleRec(row+1, col-1, DP)
8                  else
9                          DP[row][col] = pilloleRec(row+1, col-1, DP) +
10                                pilloleRec(row+1, col, DP)
11                 return DP[row][col]
12
13 long long int pillole(int N)
14         long long int[][] DP = new long long int[N][N]
15         for i = 0 to N do
16                 for j = 0 to N do
17                         if j == 0 then
18                                 DP[i][j] = 1
19                         else
20                                 DP[i][j] = 0
21         DP[0][1] = 1
22         return pilloleRec(0, N, DP)
```

Esempio riempimento tabella:

$j\backslash i$	0	1	2	3
0	1	1	2	$\swarrow$ 5
1	1	$\updownarrow\swarrow$2	$\updownarrow\swarrow$5	0
2	1	$\updownarrow\swarrow$3	0	0
3	1	0	0	0

22 Tutorati: Esercizi Utili

22.1 ASD1

Albero di Natale

Vedi Sezione Alberi Binari di Ricerca: Esempi BFS.

22.2 ASD2

Occorrenze di valori in un ABR

Algorithm 110 Occorrenze di valori in un ABR

```
1  int occorrenze(Tree T, int min, int max)
2    if T == nil then
3      return 0
4    else
5      if T.key() < min then
6        % lo mando a desta
7        return occorrenze(T.right(),min,max)
8      else if T.key() > max then
9        % lo mando a sinistra
10       return occorrenze(T.left(),min,max)
11     else
12       return 1 + occorrenze(T.right(),min,max) + occorrenze(T.left(),min,max)
```

Semplice algoritmo che sfrutta il principio della lookup in un albero binario di ricerca, restituendo il numero di occorrenze di valori compresi tra min e max (estremi inclusi). Se il valore da confrontare risulta essere minore del minimo dell'intervallo allora è inutile inoltrare la chiamata nella parte di sinstra in quanto conterrà solamente valori minori di quello visitato. Un discorso analogo può essere fatto se il valore esaminato risulta essere maggiore del massimo, in quel caso infatti propago l'algoritmo solamente nella sua parte sinistra.

Alberobello

La funzione nicetree riceve in input un array di caratteri contenente la visita in pre ordine di un albero; sfruttando la visita in pre ordine è possibile fare un visita fittizia dell'albero tenendo conto delle caratteristiche di un alberobello: se troviamo una I all'interno dell'array di caratteri sappiamo che questo avrà per forza di cose un sottoalbero destro e uno sinistro e, per come è definita la visita in pre ordine di un albero, visiteremo per forza di cose prima tutto il suo sottoalbero sinistro. Invece quando incontreremo una L sapremo di essere arrivati in una foglia e quindi che dovremo risalire ed guardare il prossimo carattere. Visto che il problema richiede l'altezza dell'albero il caso base della nostra funzione sarà per l'appunto trovare una L e quindi risalire nella ricorsione ad un livello superiore, mentre ogni volta che troveremo una I dovremo prima chiamare la ricorsione sul sottoalbero sinistro (incrementando mano a mano l'altezza) e, solamente dopo essere certi di averla visitata completamente, fare la ricorsione sul sottoalbero destro.

Algorithm 111 Alberobello

```
int nicetreeRec(ITEM[] S, int& pos, int n)
        if pos ≤ n then
        if S[pos] == ''L'' then
                pos = pos + 1
                return 0
        else
                int altezzaSx, altezzaDx
                pos = pos + 1
                altezzaSx = 1 + nicetreeRec(S, pos, n)
                altezzaDx = 1 + nicetreeRec(S, pos, n)
                return max(altezzaDx, altezzaSx)
        else
        return 0

int nicetree(ITEM[] S, int n)
        int pos = 1;
        if n > 0 then
        return nicetreeRec(S, pos, n)
        else
        return 0
```

22.3 ASD3

É un albero red-black?

Algorithm 112 É un albero red-black?

```
boolean isRedBlack(TREE T)
        % controllo sulla radice
        if T.color == RED then
        return false
        else
        return (blackHeight(T) > 0)

int blackHeight(TREE T)
        % Tutte le foglie devono essere rosse
        if T.value == nil then
        return iif(T.color == RED, -1, 1)
        % Ogni nodo rosso deve avere entrambi i figli neri
        if T.color == RED and T.parent ≠ nil and T.parent.color == RED then
        return -1
        % Il numero di nodi neri incontrati su tutti i possibili percorsi dalla
            radice ad una foglia deve essere lo stesso
        int bhL = blackHeight(T.left)
        int bhR = blackHeight(T.right)
        if bhL < 0 or bhR < 0 or bhL ≠ bhR then
        return -1
        else
        return bhL + iif(t.color == BLACK, 1, 0)
```

Per verificare se un albero è red-black è necessario assicurarsi che tutti i vincoli siano rispettati, in particolare che nessuno di essi venga violato:

- La radice deve essere nera
- Tutte le foglie sono nere

- Entrambi i figli di un nodo rosso sono neri
- Tutti i cammini semplici da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri.
  Una volta ricevuto l'albero in input l'algoritmo ne controlla la radice, restituendo false se il vincolo viene violato. A questo punto l'algoritmo si appoggia su una procedura ricorsiva che restituisce iun intero (necessario per il vincolo 4, ovvero quello dei cammini). Ogni volta che la parte ricorsiva si porta in una foglia (il valore è nil) allora controlla che il suo colore sia corretto, se si allora ritorna 1 (in quanto è un nodo nero), -1 altrimenti. Viene verificato il vincolo 3 mediante un controllo sul padre qualora il colore della foglia esaminata sia rossa. Controlla infine le altezze nere, se non combaciano, oppure un vincolo è stato violato nei livelli sottostanti (il valore ottenuto è < 0) esso torna -1. In questo modo è possibile risalire con un semplice controllo di maggiore se i vincoli sono stati rispettati o meno.

22.4 ASD4

Ricorrenza Parametrizzata rispetto a K

$$T_k(n) = \begin{cases} k^2 T_k(n/k) + n^{k/2} & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Possiamo notare che è possibile appicare il master theorem:
$a = k^2$, $b = k$, $\beta = k/2$, $\alpha = \log_n k^2 = 2$
Se $k = 2 \implies \beta = 1 \implies \alpha > \beta \implies \Theta(n^2)$
Se $k = 3 \implies \beta = 3/2 \implies \alpha > \beta \implies \Theta(n^2)$
Se $k = 4 \implies \beta = 2 \implies \alpha = \beta \implies \Theta(n^2 \log n)$
Risulta dunque chiato che l'algoritmo che va scartato è quello per $k = 4$.

22.5 ASD5

Grafo etichettato

Algorithm 113 Il grafo contiene la sottosequenza LODE?

```
1  boolean checkPath(GRAPH G, char[] l, char[] A, int n)
2          foreach u ∈ G.V() do
3          if checkLabels(G, l, A, n,1, u) then
4                  return true
5
6  boolean checkLabels(GRAPH G, char[] l, char[] A, int n, int i, int u)
7          if l[u] = A[i] then
8          if i = n then
9                  return true
10         foreach v ∈ G.adj(u) do
11                 if checkLabels(G, l, A, n, i+1, v) then
12                         return true
13         return false
```

Questo algoritmo per risolvere il problema effettua una visita a partire da ogni nodo, verificando nodo per nodo se la sequenza viene rispettata, altrimenti, interrompe la visita qualora le etichette non corrispondano. Semplicemente viene fatto passare per ogni nodo l'algoritmo *checkLabels*, il

quale in dfs controlla se il suo valore è pari al valore del vettore nella posizione passata, se si allora esso propaga l'algoritmo con la posizione successiva agli adiacenti, altrimenti non fa nulla. Qualora si arrivasse alla fine del percorso l'algoritmo ritorna true. Si noti che la complessità è superpolinomiale, in quanto è necessario seguire tutti i possibili percorsi, anche tenendo conto che il grafo e aciclico.

Griglia quadra

Soluzione con costruzione del grafo:

Algorithm 114 Griglia quadra - costruzione grafo

```
1  (GRAPH, int[]) costruisciGrafo(int[][] A, int n)
2          GRAPH G = new GRAPH
3          int[] colors = new colors[1..n]
4          % inserimento nodi
5          for i = 1 to n · n do
6          G.insertNode(i)
7          % inserimento archi
8          for riga = 1 to n do
9          for colonna = 1 to n do
10                 % id sequenziale del nodo
11                 int idNodo = colonna + (riga − 1) · n
12                 colors[idNodo] = A[riga][colonna]
13                 int idAdiacente
14                 if riga ≠ 1 then
15                         idAdiacente = colonna + (riga − 2) · n
16                         G.insertEdge(idNodo, idAdiacente)
17                 if colonna ≠ 1 then
18                         idAdiacente = (colonna − 1) + (riga − 1) · n
19                         G.insertEdge(idNodo, idAdiacente)
20                 if colonna ≠ n then
21                         idAdiacente = (colonna + 1) + (riga − 1) · n
22                         G.insertEdge(idNodo, idAdiacente)
23                 if riga ≠ n then
24                         idAdiacente = colonna + riga · n
25                         G.insertEdge(idNodo, idAdiacente)
26         return (G, colors)
27
28 int distanza1_3(int[][] A, int n)
29         GRAPH G
30         int[] colors = new int[1..n·n]
31         (G, colors) = costruisciGrafo(A, n)
32         QUEUE q = new QUEUE
33         int[] distanze = new distanze[1..G.size()]
34         foreach u ∈ G.V() do
35         if colors[u] == 1 then
36                 distanze[u] = 0
37                 q.enqueue(u)
38         else
39                 distanze[u] = +∞
40         while not q.isEmpty() do
41         NODE u = q.dequeue()
42         foreach v ∈ G.adj(u) do
43                 if distanze[v] == +∞ then
44                         distanze[v] = distanze[u] + 1
45                         if colors[v] == 3 then
46                                 return distanze[v]
47                         else
48                                 q.enqueue(v)
49         return +∞
```

La costruzione del grafo partendo dalla matrice ha un costo di $\Theta(n^2)$, e si può fare con dei semplici accorgimenti: un assegnazione univoca di un identificatore del nodo, semplicemente un numero da 1 a n^2 e un controllo sugli indici per non selezionare un nodo non esistente.
Questo algoritmo costruisce un grafo partendo da una matrice, si occupa inoltre di inserire all'interno di un vettore *colors[i]* il valore della cella della matrice associata al nodo i.
Avendo questo è semplice trovare la distanza minima dalle celle di colore 1 ad una cella di colore 3, effettuando una bfs assegnando distanza 0 dei nodi con colore pari a 1 e procedendo come insegna *Erdos*. Ritorniamo la distanza della prima occorrenza di colore 3 in quanto essendo la bfs una visita per livelli, la prima occorrenza è anche la meno distante.

Soluzione con bfs direttamente sulla matrice:

Algorithm 115 Griglia quadra

```
int grid(int[][] M, int n)
  int[] dr = [-1, 0, +1, 0] % Mosse possibili sulle righe
  int[] dc = [0, -1, 0, +1] % Mosse possibili sulle colonne
  int[] distance = new int[1...n][1...n]
  QUEUE Q = QUEUE
  for r = 1 to n do
    for c = 1 to n do
      distance[r][c] = iif(M[r][c] == 1, 0, -1)
      if M[r][c] == 1 then
        Q.enqueue(<r, c>)
  while not Q.isEmpty() do
    int, int r, c = Q.dequeue()   % Riga, colonna della cella corrente
    for i = 1 to 4 do
      nr = r + dr[i] % Nuova riga
      nc = c + dc[i] % Nuova colonna
      if 1 ≤ nr ≤ n and 1 ≤ nc ≤ n and distance[nr][nc] < 0 then
        distance[nr][nc] = distance[r][c] + 1
        if M[nr][nc] == 3 then
          return distance[nr][nc]
        else
          Q.enqueue(<nr, nc>)
```

Il principio di funzinamento è sempre lo stesso, ovvero una visita in bfs assegnando ai nodi di colore 1 la distanza 0. Il foreach della bfs viene fatto corrispondere ad un for sulle mosse possibili. Viene inoltre controllato se viene sforato uno dei possibili indici, se non viene sforato allora ci si comporta come nell'algoritmo precedente.

22.6 ASD6

Nodi a distanza d

Algorithm 116 Nodi che distano un numero di archi $\leq$ d

```
int nodiMinUgualeDistanza(GRAPH G, NODE r, int d)
        int[] distanze = new int[1..G.size()]
        NODE[] parents = new NODE[1..G.size()]
        erdos(G, r, distanze, parents)
        int nNodi = 0
        foreach u ∈ G.V() do
        if distanze[u] ≤ d then
                nNodi = nNodi + 1
        return nNodi
```

Algoritmo molto semplice, una volta trovate tutte le distanze dei nodi dalla radice grazie ad *Erdos* essa confronta le distanze con il valore d, se minore o uguale allora incrementa un contatore, che sarà ritornato alla fine dei confronti.

Albero Allopato
Un grafo e un "albero binario alloopato" se deriva da un albero binario nel modo seguente: è presente un nodo del grafo per ogni nodo dell'albero; e presente un arco per ogni coppia *(padre,figlio)*, orientato dal padre al figlio; e presente un arco da ogni foglia alla radice dell'albero binario, chiudendo così un ciclo (loop). Gli archi sono pesati da una funzione $W : E \to Z$; gli archi possono avere peso negativo, ma non esistono cicli la cui somma dei pesi sia negativa.

Algorithm 117 Albero Binario Allopato

```
1  int dfsSearch(GRAPH G, NODE x, NODE t, NODE r)
2          if x == t then
3          return 0
4          int min = +∞
5          foreach y ∈ G.adj(x) do
6          if y ≠ r then
7                  int d = dfsSearch(G, y, t, r)
8          if d + w(x, y) < min then
9                  min = d + w(x, y)
10         return min
11
12 int computeDist(GRAPH G, NODE r, NODE u, NODE v)
13         d = dfsSearch(G, u, v, r)
14         if d < +∞ then
15         return d
16         else
17                 return dfsSearch(G, u, r, −1) + dfsSearch(G, r, v, r)
```

Il compito di dfsSearch è quello di ricavare la distanza minima tra una foglia x e la foglia t con un controllo postordine. Questa procedura tornerà un numero $n < +\infty$ se l'elemento t è nel sottalbero cercato, mentre $+\infty$ se non è presente Si noti che quando viene propagata la visita in dfs non viene mai fatto per la radice.
Proviamo quindi come primo approccio quello di chiamare *dfsSearch* sui nodi u e v di cui si vuole la distanza minima, se il valore ritornto dalla funzione è $+\infty$ significa che il nodo v non è un diretto discendente di u, dunque devo necessariamente passare per la radice. Detto questo, il costo effettivo della ricerca è dunque dato dalla distanza minima da u alla radice, sommato alla distanza minima dalla radice dalla radice al nodo v, ovviamente anche qui non bisogna passare ancora per la radice. Osserviamo che *dfsSearch* funziona in questo caso, dato che sia u che v sono figli della radice dell'albero.

22.7 ASD7

Colorazione Alberi
Il problema prevede che ogni nodo abbia un colore differente rispetto a quelli adiacenti e quindi non bisogna lasciarsi ingannare dal fatto che vi siano a disposizione un numero di colori pari al numero di nodi presenti all'interno dell'albero. Si può anche facilmente intuire che un albero è un grafo bipartito in quanto possiamo suddividere i nodi in base a livelli pari e dispari; per questo motivo è bi-colorabile e dunque possiamo usare solamente 2 colori per poter colorare completamente l'albero e, per ottenere il costo minore, dobbiamo usare per forza i colori 1 e 2. L'ultima cosa di cui non possiamo essere certi è con che colore cominciare: visto che si tratta di un albero vorremmo che i livelli contenenti più nodi siano colorati con il colore 1 mentre quelli contenenti il numero minore

con il colore 2.
Di seguito viene proposta un'implementazione più intelligente che effettua una sola visita dell'albero salvando i nodi in 2 insiemi differenti e assegnando a quello con il numero maggiore il colore 1 mentre all'altro il colore 2; ovviamente viene eseguita una BFS utilizzando una coda.

Algorithm 118 Colorazione Alberi

```
1  int minColoring(TREE T)
2          if T = nil then
3          return 0
4          else
5          QUEUE q = new QUEUE
6          q.enqueue(T)
7          SET s1 = new SET
8          SET s2 = new SET
9          int nFigli = q.size()
10         bool first = true
11         while not q.isEmpty() do
12                 TREE extracted = q.dequeue()
13                 if first then
14                         s1.insert(extracted)
15                 else
16                         s2.insert(extracted)
17                 nFigli = nFigli - 1
18                 if extracted.left ≠ nil then
19                         q.enqueue(extracted.left)
20                 if extracted.right ≠ nil then
21                         q.enqueue(extracted.right)
22                 if nFigli == 0 then
23                         first = not first
24                         nFigli = q.size()
25         if s1.size() > s2.size() then
26                 return s1.size() + 2*s2.size()
27         else
28                 return s2.size() + 2*s1.size()
```

Invece ecco l'algorirtmo proposto da montresor, sostanzialmente si prova a colorare il grafo in 2 modi (cioè in un caso colorando la radice con il colore 1 mentre nell'altro colorando la radice con il colore 2) e si tiene il risultato che costa di meno.

Algorithm 119 Colorazione Alberi (by Montresor)

```
1  int countRec(TREE T, int color)
2          if T == nil then
3          return 0
4          else
5          return color + countRec(T.left, 3 - color) + countRec(T.right, 3 - color
               )
6
7  int minColoring(TREE T)
8          return min(countRec(T, 1), countRec(T, 2))
```

22.8 ASD8

Il gioco del Mikado/Shangai

Algorithm 120 Posso vincere Mikado/Shangai in un unico turno?

```
1  GRAPH buildGraph(int[] X, int[] Y, int n, int m)
2          GRAPH G = new GRAPH
3          for i = 1 to n do
4          G.insertNode(i)
5          for i = 1 to m do
6          G.insertEdge(X[i], Y[i])
7          return G
8
9  boolean isShangaiSolvable(int[] X, int[] n, int m)
10         GRAPH G = buildGraph(X, Y, n, m)
11         int time = 1
12         int[] dt = new int[1...G.size()]
13         int[] ft = new int[1...G.size()]
14         foreach u ∈ G.V() do
15         dt[u] = 0
16         ft[u] = 0
17         foreach u ∈ G.V() do
18         if dt[u] == 0 and hasCycle(G, u, time, dt, ft) then
19                 return false
20         return true
```

Il problema può essere visto come un problema di assenza di cicli in un grafo. Dove i numeri da 1 a n che identificano i bastoncini sono difatto dei nodi, mentre invece le relazioni identificate dal verbo "sovrasta" degli archi. É molto semplice dunque realizzare un grafo con i dati che ci vengono forniti (si noti che $X[i]$ sovrasta $Y[i]$) e verficare se esso è un DAG, ovvero se non presenta cicli. Per farlo possiamo semplicemente avvalerci della funzione hasCycle per i grafi orientati, che si trova nella sezione: 8.5 Grafo orientato aciclico(DAG): DFS.

Votazioni

Algorithm 121 Le votazioni

```
1  int[] election(GRAPH G, int[] status)
2          QUEUE qpp = new QUEUE
3          QUEUE qpe = new QUEUE
4          int[] distanzePP = new int[1..G.size()]
5          int[] distanzePE = new int[1..G.size()]
6          foreach u ∈ G.V() do
7          if status[u] == PP then
8                  distanzePP[u] = 0
9                  distanzePE[u] = +∞
10                 qpp.enqueue(u)
11         else if status[u] == PE then
12                 distanzePP[u] = +∞
13                 distanzePE[u] = 0
14                 qpe.enqueue(u)
15         else
16                 distanzePP[u] = +∞
17                 distanzePE[u] = +∞
18
19         while not qpp.isEmpty() then
20         NODE u = qpp.pop()
21         foreach v ∈ G.adj(u) do
22                 if distanzePP[v] == +∞ then
```

```
23                          distanzePP[v] = distanzePP[u] + 1
24                          qpp.enqueue(v)
25
26          while not qpe.isEmpty() then
27          NODE u = qpe.pop()
28          foreach v ∈ G.adj(u) do
29                  if distanzePE[v] == +∞ then
30                          distanzePE[v] = distanzePE[u] + 1
31                          qpe.enqueue(v)
32
33          int indecisi = 0
34          foreach u ∈ G.V() do
35          if distanzePP[u] == distanzePE[u] then
36                  indecisi = indecisi + 1
37
38          return indecisi
```

Per risolvere questo problema sono necessarie due visite in bfs assegnando dunque le distanze minori ai nodi dai nodi facenti parte del partito PE e una visita assegnando la distanza minima dai nodi PP. La visita viene effettuata incodando i corrispettivi membri del partito nella loro appostita coda con distanza pari a 0, la distanza viene aggiornata non appena vengono scoperti (lo stesso principio di ragionamento è presente nell'algoritmo di *Erdos*). Una volta che abbiamo le distanze minime dai membri del partito PE e PP controlliamo se qualche nodo possiede la stessa dai membri dei due schieramenti, se questo avviene incrementiamo il numero degli indecisi.
Ritorniamo infine gli indecisi totali.

22.9 ASD9

Foresta

Algorithm 122 Foresta con costruzione del grafo

```
1  (GRAPH, int[]) costruisciGrafo(int[][] A, int n)
2          GRAPH G = new GRAPH
3          int[] types = new int[1..n]
4          % inserimento nodi
5          for i = 1 to n · n do
6          G.insertNode(i)
7          % inserimento archi
8          for riga = 1 to n do
9          for colonna = 1 to n do
10         % id sequenziale del nodo
11         int idNodo = colonna + (riga − 1) · n
12         types[idNodo] = A[riga][colonna]
13         if A[riga][colonna] == 1 then
14                 if riga ≠ 1 then
15                         if A[colonna][riga − 1] == 1 then
16                         int idAdiacente = colonna + (riga − 2) · n
17                         G.insertEdge(idNodo, idAdiacente)
18         if colonna ≠ 1 then
19                 if A[colonna − 1][riga] == 1 then
20                 int idAdiacente = (colonna − 1) + (riga − 1) · n
21                 G.insertEdge(idNodo, idAdiacente)
22         if colonna ≠ n then
23                 if A[colonna + 1][riga] == 1 then
24                 int idAdiacente = (colonna + 1) + (riga − 1) · n
25                 G.insertEdge(idNodo, idAdiacente)
26         if riga ≠ n then
27                 if A[colonna][riga + 1] == 1 then
```

```
28                  int idAdiacente = colonna + riga · n
29                  G.insertEdge(idNodo, idAdiacente)
30          return (G, type)
31
32  int searchForest(int[][] A, int n)
33          int[] types = new int[1..n·n]
34          GRAPH G = new GRAPH
35          (G, types) costruisciGrafo(A, n)
36          % trovo le componenti connesse che siano foreste
37          int[] ids = new int[1..G.size()]
38          foreach u ∈ G.V() do
39                  ids[u] = 0
40          int counter = 0
41          int maxSoFar = 0
42          int maxTot = 0
43          foreach u ∈ G.V() do
44          counter = counter + 1
45          ccdfs(G, counter, u, ids, types, maxTot)
46          maxTot = max(maxTot, maxSoFar)
47          maxSoFar = 0
48          return maxTot
49
50  ccdfs(GRAPH G, int counter, NODE u, int[] id, int[] types, int& maxSoFar)
51          id[u] = counter
52          if types[u] == 1 then
53                  maxSoFar = maxSoFar + 1
54                  foreach v ∈ G.adj(u) do
55                  if id[v] == 0 then
56                          ccdfs(G, counter, v, id, types, maxSoFar)
```

L'algoritmo costruisce a partire dalla matrice una semplice grafo. Non appena il grafo è stato preparato si chiama, per tutti i nodi, la procedura in dfs chiamata *ccdfs* con una piccola modifca, essa modifca la grandezza della componente connessa per riferimento che viene opportunatamente resettata al termine della procedura; l'algoritmo inoltre non si propaga se il tipo della casella è pari a 0. Infine si tiene sempre morizzata la grandezza della componente connessa massima che verrà ritornata dalla procedura principale.

Soluzione alternativa senza la costruzione del grafo:

Algorithm 123 Foresta senza costruzione del grafo

```
1  int searchForest(int[][]A, int n)
2          maxsofar = 0
3          for i = 1 to n do
4          for j = 1 to n do
5                  if A[i, j] = 1 then
6                          maxsofar = max(maxsofar, dfs(A, i, j, n))
7          return maxsofar
8
9  int dfs(int[][]A, int i, int j, int n)
10         if 1 ≤ i ≤ n and 1 ≤ j ≤ n and A[i, j] = 1 then
11         A[i, j] = 0
12         return 1 + dfs(A, i − 1, j, n) + dfs(A, i, j − 1, n) + dfs(A, i +
13     1, j, n) + dfs(A, i, j + 1, n)
14         else
15         return 0
```

Lo scopo di questo algoritmo è analogo al precedente, in questo caso, come nel precedente si chiama sui nodi foresta la procedura in dfs.

La procedura torna un valore diverso da 0 se la cella tornata è in una posizione valida ed ovviamente è di tipo foresta.
Ogni nodo viene settato visitato modificando direttamente la matrice, in particolare viene settato il campo stesso a 0 e viene fatta una chiamata ricorsiva su tutte e quattro le direzioni consentite.
Ovviamente la procedura finale ritornerà la dimensione della foresta connessa più grande.

Nodi che raggiungo ovunque

Algorithm 124 Nodi che raggiungo ovunque

```
1  int nodiRaggiunti(GRAPH G, NODE r)
2          boolean[] visited = new boolean[1..G.size()]
3          QUEUE q = new QUEUE()
4          foreach u ∈ G.V()−{r} do
5          visited[u] = false
6          visited[r] = true
7          int nodiVisitati = 1
8          q.enqueue(r)
9          while not q.isEmpty() do
10         NODE u = q.dequeue()
11         foreach v ∈ G.adj(u) do
12                 if not visited[v] then
13                         nodiVisitati = nodiVisitati + 1
14                         visited[v] = true
15                         q.enqueue(v)
16         return nodiVisitati
17
18 int nodiOnniscenti(GRAPH G)
19         int countNodi = 0
20         foreach u ∈ G.V() do
21         if nodiRaggiunti(G, u) == G.size() then
22                 countNodi = countNodi + 1
23         return countNodi
```

Il principio di base è quello di lanciare una visita in bfs per ogni nodo che compone il grafo e controllare se tutti i nodi sono stati vistati; se questo accade allora il numero di nodi visti sarà pari al numero di nodi totali, incrementando dunque il contatore dei nodi che raggiungono ovunque.
L'algoritmo termina ritornando il numero di nodi che raggiungono ovunque.

22.10 ASD10

Prima Occorrenza

Visto che il vettore di interi che viene passato è ordinato possiamo utilizzare la ricerca binaria in modo da trovare in complessità logaritmica la prima occorrenza di un numero v. Di seguito viene proposta un'implementazione definita poco elegante ma che ci è venuta subito in mente: l'idea è quella di continuare la ricerca dicotomica nella parte sinistra dell'array quando si trova l'elemento cercato (con l'accortezza di includere all'interno di questo array anche l'elemento appena analizzato - nel caso in cui fosse effettivamente la prima occorrenza) e restituire l'indice soltanto quando si analizza uno e un solo elemento.

Algorithm 125 Prima Occorrenza

```
1  int binaryFirst(int[] A, int start, int end, int v)
2          if start == end then
3                  return start
4          else
```

```
5           int m = (start + end)/2
6           if A[m] == v then
7                   return binaryFirst(A, start, m, v)
8           else if A[m] < v then
9                   return binaryFirst(A, m+1, end, v)
10          else
11                  return binaryFirst(A, start, m-1, v)
12
13  int first(int[] A, int n, int v)
14          return binaryFirst(A, 1, n, v)
```

Nella soluzione seguente Montresor decide semplicemente di unificare i casi in un unico if else con la condizione del $\leq$

Algorithm 126 Prima Occorrenza (by Montresor)

```
1   int firstRec(int[] A, int i, int j, int v)
2           if i == j then
3           return i
4           else
5           int m = (i + j)/2
6           if A[m] ≤ v then
7                   return binaryFirst(A, i, m, v)
8           else
9                   return binaryFirst(A, j, m+1, v)
10
11  int first(int[] A, int n, int v)
12          return binaryFirst(A, 1, n, v)
```

22.11 ASD11

Valore unico

Dato un vettore ordinato di interi in cui gli elementi compaiono esattamente due volte tranne un elemento, trovare e restituire tale elemento che compare solo una volta. La soluzione più efficiente si basa su una ricerca binaria modificata. Usiamo indici tra 1 e n per il vettore. Se consideriamo l'elemento a metà del vettore, il suo indice sarà pari o dispari. Se tale indice è dispari e se il suo l'elemento successivo è uguale, vuol dire che tutti gli elementi prima sono presenti a coppie e quindi cerchiamo solo nella metà successiva al mediano, altrimenti sappiamo che tutti gli elementi dopo sono coppie e dobbiamo cerca negli elementi precedenti. Il ragionamento è lo stesso per il caso di indice dell'elemento a metà pari e il test dell'elemento precedente.

Algorithm 127 Valore unico

```
1   int singleRec(ITEM[] A, int i, int j)
2           if i = j then
3                   return A[i]
4           int m = floor((i + j)/2)
5           if m mod 2 = 1 then
6                   if A[m] = A[m + 1] then
7                           return singleRec(A, m+2, j)
8                   else
9                           return singleRec(A, i, m)
10          else
11                          if A[m - 1] = A[m] then
12                          return singleRec(A, m+1, j)
13                  else
14                          return singleRec(A, i, m - 1)
15
```

```
16 //funzione wrapper
17 int single(ITEM[] A, int n)
18         return singleRec(A, 1, n)
```

Il costo di tale soluzione è $\mathcal{O}(\log n)$.

Zero

Dato un vettore contenente n interi ($n \geq 3$) e tale per cui: il primo elemento è minore di 0, l'ultimo elemento è maggiore di 0 e per ogni indice $\leq$ di 2 vale — V[i-1] - V[i]— ≤ 1, trovare l'indice dell'elemento 0 (se esiste). La soluzione è basata su un algoritmo divide-et-impera. Possiamo notare che l'elemento 0 sarà sempre presente all'interno del vettore per via delle condizioni poste all'inizio. Per risolvere il problema, l'algoritmo divide a metà il problema trovando indice dell'elemento m a metà dell'array. Se l'elemento a tale indice è uguale a 0, abbiamo finito, altrimenti se l'elemento è più piccolo di 0 dobbiamo cercare ricorsivamente nella metà destra dell'array oppure se è maggiore dobbiamo ricercare nella metà sinistra. Il caso base considera un vettore di 3 elementi: -1, 0 e 1.

Algorithm 128 Zero

```
1  int zerorec(ITEM[] A, int i, int j)
2          int m = floor((i + j)/2)
3          if A[m] = 0 then
4                  return m
5          else if(A[m] > 0)
6                  return zerorec(A, i, m)
7          else
8                  return zerorec(A, m, j)
9
10 int zero(ITEM[] A, int n)
11         return zerorec(A, 1, n)
```

23 Esami anni passati

23.1 Esame 07/02/2019

Esercizio A1 Trovare un limite superiore, il più stretto possibile per la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + T(\lfloor n/8 \rfloor) + 2T(\lfloor n/16 \rfloor) + 1 & n > 16 \\ 1 & n \leq 16 \end{cases}$$

Soluzione A1 Non possiamo applicare il master theorem quindi proviamo a risolvere il problema per tentativi. Proviamo con il metodo di sostituzione a verificare e vale T(n) = $\mathcal{O}(n)$. Dimostriamo per induzione.

- Ipotesi induttiva: T(k) $\leq$ ck, per k < n
- Passo induttivo:

$$\begin{aligned} T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + T(\lfloor n/8 \rfloor) + 2T(\lfloor n/16 \rfloor) + 1 \\ \leq c\lfloor n/2 \rfloor + c\lfloor n/4 \rfloor + c\lfloor n/8 \rfloor + 2c\lfloor n/16 \rfloor + 1 \\ \leq \frac{1}{2}cn + \frac{1}{4}cn + \frac{1}{8}cn + \frac{1}{8}cn + 1 \\ = cn + 1 \leq cn \end{aligned}$$

  L'ultima disequazione è falsa per un termine di ordine inferiore.

  Proviamo a dimostrare che

$$\exists b{>}0, \exists c{>}0, \exists m{\geq}0, : T(n){\leq}cn - b, \forall n{\geq}m$$

- Ipotesi induttiva: T(k) $\leq$ ck-b, per k < n
- Passo induttivo:

$$\begin{aligned} T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + T(\lfloor n/8 \rfloor) + 2T(\lfloor n/16 \rfloor) + 1 \\ \leq c(lfloorn/2\rfloor - b + c\lfloor n/4 \rfloor - b + c\lfloor n/8 \rfloor - b + 2(c\lfloor n/16 \rfloor - b) + 1 \\ \leq \frac{1}{2}cn - b + \frac{1}{4}cn - b + \frac{1}{8}cn - b + \frac{2}{16}cn - 2b + 1 \\ = cn - 5b + 1 \leq cn - b \end{aligned}$$

  L'ultima disequazione è vera per ogni c e per b $\geq \frac{1}{4}$.

- Caso base: T(n)=1 $\leq$ cn-b, per tutti i valori di n compresi tra 1 e 16, ovvero:

$$c{\geq}\frac{b+1}{n}, \forall n : 1 \leq n \leq 16$$

  I valori $\frac{b+1}{n}$ sono minori o uguali di 5/4 (per T(1) vale 5/4), per 1 $\leq$ n $\leq$ 16; quindi tutte queste disequazioni sono soddisfatte da c *geq* 5/4.

Abbiamo quindi dimostrato che T(n)= $\mathcal{O}(n)$, con m= 1 e c $\geq$ 5/4.

23.2 Esercizi vari

Algorithm 129 Esercizi da schedare

```
1  int longestIncreasing(GRAPH G)
2          int[] longest = new int[1...G.size()]
3          for u = 1 to G.n do
4          longest[u] = -1
5          for u = 1 to G.n do
6          if longest[u] < 0 then
7                  longIncreasingRec(G, u, longest)
8          return max(longest)
9
10 longestIncreasingRec(GRAPH G, NODE u, int[] longest)
11         longest[u] = 0
12         foreach v ∈ G.adj(u) do
13         if longest[v] < 0 then
14                 longIncreasingRec(G, v, longest)
15         if u.p < v.p and longest[u] < longest[v] + 1 then
16                         longest[u] = longest[v] + 1
17
18 int ceil(int[] A, int inizio, int n, int v)
19         if inizio == fine then
20         if A[inizio] < v then
21                 return A[inizio]
22         else
23                 return -1
24         int meta = (inzio + fine)/2
25         if A[meta] <= v then
26         return ceil(A, meta + 1, fine, v)
27         else
28         return ceil(A, inizio, meta, v)
29
30 boolean four(int[] A, int n, int k)
31         SET S = SET
32         for i = 1 to n do
33         S.insert(A[i])
34         for i = 1 to n do
35         for j = i to n do
36                 for h = j to n do
37                         if S.contains(k - A[i] - A[j] - A[h]) then
38                                 return true
39         return false
40
41
42 int costLenRec(int[][] M, int n, int r, int c, SET terra, SET mare)
43         if M[r][c] > 0 and not terra.contains(<r, c>) then
44         terra.insert(<r, c>)
45         return 1
46         if M[r][c] < 0 and not contains(<r, c>) then
47         mare.insert(<r, c>)
48         int costo = 0
49         if r ≠ 1 then
50                 costo = costo + coastLenRec(M, int n, r - 1, c, terra, mare)
51         if r ≠ n then
52                 costo = costo + coastLenRec(M, int n, r + 1, c, terra, mare)
53         if c ≠ 1 then
54                 costo = costo + coastLenRec(M, int n, r, c - 1, terra, mare)
55         if c ≠ n then
56                 costo = costo + coastLenRec(M, int n, r1, c + 1, terra, mare)
```

```
57          return costo
58          return 0
59
60  int coastLen(int[][] M, int n, int r, int c)
61          SET terra = new SET
62          SET mare = new SET
63          return coastLenRec(M, n, r, c, terra, mare)
64
65  % sottomatrice dimensione massima
66
67  int shifting(int[] A, int inizio, int fine)
68          if inizio == fine then
69          return inizio
70          else
71          int meta = (inizio + fine)/2
72          if A[meta] < A[n] then
73                  return shifting(A, inizio, meta)
74          else
75                  return shifting(A, meta + 1, fine)
76
77  int nShift(int[] A, int n)
78          return shifting(A, 1, n) - 1
```