Note di Algoritmi e Strutture Dati: Algoritmi, Calcolabilita e Complessita'
Anno Accademico 2001/02

Eugenio Moggi

March 4, 2002

1 Algoritmi e automi

Vi sono vari modi di formalizzare la nozione di algoritmo. Tale nozione non e' assoluta, bensi' relativa ad un *modello di calcolo*. Una formalizzazione che cattura alcune (ma non tutte) le proprieta' che dovrebbe avere un algoritmo deterministico e' data dalla nozione di automa deterministico. Per tener conto della complessita' associamo ad ogni transizione dell' automa con costo a valori in un monoide ordinato.

Definition 1.1 (Monoide ordinato) *Un* **monoide ordinato** *e' un'algebra con predicati* $(M, \leq, +, 0)$

- *dove* M *e' un insieme non vuoto,* $\leq$ *e' un predicato binario,* $+$ *e' una operazione binaria, e* 0 *e' una costante;*
- $(M, \leq)$ *e' un ordine parziale, cioe'* $\leq$ *soddisfa la proprieta' riflessiva, transitiva ed anti-simmetrica:*
- $(M, +, 0)$ *e' un monoide, cioe' l'operazione* $+$ *e' associativa* $x + (y + z) = (x + y) + z$*, e* $x + 0 = x = 0 + x$*;*
- *ed inoltre* 0 *e' il minimo* $0 \leq x$*, e* $+$ *e' monotona* $x \leq x' \wedge y \leq y' \supset x + y \leq x' + y'$*.*

Definition 1.2 (Automa) *Fissati due insiemi* I *e* O*, ed un* monoide ordinato $(M, \leq, +, 0)$*, un automa deterministico* A *con input* I *ed output* O *e pesi in* M *(o piu' semplicemente un automa) e' dato da:*

- *un insieme* S_A *di stati interni*
- *una funzione di transizione pesata* $t_A: I + S_A \to (S_A + O) \times M$*.*

Introduciamo anche le seguenti nozioni derivate:

- $\tilde{S}_A \stackrel{\Delta}{=} I + S_A + O$ *e' l'insieme degli stati.*
- $\xrightarrow[A]{m} \subseteq \tilde{S}_A \times \tilde{S}_A$ *e' la relazione di transizione (pesata), cioe'* $s \xrightarrow[A]{m} s' \stackrel{\Delta}{\iff} t_A(s) = (s', m)$*.*
- $(s_i | 0 \leq i \leq n) \in \tilde{S}_A^+$ *e' una computazione (finita) di* A *da* s_0 *a* s_n *con peso* $m \stackrel{\Delta}{\iff}$
  esiste, necessariamente unica, $(m_i | 0 \leq i < n) \in M^n$ *t.c.* $\forall i < n. s_i \xrightarrow[A]{m_i} s_{i+1}$ *e* $m = \sum_{0 \leq i < n} m_i$*.*
- $\stackrel{m}{\underset{A}{\twoheadrightarrow}} \subseteq \tilde{S}_A \times \tilde{S}_A$ *e' la* chiusura transitiva *di* $\xrightarrow{m}$ *cioe'* $s \stackrel{m}{\underset{A}{\twoheadrightarrow}} s' \stackrel{\Delta}{\iff}$ *esiste una computazione non banale (cioe'* $n > 0$*) di* A *da* s *a* s' *con peso* m*.*
- $f_A: I \rightharpoonup O$ *e' la funzione (parziale) calcolata da* A*, cioe'* $f_A(x) = y \stackrel{\Delta}{\iff}$ *esiste* $m \in M$ *t.c.* $x \stackrel{m}{\underset{A}{\twoheadrightarrow}} y$
- $T_A: I \rightharpoonup M$ *e' la funzione complessita' esatta di* A*, cioe'* $T_A(x) = m \stackrel{\Delta}{\iff}$ *esiste* $y \in O$ *t.c.* $x \stackrel{m}{\underset{A}{\twoheadrightarrow}} y$

Remark 1.3 Vi sono vari modi di scegliere il monoide ordinato M, che riflettono quale uso di *risorsa* vogliamo analizzare. Se siamo interessati alla complessita' tempo useremo il monoide $(N, \leq, +, 0)$ dei numeri naturali con l'addizione, o il monoide $(R^+, \leq, +, 0)$ dei reali non negativi, o il monoide $(\overline{N}, \leq, +, 0)$ dei numeri naturali estesi con ∞. La scelta dell'addizione come operazione del monoide corrisponde al fatto che il tempo per effettuare due transizioni, una dopo l'altra', si ottiene sommando i tempi. Se siamo interessati alla complessita' spazio (di memoria) useremo il monoide $(N, \leq, \max, 0)$ dei numeri naturali con l' operazione di massimo (monoidi analoghi si possono definire su R^+ e $\overline{N}$). La scelta del massimo come operazione del monoide corrisponde al fatto che lo spazio necessario per effettuare due transizioni, una dopo l'altra', si ottiene prendendo il massimo tra gli spazi richiesti, poiche' la memoria necessaria per la prima transizione puo' essere riutilizzata dalla seconda. Se siamo interessati sia alla complessita' spazio che tempo, basta prendere il prodotto dei monoidi ordinati $(N, \leq, +, 0)$ e $(N, \leq, \max, 0)$, cioe' $(N \times N, \leq, \ldots)$. Se non siamo interessati alla complessita', basta prendere il monoide banale, che ha solo l'elemento 0.

Se un algoritmo lo rappresentiamo mediante un automa con input I ed output O, e' naturale rappresentare un problema mediante una relazione $P \subseteq I \times O$ che ad ogni input (istanza del problema) associa un insieme di output (soluzioni) accettabili. Useremo indistintamente la notazione $x\ P\ y$ e $P(x, y)$ per indicare che $(x, y) \in P$. Il primo requisito che deve soddisfare un algoritmo e' la correttezza.

Definition 1.4 (Correttezza) *Dato un automa* A *ed un problema* P *diciamo che*

- A **risolve parzialmente** $P \stackrel{\Delta}{\iff} \forall x \in I.\ f_A(x)$ *definito implica* $x\ P\ f_A(x)$*;*

- A **risolve totalmente** $P \overset{\Delta}{\iff} \forall x \in I.\ f_A(x)$ *definito e* $x\ P\ f_A(x)$.

In generale, se A risolve totalmente P, allora lo risolve parzialmente. Tuttavia, quando la funzione f_A e' totale, vale anche il viceversa. La definizione di correttezza non e' molto sensata quando vi sono $x \in I$ per i quali non esistono $y \in O$ t.c. $P(x,y)$, tuttavia ci possiamo sembre ridurre al caso in cui I e' il dominio della relazione P.

Il requisito di correttezza ignora completamento la complessita' T_A di un algoritmo. Tuttavia, disponendo di due algoritmi corretti, cioe' che risolvono un dato problema P, e' naturale preferire l'algoritmo con minore complessita' (per questo e' importante che su M vi sia una relazione d'ordine $\leq$). In certi casi (p.e. applicazioni real-time) e' importante specificare gia' a livello di problema quanto si e' disposti a tollerare in termini di costi, fissando un limite superiore $T: I \to M$. Molto spesso all'insieme I (ed analogamente per O) si associa una funzione $|\_|: I \to N$, che fornisce la **dimensione** $|x|$ di una *istanza* x del problema. In questo caso e' piu' semplice specificare un limite superiore $T: N \to M$ in funzione della dimensione dell'input.

Definition 1.5 *Dato un automa A, e due funzioni $|\_|: I \to N$ e $T: N \to M$ (che assumiamo monotona), diciamo che T e' una* **maggiorazione** *della complessita' di A (nel caso peggiore) $\overset{\Delta}{\iff} \forall x \in I.\ T_A(x) \leq T(|x|)$.*

Si noti che abbiamo richiesto che T sia una funzione totale, e quindi anche T_A deve essere totale se maggiorata da T.

Un algoritmo lo si puo' descrivere a diversi livelli di astrazione. In particolare, quello che e' considerato un passo di computazione ad un *livello di astrazione*, corrispondera' ad uno o piu' passi di computazione ad un maggior livello di dettaglio. La nozione di simulazione tra automi formalizza la corrispondenza tra un algoritmo ed un suo *raffinamento*.

Definition 1.6 (Simulazione tra automi) *Dati due automi A e B con input I ed output O e pesi in M, una simulazione di A tramite B e' una relazione $R \subseteq S_B \times S_A$ t.c.*

- $t\ \tilde{R}\ s \xrightarrow[A]{n} s'$ *implica* $\exists t' \in \tilde{S}_B, m \in M$ *t.c.* $t \overset{m}{\underset{B}{\Longrightarrow}} t'\ \tilde{R}\ s'$ *e* $m \leq n$

dove $\tilde{R}$ e' la relazione relazione $\Delta_I + R + \Delta_O \subseteq \tilde{S}_A \times \tilde{S}_B$.

In genere si suppone che R sia il grafo di una funzione parziale surgettiva da S_B ad S_A, cioe' per ogni t esiste al piu' un s t.c. $t\ R\ s$, e per ogni s esiste almeno un t t.c. $t\ R\ s$. In termini piu' intuitivi, avere una simulazione di A tramite B vuol dire che ogni transizione $s \xrightarrow[A]{n} s'$ di A puo' essere *mimata* da una transizione composita di B di costo non superiore ad n.

Proposition 1.7 *Se esiste una simulazione di A tramite B, allora i due algoritmi calcolano la stessa funzione parziale, cioe' $f_A(x) = f_B(x)$, e la complessita' di B e' maggiorata da quella di A, cioe' $T_B(x) \leq T_A(x)$.*

Quindi se abbiamo dimostrato che un algoritmo A ad alto livello (o che fa' uso di pseudo-istruzioni) e' *corretto*, cioe' risolve il problema P con un costo maggiorato da $T: N \to M$, allora ogni *implementazione corretta* di A, cioe' un algoritmo B per cui esiste una simulazione di A tramite B, risolve il problema P.

1.1 Invarianti e terminazione

Dato un automa A ed un problema P, diamo un metodo (detto dell' invariante) per dimostrare la correttezza parziale.

Proposition 1.8 *A risolve parzialmente P se esiste $Inv \subseteq I \times \tilde{S}_A$ t.c.*

- $\forall x \in I. Inv(x,x)$
- $\forall x \in I,\ s, s' \in \tilde{S}_A,\ m \in M. Inv(x,s) \land s \xrightarrow[A]{m} s' \supset Inv(x,s')$
- $\forall x \in I,\ y \in O. Inv(x,y) \supset P(x,y)$

Proof Per definizione di f_A, si deve dimostrare che se esiste una computazione $(s_i | 0 \leq i \leq n)$ da $x \in I$ ad $y \in O$, allora vale $P(x,y)$. E' facile dimostrare, per induzione su i e dalla definizione di invariante, che $Inv(x, s_i)$. In particolare, si avra' $Inv(x,y)$ e quindi $P(x,y)$. ∎

Se siamo interessati a dimostrare anche la correttezza totale, e' sufficiente dimostrare che la funzione f_A e' totale.

Proposition 1.9 *$f_A: I \rightharpoonup O$ e' totale se esiste $p: S_A \to N$ t.c. $\forall s, s' \in S_A,\ m \in M. s \xrightarrow[A]{m} s' \supset p(s) > p(s')$.*

Example 1.10 Applicare le idee generali per dimostrare la correttezza dell'algoritmo di sorting mediante selezione. Trattare le sequenze come un tipo di dato astratto. Considerare prima un algoritmo A ad alto livello, in cui l'estrazione del minimo elemento da una sequenza e' considera una operazione elementare. Darne quindi una implementazione B a piu' basso livello, e dimostrandone la correttezza esibendo una simulazione di A tramite B.

2 Schema divide-et-impera

Una tecnica generale per risolvere un problema complesso e' di decomporlo in problemi *dello stesso tipo*, ma di dimensioni minori. Formalmente, tale tecnica corrisponde a definire ricorsivamente una funzione (parziale) $f: I \rightharpoonup O$, in termini di due funzioni ausiliarie $d: I \to I\ list$ per decomporre un problema in sottoproblemi e $c: I \times (O\ list) \to O$ per costruire la soluzione al problema a partire dalle soluzioni ai sottoproblemi. In ML lo schema generale del divide-et-impera e' espresso mediante il seguente funzionale ricorsivo

```
DC : (I -> I list) -> (I * O list -> O) -> I -> O
fun DC D C x = C(x,map (DC D C) (D x));
fun F x = DC D C x; (* o piu' direttamente *)
fun F x = C(x,map F (D x));
```

In pseudo-Pascal l' algoritmo ricorsivo per calcolare f, supponendo di aver gia' disponibili degli algoritmi per d e c, e'

```
F(x)
  (x_1,...,x_k) <- D(x)           {dividi}
  for i=1 to k do y_i <- F(x_i)  {risolvi sotto-problemi}
  y <- C(x,(y_1,...,y_k))           {combina}
  return(y)
```

- Dimostrazione di terminazione degli algoritmi, ovvero f e' totale, per induzione su $|x|$. Basta verificare che
  - $(x_{i\in k}) = d(x)$ implica $\forall i \in k.|x_i| < |x|$

  si osservi che quando $|x| = 0$ bisogna avere $k = 0$.
- Dimostrazione di correttezza parziale, cioe' $P(x, f(x))$ se $f(x)$ definita, per induzione su "altezza dell'albero delle chiamate ricorsive per il calcolo di $f(x)$". Basta verificare che
  - $(x_{i\in k}) = d(x)$ e $\forall i \in k.P(x_i, y_i)$ implica $P(x, c(x, (y_{i\in k})))$

  se si sa' anche che f e' totale, allora consegue la correttezza totale.

Molto spesso e' sufficiente uno schema divide-et-impera piu' semplice, in cui esiste un k prefissato t.c. il numero di sottoproblemi in cui si decompone x e' o 0 o k.

```
F(x)
  if trivial(x)   {test}
  then y <- solve(x)      {risolvi immediatamente}
  else (x_1,...,x_k) <- D(x)              {dividi}
       for i=1 to k do y_i <- F(x_i)     {risolvi sotto-problemi}
       y <- C(x,y_1,...,y_k)              {combina}
  return(y)
```

Le tecniche di dimostrazione illustrate sopra si adattano facilmente a questa variante, per esempio:

- Dimostrazione di correttezza parziale, per induzione su "altezza dell'albero delle chiamate ricorsive per il calcolo di $f(x)$". Basta verificare che
  - $trivial(x)$ implica $P(x, solve(x))$
  - $\neg trivial(x)$ e $(x_{i\in k}) = d(x)$ e $\forall i \in k.P(x_i, y_i)$ implica $P(x, c(x, y_{i\in k}))$.

Example 2.1 Applicare le idee generali del divide-et-impera per analizzare Mergesort. Trattare le sequenze come un tipo di dato astratto. Parametrizzare lo studio della correttezza e dell'analisi di complessita' di Mergesort in funzione di una specifica delle proprieta' delle operazioni di composizione e decomposizione.

2.1 Complessita' di algoritmi ricorsivi e Master Method

Per poter fare stime di complessita' di algoritmi ricorsivi basati sulla tecnica del divide-et-impera, occorre anzitutto definire la **complessita' esatta** $T_f: I \rightharpoonup M$ dell'algoritmo F (dove M e' uno dei monoidi ordinati precedentemente introdotti per misurare la complessita'). L'approccio piu' ovvio e' di definire T_f ricorsivamente, in analogia con la definizione di f, supponendo di disporre di funzioni $T_d: I \to M$ e $T_c: I \times (O\ list) \to M$ per la complessita' esatta degli algoritmi che calcola le funzioni d e c. La definizione ricorsiva di T_f in funzione di f, T_d e T_c e' data da (dove il $+$ e' quello del monoide ordinato M)

$$T_f(x) = T_d(x) + (\sum_{i \in k} T_f(x_i)) + T_c(x, (y_{i \in k})) \quad \text{dove } (x_{i \in k}) = d(x) \text{ e } y_i = f(x_i)$$

raggruppando in un unico termine $t_f(x)$ il costo di divisione e combinazione associato ad x, cioe'

$$t_f(x) = T_d(x) + T_c(x, (y_{i \in k})) \quad \text{dove } (x_{i \in k}) = d(x) \text{ e } y_i = f(x_i)$$

possiamo riscrivere la definizione della complessita' esatta T_f nel modo seguente

$$T_f(x) = (\sum_{i \in k} T_f(x_i)) + t_f(x) \quad \text{dove } (x_{i \in k}) = d(x)$$

Nel seguito supporremo che la funzione di decomposizione produce sotto-problemi di dimensione strettamente minore, cioe' $\forall i \in k.|x_i| < |x|$. Tra le altre cose questa ipotesi garantisce che f, t_f e T_f sono totali.
Per dimostrare che $T: N \to M$ e' una maggiorazione della complessita' esatta T_f, cioe' $\forall x \in I.T_f(x) \leq T(|x|)$, basta verificare che

- $(x_{i \in k}) = d(x)$ implica $(\sum_{i \in k} T(|x_i|)) + t_f(x) \leq T(|x|)$

Il precedente criterio fornisce un criterio sufficiente per dimostrare che T e' una maggiorazione di T_f, ma non fornisce alcun aiuto per trovare T. Diamo adesso un metodo per definirsi *implicitamente* un buon cadidato per T. Supponiamo che

- $t: N \to M$ sia una maggiorazione di t_f, cioe' $t_f(x) \leq t(|x|)$
- $a: N \to N$ sia una maggiorazione sul numero dei sottoproblemi, cioe' $k \leq a(|x|)$ dove $x_{i \in k} = d(x)$
- $s: N \to N$ sia una **buona** maggiorazione per la dimensione dei sottoproblemi, cioe' $\forall i \in k.|x_i| \leq s(|x|) < |x|$ dove $x_{i \in k} = d(x)$ e $|x| > 0$
- infine supponiamo che le funzioni t, a e s siano **monotone**

allora la funzione T definita (per induzione su n) da

$$T(0) = t(0) \quad \text{e} \quad T(n) = a(n) * T(s\ n) + t(n) \quad \text{se } n > 0$$

e' **monotona** ed e' una maggiorazione di T_f. In genere siamo interessati a stimare la complessita' degli algoritmi a meno di fattori costanti, cioe' a sapere se $T \in O(g)$. Per alcune T definite mediante relazioni di ricorrenza vi sono delle formule preconfezionate, che vanno sotto il nome di Master Method. Supponiamo che T sia definita dalla seguente relazione di ricorrenza, dove $a \geq 1$ e $b > 1$, mentre t e' una funzione data

$$T(1) = t(1) \quad \text{e} \quad T(n) = a * T(\lceil n/b \rceil) + t(n) \quad \text{se } n > 1$$

e sia $c \stackrel{\Delta}{=} \log_b a$, allora valgono le seguenti implicazioni:

- se $t(n) \in O(n^{c-\epsilon})$ allora $T(n) \in \Theta(n^c)$,
- se $t(n) \in \Theta(n^c)$ allora $T(n) \in \Theta(n^c * \log n)$,
- se $t(n) \in \Omega(n^{c+\epsilon})$ (e vale una condizione di regolarita' su t) allora $T(n) \in \Theta(t)$.

Per esempio quando si analizza un algoritmo divide-et-impera, $t(n)$ corrisponde al costo di divisione e combinazione di un problema di dimensione n, a e' il numero di sottoproblemi, e b misura di quanto si riducone le dimensioni dei sottoproblemi. Il primo caso si verifica quando il costo della ricorsione predomina su quello della divisione e combinazione (quindi anche se esse fossero a costo 0 non si otterrebbero miglioramenti significativi), il terzo caso si verifica quando il costo della divisione e combinazione e' predominante. Non abbiamo esplicitato la condizione di regolarita', poiche' tutte le funzioni che si usano in pratica la soddisfano.

3 Tipi di Dato Astratti e algebre many-sorted

Si suppongono note le seguenti nozioni:

- segnatura many-sorted Σ. Indichiamo con σ un simbolo di sort e con *op* un simbolo di operazione. Scriviamo $op: \sigma_{i\in n} \to \sigma$ per indicare l'arieta' di *op*.
- l'insieme (definito induttivamente) dei Σ-termini $t \in T_\Sigma(X)_\sigma$ di sort σ e con variabili in X, dove X e' una famiglia di insiemi indiciata dalle sort di Σ, quindi X_σ e' l'insieme delle variabili di sort σ.
- Σ-algebra (parziale) A. Indichiamo con σ^A l'insieme che *interpreta* la sort σ, e $op^A: \sigma_1^A \times \ldots \times \sigma_n^A \rightharpoonup \sigma^A$ la funzione (parziale) che interpreta l'operazione *op* (con l'arieta' sopra indicata). In genere, si suppone che gli insiemi σ^A non siano vuoti. Si dice che A e' una Σ-algebra totale, se tutte le operazioni sono interpretate da funzioni totali.
- l'interpretazione $t_v^A \in \sigma^A$ del termine t nell'algebra A in funzione di una interpretazione v delle variabili, cioe' $v(x) \in \sigma^A$ se $x \in X_\sigma$. Nel caso di un'algebra parziale t_v^A puo' essere indefinita.

Un tipo di dato astratto puo' essere specificato o in modo assiomatico, cioe' dando degli assiomi (p.e. equazioni o formule della logica del primo ordine) che descrivono proprieta' che devono essere *soddisfatte* dalle operazioni, oppure dando una interpretazione *canonica*. Nell'approccio assiomatico e' naturale identificare una *implementazione* del tipo di dato astratto con un *modello*, cioe' un'algebra che soddisfa gli assiomi. Nell'altro approccio non vi e' una cosi' chiara distinzione tra specifica ed implementazione. Quindi introdurremo una nozione di simuazione di un'algebra (specifica) tramite un'altra algebra (implementazione), simile a quella vista per gli automi.

Definition 3.1 (Simulazione tra algebre) *Date due Σ-algebre (parziali) A e B, una* **simulazione** *di A tramite B e' una famiglia di relazioni $R_\sigma \subseteq \sigma^B \times \sigma^A$ indiciata dalle sort di Σ t.c.*

- $(\forall i \in n.b_i\ R_{\sigma_i}\ a_i)$ *e* $op^A(a_{i\in n})$ *definita implicano* $op^B(b_{i\in n})$ *definita e* $op^B(b_{i\in n})\ R_\sigma\ op^A(a_{i\in n})$

per ogni operazione $op: \sigma_{i\in n} \to \sigma$.

In genere e' ragionevole assumere che R_σ sia il grafo di una funzione parziale surgettiva. Quindi ogni elemento di A e' *rappresentabile* da almeno un elemento di B, ed un elemento di B puo' rappresentare al piu' un elemento di A.

Proposition 3.2 (Simulazioni e correttezza totale) *Se R e' una simulazione di A tramite B, $t \in T_\Sigma(X)_\sigma$ e' un termine, u e v sono interpretazioni delle variabili in A e B rispettivamente, allora*

- $(\forall x \in X.u(x)\ R\ v(x))$ *e* t_u^A *definita implicano* (t_v^B *definita e*) $t_v^B\ R_\sigma\ t_u^A$.

In certi casi la nozione di simulazione e' troppo forte. Per esempio, non e' possibile simulare i numeri interi usando rappresentazioni binarie a 32 bit, poiche' le operazioni aritmetiche $+$ o $-$ sono totali sugli interi, ma parziali (causano errore di overflow) sulle rappresentazioni binarie di 32 bit. In questi casi si usa una nozione piu' debole di simulazione.

Definition 3.3 (Simulazione parziale) *Date due Σ-algebre (parziali) A e B, una* **parziale simulazione** *di A tramite B e' una famiglia di relazioni $R_\sigma \subseteq \sigma^B \times \sigma^A$ indiciata dalle sort di Σ t.c.*

- $(\forall i \in n.b_i\ R_{\sigma_i}\ a_i)$ *e* $op^A(a_{i\in n})$ *e* $op^B(b_{i\in n})$ *definite implicano* $op^B(b_{i\in n})\ R_\sigma\ op^A(a_{i\in n})$

per ogni operazione $op: \sigma_{i\in n} \to \sigma$.

Quindi la differenza cruciale rispetto alla definizione di simulazione e' data dal fatto che nell'implementazione B una operazione *op* puo' essere indefinita anche quando nella specifica A *op* e' definita.

Proposition 3.4 (Simulazioni parziali e correttezza parziale) *Se R e' una simulazione parziale di A tramite B, $t \in T_\Sigma(X)_\sigma$ e' un termine, u e v sono interpretazioni delle variabili in A e B rispettivamente, allora*

- $(\forall x \in X.u(x)\ R\ v(x))$ *e* t_u^A *e* t_v^B *definite implicano* $t_v^B\ R_\sigma\ t_u^A$.

Example 3.5 Si consideri la seguente segnatura Σ per l'ADT **imsiemi finiti**:

- sorts – $elem, set, bool$
- ops – $empty: set$, $member: elem, set \to bool$, $ins, del: elem, set \to set$, $union, intersect: set, set \to set$.

- modello canonico: $elem^A = U$, $set^A = \mathcal{P}_{fin}(U)$ cioe' l'insieme dei sottoinsiemi finiti di U

Descriviamo delle Σ-algebre che corrispondono a diverse implementazioni, e diamo delle relazioni di simulazione che precisamo in che modo si *implementa* il modello canonico (solo la relazione R_{set} e' interessante, poiche' le altri sono la relazione identica):

- implementazione a bit vector (si richiede che U sia finito): $elem^B = U$, $set^B = U \to bool$ cioe' l'insieme degli array di booleani di dimensione U

  $a\ R_{set}\ X \stackrel{\Delta}{\iff} \forall x \in U.x \in X \leftrightarrow a(x) = true$

- implementazione a liste: $elem^B = U$, $set^B = U^*$ cioe' l'insieme delle sequenze finite di elementi di U

  $(x_i|i \in m)\ R_{set}\ X \stackrel{\Delta}{\iff} X$ e' l'insieme $\{x_i|i \in m\}$

- implementazione a liste senza ripetizioni: stessi carrier della precedente algebra, ma operazioni diverse

  $(x_i|i \in m)\ R_{set}\ X \stackrel{\Delta}{\iff} X$ e' l'insieme $\{x_i|i \in m\}$, e la lista non contiene ripetizioni, cioe' $\forall i, j \in m.x_i = x_j \supset i = j$

- implementazione a liste ordinate senza ripetizioni (si richiede un ordine lineare su U): stessi carrier della precedente algebra, ma operazioni diverse

  $(x_i|i \in m)\ R_{set}\ X \stackrel{\Delta}{\iff} X$ e' l'insieme $\{x_i|i \in m\}$, e la lista e' in ordine crescente, cioe' $\forall i, j \in m.i < j \supset x_i < x_j$

- implementazione (parziale) liste di lunghezza limitata e senza ripetizioni implementate con un array: $elem^B = U$, $set^B = [0..k] \times U^k$ cioe' l'insieme delle coppie lunghezza della lista e array di dimensione k

  $\langle m, (x_i|i \in k)\rangle\ R_{set}\ X \stackrel{\Delta}{\iff} 0 \leq m \leq k$, X e' l'insieme $\{x_i|i \in m\}$, e $\forall i, j \in m.x_i = x_j \supset i = j$ (si osservi che la solo i primi m elementi dell'array sono significativi).

Al fine di analizzare la complessita' di un algoritmo che usa un tipo di dato astratto, e' necessario fornire una maggiorazione della complessita' delle operazioni del tipo di dato. Tali maggiorazioni dovranno essere date in termini di parametri definibili a livello di modello canonico. E' opportuno evitare l'uso di parametri che sono specifici dell'implementazione, poiche' questo viola il principio di astrazione (cioe' l'utilizzatore del tipo di dato astratto e' costretto a conoscere i dettagli implementativi per capire cosa succede), e quindi impedisce di utilizzare un *approccio condizionale* all'analisi di complessita'. Per esempio, consideriamo l'operazione $del(x, X)$ di cancellazione dell'elemento x dall'insieme X, ed esemplifichiamo dei parametri appropriati (cioe' che si riferiscono al modello canonico) ed impropri (p.e. che fanno rifierimento all'implementazione a liste):

- parametri propri: la cardinalita' di X, la cardinalita' della sort $elem$ (quando e' finita);

- parametri impropri: la lunghezza della lista X (nell'implementazione a liste con la possibilita' di ripetizione un insieme puo' essere rappresentato con liste di diversa lunghezza), la posizione di x nella lista X.

4 Esempi di Implementazione di Tipi di Dato

4.1 Estensioni del tipo di dato set

Si consideri il tipo di dato set (dove gli elementi hanno un ordine lineare) con la seguente segnatura e relativo modello canonico

- sorts – $elem, set, bool$
- ops – $leq\colon elem, elem \to bool$, $empty\colon set$, $member\colon elem, set \to bool$, $ins, del\colon elem, set \to set$, $union, intersect\colon set, set \to set$.
- modello canonico A: $elem^A = U$, leq^A ordine lineare $\leq$ su U, $set^A = \mathcal{P}(U)$ cioe' l'insieme dei sottoinsiemi di U (non necessariamente finiti)

Operazione differenza. Si introduca una nuova operazione $diff\colon set, set \to set$ con $diff^A(X, Y) = \{x \in X | x \notin Y\}$. Si dica come implementare tale operazione (ed analizzarne la complessita') nelle varie rappresentazioni (parziali) degli insiemi finiti gia' viste a lezione (cioe' bit-vector, liste/array senza ripetizione/ordinati).

Operazione complemento. Si introduca una nuova operazione $compl\colon set \to set$ con $compl^A(X) = \{x \in U | x \notin X\}$. Quando U e' infinito tale operazione fa' uscire da dall'insieme dei sottoinsiemi finiti di U, piu' precisamente occorre rappresentare sia i sottoinsiemi finiti che quelli co-finiti di U ($X \subseteq U$ e' detto co-finito se il suo complementare e' finito). Proporre delle rappresentazioni di tale collezione di sottoinsiemi di U, dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: definire una implementazione B dei sottoinsiemi finiti e co-finiti parametricamente in una implementazione C dei soli sottoinsiemi finiti. L'idea chiave e' di definire $set^B = bool \times set^C$.

Operazione intervallo. Si introduca una nuova operazione $interval\colon elem, elem \to set$ con $interval^A(x, y) = [x, y] \stackrel{\Delta}{=} \{z \in U | x \leq z \leq y\}$. Quando $(U, \leq)$ e' ordine lineare denso (cioe' se $x < y$, allora esiste uno z t.c. $x < z < y$), un intervallo limitato puo' contenere un numero infinito di punti. Proporre delle rappresentazioni della collezione di sottoinsiemi di U che si ottiene applicando le operazioni della segnatura (ma non $compl$), dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: i sottoinsiemi di U che ci interessano sono descrivibili come unioni finite di intervalli limitati (ma non necessariamente chiusi). Si consiglia di trovare una rappresentazione univoca di tali sottoinsiemi, quindi si dovrebbero evitare intervalli che si intersecano o la cui unione forma un intervallo.

Operazioni complemento ed intervallo. Si consideri l'estensione del tipo di dato set con entrambe le operazioni $compl$ e $interval$. Proporre delle rappresentazioni della collezione di sottoinsiemi di U che si ottiene applicando le operazioni della segnatura, dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: i sottoinsiemi di U che ci interessano sono descrivibili come unioni finite di intervalli non necessariamente limitati, come p.e. $(, x] \stackrel{\Delta}{=} \{z \in U | z \leq x\}$ e $[x,) \stackrel{\Delta}{=} \{z \in U | x \leq z\}$. Si consiglia di trovare una rappresentazione univoca di tali sottoinsiemi, quindi si dovrebbero evitare intervalli che si intersecano o la cui unione forma un intervallo. Un approccio alternativo e' basato sulla rappresentazione della funzione caratteristica di un tale sottoinsieme, che avra' necessariamente un numero finito di punti di discontinuita'.

4.2 Dizionari: alberi di ricerca e B-tree

Una implementazione del tipo di dato dizionario (e sue estensioni) si ottiene utilizzando gli alberi binari di ricerca. Formalmente tale implementazione la possiamo descrivere nel modo seguente:

- la sort *elem* e' interpretata dall'insieme U, che si suppone avere un ordine lineare $\leq$
- la sort *set* e' interpretata dall'insieme B degli alberi binari con nodi etichettati da U, in un linguaggio ML-like B lo possiamo descrivere come `datatype B = empty | node of B*U*B`

Prima di descrivere l'implementazione delle operazioni, e' importante definire la relazione di simulazione R, che stabilisce quando un albero binario rappresenta un sottoinsieme finito, $t\ R\ X \stackrel{\Delta}{\iff} X = el(t) \wedge bst(t)$, dove le funzioni el e bst sono definite per induzione su $t \in B$:

- $el\colon B \to \mathcal{P}_{fin}(U)$ restituisce l'insieme degli elementi in t

  $el(empty) = \emptyset \quad , \quad el(node(t_1, u, t_2)) = el(t_1) \cup \{u\} \cup el(t_2)$
- $bst\colon B \to bool$ e' un predicato che dice se t e' un "binary search tree"

  $bst(empty) = T \quad , \quad bst(node(t_1, u, t_2)) = bst(t_1) \wedge bst(t_2) \wedge (\forall x \in el(t_1).x < u) \wedge (\forall x \in el(t_2).u < x)$
- $h\colon B \to nat$ e' la funzione che restituisce l'altezza di t

  $h(empty) = 0 \quad , \quad h(node(t_1, u, t_2)) = 1 + \max(h(t_1), h(t_2))$

E' facile vedere che $bst(t)$ implica che un elemento di U occorre al piu' una volta in t. Le operazioni di inserzione *ins* e cancellazione *del* (e quelle ausiliarie *min* e *delmin*) si definiscono per induzione $t \in B$ in un linguaggio ML-like

```
fun ins(x,empty) = node(empty,x,empty)
  | ins(x,node(t1,u,t2)) =
    if x<u then node(ins(x,t1),u,t2)
    elsif u<x then node(t1,u,ins(x,t2))
    else node(t1,u,t2);

fun min empty = raise empty | min node(empty,u,t2) = u | min node(t1,u,t2) = min t1;
fun delmin empty = raise empty
  | delmin node(empty,u,t2) = t2
  | delmin node(t1,u,t2) = node(delmin t1,u,t2);

fun del(x,empty) = empty
  | del(x,node(t1,u,t2)) =
    if x<u then node(del(x,t1),u,t2)
    elsif u<x then node(t1,u,del(x,t2))
    elsif empty(t2) then t1
    else node(t1,min t2,delmin t2)
```

La complessita' di tali operazioni e' $O(h(t))$, se $bst(t)$ allora $\log(n+1) \leq h(t) \leq n$, dove n e' la cardinalita' dell'insieme $el(t)$ rappresentato da t. Quindi se riusciamo a mantenere t (quasi) bilanciato, possiamo ottenere una complessita' $O(\log n)$.

B-tree. Descriveremo una variante degli alberi binari di ricerca in cui le operazioni per i dizionari (*member*, *ins* e *del*) hanno complessita' $O(\log n)$. L'idea chiave e' di usare alberi di ricerca perfettamente bilanciati i cui nodi non foglia abbiamo 2 o 3 figli (ma gli elementi nei nodi non foglia servono solo come informazione per guidare la ricerca).

Remark 4.1 L'idea si puo' generalizzare permettendo ad ogni nodo non foglia di avere tra $m \geq 3$ ed $\lceil m/2 \rceil$ figli (ma la radice puo' avere meno figli). La possibilita' di scegliere il parametro $m \geq 3$ e' particolarmente utile quando si deve memorizzare tali alberi in memoria secondaria, poiche' si puo' scegliere m in base alla dimensione di un blocco.

In questa implementazione la sort *set* e' `datatype set = empty | root of T` e fa' uso di un tipo ausiliario (che non e' in grado di rappresentare l'insieme vuoto) `datatype T = leaf of U | node2 of T*U*T | node3 of T*U*T*U*T`. Per definire la relazione di simulazione che stabilisce quando un B-tree rappresenta un insieme abbiamo bisogno di alcune funzioni ausiliarie

- $el: T \to \mathcal{P}_{fin}(U)$ restituisce l'insieme degli elementi in t

  $el(leaf(u)) = \{u\}$
  $el(node2(t_0, x_1, t_1)) = el(t_1) \cup el(t_2)$
  $el(node3(t_0, x_1, t_1, x_1, t_2)) = el(t_1) \cup el(t_2) \cup el(t_3)$

  el si estende nel modo ovvio al set:

  $el(empty) = \emptyset \quad , \quad el(root(t)) = el(t)$

- $h: T \to nat$ e' la funzione che restituisce l'altezza di t

  $h(leaf(u)) = 0$
  $h(node2(t_0, x_1, t_1)) = 1 + \max(h(t_0), h(t_1))$
  $h(node3(t_0, x_1, t_1, x_2, t_2)) = 1 + \max(h(t_0), h(t_1), h(t_2))$

- $bt: T \to bool$ e' un predicato che dice se t e' un albero bilanciato

  $bt(leaf(u)) = T$
  $bt(node2(t_0, x_1, t_1)) = bt(t_0) \wedge bt(t_1) \wedge h(t_0) = h(t_1)$
  $bt(node3(t_0, x_1, t_1, x_2, t_2)) = bt(t_0) \wedge bt(t_1) \wedge bt(t_2) \wedge h(t_0) = h(t_1) = h(t_2)$

- $st: T \to bool$ e' un predicato che dice se t e' un albero di ricerca

  $st(leaf(u)) = T$
  $st(node2(t_0, x_1, t_1)) = st(t_0) \wedge st(t_1) \wedge (\forall u \in el(t_0).u < x_1) \wedge (\forall u \in el(t_1).x_1 \leq u)$
  $st(node3(t_0, x_1, t_1, x_2, t_2)) = st(t_0) \wedge st(t_1) \wedge st(t_2) \wedge (\forall u \in el(t_0).u < x_1) \wedge (\forall u \in el(t_1).x_1 \leq u < x_2) \wedge (\forall u \in el(t_2).x_2 \leq u)$

- la relazione di simulazione $R \subseteq T \times \mathcal{P}_{fin}(U)$ e' data da

  $t\ R\ X \stackrel{\Delta}{\iff} X = el(t) \wedge bt(t) \wedge st(t)$

Remark 4.2 Vi sono diverse definizioni di B-tree che variano per dettagli marginali (p.e. vedi lucidi di Catania su B-tree), ma in tutte le definizioni si richiede che i B-tree siano bilanciati. Tra le possibili definizioni abbiamo scelto quella che sembra semplificare al massimo la descrizione degli algoritmi.

Il test di appartenenza all'insieme rappresentato da un B-tree e' simile a quello per gli alberi di ricerca binari, ed ha complessita' $O(\log n)$ poiche' un B-tree e' per definizione bilanciato. La principale differenza riguarda il fatto in un B-tree solo le foglie corrispondono ad elementi nell'insieme rappresentato.

```
fun member(u,leaf(x)) = u=x
  | member(u,node2(t0,x1,t1)) =
    if u<x1 then member(x,t0) else member(x,t1)
  | member(u,node3(t0,x1,t1,x2,t2)) =
    if u<x1 then member(x,t0) elsif u<x2 member(x,t1) else member(x,t2)
```

Per le operazioni di inserimento e cancellazione si ha una prima fase, simile a quello che si fa' con gli alberi di ricerca binari, in cui si discende il B-tree per identificare il punto in cui effetturare l'inserimento o la cancellazione. Questo pero' non basta, poiche' per rimanere in relazione R potrebbe essere necessario un ribilanciamento. Si ha quindi una seconda fase in cui si risale verso la radice effettuando delle operazioni locali di ribilanciamento che coinvolgono solo i nodi lungo il cammino. Per implementare la seconda fase e' necessario usare delle funzioni su T del seguente tipo

- $ins: U \times T \to ok(T) \mid over(T \times U \times T)$. Se $ins(u, t) = ok(t')$ vuol dire che non e' necessario ribilanciare il padre di t, poiche' $h(t') = h(t)$. Se $ins(u, t) = over(t_0, u_1, t_1)$ vuol dire non e' stato possibile inserire u senza incrementare l'altezza, ovvero il B-tree $node2(t_0, u_1, t_1)$ rappresenta il risultato di $ins(u, t)$ ed ha altezza $h(t) + 1$.

- $del: U \times T \to empty \mid ok(T) \mid under(T)$. Se $del(u, t) = empty$ vuol dire che t era una foglia, ed e' stata cancellata. Se $del(u, t) = ok(t')$ vuol dire che non e' necessario ribilanciare il padre di t, poiche' $h(t') = h(t)$. Se $del(u, t) = under(t')$ vuol dire non e' stato possibile cancellare u senza diminuire l'altezza, ovvero il B-tree t' rappresenta il risultato di $del(u, t)$ ed ha altezza $h(t) - 1$.

Le operazioni ins e del si estendono alla implementazione della sort set nel seguente modo:

```
fun ins(u,empty) = root(leaf(u))
  | ins(u,root(t)) = case ins(u,t) of ok(t') => root(t') | over(t0,x,t1) => root(node2((t0,x,t1))
fun del(u,empty) = empty
  | del(u,root(t)) = case del(u,t) of empty => empty | ok(t') => root(t') | under(t') => root(t')
```

Ci limitiamo a dare i casi piu' significativi delle definizioni di *ins* e *del*.

```
fun ins(u,leaf(x)) = if u<x then over(leaf(u),x,leaf(x))
                     elsif u=x then ok(leaf(x)) else over(leaf(x),u,leaf(u))
  | ins(u,node2(t0,x1,t1)) =
    if u<x1 then case ins(x,t0) of
                 ok(t') => ok(node2(t',x1,t1)) |
                 over(p0,y1,p1) => ok(node3(p0,y1,p1,x1,t1))
    else ...
  | ins(u,node3(t0,x1,t1,x2,t2)) =
    if u<x1 then case ins(x,t0) of
                 ok(t') => ok(node3(t',x1,t1,x2,t2)) |
                 over(p0,y1,p1) => over(node2(p0,y1,p1),x1,node2(t1,x2,t2))
    elsif ...

fun del(u,leaf(x)) = if u=x then empty else ok(leaf(x))
  | del(u,node2(t0,x1,t1)) =
    if u<x1 then case del(x,t0) of
       empty => under(t1) |
       ok(t') => ok(node2(t',x1,t1)) |
       under(t') => case t1 of
          node2(p0,y1,p1) => under(node3(t',x1,p0,y1,p1) |
          node3(p0,y1,p1,y2,p2) => ok(node2(node2(t',x1,p0),y1,node2(p1,y2,p2)))
    else ...
  | del(u,node3(t0,x1,t1,x2,t2)) =
    if u<x1 then case del(x,t0) of
       empty => ok(node2(t1,x2,t2)) |
       ok(t') => ok(node3(t',x1,t1,x2,t2)) |
       under(t') => case t1 of
          node2(p0,y1,p1) => ok(node2(node3(t',x1,p0,y1,p1),x2,t2)) |
          node3(p0,y1,p1,y2,p2) => ok(node3(node2(t',x1,p0),y1,node2(p1,y2,p2),x2,t2))
    elsif ...
```

E' possibile implementare sui B-tree anche le operazioni insiemistiche $union, intersect: set, set \to set$. Tali operazioni sono gia' implementate in modo ottimale (cioe' lineare) quando si adotta la rappresentazione a liste ordinate degli insiemi. Quindi, usando delle funzioni di conversione da B-tree a liste ordinate e viceversa, che siano calcolabili in tempo lineare (nella cardinalita' dell'insieme rappresentato) si puo' definire (per composizione) una implementazione ottimale delle suddette operazioni. La funzione di conversione $b2l: T \to U^*$ corrisponde ad una visita del B-tree

```
fun b2l(leaf(u)) = [u]
  | b2l(node2(t0,x1,t1)) = b2l(t0)@b2l(t1)
  | b2l(node3(t0,x1,t1,x2,t2)) = b2l(t0)@b2l(t1)@b2l(t2)
```

la complessita' di *b2l* e' lineare se la contatenazione di liste puo' essere effettuata in tempo costante (p.e. matenendo un puntatore all'ultimo elemento). La funzione di conversione $l2b: U^* \to T$ e' piu' complessa. Data una lista non vuota $l = (x_0, \ldots, x_n)$ si costrusce prima la lista di tutte le foglie separate da elementi di controllo, cioe' $(leaf(x_0), x_1, leaf(x_1), \ldots, leaf(x_n))$. Quindi si itera una procedura che prende una lista di B-tree di altezza h separati d aelementi di controllo $(t_0, x_1, t_1, x_2, t_2, x_3, \ldots)$ e restituisce una lista analoga,ma con B-tree di altezza $h+1$, cioe' $(node3(t_0, x_1, t_1, x_2, t_2), x_3, \ldots)$ [**nota:** occorre trattare a parte i casi in cui la lista ha meno di 5 B-tree]. Alla fine si ha una lista contenete un solo B-tree t, che sara' il risultato di $l2b(l)$.

Concludiamo questa sezione descrivendo il tipo di dato per B-tree con $m \geq 3$ (si lascia come esercizio la definizione della relazione di simulazione e l'implementazione delle operazioni):

```
datatype set = root of record j:[0..l]; x:array[1..l-2] of U; t:array[0..l-1] of T end;
datatype T = leaf of U | node of record j:[1..u]; x:array[1..u] of U; t:array[0..u] of T end;
```

dove $u = m - 1$ e $l = \lceil m/2 \rceil - 1$. Il campo j individua la parte significativa degli array x e t, in particolare la radice puo' avere tra 0 ed $\lceil m/2 \rceil - 1$ figli, mentre un nodo interno puo' avere tra $\lceil m/2 \rceil$ ed m figli.

4.3 Code di priorita': Heap

Le code di priorita' sono una variante del tipo di dato dizionario in cui sugli elementi si ha un ordine lineare e si vuole avere una operazione $min: set \to elem$ e $delmin: set \to set$ [**nota:** *delmin* si puo' derivare usando *min* e *del*].

Remark 4.3 Un caso tipico in cui si usano le code di priorita' e' nella scelta dei processi da mandare in esecuzione, in genere il criterio che si adotta e' di selezionare il processo a piu' alta priorita' tra quelli che possono essere mandati in esecuzione. In effetti, in questo caso si possono avere piu' processi con la stessa priorita', quindi e' piu' corretto modellare la situazione con una funzione (parziale) che assegna ad ogni process-id una priorita' (p.e. un numero naturale).

Se consideriamo le implementazioni dei dizionari sinora esaminate si vede che: l'implementazione a liste ordinate permette di implementare *min* e *delmin* in tempo costante [ma *ins* richiede tempo lineare], mentre l'implementazione a B-tree permette di implementare *min* e *delmin* in tempo $O(\log n)$.

5 Algoritmi su Grafi

Un grafo orientato/diretto G e' una coppia (V, E) dove $E \subseteq V \times V$. Gli elementi di V si chiamano nodi/vertici mentre gli elementi di E si chiamano archi. Si riporta la terminologia e notazione che useremo in relazione ai grafi (vedi anche Sezione 5.4 del Cormen):

- $e: u \to v$ significa che e e' un arco da u a v, cioe' $e = (u, v) \in E$. Si dice anche che e esce da u ed entra in v. Un arco $e: u \to u$ e' detto **cappio**.
- v **adiacente** ad u significa che $(u, v) \in E$.
- ad un nodo u sono associati i seguenti numeri naturali:

  grado entrante (in-degree) = numero di archi entranti in u
  grado uscente (out-degree) = numero di archi uscenti da u
  grado = grado entrante + grado uscente.
- un **cammino** p da u a v (notazione $p: u \leadsto v$) e' una sequenza di nodi $(u_i | 0 \leq i \leq n)$ t.c. $u = u_0$, $v = u_n$ e per ogni $i < n$ la coppia (u_i, u_{i+1}) e' un arco. Il numero naturale n e' il numero di archi del cammino p (n puo' essere $= 0$). p e' un **cammino semplice** sse i nodi di p (e quindi anche gli archi) sono distinti. p e' un **ciclo** sse $p: u \leadsto u$ e p contiene almeno un arco, ed inoltre e' un **ciclo semplice** se i nodi sono distinti, tranne il primo e l'ultimo. v e' **raggiungibile** da u sse esiste un cammino $p: u \leadsto v$.
- una **componente fortemente connessa** di G e' una classe di equivalenza rispetto alla relazione di equivalenza R su V definita da uRv sse u e' raggiungibile da v e viceversa.

Elenchiamo alcune proprieta' di interesse per un grafo orientato G:

- G e' **semplice** sse non contiene cappi.
- G e' **aciclico** (DAG) sse non contiene cicli semplici. Se esiste un ciclo $p: u \leadsto u$, allora esiste un segmento di p che e' un ciclo semplice. Quindi un grafo e' aciclico sse non contiene cicli. G e' una **foresta** sse e' aciclico ed ogni nodo ha in-degree ≤ 1.
- G e' **fortemente connesso** sse per ogni coppia di nodi (u, v) esiste un cammino $p: u \leadsto v$.

I Grafi orientati possono essere visti come strutture relazionali per la segnatura con una sort ed un predicato binario, quindi si possono definire (nel modo usuale) le nozioni di

- omomorfismo $f: G_1 \to G_2$ sse $f: V_1 \to V_2$ t.c. $(u, v) \in E_1$ implica $(fu, fv) \in E_2$
- isomorfismo $f: G_1 \to G_2$ sse $f: V_1 \to V_2$ bigettiva e t.c. $(u, v) \in E_1$ sse $(fu, fv) \in E_2$
- G_1 sottografo di G_2 sse V_1 sottoinsieme di V_2 e $(u, v) \in E_1$ implica $(u, v) \in E_2$
- G_1 sottografo di G_2 generato da V_1 (con V_1 sottoinsieme di V_2) sse $\forall u, v \in V_1.(u, v) \in E_1 \leftrightarrow (u, v) \in E_2$.

5.1 Grafi: varianti

Un grafo non orientato si puo' vedere come un caso particolare di grafo orientato senza cappi e t.c. $(u, v) \in E$ implica $(v, u) \in E$. Le nozioni introdotto per i grafi orientati si trasferiscono quasi immutati ai grafi non orientati, le principali differenze riguardano

- grado di u = numero di archi entranti in u = numero di archi uscenti da u
- un cammino/ciclo $p: u \leadsto v$ in un grafo non orientato corrisponde ad un cammino/ciclo semplice (nel corrispondente grafo orientato).

Esistono molte variazioni della nozione di grafo orientato (e non orientato):

- Multigrafo (V, E, dom, cod) con $dom, cod: E \to V$, cioe' vi possono essere piu' archi tra due nodi prefissati
- Ipergrafo (V, E, inc) con $inc: E \to V^+$, cioe' un iperarco identifica una sequenza finita di nodi $(\overline{u}, v)$, dove $\overline{u}$ e' la sequenza dei nodi sorgente, e v e' il nodo destinazione
- Grafo con nodi etichettati (V, E, l) con $l: V \to L$, cioe' ad ogni node u e' assegnata una etichetta $l(u)$
- Grafo con archi etichettati (V, E, l) con $l: E \to L$, cioe' ad ogni arco e e' assegnata una etichetta $l(e)$.
- Grafo ordinato (V, E) se gli archi uscenti da un nodo sono ordinati linearmente.

5.2 Rappresentazione di grafi orientati

Le rappresentazioni piu' comunemente utilizzate per un grafo (orientato) sone quelle a matrice di adiacenza e a liste di adiacenza. Supponiamo fissato l'insieme dei nodi V, allora *astrattamente* un grafo $G = (V, E)$ puo' essere rappresentato

- con un array di insiemi di adiacenza `A: array [V] of "sottoinsiemi di V"`, la relazione di corrispondenza tra grafo e sua rappresentazione e' data da: (u, v) e' un arco sse v e' in $A[u]$.

  Vi e' una rappresentazione duale, in cui si associa ad ogni nodo u, l'insieme dei nodi incidenti, ovvero: (v, u) e' un arco sse v e' in $A[u]$.

- con un insieme di coppie ordinate `B: "sottoinsiemi di VxV"`, la relazione di corrispondenza tra grafo e sua rappresentazione e' data da: (u, v) e' un arco sse (u, v) e' in B.

Queste due rappresentazioni astratte possono essere specializzate scegliendo una delle rappresentazioni dei sottoinsiemi (bit-vector, sequenza senza ripetizioni, sequenza ordinata senza ripetizioni).

- se scegliamo di rappresentare i sottoinsiemi con dei bit-vetcors otteniamo la rappresentazione a matrice di adiacenza `M: array [V,V] of boolean`, $M[u, v]$ true sse (u, v) e' un arco.

- se scegliamo di rappresentare i sottoinsiemi con sequenze senza ripetizioni otteniamo la rappresentazione a sequenze di adiacenza (che a loro volta possono essere rappresentate con array, files, o liste con puntatori). Usando le sequenze senza riperizioni si rappresentano di fatto grafi ordinati.

Nel caso delle foreste (viste come grafi orientati semplici e con in-degree ≤ 1), si puo' adottare una rappresentazione piu' compatta `P : array [V] of V+{nil}`, che sfruttare la proprieta' che ogni nodo ha al piu' un padre, la corrispondenza tra foresta e rappresentazione e' data da: u padre di v sse $u = P[v]$. Non tutti gli array P corrispondono ad una foresta, p.e. P potrebbe rappresentare un grafo costituito da un ciclo semplice. Se $n = |V|$, si puo' convenire di enumerare i nodi da 1 ad n (ed identificare nil con 0) in modo che se v_i e' padre di v_j allora $i < j$.

5.3 Topological sorting

Possiamo utilizzare un grafo orientato per rappresentare le dipendenze tra diverse attivita', piu' precisamente una attivita' (o evento) puo' essere identificata con un nodo, e la presenza di un arco $u \to v$ indica che l'attivita' v puo' iniziare solo dopo il completamento dell'attivita' u. In questo contesto si e' interessati a risolvere i seguenti problemi:

- trovare le attivita' che non dipendono dal completamento di altre, ovvero che hanno in-degree $= 0$
- verificare l'assenza di cicli (che renderebbero impossibile l'espletamento di alcune attivita')
- trovare un ordine di esecuzione delle attivita' compatibile con i vincoli imposti dal grafo
- trovare una tempistica ottimale di espletamento delle attivita' (alcune delle quali potrebbero essere eseguite in parallelo). In quest'ultimo caso si potrebbe etichettare il nodo u con il tempo richiesto per svolgere l'attivita', e l'arco (u, v) con il tempo che deve intercorrere tra la fine dell'attivita' u e l'inizio dell'attivita' v.

Per tutti questi problemi e' possibile dare degli algoritmi che lavorano in tempo $O(|V| + |E|)$. Descriveremo gli algoritmi usando la rappresentazione astratta dei grafi come array di insiemi di adiacenza. Prima di procedere alla descrizione degli algoritmi diamo una definizione piu' formale dei problemi:

- calcolo dello in-degree (grado entrante). `input  G grafo` ,
  `output IN:array [V] of Nat` t.c. $In[u] =$ in-degree di u per ogni $u \in V$

- test di aciclicita'. `input  G grafo` ,
  `output nocicle:Bool` t.c. *nocicle* sse G non contiene cicli

- topological sorting (di un grafo aciclico). `input  G grafo` ,
  `output l lista di nodi` t.c. $l = (v_i | i \in |V|)$ senza ripetizioni e $\forall i, j \in |V|. v_i \to v_j \supset i < j$

- tempistica ottimale. `input  G grafo con archi etichettati in Time` , con Time monoide ordinato
  `output T:array [V] of Time` t.c. T e' la tempistica ottimale compatibile con il grafo, dove T e' compatibile con il grafo se $\forall (u, v) \in E. T[u] + l(u, v) \leq T[v]$, e T e' migliore di T' se $\forall u \in V. T[u] \leq T'[u]$.

5.3.1 Calcolo dello in-degree

```
input  A:array [V] of "sottoinsiemi di V"
output IN:array [V] of Nat

for v in V do "azzera IN[v]" {inizializzazione di IN}

for u in V do
  for v in A[u] do
    "incrementa IN[v]"
```

Lo out-degree di u e' semplicemente la cardinalita' di $A[u]$, se i sottoinsiemi sono rappresentati con un array allora la cardinalita' la di $A[u]$ si calcola in $O(1)$, e quindi l'out-degree si calcola in $O(|V|)$.

5.3.2 Topological sorting

```
input  A:array [V] of "sottoinsiemi di V"
output l:coda
var    IN:array [V] of Nat {vettore con in-degrees}
       X0:coda/stack {insieme dei nodi con in-degree 0}

"inizializza IN" {usando algoritmo per calcolo dell'in-degree}
X0 <- {u|IN[u]=0}
l  <- empty

while "X0 non vuoto" do
  "rimuovi un elemento u da X0"
  "inserisci u in l"
  for v in A[u] do  {rimuovi u dal grafo, aggiornando IN e X0}
    "decrementa IN[v]"
    if IN[v]=0 then "aggiungi v ad X0"
```

Se al termine del ciclo while l'array IN non ha tutti gli elementi nulli, allora vuol dire che il grafo contiene un ciclo. Le dipendenze tra le celle di uno spreadsheet possono essere rappresentate mediante un grafo ordinato, l'aciclicita' del grafo significa che alle celle e' possibile assegnare un valore calcolato, ed il topological sorting indica in che ordine calcolare il valore delle singole celle (per essere sicuri che i valori di tutte le celle da cui tale valore dipende siano gia' stati calcolati). La correttezza dell'algoritmo si puo' dimostrare con la tecnica dell'invariante, piu' precisamente le seguenti proprieta' valgono prima di eseguire un ciclo dello while

- l e' una lista senza ripetizioni, e $\forall(u,v) \in E$ se $v \in l$ allora u precede v in l
- $IN[u] =$ in-degree di u nel grafo G a cui sono stati tolti tutti gli archi uscenti da nodi in l
- $X0$ e' l'insieme dei nodi u t.c. $IN[u] = 0$ e $u \notin l$.

5.3.3 Tempistica ottimale

L'algoritmo ha la stessa struttura di quello per il topological sorting.

```
input  A: array [V] of "funzione parziale da V a Time"
output T:array [V] of Time
var    IN:array [V] of Nat {vettore con in-degrees}
       X0:coda/stack {insieme dei nodi con in-degree 0}

"inizializza IN" {usando algoritmo per calcolo dell'in-degree}
X0 <- {u|IN[u]=0}
for u in V do T[u]<-0

while "X0 non vuoto" do
  "rimuovi un elemento u da X0"
  for v adiacente ad u do {rimuovi u dal grafo, aggiornando IN e X0}
```

```
    T[v]<-max(T[v],T[u]+l(u,v))
    "decrementa IN[v]"
    if IN[v]=0 then "aggiungi v ad X0"
```

Esercizio: dare un invariante per dimostrare la correttezza parziale.

5.4 Visite di un grafo orientato

Fissato un nodo r di un grafo orientato $G = (V, E)$ diciamo che una sequenza $S = ((u_i, v_i)|1 \leq i \leq k)$ di archi e' una **visita** di G a partire da r sse

- l' i-esimo arco (u_i, v_i) della sequenza e' t.c. $u_i \in V_{i-1}$ mentre $v_i \notin V_{i-1}$, dove V_i e' definito per induzione su $0 \leq i \leq k$: $V_0 = \{r\}$ e $V_{i+1} = V_i \cup \{v_{i+1}\}$
- non esiste un arco (u, v) t.c. $u \in V_k$ ma $v \notin V_k$.

Ogni visita soddisfa le seguenti proprieta':

- V_k e' l'insieme dei nodi raggiungibili da r.

  Dimostrazione per assurdo. Se v e' raggiungibile da r ma non e' in V_k, si consideri il primo nodo u in un cammino da r a v che non sta in V_k, si ottenga una contraddizione con la seconda condizione nella definizione di visita.
- il grafo $G_i = (V_i, E_i)$ con $0 \leq i \leq k$, dove E_i e' l'insieme dei primi i archi in S, e' un albero con radice r.

Il seguente algoritmo fornisce uno schema generale per produrre una visita di $G = (V, E)$ a partire da r.

```
input  A:array [V] of "sottoinsiemi di V" {liste di adiacenza per G}
       r:V {nodo di partenza}
output S:coda di archi
var    U:sottoinsieme di V {dei nodi visitati}
       F:sottoinsieme di V {dei nodi visitati frontiera}

U <- F <- {r}
S <- empty

while "F non vuoto" do
  u <- "un elemento di F"
  if A[u] incluso in U
    then "rimuovi u da F"
    else v <- "un elemento di A[u]-U"
         "inserisci v in U ed F"
         "inserisci (u,v) in coda a S"
```

Le visite si possono sfruttare per risolvere una varieta' di problemi sui grafi: raggiungibilita', test di connettivita' (forte), calcolo delle componenti (fortemente) connesse. Per esempio, per stabilire se un grafo orientato G e' fortemente connesso basta prendere un nodo r e verificare se tutti i nodi sono raggiungibili da r sia nel grafo G che nel suo *duale* (ottenuto invertendo la direzione degli archi). Se S e' una visita, introduciamo la seguente notazione

- v_i e' l'i-esimo nodo visitato, e si conviene di indicare r con v_0. Quindi l'i-esimo arco di S e' della forma (u_i, v_i).
- p_i e' la posizione nella lista dei nodi visitati di u_i, cioe' il padre di v_i nell'albero associato ad S, ovvero $u_i = v_{p_i}$. Quindi p_0 e' indefinito, e $p_i < i$ se $i > 0$.
- V_i e' l'insieme $\{v_0, ..., v_i\}$.

5.4.1 Algoritmi di visita BFS e DFS

Vi sono due visite che si possono effettuare in modo efficiente, cioe' in $O(|V| + |E|)$, la visita in ampiezza (detta BFS) e quella in profondita' (detta DFS). Nella visita BFS si implementa F come una coda

```
var U:sottoinsieme di V {dei nodi visitati}
    F:coda di V {dei nodi visitati frontiera}

U <- F <- {r}
S <- empty

while "F non vuota" do
  "estrai il primo elemento u di F"
  for v in A[u] do
    if "v non in U"
      then "inserisci v in U ed F"
           "inserisci (u,v) in coda a S"
```

Nella visita DFS si implementa F come uno stack, o piu' semplicemente si usa una procedura ricorsiva

```
var U:sottoinsieme di V {dei nodi visitati}
U <- empty
S <- empty
DFS(r)
dove
DFS(u:V)
"inserisci u in U"
for v in A[u] do
  if "v not in U"
    then "inserisci (u,v) in coda a S"
         DFS(v)
```

Vediamo come si adatta l'algoritmo DFS per determinare le componenti connesse di un grafo NON ORIENTATO.

```
input  A:array [V] of sottoinsiemi di V
output c:array [V] of 0..|V| {c[v] numero della componente connessa}
var    k: 0..|V| {numero componenti connesse}
       U:sottoinsieme di V {dei nodi visitati}

U <- empty
k <- 0
for v in V do c[v] <- 0

while "V-U non vuoto" do
  k <- k+1
  r <- "un elemento di V-U"
* invariante:
* 0 < c[v] <= k se v in U, c[v] = 0 altrimenti
* {v|c[v]=i} componente connessa se 0 < i < k
* c[v]=k sse v in U e v nella stessa componente connessa di r
  DFS'(r)

DFS'(u:V)
"inserisci u in U"
c[u] <- k
for v in A[u] do
  if "v not in U" then DFS'(v)
```

5.4.2 Correttezza e terminazione degli algoritmi di visita

In questa sezione formaliziamo gli algoritmi di visita mediante automi deterministici, dimostrandone la correttezza parziale (con la tecnica dell'invariante) e la terminazione. Per ottenere degli automi deterministici occorre risolvere il non-determinismo in alcuni passi dell'algoritmo di visita. Il problema della visita di un grafo possiamo formularlo nel modo seguente (supponiamo che l'insieme V dei nodi sia fissato)

- l'insieme I degli input e' costituito dalle coppie (E, r) con $E \subseteq V \times V$ e $r \in V$

- l'insieme O degli output e' $(V \times V)^*$
- il problema $P \subseteq I \times O$ e' $(E,r)\ P\ S$ sse S e' una visita di $G = (V,E)$ a partire da r.

L'algoritmo generale di visita si puo' formalizzare con il seguente automa

- gli stati interni sono della forma (E,S,U,F) con $E \subseteq V \times V$, $S \in (V \times V)^*$ e $U,F \subseteq V$. U e' l'insieme dei nodi visitati, mentre F e' l'insieme dei potenziali nodi frontiera.
- la relazione di transizione e' data da
  - $(E,r) \Longrightarrow (E, nil, \{r\}, \{r\})$ inizializzazione
  - $(E,S,U,\emptyset) \Longrightarrow S$ output
  - $(E,S,U,F) \Longrightarrow (E,S,U,F-u)$ se u *primo elemento* di F e $A[u] \subseteq U$, dove $A[u] = \{v|(u,v) \in E\}$
  - $(E,S,U,F) \Longrightarrow (E, S\colon(u,v), U+v, F \cup v)$ se u *primo elemento* di F e v *primo elemento* di $\{v \in A[u]|v \notin U\}$

Per dimostrare la correttezza parziale dell'algoritmo utilizziamo il l'invariante costituito dalle coppie input (E,r) stato interno (E,S,U,F) t.c.

- S soddisfa la prima proprieta' per essere una visita di G a partire, cioe' $S = ((u_i,v_i) \in E|1 \leq i \leq k)$ e $\forall i \in [1..k].u_i \in V_{i-1} \wedge v_i \notin V_{i-1}$, dove $V_i = \{r\} \cup \{v_j|1 \leq j \leq i\}$
- $U = V_k$
- $\{u \in U|A[u] \not\subseteq U\} \subseteq F \subseteq U$

Per dimostrare la terminazione dell'algoritmo definiamo una funzione $p(E,S,U,F) = 2 * |V-U| + |F|$, dove $|F|$ e' la cardinalita' dell'insieme F. Occorre quindi dimostrare che ogni transizione tra stati interni fa' diminuire il valore di p.
L'algoritmo di visita BFS si puo' formalizzare con il seguente automa

- gli stati interni sono della forma (E,S,U,F) con $E \subseteq V \times V$, $S \in (V \times V)^*$, $U \subseteq V$ e $F \in V^*$ U e' l'insieme dei nodi visitati, mentre F e' la coda dei potenziali nodi frontiera.
- la relazione di transizione e' data da
  - $(E,r) \Longrightarrow (E, nil, \{r\}, r)$ inizializzazione
  - $(E,S,U,nil) \Longrightarrow S$ output
  - $(E,S,U,u\colon F) \Longrightarrow (E, S@((u,v)|v \in A'[u]), U \cup A'[u], F \cdot A'[u])$, dove $A'[u]$ e' la lista/insieme $[v \in A[u]|v \notin U]$

Per dimostrare la correttezza parziale dell'algoritmo utilizziamo lo *stesso* invariante usato per la correttezza parziale dell'algoritmo generale di visita, identificando la lista F con l'insieme dei suoi elementi Per dimostrare la terminazione dell'algoritmo definiamo la funzione $p(E,S,U,F) = |V-U| + |F|$, dove $|F|$ e' la lunghezza della lista F. Occorre quindi dimostrare che ogni transizione tra stati interni fa' diminuire il valore di p.

6 Cammini minimi

In questa sezione si considera il problema di trovare i cammini minimi/calcolare le distanze tra nodi di un grafo G con archi etichettati da numeri reali (che posso essere interpretati come lunghezze o costi). Per il corretto funzionamento di alcuni algoritmi (p.e. quello di Dijkstra) e' essenziale che i numeri reali siano ≥ 0 (e le dimostrazioni si semplificano ulteriormente se sono > 0). Inoltre gli algoritmi possono essere generalizzati al caso in cui le etichette, invece che numeri reali, sono elementi di *semianelli chiusi*. Tali generalizzazione saranno discusse in seguito. Diamo le definizioni necessarie per definire formalmente il problema:

- la **lunghezza** di un cammino $p = (u_i|0 \leq i \leq n)$ di G e' data da $l(p) \stackrel{\Delta}{=} \sum_{0\leq i<n} l(u_i, u_{i+1})$, cioe' la somma delle lunghezze degli archi che compongono il cammino p. In particolare, $l(p) = 0$ quando il numero di archi $n = 0$.
- la **distanza** di v da u in G e' data da $d(u, v) \stackrel{\Delta}{=} \inf\{l(p)|p: u \leadsto v\}$, cioe' l'estremo inferiore delle lunghezze dei cammini da u a v. In particolare, $d(u, v) = +\infty$ quando non esistono cammini $p: u \leadsto v$. In presenza di archi di lunghezza negativa e' possibile avere anche $d(u, v) = -\infty$. p.e. se $u = v$ e si ha un cappio con $l(u, u) = -1$ (infatti piu' giri si fa' intorno al cappio, e piu' si riduce la lunghezza del cammino).
- dato $U \subseteq V$ diciamo che il cammino $p = (u_i|0 \leq i \leq n)$ ha nodi intermedi in $U \stackrel{\Delta}{\iff} u_i \in U$ quando $0 < i < n$. Inoltre definiamo la distanza $d_U(u, v) \stackrel{\Delta}{=} \inf\{l(p)|p: u \leadsto v \wedge p$ ha nodi intermedi in $U\}$. Proprieta', che discendono immediatamente dalla definizione, sono: $d_U(u, v)$ decresce al crescere di U, e $d_V(u, v) = d(u, v)$.

Per rappresentare un grafo G con archi etichettati con numeri reali (ed eventualmente $-\infty$) si puo' usare una matrice M di dimensioni $V \times V$ a valori nei reali estesi con $+\infty$ (e $-\infty$), piu' precisamente M rappresenta $G \stackrel{\Delta}{\iff}$

- $M[u, v] = +\infty$ sse $(u, v) \notin E$
- $M[u, v] = l < +\infty$ sse $(u, v) \in E$ e $l(u, v) = l$

6.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra serve a trovare i cammini minimi/calcolare le distanze di tutti i nodi da un nodo prefissato r in un grafo con gli archi etichettati da numeri reali (che posso essere interpretati come lunghezze o costi). Per il corretto funzionamento dell'algoritmo e' essenziale che i numeri reali siano ≥ 0 (e le dimostrazioni si semplificano ulteriormente se i pesi sono > 0). L'algoritmo generalizza al caso pesato il comportamento delle visite BFS, cioe' visita i nodi del grafo in ordine di distanza crescente da r. Il punto cruciale e' come determinare il prossimo nodo da visitare.

```
input  M:array [V,V] of pesi {matrice delle lunghezze degli archi}
       r:V
output d:array [V] of pesi {vettore delle distanze da r}
var    U:sottoinsiemi di V {dei nodi visitati}
*inizializzazione
U <- {r}
for v in V do if v=r then d[v] <- 0 else d[v] <- M[r,v]
*invariante
*r in U
*d[v]=d(r,v) se v in U, d_U(r,v) altrimenti
*d(r,u) <= d(r,v) se u in U ed v in V-U
while V-U non vuoto do
  v <- un elemento di V-U che minimizza d[_]
  U <- U+{v}
  for x in V-U do if d[v]+M[v,x] < d[x] then d[x] <- d[v]+M[v,x]
```

Le seguenti proprieta' dei grafi con pesi > 0 sono essenziali per dimostrare la correttezza dell'algoritmo:

- $d(r, v) = 0$ sse $v = r$
- se $p: r \leadsto u$ e $q: u \leadsto q$ sono cammini t.c. $l(p) + l(q) = d(r, v)$, allora $l(p) = d(r, u) \leq d(r, v)$
- se $U \subseteq V$ e' t.c. $r \in U$ e $d(r, u) \leq d(r, v)$ per ogni $u \in U$ e $v \in V - U$, mentre $w \in V - U$ e' t.c. $\forall v \in V - U.d_U(r, w) \leq d_U(r, v)$, allora

1. $\forall v \in V - U.d(r, w) \leq d(r, v)$ e $d(r, w) = d_U(r, w)$
   si consideri il primo nodo $\notin U$ del cammino $p: r \rightsquigarrow v$ di lunghezza minima
2. $\forall v \in V - (U + w).d_{U+w}(r, v) = \min\{d_U(r, v), d_U(r, w) + l(w, v)\}$
   si consideri il penultimo nodo del cammino $p: r \rightsquigarrow v$ di lunghezza minima con nodi intermedi in $U + w$.

Per calcolarsi i cammini minimi basta mantenere un array p t.c. $p[v]$ contenga il padre di v nel cammino minimo da r a v con nodi intermedi in U.

Esercizio. Si identifichi il punto della dimostrazione di correttezza che non funziona in presenza di pesi negativi.

6.2 Algoritmo di Floyd

L'algoritmo di Floyd serve a trovare i cammini minimi/calcolare le distanze tra tutte le possibili coppie di nodi in un grafo pesato. Al contrario dell'algoritmo di Dijkstra non e' necessario assumere che i pesi siano non negativi. Inoltre, l'algoritmo di Floyd puo' essere generalizzato al caso di pesi a valori in un *semianello chiuso*. In questo modo, problemi apparentemente non correlati (sia su grafi che su matrici) possono essere risolti con l'algoritmo di Floyd scegliendo opportunamente il semianello chiuso.
L'idea base dell'algoritmo e' di calcolarsi le matrici D_k di dimensione $V \times V$ t.c. $D_k[u, v]$ e' la lunghezza del cammino minimo da u a v i cui nodi intermedi (cioe' senza considerare il primo ed ultimo nodo del cammino) sono contenuti nei primi k nodi del grafo (dove l'ordine lineare dei nodi puo' essere scelto in modo arbitrario). Poiche' l'ordine sui nodi e' del tutto irrilevante, potremo ignorarlo e considerare invece le matrici $D_U[u, v] = d_U(u, v)$ con $U \subseteq V$. L'idea fondamentale e' come calcolare D_{U+w}, con $w \in V - U$, a partire da D_U. Un cammino da u a v con nodi intermedi un $U + w$ deve necessariamente essere di una delle seguenti forme:

- un cammino $p: u \rightsquigarrow v$ con nodi intermedi in U, e scriveremo $p: u \rightsquigarrow_U v$
- un cammino $p: u \rightsquigarrow v$ che ha w come nodo intermedio esattamente $k > 0$ volte, e quindi avremo che p e' ottenuto concatenando i cammini propri (cioe' con almeno un arco)
  - $p_0: u \rightsquigarrow_U w$
  - $p_i: w \rightsquigarrow_U w$ con $0 < i < k$
  - $p_k: w \rightsquigarrow_U v$

A questo punto si tratta di esaminare come puo' essere fatto il cammino $p: u \rightsquigarrow_{U+w} v$ di lunghezza minima:

- se $p: u \rightsquigarrow_U v$ allora la sua lunghezza sara' $D_U[u, v]$
- se $p: u \rightsquigarrow_U w \rightsquigarrow_U v$ allora la sua lunghezza sara' $D_U[u, w] + D_U[w, v]$
- se invece esiste un ciclo $q: w \rightsquigarrow_U w$ di lunghezza < 0, e quindi $D_U[w, w] < 0$, allora vi sono cammini $p: u \rightsquigarrow_{U+w} v$ di lunghezza negativa arbitrariamente grande (ripetendo il ciclo q). Indicheremo questa terza eventualita' ponendo $D_{U+w}[u, v] = -\infty$.

Occorre fare attenzione nel manipolare $+\infty$ e $-\infty$. Il primo indica l'assenza di cammini, l'altro la presenza di cammini con lunghezza negativa grande a piacere. Con questa interpretazione e' sensato convenire che $(+\infty) + (-\infty) = +\infty$.

```
input  M:array [V,V] of pesi {+infinito indica assenza di arco}
output D:array [V,V] of pesi {+infinito indica assenza di cammini, -infinito
         indica cammini di lunghezza negativi arbitrariamente grande}
var    D:array [0..|V|,V,V] of pesi
for x,y in V do if x=y then D[0,x,y] <- min(0,M[x,y]) else D[0,x,y] <- M[x,y]

for k <- 1 to |V| do
  for x,y in V do
    D[k,x,y] <- min(D[k-1,x,y] , D[k-1,x,v_k]+D[k-1,v_k,v_k]^*+D[k-1,v_k,y])

for x,y in V do D[x,y] <- D[|V|,x,y]
```

dove $c^* = 0$ se $c \geq 0$, e $c^* = -\infty$ se $c < 0$. Per risparmiare memoria, si puo' evitare di mantenere un array di matrici e lavorare direttamente sulla matrice di output D, in tal caso occorre fare attenzione a sovrascrivere un valore solo quando non serve piu' per i calcoli intermedi.

```
input  M:array [V,V] of pesi
output D:array [V,V] of pesi
for x,y in V do
  if x=y and 0<=D[x,y] then D[x,y] <- 0 else D[x,y] <- M[x,y]

for k <- 1 to |V| do
  D[v_k,v_k] <- D[v_k,v_k]^* {ovvero 0 o -infinito}
  for x,y in V-v_k do
    D[x,y] <- min(D[x,y] , D[x,v_k]+D[v_k,v_k]+D[v_k,y])
  for x in V-v_k do
    D[x,v_k] <- D[x,v_k]+D[v_k,v_k]
  for y in V-v_k do
    D[v_k,y] <- D[v_k,v_k]+D[v_k,y]
```

Con questa ottimizzazione la complessita' spazio passa da $O(|V|^3)$ a $O(|V|^2)$, mentre la complessita' tempo rimane in $O(|V|^3)$. Dal punto di vista della complessita' l'algoritmo di Floyd e' equivalente ad eseguire l'algoritmo di Dijkstra $|V|$ volte, una per ogni $r \in V$. Tuttavia, l'algoritmo di Dijkstra e' corretto solo nell'ipotesi di pesi non negativi, mentre Floyd non richiede ipotesi aggiuntive. Infine consideriamo due varianti dell'algoritmo di Floyd, che oltre alla distanza $d(x, y)$ permettono di calcolare anche il cammino minimo da x a y, se esiste (cioe' se $-\infty < D[x, y] < +\infty$). In entrambe le varianti abbiamo in output matrice `D` delle distanze e una matrice `P:array [V,V] of V+{nil}` t.c. $P[x, y]$ contiene un nodo del cammino minimo da x a y. Nella prima variante (similmente a quanto fatto per Dijkstra) $P[x, y]$ e' il penultimo nodo del cammino minimo da x a y

```
input  M:array [V,V] of pesi
output D:array [V,V] of pesi
       P:array [V,V] of V+{nil}
for x,y in V do
  if x=y and 0<=M[x,y]
  then D[x,y] <- 0; P[x,y] <- nil
  else D[x,y] <- M[x,y]; P[x,y] <- x

for k <- 1 to |V| do
  D[v_k,v_k] <- D[v_k,v_k]^*; P[v_k,v_k] <- nil
  for x,y in V-v_k do
    if D[x,v_k]+D[v_k,v_k]+D[v_k,y] < D[x,y]
    then D[x,y] <- D[x,v_k]+D[v_k,v_k]+D[v_k,y]; P[x,y] <- P[v_k,y]
  for x in V-v_k do
    D[x,v_k] <- D[x,v_k]+D[v_k,v_k]
  for y in V-v_k do
    D[v_k,y] <- D[v_k,v_k]+D[v_k,y]
```

Nella seconda variante $P[x, y] = v_k$ sse v_k e' il nodo intermedio sul cammino minimo da x a y con il valore k piu' grande. La seconda variante e' piu' in sintonia con l'algoritmo di Floyd, tuttavia la ricostruzione del cammino minimo da x a y richiede una procedura ricorsiva, invece che iterativa.

```
input  M:array [V,V] of pesi
output D:array [V,V] of pesi
       P:array [V,V] of V+{nil}
for x,y in V do
  if x=y and 0<=D[x,y] then D[x,y] <- 0 else D[x,y] <- M[x,y]
  P[x,y] <- nil

for k <- 1 to |V| do
  D[v_k,v_k] <- D[v_k,v_k]^*; P[v_k,v_k] <- nil
  for x,y in V-v_k do
    if D[x,v_k]+D[v_k,v_k]+D[v_k,y] < D[x,y]
    then D[x,y] <- D[x,v_k]+D[v_k,v_k]+D[v_k,y]; P[x,y] <- v_k
  for x in V-v_k do
    D[x,v_k] <- D[x,v_k]+D[v_k,v_k]
  for y in V-v_k do
    D[v_k,y] <- D[v_k,v_k]+D[v_k,y]
```

6.3 Semi-anelli chiusi e chiusura transitiva

Descriveremo un algoritmo per calcolare la *chiusura transitiva* di una matrice a valori in un *semi-anello chiuso*. L'algoritmo e' una generalizzazione (e lieve variazione) dell'algoritmo di Floyd per calcolare le distanze tra tutte le coppie di nodi. Variando opportunamente il semi-anello si possono risolvere (con lo stesso algoritmo astratto) una varieta' di problemi su grafi e matrici.

6.4 Semi-anelli chiusi

Introduciamo le nozioni matematiche preliminari per descrivere il problema e giustificare la correttezza dell'algoritmo. Un **semi-anello** $(A, +, *, 0, 1)$ e' una struttura algebrica con due operazioni binarie $+$ e $*$ e due costanti 0 ed 1 che soddisfano i seguenti assiomi

- associativita' del $+$: $(x + y) + z = x + (y + z) \quad 0 + x = x = x + 0$
- commutativita' del $+$: $x + y = y + x$
- associativita' del $*$: $(x * y) * z = x * (y * z) \quad 1 * x = x = x * 1$
- distributivita': $(x + y) * z = (x * z) + (y * z) \quad z * (x + y) = (z * x) + (z * y) \quad 0 * z = 0 = z * 0$

Un **semi-anello chiuso** e' un semi-anello dove si possono effettuare sommatorie arbitrarie $\sum_{i \in I} a_i$, dove I e' un insieme qualsiasi (usando 0 e + si possono calcolare solo sommatorie su insiemi finiti), e dove gli assiomi dei semi-anelli che riguardano 0 e + devono essere estesi al caso di sommatorie arbitrarie:

- associativita' di $\sum$: $\sum_{i \in I}(\sum_{j \in J_i} x_{i,j}) = \sum_{i \in I, j \in J_i} x_{i,j}$ dove J_i e' un insieme per ogni $i \in I$
- commutativita' di $\sum$: $\sum_{i \in I} x_i = \sum_{j \in J} x_{f(j)}$ dove $f \colon J \to I$ e' una funzione bigettiva
- associativita' del $*$ (come per i semi-anelli): $(x * y) * z = x * (y * z) \quad 1 * x = x = x * 1$
- distributivita': $(\sum_{i \in I} x_i) * z = \sum_{i \in I}(x_i * z) \quad z * (\sum_{i \in I} x_i) = \sum_{i \in I}(z * x_i)$

In un semi-anello chiuso e' possibile definire due operazioni unarie

- **chiusura transitiva** $a^+ = \sum_{n>0} a^n$
- la **chiusura riflessiva e transitiva** $a^* = 1 + a^+ = \sum_{n \geq 0} a^n$

dove a^n e' il prodotto di $a \in A$ con se' stesso n volte, cioe' $a^0 = 1$ e $a^{n+1} = a * a^n$. Vi sono innumerevoli esempi di semi-anelli (chiusi), per esempio:

1. $(\mathsf{N}, +, *, 0, 1)$, cioe' i numeri naturali N con le usuali operazioni aritmetiche, questo semi-anello NON E' CHIUSO
2. $(\overline{\mathsf{N}}, +, *, 0, 1)$, cioe' i numeri naturali estesi con $+\infty$ e con le usuli operazioni opportunamente estese: $x + (+\infty) = +\infty$, $0 * (+\infty) = 0$, $n * (+\infty) = +\infty$ se $n > 0$
3. $(\overline{\mathsf{N}}, \min, +, +\infty, 0)$, cioe' i numeri naturali estesi con $+\infty$ con le operazioni di minimo e somma (esercizio: verificare la proprieta' distributiva)
4. $(\overline{\mathsf{N}}, \max, \min, 0, +\infty)$, cioe' i numeri naturali estesi con $+\infty$ con le operazioni di massimo e minimo (esercizio: verificare la proprieta' distributiva)
5. vi sono le ovvie varianti ottenute sostituendo a N i reali/razionali non negativi $\mathsf{R}_0/\mathsf{Q}_0$
6. $(\mathsf{B}, \vee, \wedge, \perp, \top)$, cioe' i valori booleani con le operazioni logiche or $\vee$ e and $\wedge$ e le costanti falso $\perp$ e vero $\top$
7. $(\overline{\mathsf{R}}, \min, +, +\infty, 0)$, cioe' i reali estesi con $+\infty$ e $-\infty$, dove $x + (+\infty) = +\infty$ anche quando $x = -\infty$
8. $(\overline{\mathsf{R}}, \max, +, -\infty, 0)$, cioe' i reali estesi con $+\infty$ e $-\infty$, dove $x + (-\infty) = -\infty$ anche quando $x = +\infty$
   Si noti che $(\mathsf{N}, \max, +, 0, 0)$, NON e' un semi-anello.

In alcuni di questi semi-anelli le operazioni soddisfano proprieta' aggiuntive, p.e.: tutte le interpretazioni del $*$ soddisfano la proprieta' commutativa $x * y = y * x$; quando $+$ e' interpretato da min o max, allora vale l'idempotenza $x + x = x$. Per esercizio si provi a definire esplicitamente le operazioni di chiusura. Vi sono costruzioni che permettono di definire nuovi semi-anelli (chiusi) a partire da semi-anelli (chiusi) piu' semplici, per esempio

- Se A_1 e A_2 sono semi-anelli, allora $A_1 \times A_2$ e' il semi-anello con le operazioni definite componente per componente, p.e. $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$.

  In modo analogo si puo' definire il prodotto $\prod_{i \in I} A_i$ di una famiglia di semi-anelli.

- Se A e' un semi-anello, allora l'insieme delle matrici $V \times V$ (con V insieme finito) a coefficienti in A e' un semi-anello con le operazioni sulle matrici definite nel modo usuale, cioe' $(M + N)[u, v] = M[u, v] + N[u, v]$ e $(M * N)[u, v] = \sum_{w \in V} (M[u, w] * N[w, v])$.

  Se A e' un semi-anello chiuso, allora l'insieme V puo' essere anche essere infinito.

Se i semi-anelli di partenza sono chiusi, e' facile vedere che, anche i semi-anelli definiti da tali costruzioni sono chiusi.

6.5 Operazioni su matrici e cammini

Vi e' una corrispondenza biunivoca tra matrici M di dimensione $V \times V$ a coefficienti in un semi-anello A e grafi G con insieme dei nodi V ed archi con pesi in $A - \{0\}$ data da: (x, y) e' un arco di G con etichetta $a \neq 0$ sse $M[x, y] = a$. Quando A e' il semi-anello B dei booleani (e quindi 0 e' interpretato dal falso $\perp$), allora M non e' altro che la matrice di adiacenza di G.

Data una matrice M a coefficienti in A (e G il grafo corrispondente), possiamo definire i seguenti valori derivati:

- $M[p]$. Sia $p: x \rightsquigarrow y$ un cammino di G, definiamo il suo peso $M[p]$ come il prodotto in A dei pesi che si incontrano lungo il cammino, cioe' $M[x_0, ..., x_n] = M[x_0, x_1] * ... * M[x_{n-1}, x_n]$ (e si pone $M[x] = 1$). Si osservi che $M[p] = 0$, se la sequenza di nodi non e' un cammino in G, poiche' $M[x_{i-1}, x_i] = 0$ per qualche i.

  $M^n[x, y]$ risulta uguale alla sommatoria in A di $M[p]$ con p che varia sui cammini da x a y in G con esattamente n archi. Poiche' $M[p] = 0$ se p e' una sequenza di nodi che non corrisponde ad un cammino, tali addendi non contribuiscono alla sommatoria.

  $M^+[x, y]$ (cioe' la chiusura transitiva di M) risulta uguale alla sommatoria in A di $M[p]$ con p che varia sui cammini propri (cioe' con almeno un arco) da x a y.

- $M^+_U[x, y]$ e' la sommatoria di $M[p]$ con p che varia sui cammini propri da x a y con nodi intermedi in U.

  Dato $v \in V - U$ si dimostra che $M^+_{U+v}[x, y] = M^+_U[x, y] + (M^+_U[x, v] * (M^+_U[v, v])^* * M^+_U[v, y])$

  Quindi formalmente abbiamo la stessa equazione alla base dell'algoritmo di Floyd per il calcolo delle distanze.

Si puo' definire anche $M^*_U[x, y]$, pero' in generale non soddisfa la stessa definizione ricorsiva di M^+, a meno che la sommatoria in A non soddisfi anche la proprieta' di idempotenza (cioe' la versione infinitaria di $x + x = x$).

6.6 Algoritmo di Floyd per calcolare la chiusura transitiva

Sia A un semi-anello chiuso, su cui non faremo alcune ipotesi tranne il fatto di essere in grado di calcolare $x + y$, $x * y$, 0, 1 e x^*. Diamo l'algoritmo (stile Floyd) per calcolare la chiusura transitiva M^+ di una matrice $V \times V$ a valori in A e con V finito. Le operazioni $x + y$, $x * y$ e x^* sono quelle del semi-anello chiuso A.

```
input:  M: array [V,V] of A
output: M^+: array [V,V] of A
var: M': array [1..|V|] of array [V,V] of A
*inizializzazione
M'[0] <- M
for k <- 1 to |V| do
for x,y in V do
M'[k,x,y] <-  M'[k-1,x,y]+(M'[k-1,x,v_k]*(M'[k-1,v_k,v_k]^*)*M'[k-1,v_k,y])
M^+ <- M'[|V|]
```

Tale algoritmo e' interessante perche' instanziando A con opportuni semi-anelli chiusi le matrici M^+ e $M^*(= I + M^+)$ forniscono la soluzione a svariati problemi relativi a grafi pesati. Per esempio:

- $M^*[x, y]$ e' il numero di cammini da x a y, se $A = (\overline{\mathsf{N}}, +, *, 0, 1)$ e $M[x, y] = 1$ se (x, y) e' un arco, 0 altrimenti.
- $M^*[x, y]$ e' la distanza da x a y (la lunghezza del cammino di lunghezza minima), se $A = (\overline{\mathsf{R}}, \min, +, +\infty, 0)$ e $M[x, y]$ e' la lunghezza dell'arco (x, y).
- $M^*[x, y]$ e' la lunghezza del cammino di lunghezza massima, se $A = (\overline{\mathsf{R}}, \max, +, -\infty, 0)$ e $M[x, y]$ e' la lunghezza dell'arco (x, y).
- $M^*[x, y]$ sse esiste un cammino da x a y, se $A = (Bool, or, and, false, true)$ e M e' la matrice di adiacenza.
- $M^*[x, y]$ e' la capacita' del cammino da x a y con capacita' massima, se $A = (\overline{\mathsf{N}}, \max, \min, 0, +\infty)$ e $M[x, y]$ e' la capacita' dell'arco (x, y).

Esercizio: trovare un semi-anello chiuso A in modo che da $M^*[x, y]$ sia ricavi il numero di cammini di lunghezza minima da x a y (serve assumere che gli archi abbiano lunghezza positiva?).

7 Esercizi

7.1 Costruzione di semi-anelli

Supponiamo fissato un semi-anello A, non necessariamente chiuso, ed una sua implementazione (per semplicita' si supponga che le implementazioni delle operazioni su A abbiano costo costante).

Semi-anello delle funzioni. Dato un qualunque insieme X, l'insieme delle funzioni $X \to A$ ha una struttura di semi-anello ottenuta estendendo in modo puntuale le operazioni su A, cioe'

- $f_1 + f_2$ e' la funzione t.c. $(f_1 + f_2)(x) = f_1(x) + f_2(x)$
- $f_1 * f_2$ e' la funzione t.c. $(f_1 * f_2)(x) = f_1(x) * f_2(x)$

In generale il semi-anello $X \to A$ non e' implementabile (perche' le funzioni sono oggetti infinitari, quando X e' un insieme infinito).

Semi-anello dei polinomi in x. Si consideri il semi-anello $A[x]$ dei polinomi in una variabile, un polinomio puo' essere descritto come una somma formale di monomi $p(x) = \sum_{i=0}^{n} a_i * x^i$ con $a_n \neq 0$, n e' detto il grado del polinomio p. Ad ogni polinomio resta associata una funzione $A \to A$, e le operazioni $+$ e $*$ sui polinomi (come somme formali) sono consistenti con quelle definite su $A \to A$.
Implementare il tipo di dato $A[x]$, supponendo di disporre di una implementazione di A, ed analizzare gli algoritmi per le operazioni del semi-anello $A[x]$ in funzione del grado.
OSSERVAZIONE: in alcuni semi-anelli, p.e. quello dei reali, un polinomio di grado $\leq n$ e' univocamente determinato dai valori che assume in $n+1$ punti distinti $x_0, \ldots, x_n$. In questi casi si puo' adottare una rappresentazione alternativa dei polinomi. Analizzare il costo (in funzione del grado del polinomio) per passare da una rappresentazione all'altra.

Semi-anello generato dalle matrici $X \times X$ quasi ovunque nulle. Quando X e' infinito, sulle matrici $X \times X$ quasi ovunque nulle e' possibile definire le operazioni $+$ e $*$, ma l'unita' per $*$, cioe' la matrice identita' I non e' quasi ovunque nulla. Si estenda l'insieme delle matrici $X \times X$ quasi ovunque nulle in modo da ottenere un semi-anello implementabile.
SUGGERIMENTO: si faccia vedere che le matrici della forma $a * I + M$ con $a \in A$ e M matrice quasi ovunque nulla, formano un semi-anello. Implementare tale semi-anello, ed analizzare gli algoritmi per le operazioni in funzione del numero di coefficienti non nulli di M.

Insiemi ereditari. Si consideri l'estensione del tipo di dato set con una operazione $include\colon set \to elem$ che permette di vedere un insieme come un elemento. A livello di modello canonico $include^A$ dovrebbe corrispondere all'inclusione $\mathcal{P}_{fin}(U) \subseteq U$, quindi l'interpretazione di $elem$ non puo' essere scelta in modo indipendente dalla interpretazione di set, ovvero l'interpretazione delle due sort e' mutuamente ricorsiva. Quello che si puo' fare e' definire U in modo induttivo, a partire da un insieme U_0 di elementi (che non sono insiemi):

- $U_{n+1} = U_0 + \mathcal{P}_{fin}(U_n)$
- $U_n \subset U_{n+1}$, la dimostrazione e' per induzione su n
- $U = \cup_n U_n$, e si dimostra che $U = U_0 + \mathcal{P}_{fin}(U)$ (U e' il piu' piccolo insieme che soddisfa tale proprieta').

Un'altro problema e' come estendere un ordine lineare su U_0 ad un ordine lineare su U, anche questo puo' essere definito in modo induttivo, sulla falsariga della definizione di U_n. L'idea generale e' di definire modello canonico A, l'implementazione B e relazione di simulazione R adattando la tecnica divide-et-impera ai tipi di dato, ovvero si sta' definendo un tipo di dato in modo ricorsivo. Per l'implementazione B (mediante liste senza ripetizione) definiremo l'insieme V per interpretare $elem$ per analogia con la definizione di U

- $V_{n+1} = U_0 + V_n^*$
- $V_n \subset V_{n+1}$, la dimostrazione e' per induzione su n
- $V = \cup_n V_n$, e si dimostra che $V = U_0 + V^*$ (V e' il piu' piccolo insieme che soddisfa tale proprieta').

Intutivamente la relazione di simulazione $R \subseteq V \times U$ dovrebbe soddisfare le seguenti proprieta'

- se $v \in U_0$, allora $vRu \stackrel{\Delta}{\iff} v = u$
- se $v = (v_i|i \in m) \in V^*$, allora $vRu \stackrel{\Delta}{\iff} \forall i \in m.\exists u_i \in U.v_iRu_i \wedge u = \{u_i|i \in m\} \wedge$ "v e' senza ripetizioni"

il punto delicato e' cosa vuol dire "v e' senza ripetizioni", o piu' precisamente cosa deve essere l'interpretazione $=^B: V \times V \to \mathit{bool}$. La risposta giusta e' che $=^B$ deve implementare $=^A$, quindi $v_i =^B v_j$ sse $u_i = u_j$. Quindi quello che si vuole richiedere e' che la lista $(u_i|i \in m)$ e' senza ripetizioni. La definizione di $R \subseteq V \times U$ con le proprieta' richiesta e' $R = \cup R_n$, dove $R_n \subseteq V_n \times U_n$ e' definita per induzione

- $vR_0u \stackrel{\Delta}{\iff} v = u$ (sia v che u sono in U_0)
- $vR_{n+1}u \stackrel{\Delta}{\iff} v = u$, quando $v \in U_0$, altrimenti
  $(v_i|i \in m)R_{n+1}u \stackrel{\Delta}{\iff} \forall i \in m.\exists u_i \in U.v_iR_nu_i \wedge u = \{u_i|i \in m\} \wedge (u_i|i \in m)$ e' senza ripetizioni.
- $R_n = R_{n+1} \cap V_n \times U_n$, la dimostrazione e' per induzione su n

Il seguente codice definisce l'implementazione della uguaglianza su V in un linguaggio ML-like

```
type U0 0 ... ; (* tipo degli elementi che non sono insiemi *)
datatype V = elem of U0 | set of V list;
- fun forall nil = true | forall (b::l) = b andalso (forall l);
val forall = fn : bool list -> bool
- fun never nil = true | never (false::l) = never l | never (true::l) = false;
val never = fn : bool list -> bool
fun existsunique nil = false
  | existsunique (false::l) = existsunique l
  | existsunique (true::l) = never l;
val existsunique = fn : bool list -> bool

fun eq (elem u1,elem u2) = u1=u2
  | eq (set l1,set l2) =
    forall (map (fn x => existsunique (map (fn y => eq(x,y)) l2)) l1) andalso
    forall (map (fn x => existsunique (map (fn y => eq(x,y)) l1)) l2)
  | eq (_,_) = false;
```

Una migliore implementazione e' data dalla rappresentazione mediante liste ordinate. Per fare cio' occorre estendere un ordine lineare su U_0 ad un ordine lineare su U. La scelta piu' ovvia e' data dall'ordine lessicografico

$$\{u_i|i \in m\} < \{u'_j|j \in n\} \stackrel{\Delta}{\iff} \begin{array}{l} \forall i.u_i < u_{i+1} \wedge \forall j.u'_j < u'_{j+1} \wedge \\ (m < n \wedge \forall i.u_i = u'_i) \vee (\exists k < m, n.u_k < u'_k \wedge \forall i < k.u_i = u'_i) \end{array}$$

Formalmente si dovrebbe definire $<$ in termini di opportuni ordini lineari $<_n \subseteq U_n \times U_n$. Nella implementazione mediante liste ordinate l'ordine lessicografico e' definibile nel modo seguente

```
fun lt (elem x,elem y) = x<y
  | lt (elem x,set l2) = true
  | lt (set nil,set(y::l2)) = true
  | lt (set(x::l1),set(y::l2)) = lt(x,y)
  | lt (set(x::l1),set(y::l2)) = x=y andalso lt(set l1,set l2)
  | lt (_,_) = false;
fun ok (elem x) = true
  | ok (set(x::nil)) = ok x
  | ok (set(x::y::l)) = ok x andalso lt(x,y) andalso ok(set(y::l));
```

Il predicato *ok* identifica gli elementi di V che rappresentano un (unico) elemento di u.

7.2 Implementazioni alternative

Chiusura transitiva di una matrice sparsa. Abbiamo dato un algoritmo per il calcolo della chiusura transitiva di una matrice M di dimensione $n \times n$ a coefficenti in un semi-anello A, supponendo che M fosse rappresentata mediante un array. Si vuole adattare l'algoritmo in mod che operi su matrici di dimensione $\mathsf{N} \times \mathsf{N}$ ma quasi ovunque nulle. Tali matrici non possono essere rappresentate mediante un array di dimensione prefissata, ma solo mediante liste, p.e. liste in $\mathsf{N} \times \mathsf{N} \times A$ ordinate nelle prime due componenti. Sviluppare i dettagli dell'algoritmo per tale rappresentazione.

OSSERVAZIONE: un possibile approccio consiste nel trovare il minimo n t.c. tutti i coefficenti non nulli di M si trovano nel quadrato $n \times n$, rappresentare M con un array $n \times n$, usare l'algoritmo visto a lezione, quindi convertire l'array $n \times n$ che rappresenta M^+ nella rappresentazione a liste. Tuttavia la complessita' tempo di tale algoritmo non e' maggiorabile in funzione del numero di coefficenti non nulli di M (p.e. se l'unico elemento non nullo di M e' in posizione $(10^9 \times 10^9)$ l'algoritmo lavorerebbe in tempo 10^{3*9}, mentre la chiusura transitiva si puo' calcolare in $O(1)$.

Tipo di dato set implementato con 3^+-alberi. Per l'implementazione basata sui 3^+-alberi ci siamo limitati a dare algoritmi con complessita' tempo $O(\log n)$ per le operazioni del tipo di dato dizionarip, escludendo le operazioni insiemistiche *union* e *intersect*. Si diano degli algoritmi per tali operazioni che lavorino in tempo $O(n_1 + n_2)$.

SUGGERIMENTO: l'approccio piu' semplice di procedere e' di trovare degli algoritmi di conversione per passare dalla rappresentazione a 3^+-albero a quella a lista ordinata e viceversa, che lavorino in tempo $O(n)$. La conversione piu' difficile e' da lista ordinata a 3^+-albero. Si consiglia di usare 2-nodi a tutti i livelli tranne che negli ultimi 2 livelli di nodi interni. Si osservi che i 3^+-alberi con 2 livelli di nodi interni possono avere un numero di nodi foglia che va' da un minimo di 4 ad un massimo di 9.

Varianti di algoritmi su grafi. Vari algoritmi su grafi (p.e. topological sorting, visite BFS e DFS) lavorano in tempo $O(V + E)$ quando si adotta una rappresentazione a liste di adiacenza. Voglia cambiare la rappresentazione del grafo usando una lista di liste di adiacenza (invece di un array di liste di adiacenza), in modo da non dover fissare a priori il numero di nodi del grafo, ma senza peggiorare la complessita' tempo di tali algoritmi. Sviluppare i dettagli di tale rappresentazione a lista di liste di adiacenza.

SUGGERIMENTO: l'idea chiave e' di associare ad ogni nodo un campo ausiliario per contenere informazioni temporanee, che negli algoritmi visti a lezione venivano tenute in array ausiliari (p.e. nodi visitati o in-degree). Descrivere l'implementazione in un linguaggio simile a C o Java.

Numero cammini in un DAG. Dato un DAG G si vuole calcolare per ogni nodo u quanti cammini ci sono che iniziano in un nodo radice ed arrivano ad u (r e' una radice se non ha nodi entranti). Adattare l'algoritmo del topological sorting per calcolare tali valori.

Implementazione di un foglio elettronico. Si implementi un foglio elettronico dove ogni cella puo' contenere una costante intera n o una operazione binaria *op* (che puo' essere $+$, $-$ o $*$) applicata a due celle (che possono anche essere la stessa). Le operazione che si desidera avere sono:

- $new(){:}\,cell$ crea una nuova cella e la inizializza a 0
- $set(x{:}\,cell, n{:}\,int)$ assegna n ad x
- $set_{op}(x, x_1, x_2{:}\,cell)$ assegna ad x l'espressione $x_1 op x_2$
- $def(x{:}\,cell){:}\,bool$ dice se la cella x ha un valore (ci potrebbero essere dei cicli nel grafo delle dipendenze tra le celle)
- $value(x{:}\,cell){:}\,int$ restituisce il valore della cella x, se definito

Ci sono due approcci al calcolo dei valori: si ricalcolano tutti ogni volta che si effettua una operazione *set* (in tal caso def e $value$ hanno costo costante), oppure si ricalcola solo i valori strettamente necessari per rispondere ad una richiesta def o $value$ (in tal caso le *set* hanno costo costante).

SUGGERIMENTO: si puo' mantenere per ogni cella anche la lista delle celle che fanno riferimento ad essa, oltr alla informazione sulle celle (al piu' 2) a cui essa fa' riferimento. In entrambi gli approcci si cerchi di ottenere degli algoritmi che dipendono solo dal numero di celle che vanno effettivamente ricalcolate, invece che dal numero di celle che fanno parte del foglio elettronico.

8 Programmazione dinamica

Presentiamo alcuni esempi concreti che illustrano la tecnica della programmazione dinamica. Tale tecnica si puo' applicare ogni volta che un problema puo' essere riformulato mediante una definizione ricorsiva (stile divide-et-impera). La differenza essenziale tra un algoritmo di programmazione dinamica e quello divide-et-impera (indotti dalla stessa definizione ricorsiva del problema), consiste nell'evitare di ricalcolare la soluzione dei sottoproblemi gia' risolti. Quindi se un sottoproblema viene ricalcolato molte volte da un algoritmo divide-et-impera, si puo' ridurre drasticamente il tempo di calcolo (questo puo' pero' richiedere di pagare una penale in termini di spazio necessario per memorizzare le soluzioni ai sottoproblemi). Un modo semplice di analizzare i trade-offs tra le due tecniche (programmazione dinamica e divide-et-impera) e' di considerare il grafo orientato delle chiamate ricorsive: i nodi sono le istanze del problema (cioe' i possibili input $x \in I$) e gli archi vanno da un problema ai suoi sottoproblemi immediati, cioe' quelli menzionati espicitamente nella definizione ricorsiva (quindi i casi base non hanno archi uscenti). Se la definizione ricorsiva e' ben fondata, il grafo delle chiamate ricorsive e' aciclico.

- per risolvere il problema x con il divide-et-impera si ricalcola la soluzione di un sottoproblema x' tante volte quanti sono i cammini da x a x' nel grafo delle chiamate ricorsive
- per risolvere il problema x con la programmazione dinamica si calcola la soluzione di un sottoproblema x' al piu' una volta.

Quindi se il grafo e' un albero, la programmazione dinamica non fornisce alcun vantaggio rispetto al divide-et-impera. La successione di Fibonacci.

- f(0) = f(1) = 1
- f(i) = f(i-1) + f(i-2)

fornisce un semplice esempio per illustrare i vantaggi della programmazione dinamica rispetto al divide-et-impera:

- l'algoritmo divide-et-impera (corrispondente alla definizione ricorsiva di $f(i)$) per calcolare $f(n)$ ricalcolera' $f(n-i)$ volte il valore $f(i)$, e poiche' $f(2i) \geq 2^i$ il tempo di calcolo sara' esponenziale in n.
- l'algoritmo di programmazione dinamica invece richiede tempo e spazio lineari in n.
- facendo un'analisi piu' accurata del grafo delle chiamate ricorsive, si vede che non e' necessario mantenere tutti i valori di $f(i)$ per $i \leq n$, basta conservare gli ultimi due, quindi si richiede solo uno spazio costante e tempo lineare.

In sostanza si riottiene l'algoritmo iterativo efficiente, applicando le idee della programmazione dinamica ed ottimizzando l'uso dello spazio sulla base del grafo delle chiamate ricorsive.

8.1 Coefficienti binomiali

Definiamo $b(j, k)$ per $j, k \geq 0$ come il coefficiente binomiale $\binom{j+k}{k} \triangleq \frac{(j+k)!}{j! * k!}$. La definizione ricorsiva dei coefficienti binomiali puo' essere riscritta in termini di $b(j, k)$ come segue

- $b(0, k) = b(j, 0) = 1$ per $j, k \geq 0$
- $b(j, k) = b(j-1, k) + b(j, k-1)$ per $j, k > 0$

Quindi per calcolarci il valore $b(j, k)$ si dovranno calcolare i coefficienti per le coppie (x, y) nel rettangolo $[0..j] \times [0..k]$. Diamo un algoritmo bottom-up che riempie una matrice di interi positivi compatibilmente con la definizione ricorsiva di b, cioe' quando si riempie la posizione (x, y), con $x, y > 0$, devono essere gia' state riempite le posizioni $(x-1, y)$ e $(x, y-1)$.

```
var B: array [0..j,0..k] of integer
*si riempie B per righe crescenti e
*   all'interno della stessa riga per colonna crescente
for y <- 0 to k do B[0,y] <- 1
for x <- 1 to j do
  B[x,0] <- 1
  for y <- 1 to k do
    B[x,y] <- B[x-1,y] + B[x,y-1]
```

Si poteva procedere anche per colonne crescenti, e all'interno della stessa colonna per riga crescente.

8.2 Il problema dello zaino

Il *problema decisionale* dello zaino puo' essere formulato come segue:

- input: k; $k_1, \ldots, k_n$ (numeri naturali)
- output: esiste un sottoinsieme $J \subseteq \{1, \ldots, n\}$ t.c. $k = \sum_{j \in J} k_j$

Per risolverlo con le tecniche della programmazione dinamica occorre riformularlo in termini di una definizione ricorsiva. Definiamo il predicato $P(j, h)$ su $[0..n] \times [0..k]$ come $P(j, h)$ sse il problema dello zaino con input $(h; k_1, ..., k_j)$ e' risolubile. Quindi il problema che vogliamo risolvere equivale al calcolo di $P(n, k)$.
A questo punto possiamo dare una definizione ricorsiva di $P(j, h)$:

- $P(j, 0) = true$, per ogni $j \geq 0$
- $P(0, h) = false$, per ogni $h > 0$
- $P(j, h)$ sse $P(j-1, h) \vee (h \geq k_j \wedge P(j-1, h-k_j))$, per ogni $j, h > 0$

Diamo un algoritmo bottom-up che riempie una matrice di boolean compatibilmente con la definizione ricorsiva di P, quindi nel caso peggiore quando si riempie la posizione (j, h), con $j, h > 0$, devono essere gia' state riempite le posizioni $(j-1, h)$ e $(j-1, h-k_j)$.

```
var P: array [0..n,0..k] of boolean
*si riempie P per righe crescenti e
*   all'interno della stessa riga per colonna crescente
*   (ma si poteva procedere in qualsiasi ordine)
P[0,0] <- true
for y <- 1 to k do P[0,y] <- false
for x <- 1 to n do
  P[x,0] <- true
  for y <- 1 to k do
*si utilizza un if-then-else in modo da
*evitare chiamate ricorsive inutili
    if P[x-1,y] then P[x,y] <- true
    elsif y < k_x then P[x,y] <- false
    else P[x,y] <- P[x-1,y-k_x]
```

Con il precedente algoritmo si tende a fare del lavoro in piu' di quello strettamente necessario, p.e. per calcolare $P(n, k)$ non abbiamo bisogno di calcolarci $P(n, h)$ per $h < k$.
Un metodo generale per evitare del lavoro inutile, e' di utilizzare un algoritmo demand-driven, che ricorda l'implementazione ricorsiva della visita DFS, ovvero si usa un array per tenere traccia di quali sotto-problemi sono stati sinora risolti (in analogia con quello dei nodi sinora visitati), e se un sotto-problema non e' stato risolto, si effettuano le stesse chiamate di un algoritmo divide-et-impera.

```
var P: array [0..n,0..k] of boolean+{NIL}
{NIL indica che P[x,y] non e' stato calcolato}
*inizializzazione di P: nessun valore e' stato calcolato
for x <- 0 to n do
  for y <- 0 to k do
    P[x,y] <- NIL
P(n,k)
```

$P(x, y)$ e' una procedura ricorsiva che usa come variabili globali P ed i k_j. Tale procedura puo' essere definita nel modo seguente.

```
P(x,y) =
if P[x,y]=NIL then if y = 0 then P[x,y] <- true
elsif x = 0 then P[x,y] <- false
elsif P(x-1,y) then P[x,y] <- true
elsif y < k_x then P[x,y] <- false
else P[x,y] <- P(x-1,y-k_x)
return(P[x,y])
```

In questo modo si calcola solo i $P[x, y]$ strettamente necessari per il calcolare $P[n, k]$. Poiche' dobbiamo inizializzare comunque P, la complessita' tempo e' $O(n * k)$ come l'algoritmo bottom-up. Tuttavia, se si usa una diversa rappresentazione dei sottoinsiemi di $[0..n] \times [0..k]$ si potrebbe ottenere dei vantaggi a livello di complessita' tempo.

Lo stack per le chiamate ricorsive non puo' crescere piu' di $O(n)$, che in genere sara' trascurabile rispetto allo spazio occupato da P.

E' abbastanza facile modificare i precedenti algoritmi che risolvono il problema decisionale dello zaino per restituire anche la soluzione, cioe' risolvere il *problema di ricerca*

- input: k; $k_1, \ldots, k_n$ (numeri naturali)
- output: un sottoinsieme $J \subseteq \{1, \ldots, n\}$ t.c. $k = \sum_{j \in J} k_j$ (se J esiste)

Il grafo delle chiamate ricorsive resta invariato, tuttavia oltre all'informazione booleana $P[j, h]$ su l'esistenza o meno della soluzione si dovra' fornire anche un sottoinsieme soluzione $I[j, h]$ ($I[j, h]$ non e' significativo se $P[j, h]$ e' falso).

```
var P: array [0..n,0..k] of boolean
var I: array [0..n,0..k] of sottoinsiemi of 1..n
P[0,0] <- true; I[0,0] <- vuoto
for y <- 1 to k do P[0,y] <- false
for x <- 1 to n do
  P[x,0] <- true; I[x,0] <- vuoto
  for y <- 1 to k do
    if P[x-1,y] then P[x,y] <- true; I[x,y] <- I[x-1,y]
    elsif y < k_x then P[x,y] <- false
    elsif P[x-1,y-k_x] then P[x,y] <- true; I[x,y] <- I[x-1,y-k_x] + {x}
    else P[x,y] <- false
```

Una riduzione nella occupazione di memoria si puo' ottenere sostituendo la tabella I con una tabella Q di booleani t.c. $Q[j, h]$ sse $j > 0$ e $j \in I[j, h]$. Infatti e' possibile ricostruire I a partire da P e Q sfruttando la seguente definizione ricorsiva

- $I[j, h]$ indefinito, se $P[j, h] = false$
- $I[j, h] = \emptyset$, se $h = 0$
- $I[j, h] = I[j-1, h-k_j] \cup \{j\}$, se $Q[j, h] = true$ e $h \geq k_j$
- $I[j, h] = I[j-1, h]$, se $Q[j, h] = false$

Quindi date le tabelle P e Q possiamo calcolare $I[j, h]$ in $O(j)$.

8.3 Moltiplicazione ottimale di matrici

Il *problema del valore ottimale* di moltiplicazioni scalari per calcolare la moltiplicazione n matrici rettangolari e' formulato come segue

- input: $(m_0, \ldots, m_n)$ (numeri naturali)
- output: numero ottimale di moltiplicazioni scalari per moltiplicare n matrici $M_1 * \ldots * M_n$ con M_i di dimensione $m_{i-1} \times m_i$

Definiamo ricorsivamente la funzione $MO(i, j)$ con $0 \leq i < j \leq n$ che da' il numero ottimale di moltiplicazioni scalari per moltiplicare le matrici $M_i * \ldots * M_j$

- $MO(i, i+1) = 0$, se $0 \leq i < n$
- $M(i, j) = \min_{i<k<j}(MO(i, k) + MO(k, j) + m_i * m_k * m_j)$, se $0 < i + 1 < j \leq n$

Si ha un notevole riuso della soluzione ad un sotto-problema. Infatti per risolvere il problema che ci interessa, ovvero $MO(0, n)$ dobbiamo risolvere $O(n^2)$ sotto-problemi. L'approccio bottom-up parte dalla diagonale $\{(i, i+1) | 0 \leq i < n\}$ e procede per diagonali $\{(i, j) | j - i = k \wedge 0 \leq i, j \leq n\}$ con k crescente.

8.4 Altri esempi e raffinamenti

L'algoritmo di Floyd per calcolare la chiusura transitiva di una matrice e quello per trovare un cammino di lunghezza minima tra ogni coppia di nodi si possono vedere come esempi di programmazione dinamica, dove il grafo delle chiamate ricorsive puo' essere strutturato come un array tridimensionale.

Negli algoritmi di programmazione dinamica considerati sinora non ci siamo preoccupati di minimizzare il consumo di memoria a parita' di complessita' tempo. Una analisi piu' accurata della struttura del grafo delle chiamate ricorsive mostra che e' possibile ridurre drasticamente l'uso della memoria negli algoritmi bottom-up considerati:

- per calcolare i coefficienti binomiali basta mantenere una riga $B[x,\_]$ (oppure una colonna $B[\_,y]$) della tabella.
- per il problema dello zaino l'unica possibilita' e' mantenere una riga $P[x,\_]$ della tabella.
- per calcolare la chiusura transitiva basta mantenere una matrice $M'[k,\_,\_]$ ed effettuare gli aggiornamenti nel seguente ordine: prima $M'[x,y]$ con x,y diversi da v_k, poi $M'[x,v_k]$ e $M'[v_k,y]$ con x,y diversi da v_k, ed infine $M'[v_k,v_k]$

```
input:  M: array [V,V] of A
output: M^+: array [V,V] of A
var: M': array [V,V] of A
M' <- M
for k <- 1 to |V| do
  for x,y in V-{v_k} do
    M'[x,y] <-  M'[x,y]+(M'[x,v_k]*(M'[v_k,v_k]^*)*M'[v_k,y])
  for x in V-{v_k} do
    M'[x,v_k] <-  M'[x,v_k]*(M'[v_k,v_k]^*)
  for y in V-{v_k} do
    M'[v_k,y] <- (M'[v_k,v_k]^*)*M'[v_k,y]
  M'[v_k,v_k] <- M'[v_k,v_k]*(M'[v_k,v_k]^*)
M^+ <- M'
```

Per gli algoritmi demand-driven e' molto piu' difficile capire quando un elemento della tabella non serve piu', poiche' il modo di riempire la tabella procede in modo alquanto irregolare.