Strutture dati - Parte 3

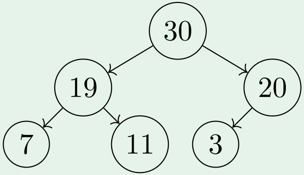
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

25 maggio 2022

Mucchi (Heaps)

Una struttura parzialmente ordinata

- Un *mucchio* (*heap*) è una struttura dati ad albero la chiave del nodo padre è sempre maggiore (max-heap) di quella dei figli
  - Nessuna relazione sussiste tra le chiavi di due fratelli
  - É possibile definirne una variante in cui la chiave del padre è sempre minore di quella dei figli (min-heap)
- Se l'albero è binario, parliamo di mucchi binari (binary heaps)
  - Manteniamo lo heap come un albero quasi-completo
- Esempio di max-heap:



Mucchi (Heaps)

Proprietà e usi pratici

- Gli heap, in particolare gli heap binari, trovano uso per:
  - Implementare code con priorità
  - Ordinare vettori (proposti originariamente per questo)
- Per tutti gli usi più comuni, è conveniente materializzare lo heap sempre come struttura dati implicita
  - É un albero binario quasi-completo $\rightarrow$ le foglie mancanti sono quelle che occupano la parte finale dell'array in cui è stoccato
  - Avremo un attributo $A.heapsize$ che indica il numero di elementi dello heap e $A.length$ che contiene la lunghezza dell'array di supporto: $A.heapsize \leq A.length$
- Le operazioni su un max-heap sono: $\textsc{Max}$, $\textsc{Inserisci}$, $\textsc{Cancella-Max}$, $\textsc{Costruisci-Max-Heap}$, $\textsc{Max-Heapify}$
- In un max-heap l'elemento con chiave più grande è la radice

Mucchi (Heaps)

Code con priorità

- Una coda con priorità è una struttura dati a coda in cui è possibile dare una priorità numerica agli elementi all'interno
- Elementi con priorità maggiore verranno estratti sempre prima di elementi con priorità minore indipendentemente dall'ordine di inserimento
- L'implementazione più comune di una coda con priorità è un max-heap
  - La priorità di un elemento è data dalla sua chiave
- Per implementare le primitive necessarie (MAX, INSERISCI, CANCELLA-MAX) necessitiamo di una procedura di supporto:MAX-HEAPIFY

Mucchi (Heaps)

Max-Heapify

- Max-Heapify(A, n) riceve un array e una posizione in esso: assume che i due sottoalberi con radice stoccata in Left$(n) = 2n$ e Right$(n) = 2n + 1$ siano dei max-heap [a]
- Modifica A in modo che l'albero radicato in n sia un max-heap
- Consente rendere un array A un max-heap come segue:

Costruisci-Max-Heap(A)

$A.heapsize \leftarrow A.length$
for $i \leftarrow \lfloor \frac{A.length}{2} \rfloor$ **downto** 1
    Max-Heapify(A, i)

[a] Convenzione: gli indici di A vanno da 1 a $A.length$

Mucchi (Heaps)

Max-Heapify

```
Max-Heapify(A, n)
  l ← LEFT(n)
  r ← RIGHT(n)
  if l ≤ A.heapsize and A[l] > A[n]
        posmax ← l
  else posmax ← n
  if r ≤ A.heapsize and A[r] > A[posmax]
        posmax ← r
  if posmax ≠ n
        Swap(A[n], A[posmax])
        Max-Heapify(A, posmax)
```

- La procedura causa la discesa del nuovo valore verso le foglie sino al punto in cui è maggiore dei figli
- Complessità: nel caso pessimo $\mathcal{O}(log(n))$ in un heap contenente n elementi

Code con priorità

$\textsc{Max}(A)$ (esamina l'elemento a priorità massima)

```
MAX(A)
  return A[1]
```

- L'ispezione è $\mathcal{O}(1)$

$\textsc{Cancella-Max}(A)$ (estrae l'elemento a massima priorità)

```
CANCELLA-MAX(A)
  if A.heapsize < 1
      return ⊥
  max ← A[1]
  A[1] ← A[A.heapsize]
  A.heapsize ← A.heapsize − 1
  MAX-HEAPIFY(A, 1)
  return max
```

- Estrarre l'elemento a priorità massima costa $\mathcal{O}(\log(n))$
- Mediamente, il costo è inferiore
- Più efficace di un vettore ordinato $\mathcal{O}(\log(n))$ contro $\mathcal{O}(n)$

Code con priorità

$\text{INSERISCI}(A, key)$ (accoda un nuovo elemento)

$\text{INSERISCI}(A, key)$

```
A.heapsize ← A.heapsize + 1
A[A.heapsize] ← key
i ← A.heapsize
while i > 1 and A[PARENT(i)] < A[i]
        SWAP(A[PARENT(i)], A[i])
        i ← PARENT(i)
```

- Inserisce l'elemento nuovo come ultima foglia
- Fa scalare l'elemento fin quando non è minore del padre
- Complessità nel caso pessimo: $\mathcal{O}(\log(n))$

Code con priorità

Riassunto delle complessità

- Una coda con priorità implementata con uno heap binario ha un costo, sia per l'accodamento che per l'estrazione, pari a $\mathcal{O}(\log(n))$
- Inserendo gli elementi uno alla volta, si ha un costo complessivo di $\mathcal{O}(n \log(n))$ per la costruzione dell'intera coda
- Apparentemente, il costo è identico a quello della $\textsc{Costruisci-Max-Heap}$
- É in realtà possibile dimostrare che $\textsc{Costruisci-Max-Heap}$ risulta essere in grado di costruire lo heap in $\mathcal{O}(n)$

Un limite più preciso

- Uno heap binario è sempre alto $\lfloor \log(n) \rfloor$, il numero di nodi con distanza dalle foglie di h archi è $\leq \frac{2^{\lfloor \log(n) \rfloor}}{2^h} \leq \frac{n}{2^h}$
- Calcolare $\textsc{Max-Heapify}$ per un nodo, richiede al più $\mathcal{O}(h)$ spostamenti verso il basso
- Calcoliamo il costo complessivo, sommando, per ogni livello, il costo di mucchificare ogni elemento di esso; otteniamo: $\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{n}{2^h} \mathcal{O}(h) = n\mathcal{O}(\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h})$, dove, ricordando che $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converge, abbiamo $\mathcal{O}(n)$
- É noto anche il risultato esatto del numero massimo di confronti: $2n - 2s_2(n) - e_2(n)$ dove $s_2(n)$ è il numero di cifre 1 nella rappresentazione binaria di n e $e_2(n)$ è l'esponente del fattore 2 nella fattorizzazione di n

Ordinamento di un vettore

Ordinare con i mucchi

- Ordinare un array in ordine crescente può essere fatto trovando il massimo tra i suoi elementi e posizionandolo alla fine, quindi ripetendo il procedimento sulla parte disordinata
  - Praticamente, è il $\textsc{SelectionSort}$
- Tuttavia, $\textsc{SelectionSort}$ è $\mathcal{O}(n^2)$ perchè trovare il massimo nella porzione disordinata costa $\mathcal{O}(n)$
- Cosa succede se rendiamo prima l'array un max-heap?

Ordinamento

HeapSort(A)

HeapSort(A)

Costruisci-Max-Heap(A)
for $i \leftarrow A.length$ **downto** 2
    Swap($A[1], A[i]$)
    $A.heapsize \leftarrow A.heapsize - 1$
    Max-Heapify($A, 1$)

- Riordino gli elementi di A in un max-heap
- Scambio il più grande con l'ultima foglia
- Decremento la dimensione dello heap e riordino l'elemento messo in testa

HeapSort

Considerazioni

- HEAPSORT ha complessità $\mathcal{O}(n \log(n))$ nel caso pessimo: è la migliore possibile
- Necessita solamente di $\mathcal{O}(1)$ in spazio ausiliario (ordina sul posto) a differenza di MERGESORT
- Nelle implementazioni pratiche, nel caso medio, è più lento di QUICKSORT: HEAPSORT ha un costo lineare sommato a $\mathcal{O}(n \log(n))$ che viene sempre pagato
- Resta il vantaggio della complessità di caso pessimo garantita
- Come la versione di MERGESORT out-of-place HEAPSORT non è stabile

Grafi

Una struttura dati molto flessibile

- La struttura dati più naturale per rappresentare un insieme di oggetti legati da una generica relazione tra di loro è il *grafo*
- La relazione tra oggetti è rappresentata da un insieme di coppie di oggetti (ordinate o meno)
- Esempio: Una mappa stradale: nodi $\rightarrow$ città, relazione $\rightarrow$ le città sono collegate da una strada

Definizione (Grafo)

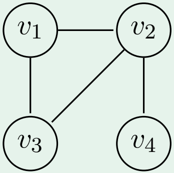
Un grafo è una coppia $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ con $\mathbf{V}$ un insieme di *nodi* (detti anche *vertici*) ed $\mathbf{E}$ un insieme di *archi* (detti anche *lati*).

Grafi

Nomenclatura

- Se un grafo ha $|\mathbf{V}|$ nodi, esso ha al più $|\mathbf{V}|^2$ archi
- Due nodi collegati da un arco si dicono *adiacenti*
- Un *cammino* tra due nodi v_1, v_2 è un insieme di archi di cui il primo ha origine in v_1, l'ultimo termina in v_2 e ogni nodo compare almeno una volta come destinazione di un arco che come sorgente

Esempio



- $\mathbf{V} = \{v_1, v_2, v_3, v_4\}$
- $\mathbf{E} = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4)$
  $(v_2, v_1), (v_3, v_1), (v_3, v_2), (v_4, v_2)\}$
- Cammino v_3–v_4: $(v_3, v_2), (v_2, v_4)$

Grafi

Nomenclatura - 2

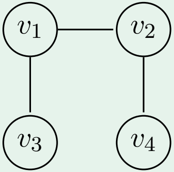
- Un grafo è detto *orientato* (*directed graph*) se la coppia di nodi che costituisce un arco è ordinata.
  - In altre parole, se ciò che collega i nodi ha un "verso"
  - I suoi archi sono detti archi orientati (*arcs*, non *edges*)
  - Esempio: un albero è un grafo orientato
- Un grafo orientato viene rappresentato indicando gli archi come frecce che puntano al secondo nodo della coppia
- In un grafo non orientato, l'insieme degli archi può essere rappresentato in modo compatto dato che se $(v_1, v_2) \in \mathbf{E}$, allora anche $(v_2, v_1) \in \mathbf{E}$

Grafi

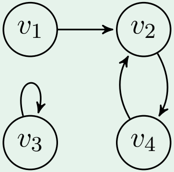
Nomenclatura - 2

- grafo *connesso*: esiste un percorso per coppia di nodi
- grafo *completo* (*completamente connesso*): esiste un arco per ogni coppia di nodi
- Un percorso è un *ciclo* se il nodo di inizio e di fine coincidono
  - Il ciclo è orientato se segue la direzione degli archi
- Un grafo privo di cicli è detto *aciclico*

Esempi



Grafo
non orientato
connesso
aciclico



Grafo
orientato
non connesso
ciclico

Grafi

Rappresentazione in memoria

- Sono possibili due strategie per rappresentare un grafo: liste di adiacenza e matrice di adiacenza
- Liste di adiacenza:
  - Un vettore di liste lungo $|\mathbf{V}|$, indicizzato dai nomi dei nodi
  - Ogni lista contiene i nodi adiacenti all'indice della sua testa
- Matrice di adiacenza:
  - Una matrice di valori booleani $|\mathbf{V}| \times |\mathbf{V}|$, con righe e colonne indicizzate dai nomi dei nodi
  - la cella alla riga i, colonna j contiene 1 se l'arco (v_i, v_j) è presente nel grafo (0 altrimenti)

Grafi

Rappresentazioni a confronto

- Complessità spaziale: Liste: $\Theta(|\mathbf{V}| + |\mathbf{E}|)$, Matrice $\Theta(|\mathbf{V}|^2)$
  - La rappresentazione a liste è più compatta se il grafo è sparso ovvero se il numero di archi è "basso": $|\mathbf{E}| \ll |\mathbf{V}|^2$
- Complessità temporale per determinare :
  - Se (v_1, v_2) appartiene a un grafo: Liste: $\mathcal{O}(|\mathbf{V}|)$, Matrice $\mathcal{O}(1)$
  - Il numero di archi o_e uscenti da un nodo: Lista: $\Theta(o_e)$, Matrice $\mathcal{O}(|\mathbf{V}|)$

Ottimizzazioni per grafi non orientati

- La matrice di adiacenza di un grafo non orientato è simmetrica rispetto alla diagonale principale: posso stoccarne solo metà
- Liste di adiacenza: posso stoccare solo uno dei due archi e raddoppiando il tempo di ricerca per un nodo adiacente

Grafi

Operazioni su grafi

- Le operazioni su grafi sono tipicamente di ispezione:
  - Visita in ampiezza
  - Visita in profondità
- ... o vanno a determinare proprietà del grafo:
  - Trovare le componenti connesse
  - Ordinamento topologico
  - Percorso più breve tra due nodi
  - Individuare cicli

Visita in ampiezza (Breadth First Search, BFS)

- La strategia di visita *in ampiezza* visita tutti i nodi di un grafo $\mathcal{G}$ a partire da uno nodo sorgente s
  - Ordine di visita: vengono visitati tutti i nodi con un cammino tra loro e s lungo n passi, prima di visitare quelli con un cammino lungo $n+1$
- La visita di un grafo è più problematica di quella di un albero: possono essere presenti cicli
  - Evitiamo di iterare all'infinito colorando i nodi mentre li visitiamo:
    - Nodo bianco: deve essere ancora visitato
    - Nodo grigio: il nodo è stato visitato, devono essere visitati quelli adiacenti ad esso
    - Nodo nero: sono stati visitati sia il nodo che quelli adiacenti

Visita in ampiezza – Schema

- Memorizziamo in una coda (semplice, senza priorità) i nodi ancora da visitare
- La coda è inizializzata con la sola sorgente
- Estraiamo un nodo dalla coda e:
  - Visitiamo i vicini bianchi
  - Li coloriamo di grigio (possiamo calcolare la loro distanza da s)
  - Li accodiamo affinchè siano visitati a loro volta
- Marchiamo quindi il nodo estratto come nero e riprendiamo estraendo il successivo

Visita in ampiezza

Pseudocodice - VisitaAmpiezza(G, s)

```
VisitaAmpiezza(G, s)
 1  for each n ∈ V \ {s}
 2      n.color ← white
 3      n.dist ← ∞
 4  s.color ← grey
 5  s.dist ← 0
 6  Q ← ∅
 7  Enqueue(Q, s)
 8  while ¬IsEmpty(Q)
 9      curr ← Dequeue(Q)
10      for each v ∈ curr.adiacenti
11          if v.color = white
12              v.color ← grey
13              v.dist ← curr.dist + 1
14              Enqueue(Q, v)
15      curr.color ← black
```

- Linee 1–7 : inizializzano tutti i nodi come bianchi
- Linee 8–15: effettuano la visita del grafo
- N.B. Ogni arco è visitato una sola volta (a partire dal nodo sorgente)
- Complessità totale: $\mathcal{O}(|\mathbf{V}| + |\mathbf{E}|)$

Grafi

Utilizzi

- Come per le visite degli alberi è possibile stampare i nodi in una visita di un grafo
- VisitaAmpiezza si trasforma in algoritmo di ricerca
  - Basta inserire un controllo appena si sta per accodare un nuovo elemento: se è quello corretto lo si ritorna

Visita in profondità

- Diversamente dalla visita in ampiezza, visitiamo prima i nodi adiacenti a quello dato, poi il nodo stesso
  - Segue i cammini "fino in fondo" prima di visitare i vicini del nodo di partenza
  - Il codice è identico a VisitaAmpiezza sostituendo la coda con una pila (condivide quindi anche le complessità)

Grafi

Componenti connesse

- É detta *componente connessa* di un grafo $\mathcal{G}$ un insieme $\mathbf{S}$ di nodi tali per cui esiste un cammino tra ogni coppia di essi, ma nessuno di essi è connesso a nodi $\notin \mathbf{S}$
- Individuare le componenti connesse in un grafo equivale ad etichettare i nodi con lo stesso valore se appartengono alla stessa componente
- Molto utile nella pratica:
    - Individua punti non serviti nella mappa di una città se rappresentata con un grafo
    - Se il grafo rappresenta la relazione di amicizia di un social network equivale a individuare le comunità

Componenti connesse

ComponentiConnesse(G) (etichette intere per le comp.)

ComponentiConnesse(G)

for each $v \in \mathbf{V}$
    $v.etichetta \leftarrow -1$
$eti \leftarrow 1$
for each $v \in \mathbf{V}$
    if $v.etichetta = -1$
        VisitaEdEtichetta(G, v, eti)
        $eti \leftarrow eti + 1$

- VisitaEdEtichetta funziona come VisitaAmpiezza o VisitaProfondità ma imposta a *eti* il campo *etichetta* del nodo visitato
- Complessità: $\mathcal{O}(|\mathbf{V}|)$, ogni nodo viene visitato una sola volta

Ordinare i nodi di un grafo

Definizione (Predecessore)

Dato un grafo orientato, il *predecessore* di un nodo v è un nodo u tale per cui esiste un cammino da u a v

- Percorrendo il grafo lungo gli archi, a partire da u *possiamo* raggiungere v
  - non lo raggiungiamo necessariamente, possono esserci biforcazioni

Ordinamento Topologico

- Un valore utile da calcolare per un grafo *orientato aciclico* è il cosidetto ordinamento topologico
- L'ordinamento topologico è una sequenza di nodi del grafo tale per cui nessun nodo compare prima di un suo predecessore
  - N.B. L'ordinamento topologico non è unico!

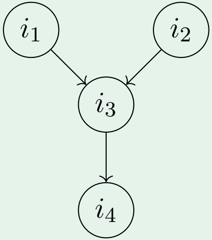
Esempio

Grafo delle dipendenze dati (DDG) $(\mathbf{V}, \mathbf{E})$

- $\mathbf{V}$: istruzioni $i_j, j \in \mathbb{N}$; $\mathbf{E}$: $(i_j, i_k) \in \mathbf{E}$ sse i_k usa il risultato prodotto da i_j

Pitagora(a, b)

$aq \leftarrow a^2$
$bq \leftarrow b^2$
$cq \leftarrow aq + bq$
return $\sqrt{cq}$



- Ordinamenti topologici validi: i_1, i_2, i_3, i_4 e i_2, i_1, i_3, i_4
- Sono ordini in cui si può eseguire le istruzioni, certi di avere già calcolato gli operandi

Calcolare l'ordinamento topologico

Osservazione

- Se un grafo non è connesso, le componenti connesse possono essere ordinate in qualunque modo l'una rispetto all'altra

Idea della procedura

- Per calcolare un ordinamento topologico è possibile riusare la procedura di VisitaProfondità
- Quando coloriamo un nodo di nero lo inseriamo in testa ad una lista

Ordinamento Topologico

OrdinamentoTopologico(G)

for each $v \in \mathbf{V}$
  $v.color \leftarrow white$
for each $v \in \mathbf{V}$
  if $v.color = white$
    VisitaProfOT(G, v, L)
return L

VisitaProfOT(G, s, L)

$s.color \leftarrow grey$
for each $v \in curr.adiacenti$
  if $v.color = white$
    VisitaProfOT(G, v, L)
$s.color \leftarrow black$
PushFront(L, s)

Il percorso più breve

Rivisitando l'algoritmo di Dijkstra

- Trova, dato un grafo orientato e un suo nodo s, i percorsi più brevi da un nodo a qualunque altro
  - Funziona sia su di un grafo classico, che su di un grafo *pesato* ovvero con archi dotati di un valore intero
- Principio di funzionamento
  - Inserisco ogni $v \in \mathbf{V} \setminus \{s\}$ in un insieme $\mathbf{Q}$ dopo aver impostato il suo attributo distanza a ∞ ed il $v.pred$ a NIL
  - Inserisco s in $\mathbf{Q}$ dopo aver impostato $s.dist \leftarrow 0$, $s.pred \leftarrow NIL$
  - Fin quando $\mathbf{Q}$ non è vuoto, estraggo il nodo c con $dist$ minima e controllo per ogni adiacente a se hanno distanza minore di $c.dist + peso(c, a)$
    - Se questo accade imposto $a.pred \leftarrow c, a.dist \leftarrow c.dist + peso(c, a)$

Il percorso più breve

Rivisitando l'algoritmo di Dijkstra

- La proposta originale di Dijkstra stocca l'insieme $\mathbf{Q}$ come un vettore
  - L'algoritmo effettua nel caso pessimo (grafo completamente connesso) $\mathcal{O}(|\mathbf{V}|)$ accessi ad ogni controllo per le distanze
  - Viene effettuato un controllo per ogni nodo del grafo $\rightarrow$ Comp. temporale $\mathcal{O}(|\mathbf{V}|^2)$
- Possiamo migliorare la complessità memorizzando l'insieme $\mathbf{Q}$ come una coda (min-heap) con priorità (la distanza)
  - Ogni estrazione di nodo a priorità minima costa $\mathcal{O}(1)$
  - Ogni cambio di priorità $\mathcal{O}(\log(|\mathbf{V}|))$

Dijkstra Ottimizzato

$\textsc{DijkstraQueue}(G, s)$

1 $Q \leftarrow \varnothing$
2 $s.dist \leftarrow 0;\,;\ \textsc{AccodaPri}(Q, s, 0)$
3 **for each** $v \in \mathbf{V}$
4   **if** $v \neq s$
5         $v.dist \leftarrow \infty$
6         $v.pred \leftarrow NIL$
7     $\textsc{AccodaPri}(Q, v, v.dist)$
8 **while** $Q \neq \varnothing$
9     $u \leftarrow \textsc{CancellaMin}(\mathrm{Q})$
10   **for each** $v \in u.succ$
11         $ndis \leftarrow u.dist + peso(u, v)$
12   **if** $v.dist > ndis$
13             $v.dist \leftarrow ndis$
14             $v.prev \leftarrow u$
15             $\textsc{DecrementaPri}(Q, v, ndis)$

- Le righe 3–7 inizializzano la coda: costo $\mathcal{O}(|\mathbf{V}|\log(|\mathbf{V}|)$
- Le righe 8–15 visitano ogni arco una volta (grafo come lista di adiacenze): Costo $\mathcal{O}(|\mathbf{E}|\log(|\mathbf{V}|))$
- Compl. totale $\mathcal{O}((|\mathbf{E}| + |\mathbf{V}|)\log(|\mathbf{V}|))$

Individuare cicli

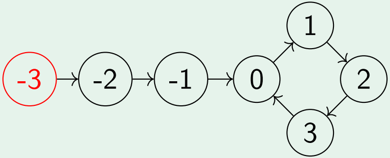
Un problema ricorrente

Dato un grafo orientato, per cui ogni nodo ha un solo successore determinare, dato un nodo di partenza, se il cammino che parte da esso ha cicli

- Utile anche nel caso in cui i successori siano molteplici ma ci sia una regola per sceglierne uno
  - Esempio: Il calcolo fatto da un FSA sta ciclando su un insieme finito di stati?
- Altrettanto utile se la relazione è una funzione matematica: il grafo può non essere materializzato in memoria
  - Esempio: Il calcolo fatto da una MT/programma sta ciclando su un insieme *finito* di configurazioni al posto di terminare?
  - Esempio 2: Una successione matematica dove $x_i = f(x_{i-1})$ si ripete periodicamente? (utile test per generatori di numeri casuali)

Algoritmo di Floyd

La lepre e la tartaruga - Idea



- Immaginiamo che il cammino su cui vogliamo individuare il ciclo sia una pista
- Usiamo due riferimenti t e l che spostiamo a ogni passo:
  - Nel caso di t, dal nodo a cui punta al successore (1 "passo")
  - Nel caso di l, dal nodo a cui punta al successore del successore (2 "passi")
- Entrambi partono dal nodo iniziale (-3 in figura)
- Se esiste un ciclo, essi sono destinati a "incontrarsi" (l doppierà prima o poi t)

Algoritmo di Floyd

La lepre e la tartaruga - Osservazioni e obiettivi

- Chiamiamo C la lunghezza del ciclo (4 nell'esempio) e T quella della "coda" che lo precede (3 nell'esempio)
- Quando t ha effettuato T passi, l si trova sicuramente nella porzione ciclica del grafo (l percorre strettamente più passi)
- Ad ogni mossa successiva l guadagna su t una posizione: la raggiunge sicuramente!
  - Sfruttiamo questo fatto per riconoscere l'esistenza di un ciclo
- Con un po'di analisi siamo in grado di ricavare anche quanto valgono T e C

Algoritmo di Floyd

La lepre e la tartaruga - Riconoscere l'esistenza del ciclo

- N.B. due posizioni $> T$ a distanza di C passi sono la stessa posizione
- Scriviamo per comodità T come $T = qC + r$
- Dopo T mosse l ha quindi effettuato $2T$ passi, di cui T nel ciclo
  - Osserviamo che la sua posizione è quindi $T = qC + r \equiv_C r$ nel ciclo
- Dopo altre $C - r$ mosse, t si trova $C - r$ posizioni all'interno del ciclo, l si trova in $r + 2(C - r) = 2C - r \equiv_C C - r$: si sono incontrate
- Il numero di mosse totali prima dell'incontro è quindi $T + C - r = (qC + r) + C - r = (q + 1)C$: si incontrano dopo un numero di mosse multiplo della lunghezza del ciclo

Algoritmo di Floyd

La lepre e la tartaruga - Calcolare T e C

- l e t hanno compiuto $(q+1)C$ passi
- Obiettivo: trovare T
  - Riporto t all'inizio, faccio muovere sia l che t, un passo alla volta contando i passi
  - dopo $T = qC + r$ passi, t si trova all'inizio del ciclo, l si trova a $(q+1)C + qC + r \equiv_C qC + r$: si incontrano!
- Per trovare C mi basta tenere ferma t e far procedere l di un passo alla volta, contando i passi, fin quando non si reincontrano

Riconoscimento di cicli

FloydLT(G, x)

```
FloydLT(G, x)
1  t ← x.succ
2  l ← x.succ.succ
3  while l ≠ t
4      t ← t.succ
5      l ← l.succ.succ
6  T ← 0
7  t ← x
8  while l ≠ t
9      t ← t.succ
10     l ← l.succ
11     T ← T + 1
12 l ← t
13 C ← 0
14 while l ≠ t
15     l ← l.succ
16     C ← C + 1
17 return T, C
```

- Le righe 1–5 trovano il ciclo quando l riprende t
- Le righe 6–11 calcolano T facendo ripartire t
- Le righe 14-16 calcolano C tenendo t ferma come segnaposto per l
- Complessità temporale: $\Theta(T + C - r + T + C) = \Theta(2(T + C) - r)$
- Complessità spaziale: $\Theta(1)$