Problem Set 3

docente: Luciano Gualà

Esercizio 1
Sia T un albero binario di n nodi con radice r in cui ogni nodo ha un valore non negativo associato. La *profondità* di un nodo v è il numero di archi del cammino da v alla radice. I nodi che si incontrano lungo tale cammino (v compreso) sono detti *antenati* di v. Diremo che un nodo v è *generazionalmente profondo* se la sua profondità è strettamente maggiore del valore di un suo antenato di valore minimo.

Si assuma che T è mantenuto attraverso una struttura collegata e che ogni nodo v abbia associato i seguenti campi: puntatori al padre e ai figli ($v.p$, $v.s$, $v.d$) e valore del nodo ($v.val$). Si progetti un algoritmo con complessità temporale $O(n)$ che, preso T, restituisca il numero di nodi generazionalmente profondi di T. Si fornisca lo pseudocodice dettagliato dell'algoritmo.

Soluzione esercizio 1
L'idea è quella di effettuare una visita dell'albero in modo tale che, quando si visita un generico nodo v, sia possibile capire se v è generazionalmente profondo e quindi conteggiarlo nel numero complessivo di nodi generazionalmente profondi. Come succede spesso quando si progettano algoritmi di visita, la cosa complicata è capire come raccogliere informazioni durante il processo. Qui per esempio occorre tenere traccia della profondità dei nodi, del valore dell'antenato di valore minimo, e trovare un modo per contare il numero di nodi generazionalmente profondi da restituire in output. Una soluzione è questa: diamo un algoritmo ricorsivo `GenProf`(v, h, m) che restituisce il numero di nodi generazionalmente profondi del sottoalbero radicato in v assumendo che la profondità di v sia h e che il valore dell'antenato di valore minimo (escluso v) sia m. La chiamata iniziale sarà `GenProf`$(r, 0, +\infty)$. Lo pseudocodice dell'algoritmo è il seguente:

Algorithm 1: `GenProf`(v, h, m)

```
if v = null then
    return 0
m ← min{m, v.val} ;
if h > m then
    return 1+GenProf(v.s, h + 1, m)+GenProf(v.d, h + 1, m)
else
    return GenProf(v.s, h + 1, m)+GenProf(v.d, h + 1, m)
```

L'algoritmo ha chiaramente complessità temporale $O(n)$: ogni nodo è visitato una sola volta e il tempo speso per esso è costante.

Esercizio 2 *(Aggiungendo un'operazione a una Pila)*
Progettare una struttura dati che implementa un tipo di dato *Pila* che mantiene una sequenza di elementi con chiave e che, oltre le classiche operazioni di `Top`, `Pop` e `Push`, consente

un'operazione aggiuntiva chiamata `Min`. Tale operazione restituisce il puntatore all'elemento di chiave minima contenuto nella pila. Tutte le operazioni devono avere complessità temporale $O(1)$ nel caso peggiore.

Soluzione esercizio 2
La struttura dati è una variazione della classica implementazione del tipo di dato *Pila* tramite una lista. Ogni ogni elemento e della lista conterrà, oltre ai campi relativi alla chiave ($e.key$), all'oggetto ($e.obj$) ed al puntatore all'elemento successivo ($e.next$) anche un puntatore all'oggetto con chiave minore tra quelli che lo seguono, oppure a se stesso ($e.min$). Una rappresentazione della struttura è mostrata in figura 1.

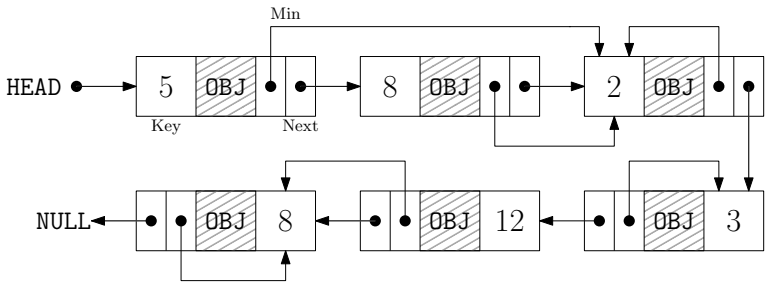

Figura 1: Rappresentazione della struttura dati che implementa il tipo "Pila con Minimo".

Se supponiamo di saper mantenere una simile struttura, l'implementazione della funzione `Min` diventa banale: è sufficiente restituire il puntatore contenuto nel primo elemento della pila. Mostriamo ora come adattare le altre operazioni per tener conto delle modifiche effettuate. Chiamiamo `HEAD` il puntatore al primo elemento della pila.

L'operazione `Push` prende in input un oggetto e la relativa chiave, crea un nuovo elemento e ed inizializza i relativi campi. Al puntatore $e.min$ viene assegnato l'indirizzo dell'oggetto con chiave minore tra quello appena inserito e quello dell'elemento puntato da `HEAD` (se esiste). Infine al puntatore $e.next$ viene assegnato l'indirizzo di `HEAD`, ed in `HEAD` viene memorizzato l'indirizzo di e.

Le operazioni `Pop` e `Top` rimangono sostanzialmente invariate.

Di seguito è riportata una possibile implementazione della struttura in un ipotetico linguaggio di alto livello simile al C.

Algorithm 2: Implementazione del tipo di dato "Pila com Minimo"

```
struct Element
    key Key
    obj Obj
    Element* Min
    Element* Next

Element* HEAD = NULL

procedure Push(key k, obj o)
    Element* e = new Element()
    e → Key = k
    e → Obj = o
    e → Min = e
    e → Next = HEAD
    if HEAD ≠ NULL ∧ HEAD → Min → Key < k then
        e → Min = HEAD → Min
    HEAD = e

procedure Pop()
    if Top ≠ NULL then
        e = HEAD → Next
        delete HEAD
        HEAD = e

funcion Top()
    if Top ≠ NULL then
        return HEAD → Obj
    return NULL

funcion Min()
    if Top ≠ NULL then
        return HEAD → Min → Obj
    return NULL
```

Esercizio 3 *(Un oracolo per il problema del* Minimum Range Query*)*
Sia A un vettore di n valori reali. Progettare un algoritmo che, dato A, costruisca un *oracolo* (ovvero una struttura dati) che sia in grado di rispondere in tempo $O(1)$ a *query* (ovvero domande) del seguente tipo: dati due interi i, j, calcolare l'indice dell'elemento di valore minimo nella porzione $A[i; j]$ del vettore.

Si noti che una soluzione semplice al problema è quella di precalcolare tutte le risposte alle $\Theta(n^2)$ query e memorizzarle in una matrice. In questa soluzione, però, l'oracolo (ovvero la matrice delle risposte) ha dimensione $\Theta(n^2)$. Vogliamo, invece, fare meglio in termini di memoria occupata dall'oracolo la cui dimensione richiediamo essere $O(n \log n)$. Non imponiamo invece nessun vincolo sulla complessità temporale necessaria per costruire l'oracolo.

Suggerimento: un'idea potrebbe essere quella di memorizzare solo le risposte a un sottoinsieme delle $\Theta(n^2)$ query. Tale sottoinsieme deve avere dimensione $O(n \log n)$ e deve comunque consentire di rispondere a una generica query in tempo costante.

Soluzione esercizio 3
La struttura è costituita da una collezione di vettori A_k con $0 \leq k \leq \lfloor \log n \rfloor$. Nell'$i$-esima cella del vettore A_k sarà memorizzato l'indice del minimo tra (al più) 2^k elementi del vettore A, in particolare tra quelli con indice compreso tra i ed $i+2^k-1$. Secondo questa definizione si ha $A_0[i] = i$. Si noti che $n \cdot \sum_{i=0}^{\lfloor \log n \rfloor} 2^i = O(n^2)$ è un upper bound al tempo richiesto per costruire tale struttura e che lo spazio complessivamente occupato dai vettori è $O(n \log n)$, come richiesto.

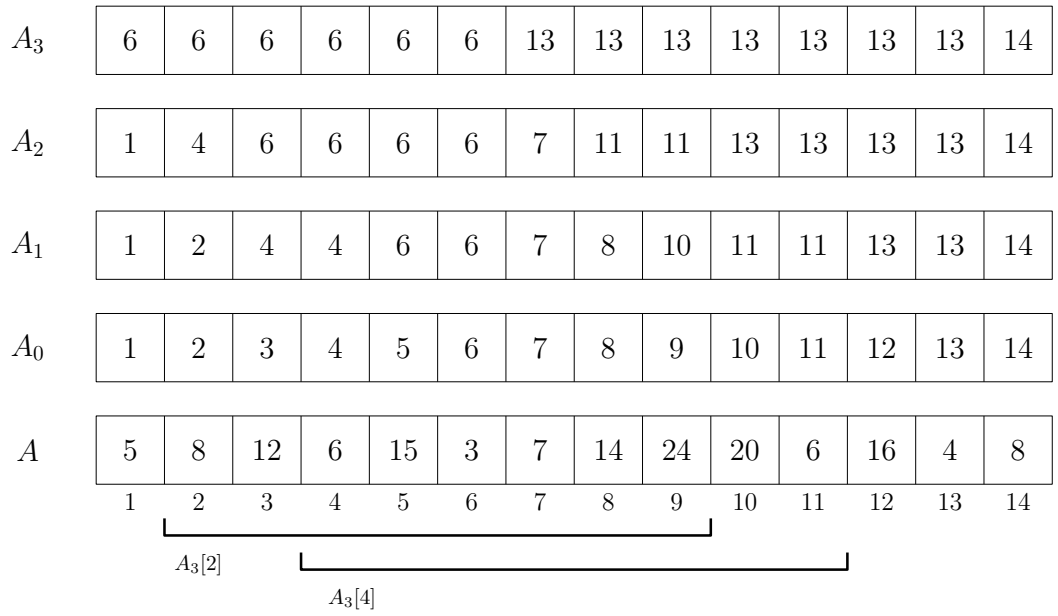

Figura 2: Oracolo per il problema del `Minimum Range Query`.

Dopo aver costruito tutti i vettori A_k è possibile rispondere in tempo costante alle query. Per conoscere il minimo del sottovettore $A[i; j]$ si consida il numero $\ell = j - i + 1$ di elementi di $A[i, j]$, e si procede nel seguente modo:

- Si determina l'indice k del vettore in cui sono stati precalcolati i minimi per gruppi di elementi di dimensione pari ad ℓ, se ℓ non è una potenza di 2 si considera il vettore di indice immediatamente inferiore. Vale: $k = \lfloor \log \ell \rfloor$.
- Si accede opportunamente a 2 elementi del vettore A_k. Ognuno dei due conterrà l'indice dell'elemento minimo in una porzione del vettore A. Gli elementi saranno scelti in modo tale che, complessivamente, vengano considerati tutti (e soli) gli elementi tra $A[i]$ ed $A[j]$. Ciò può essere fatto accedendo a: $A_k[i]$ e $A_k[j - 2^k + 1]$.

- Tra i due indici contenuti in $A_k[i]$ e $A_k[j - 2^k + 1]$ si seleziona quello che corrisponde all'elemento di valore minimo. Dal momento che l'operazione di minimo è associativa, l'indice ottenuto corrisponde proprio al minimo elemento in $A[i; j]$.

Ad esempio nel vettore di figura 2, l'elemento di indice minimo tra $A[2]$ ed $A[11]$ può essere trovato considerando $A_3[2]$ e $A_3[4]$. Il primo elemento contiene l'indice del valore minimo in $A[2; 9]$ mentre il secondo contiene quello del valore minimo in $A[4; 11]$. Il valore minimo risulta essere $A[6] = 3$.

Si noti che, per la scelta di k, non è mai possibile che i due intervalli considerati non abbiano intersezione: se così non fosse si sarebbe potuto scegliere un valore di k superiore.

Esercizio 4 *(Un algoritmo di programmazione dinamica per aiutare George Martin)*
La prossima stagione del Trono di Spade durerà n puntate e George R. R. Martin vuole aumentare la propria popolarità. Sa come fare: farà morire alcuni dei suoi personaggi. Sa che se fa morire un personaggio nella puntata i, guadagnerà $p_i \geq 0$ punti in popolarità. Se ne facesse morire due, sempre nella puntata i, l'effetto drammatico sarebbe più forte, ma non il *doppio* più forte; infatti guadagnerebbe $p_i + \frac{p_i}{2}$ punti in popolarità. In generale, se facesse morire j personaggi nella puntata i il suo guadagno in punti di popolarità sarebbe $\sum_{t=1}^{j} \frac{p_i}{2^{t-1}}$. Come a dire, l'effetto drammatico della violenza decresce esponenzialmente all'aumentare dei personaggi che muoiono. Ci si abitua a tutto, del resto. Inoltre Martin ha un altro problema: sa che il pubblico, ormai affezionato al suo sadismo, smetterebbe di guardare la serie se ci fossero più di due puntate consecutive senza un morto e, come se non bastasse, questa non sarà l'ultima stagione della serie TV, quindi è bene che qualche personaggio a cui il pubblico è affezionato resti in vita. Ancora per un po', almeno. Martin, a tal proposito, ha stimato che, nell'economia della narrazione, può sacrificare fino a k personaggi. Ma quando? Aiutatelo progettando un algoritmo di programmazione dinamica che calcoli la strategia che gli faccia guadagnare il più possibile in popolarità.

Soluzione esercizio 4
Come è tipico, nel progettare un algoritmo di programmazione dinamica la difficoltà principale risiede nel trovare un opportuno insieme di sottoproblemi, e nel cercare questi sottoproblemi è conveniente farsi guidare dalla struttura della soluzione cercata. Nello specifico, se si considera una strategia ottima per Martin, possiamo chiederci: muore qualcuno nell'ultima puntata? Se sì, quanti personaggi perdono la vita? E visto il vincolo sul numero massimo di puntate consecutive che possono esseci senza morti, quando è stata l'ultima volta che è morto qualcuno? Un possibile insieme di sottoproblemi che tiene conto di queste cose è il seguente.

Definiamo $\texttt{Opt}(i, h)$ come la massima popolarità ottenibile (in termini di punti di popolarità) da Martin nelle prime i puntate con il vincolo che nella puntata i muore qualcuno, e sapendo che Martin ha a disposizione (fino ad) h personaggi da sacrificare.

Ora, visto che in almeno una delle ultime puntate qualcuno deve morire, possiamo "indovinare" l'ultima puntata della soluzione ottima in cui questo avviene. E quindi la soluzione cercata sarà $\max\{\texttt{Opt}(n, k), \texttt{Opt}(n - 1, k), \texttt{Opt}(n - 2, k)\}$.

Si noti che il numero di sottoproblemi è $O(nk)$. La soluzione del generico sottoproblema è riportata di seguito. Il max più esterno (quello su j) serve per "indovinare" il numero di personaggi che nella soluzione ottima del sottoproblema muore nella puntata i.

$$\texttt{Opt}(i,h) = \max_{j=1,\ldots,h} \left\{ \sum_{t=1}^{j} \frac{p_i}{2^{t-1}} + \max\{\texttt{Opt}(i-1,h-j), \texttt{Opt}(i-2,h-j), \texttt{Opt}(i-3,h-j)\} \right\}.$$

I casi base (da considerare in ordine) sono: $\texttt{Opt}(i,h) = 0$ se $i \leq 0$ e $\texttt{Opt}(i,0) = -\infty$ se $i > 0$.

A questo punto l'algoritmo che calcola tutti i sottoproblemi e li memorizza in una matrice è semplice ed è lasciato come esercizio allo studente. Che complessità ha l'algoritmo?