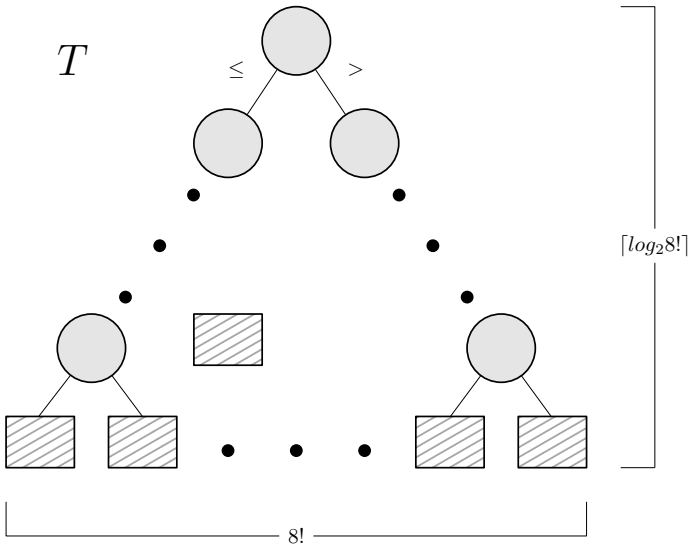
Problem Set 3

docente: Luciano Gualà

Esercizio 1 *(una domanda semplice ma non troppo)*
Dire se può esistere un algoritmo di ordinamento basato su confronti che ordina un insieme di 8 elementi facendo nel caso peggiore al più 15 confronti. Motivare la risposta.

Soluzione esercizio 1
Consideriamo un **qualsiasi** algoritmo di ordinamento A basato su confronti. Tale algoritmo può essere rappresentato tramite un opportuno albero binario di decisione T in cui ogni nodo interno rappresenta un confronto tra due elementi ed ogni foglia rappresenta un possibile output di A, cioè una particolare sequenza ordinata degli elementi in input.

Dal momento che l'algoritmo A deve ordinare correttamente qualsiasi insieme di 8 elementi (forniti in qualsiasi ordine), l'albero T deve contenere almeno tante foglie quante le possibili permutazioni di 8 elementi. Dunque il numero di foglie di T è almeno 8!.



È noto che per un qualsiasi albero binario T', l'altezza $h(T')$ è almeno logaritimica nel numero di foglie $\ell(T')$, più precisamente vale: $h(T') \geq \log_2 \ell(T')$.

Applicando la precedente proprietà all'albero T si ottiene: $h(T) \geq \log_2 8! = \log_2 40320 > 15.299$ per cui l'altezza di T deve essere almeno pari a 16. Ciò significa che esiste un cammino, dalla radice ad una foglia di T che attraversa almeno 16 nodi interni (radice inclusa), dunque l'algoritmo A (**qualunque esso sia**) deve eseguire, nel caso peggiore, più di 15 confronti.

Esercizio 2
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 2
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 3
Nella Striscia Di Mezzo si combatte una guerra sanguinaria fra le Forze Del Bene e le Forze Del Male. La Striscia si estende da ovest a est ed è composta da n territori, numerati (da ovest a est) da 1 a n. Il territorio i-esimo confina con il territorio $(i-1)$-esimo e $(i+1)$-esimo, $i = 2, 3, \ldots, n-1$. Il territorio 1 confina solo con il territorio 2 ed il territorio n confina solo con il territorio $n-1$. Ogni territorio è controllato da uno dei due schieramenti e, per ogni territorio controllato dalle Forze Del Bene, sono noti il numero di armate che

difendono tale territorio. Una possibile situazione nella Striscia Di Mezzo è rappresentata nella seguente figura, dove i territori controllati dalle Forze Del Male sono rappresentati da teschi, mentre per gli altri territori (controllati dalle Forze Del Bene) è indicato il numero di armate presenti sul territorio:

Se le Forze Del Male di un territorio i vogliono conquistare un territorio j controllato dalle Forze Del Bene, devono farsi strada sconfiggendo tutte le armate delle Forze del Bene che si trovano nei territori tra i e j (j compreso). Per ogni territorio j controllato dalle Forze Del Bene, definiamo il *fattore di difesa* δ_j come il numero minimo di armate che le Forze Del Male devono sconfiggere per conquistare il territorio j. Ad esempio, il territorio con indice 1 in figura ha fattore di difesa $\delta_1 = 5$ mentre quelli con indici 8 e 9 hanno fattori di difesa $\delta_8 = 9$ e $\delta_9 = 7$.

Si progetti un algoritmo che, preso in input in intero $k \in \{1, 2, \ldots, n\}$, calcola in tempo $O(n + k \log n)$ i k territori meglio difesi dalle Forze del Bene (cioè i k territori con i più alti fattori di difesa).

Suggerimento: è possibile calcolare tutti i fattori di difesa δ_j dei territori controllati dalle Forze Del Bene in tempo $O(n)$.

Soluzione esercizio 3

Come suggerito, procederemo nel seguente modo: calcoleremo tutti i fattori di difesa in tempo $O(n)$ e li metteremo in un vettore $\delta[1 : n]$, e poi estrarremo i k massimi da δ (corrispondenti ai territori meglio difesi) in tempo $O(n+k \log n)$. Per calcolare tutti i fattori di difesa ci avvarremo di due vettori ausiliari di $n + 1$ elementi $\delta_O[0 : n]$ e $\delta_E[1 : n + 1]$. Intuitivamente $\delta_O[i]$ e $\delta_E[i]$ contengono il numero minimo di armate che le forze del male devono distruggere per conquistare il territorio i quando vengono rispettivamente da ovest o da est. Per calcolare in tempo $O(n)$ tutti i $\delta[i]$ possiamo procedere nel seguente modo:

Algorithm 2: `CalcoloFattoriDiDifesa`(T)

$\delta_O[0] = +\infty$;
for $i = 1, \ldots, n$ **do**
  if $T[i]$ *non contiene un teschio* **then**
    $\delta_O[i] = \delta_O[i-1] + T[i]$
  else
    $\delta_O[i] = 0$
$\delta_E[n+1] = +\infty$;
for $i = n, n-1, \ldots, 2, 1$ **do**
  if $T[i]$ *non contiene un teschio* **then**
    $\delta_E[i] = \delta_E[i+1] + T[i]$
  else
    $\delta_E[i] = 0$
for $i = 1, \ldots, n$ **do**
  $\delta[i] = \min\{\delta_O[i], \delta_E[i]\}$
return δ

A questo punto i k territori meglio difesi possono essere ottenuti estraendo i k valori più grandi da δ. Usando un max-heap è possibile effettuare queste estrazioni in un tempo complessivo di $O(k \log n)$, mentre la costruzione dell'heap richiede tempo $O(n)$.

Algorithm 3: `TerritoriMeglioDifesi`(δ, k)

$H \leftarrow$ Array con n coppie
for $i \leftarrow 1$ **to** n **do**
  $H[i] \leftarrow (i, \delta[i])$

`// Costruisci un Max-Heap rispetto ai valori` $\delta[i]$`.`
`Heapify(`H`)` `//` $O(n)$

for $i \leftarrow 1$ **to** k **do**
  yield `Max(`H`)` `//` $O(1)$
  `DeleteMax(`H`)` `//` $O(\log n)$

Esercizio 4
Sia A una matrice $n \times n$ di numeri non negativi. Progettare un algoritmo che, data A, costruisca un *oracolo* (ovvero una struttura dati) che sia in grado di rispondere a *query* (ovvero domande) del seguente tipo:

- `query1`(i, j, b, h): dati quattro interi i, j, b, h, restituire la somma degli elementi della sottomatrice di A delimitata dai quattro elementi $A[i, j], A[i, j + b - 1], A[i + h - 1, j], A[i + h - 1, j + b - 1]$.

- `query2`(x): dato un valore x, restituire, se esistono, una coppia di indici (i, j) tale che la somma degli elementi della sottomatrice delimitata dai quattro elementi $A[1, 1], A[1, j], A[i, 1], A[i, j]$ è uguale a x.

L'oracolo deve avere dimensione $O(n^2)$ e l'algoritmo che lo costruisce deve avere complessità temporale lineare (nella dimensione dell'istanza), ovvero $O(n^2)$. Per quanto riguarda la complessità delle query, `query1` e `query2` devono poter essere risposte rispettivamente in tempo costante e tempo $O(n)$.

Un'ulteriore domanda: sempre tenendo la dimensione dell'oracolo a $O(n^2)$ e il tempo di risposta costante a `query1`, concedendosi tempo di costruzione $O(n^2 \log n)$, è possibile portare il tempo richiesto per la `query2` da $O(n)$ a $O(\log n)$?

Soluzione esercizio 4

Consideriamo la matrice A come indicizzata a partire da 1, in modo che valga $1 \leq i, j \leq n$. L'algoritmo costruisce come oracolo una matrice B di dimensione $(n+1)\times(n+1)$, indicizzata a partire da 0. Tale matrice sarà riempita dinamicamente in modo che la cella $B[i, j]$ con $1 \leq i, j \leq n$ contenga la somma degli elementi della sottomatrice di A delimitata da $A[1, 1], A[1, j], A[i, 1], A[i, j]$. Gli elementi $B[i, j]$ per cui $i = 0$ o $j = 0$ a corrispondono intuitivamente a sottomatrici "vuote" e vengono inizializzati a 0.

A partire dalle somme contenute nella matrice B faremo vedere come sarà possibile rispondere alle query 1 e 2 in tempo rispettivamente $O(1)$ e $O(n)$.

Concentriamoci prima sulla costruizione dell'oracolo. Per riempire B in tempo $O(n^2)$ consideriamo i suoi elementi in ordine di riga e notiamo che ognuno di essi può essere calcolato con un numero di operazioni costante, facendo ricorso ad alcuni degli elementi di B già esaminati. In particolare se $i > 0$ e $j > 0$ vale la seguente uguaglianza:

$$B[i, j] = B[i, j-1] + B[i-1, j] + A[i, j] - B[i-1, j-1] \tag{1}$$

In cui la differenza è necessaria dal momento che nella somma $B[i, j-1]+B[i-1, j]$, gli elementi della matrice A fino alla riga $i-1$ e fino alla colonna $j-2$ compaiono due volte. Una rappresentazione grafica dell'equazione (1) è mostrata in figura 1.

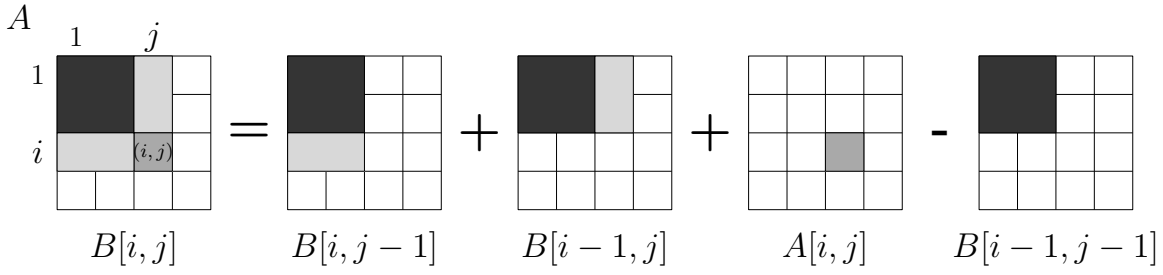


Figura 1: Rappresentazione grafica dell'equazione (1). Le aree colorate della matrice A identificano gli elementi la cui somma è contenuta nell'elemento indicato della matrice B.

Algorithm 4: `CostruzioneOracolo`(A, n)

$B \leftarrow$ Matrice di $(n+1) \times (n+1)$ elementi interi, indicizzata da 0.
`// Inizializza la prima riga e la prima colonna`
for $k \leftarrow 0$ **to** n **do**
  $B[k, 0] \leftarrow 0$
  $B[0, k] \leftarrow 0$

`// Calcola i rimanenti elementi utilizzando l'equazione` (1).
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $B[i, j] = B[i, j-1] + B[i-1, j] + A[i, j] - B[i-1, j-1]$
return B

Concentriamoci ora sugli algoritmi che rispondono alle query. Per quanto riguarda la prima, dato l'oracolo (la matrice) B, è possibile rispondere in tempo costante alle query della forma `query1`(i, j, b, h) osservando che vale la seguente equazione:

$$\begin{aligned} \texttt{query1}(i, j, b, h) &= \sum_{x=i}^{i+h-1} \sum_{y=j}^{j+b-1} A[x, y] = \\ &= B[i+h-1, j+b-1] - B[i+h-1, j-1] - B[i-1, j+b-1] + B[i-1, j-1] \end{aligned} \tag{2}$$

Dunque per calcolare la somma degli elementi della sottomatrice A richiesta sono sufficienti 4 accessi alla matrice B. Una rappresentazione grafica dell'equazione (2) è mostrata in figura 2.

Algorithm 5: `Query1`(i, j, b, h, B)

`// Calcola la somma della sottomatrice utilizzando l'equazione` (2).
return $B[i+h-1, j+b-1] - B[i+h-1, j-1] - B[i-1, j+b-1] + B[i-1, j-1]$

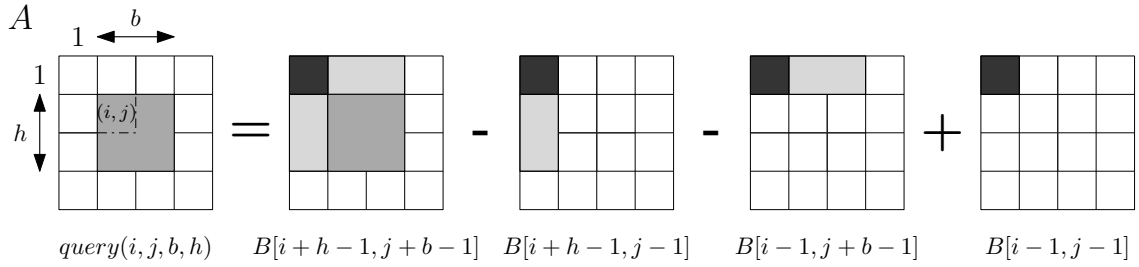


Figura 2: Rappresentazione grafica dell'equazione (2). Le aree colorate della matrice A identificano gli elementi la cui somma è contenuta nell'elemento indicato della matrice B.

Per quanto riguarda la seconda query, l'osservazione cruciale è che la matrice B ha una forma particolare: gli elementi sono ordinati in modo crescente sia per riga che per colonna. Questo facilita il compito di rispondere alla query che di fatto chiede di dire se esiste una cella della matrice B che contiene il valore x.

L'idea dell'algoritmo è la seguente. Parto dalla cella della matrice B in alto a destra, quella di indici $(1, n)$. Se contiene il valore x l'ho trovato, altrimenti, se $B[1, n] < x$ so con certezza che nella prima riga x non potrà essere presente (perché $B[1, n]$ è l'elemento più grande della prima riga), escludo quindi la prima riga dalla mia ricerca e mi sposto sulla cella $(2, n)$ (è come se avessi cancellato la prima riga). Se invece $B[1, n] > x$ so con certezza che x non può essere presente nell'ultima colonna (perché $B[1, n]$ è l'elemento più piccolo dell'ultima colonna), quindi escludo l'ultima colonna e mi sposto sulla cella $(1, n-1)$ (è come se stessi cancellando l'ultima colonna). In ogni caso ho in un certo senso "eliminato" una riga o una colonna e mi sono spostato nella cella in alto a destra della matrice rimanente. Ripeto il ragionamento a partire dalla nuova matrice.

Lo pseudocodice dell'algoritmo che risponde alla seconda query è riportato di seguito. L'algoritmo restituisce una coppia (i, j) tale che $B[i, j] = x$, $(-1, -1)$ se tale coppia non esiste. La complessità dell'algoritmo è chiaramente $O(n)$, perché il ciclo interno ad ogni passo o incrementa i o decrementa j. La dimostrazione formale di correttezza invece è lasciata per esercizio allo studente.

Algorithm 6: `Query2`(B, x)

```
i ← 1; j ← n
while i ≤ n e j ≥ 1 do
    if B[i, j] = x then
        return (i, j)
    if B[i, j] < x then
        i ← i + 1
    else
        j ← j − 1
// x non è presente in B
return (−1, −1)
```

Ultima osservazione sulla *domanda ulteriore*: è possibile rispondere alla `query2` in tempo $O(\log n)$ in tanti modi. Uno è questo. Si costruisce un albero AVL i cui elementi (nodi) sono tutte le $\Theta(n^2)$ celle di B, la cella (i, j) ha chiave $B[i, j]$. Rispondere alla `query2` vuol dire cercare l'elemento con chiave x nell'albero, cosa che costa $O(\log n^2) = O(\log n)$. Costruire tutto l'abero invece ha costo $O(n^2 \log n)$, dato che bisogna fare n^2 inserimenti ognuno di costo $O(\log n)$.