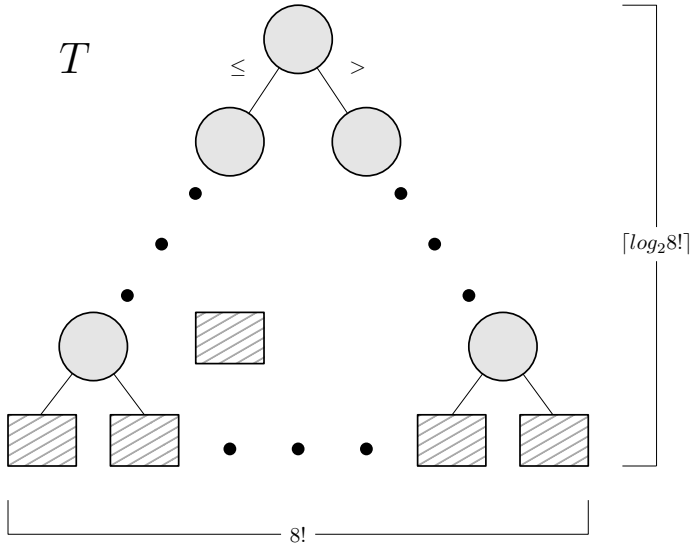
Problem Set 3

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
Siano dati n punti disposti sul piano Euclideo, dove il punto p_i ha coordinate (x_i, y_i), $i = 1, \ldots, n$. Si progetti un algoritmo che, preso in input l'insieme degli n punti, un valore $k \in \{1, 2, \ldots, n\}$ e un ulteriore punto *target* t di coordinate (x_t, y_t), restituisca i k punti dell'insieme che sono più vicini a p_t (rispetto alla distanza euclidea). L'algoritmo deve avere complessità temporale (nel caso peggiore) $O(n + k \log n)$.

Soluzione esercizio 2
Notiamo che è possibile calcolare la distanza di ogni punto p_i dal punto target t in un numero costante di operazioni, tramite la ben nota formula:

$$d(p_i, t) = \sqrt{(x_i - x_t)^2 + (y_i - y_t)^2}$$

L'algoritmo inizializza un array D di dimensione n, in cui ogni elemento è associato ad un punto. In particolare nella cella i sarà contenuta la coppia (p_i, d_i) in cui $d_i = d(p_i, t)$ è la distanza del punto p_i dal punto target t.

Tramite la procedura `Heapify`, si riordinano gli elementi di D in modo da formare un Min-Heap rispetto al secondo elemento di ogni coppia. La costruzione dell'heap richiede tempo $O(n)$.

A questo punto i k punti più vicini a t possono essere ottenuti tramite k estrazioni del minimo da D, ognuna delle quali richiede tempo $O(\log n)$.

Algorithm 1: `FindKNearestNeighbors`$(p_1, \ldots, p_n)$

```
D ← Array con n elementi del tipo (punto, distanza)
for i ← 1 to n do
    D[i] ← sqrt((x_i − x_t)^2 + (y_i − y_t)^2)

// Costruisci un Min-Heap rispetto alle distanze.
Heapify(D)                                          // O(n)

for i ← 1 to k do
    yield Min(D)                                    // O(1)
    DeleteMin(D)                                    // O(log n)
```

Esercizio 3
Sia A una matrice $n \times n$ di interi. Progettare un algoritmo che, data 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 quattro interi i, j, b, h, quale è la somma degli elementi della sottomatrice di A delimitata dai quattro elementi $A[i, j], A[i, j + b - 1], A[i + h - 1, j], A[i + h - 1, j + b - 1]$?

L'algoritmo di costruzione dell'oracolo deve avere complessità temporale $O(n^2)$, mentre l'algoritmo di interrogazione dell'oracolo, come già detto, deve avere complessità temporale $O(1)$.

Soluzione esercizio 3
Consideriamo la matrice A come indicizzata a partire da 1, in modo che valga $1 \leq i, j \leq n$. L'algoritmo alloca 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 sarà possibile rispondere alle query in tempo $O(1)$.

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.

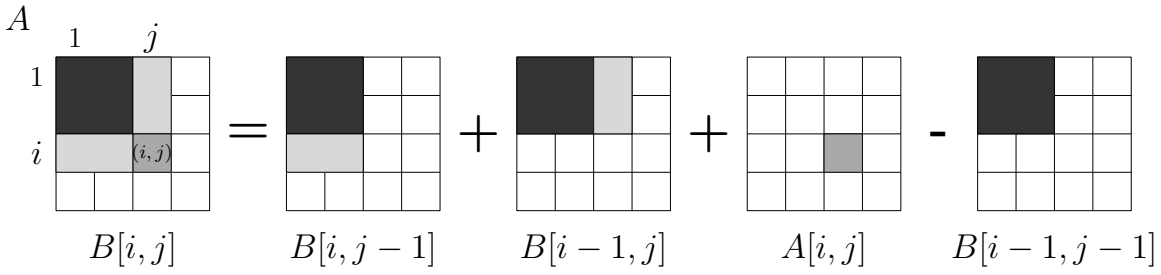


Figura 1: Rappresentazione grafica dell'equazione (1). Le aree colorate della matrice A identificano gli elementi la cui somma è contenuta nell'elemento indicato della matrice B.

Algorithm 2: `Preprocess`(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**
    $B[i, j] = B[i, j-1] + B[i-1, j] + A[i, j] - B[i-1, j-1]$

A questo punto è possibile rispondere velocemente alle query della forma $Query(i, j, b, h)$ osservando che vale la seguente equazione:

$$Query(i, j, b, h) = \sum_{x=i}^{i+h-1} \sum_{y=j}^{j+b-1} A[x, y] = \\ = B[i+h-1, j+b-1] - B[i+h-1, j-1] - B[i-1, j+b-1] + B[i-1, j-1] \tag{2}$$

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 3: `Query`(i, j, b, h)

Sia B la matrice costruita durante la fase di preprocessing.

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

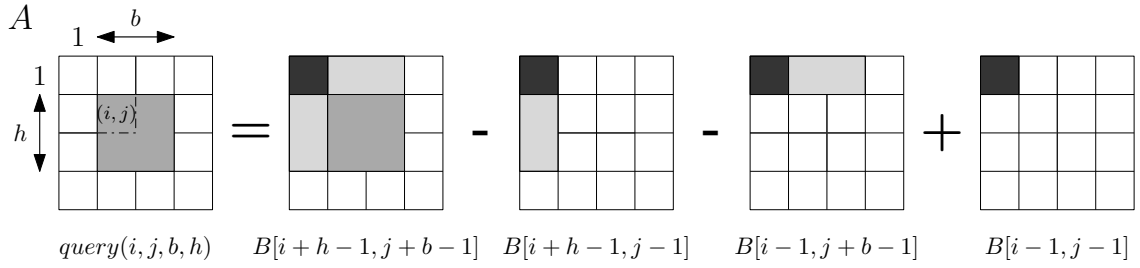


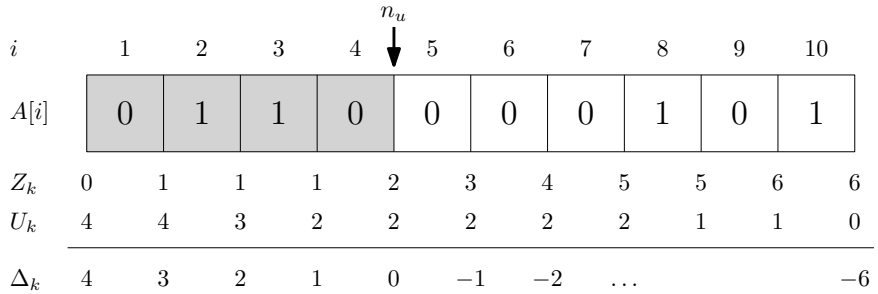
Figura 2: Rappresentazione grafica dell'equazione (2). Le aree colorate della matrice A identificano gli elementi la cui somma è contenuta nell'elemento indicato della matrice B.

Esercizio 4 *(tagliare un vettore in streaming)*
Vogliamo studiare il seguente problema: dato un array $A[1;n]$ di n bit, ovvero dove $A[i] \in \{0,1\}$, per ogni i, vogliamo trovare un punto dove tagliare l'array in modo che la metà di sinistra contenga un numero di zeri uguale al numero di uni nella metà di destra. Più formalmente, vogliamo trovare un indice k (ammesso che esista), tale che la porzione $A[1;k]$ contenga un numero di zeri pari al numero di uni nella porzione $A[k+1,n]$. Permettiamo anche i due tagli degeneri che corrispondono a $k=0$ e a $k=n$, in tal caso assumiamo che $A[1;0]$ e $A[n+1;n]$ contengono 0 zeri e 0 uni.

Vogliamo progettare un algoritmo con complessità temporale $O(n)$ che risolve il problema in uno scenario in cui il vettore ha una dimensione tale da non poter essere mantenuto in memoria interamente. In particolare, assumiamo che possiamo usare solo una quantità di memoria costante (rispetto a n) e che l'unico modo per leggere il vettore è chiedere che ci vengano snocciolati uno alla volta, da sinistra a destra, i suoi elementi. Un algoritmo così fatto è spesso chiamato algoritmo in *streaming*, perché può leggere l'input una sola volta come se fosse un flusso di dati. Riassumendo, quindi, vogliamo progettare un algoritmo con complessità temporale $O(n)$, memoria ausiliaria $O(1)$, e che può leggere una sola volta il vettore A da sinistra a destra. Si argomenti in modo preciso la correttezza dell'algoritmo.

Soluzione esercizio 4
Sia n_u il numero di elementi pari ad 1 in A. Consideriamo gli indici $0 \le k \le n$ per cui è possibile suddividere l'array e definiamo le quantità Z_k ed U_k pari, rispettivamente, al numero di 0 contenuti nel vettore $A[1;k]$ ed al numero di 1 contenuti nel vettore $A[k+1;n]$.

La quantità $\Delta_k = U_k - Z_k$ rappresenta il numero di 1 in eccesso nella porzione destra del vettore, rispetto al numero di 0 nella porzione sinistra. È facile convincersi che $\Delta_0 = n_u - 0 = n_u$.



Esaminiamo le possibili suddivisioni in maniera incrementale e notiamo che, per $k > 0$, vale $\Delta_{k+1} = \Delta_k - 1$ da cui segue $\Delta_k = n_u - k$. Per convincersene è sufficiente considerare il valore dell'elemento $A[k+1]$ che "si sposta" dal sottovettore destro a quello sinistro quando si passa dalla suddivione in posizione k a quella in posizione $k+1$:

- Se $A[k+1] = 0$ allora il numero di 0 nel sottovettore sinistro aumenta di 1 rispetto a quello della suddivisione in posizione k, mentre il numero di 1 nel sottovettore destro rimane invariato.

- Se $A[k+1] = 0$ allora il numero di 0 nel sottovettore sinistro rimane inviariato mentre il numero di 1 nel sottovettore destro diminuisce di 1 rispetto a quello della suddivisione in posizione k.

Dalle considerazioni precedenti segue scegliendo $k = n_u$ si avrà $\Delta_{n_u} = n_u - n_u = 0$, il che significa i numeri di 0 e di 1 nei vettori $A[1; n_u]$ e $A[n_u + 1; n]$ coincidono.

In definitiva è sufficiente analizzare il vettore una sola volta, da sinistra a destra, e contare il numero di elementi pari ad 1. Il valore ottenuto sarà proprio la soluzione cercata.