Studenti:
Martedì Gaetano (M63001226)
Salzillo Biagio (M63001227)

Corso di Algoritmi e Strutture Dati
Homework #2

- **Problema 2.1**

  Si consideri il seguente problema di minimo percorso vincolato. Sia data una matrice rettangolare di 1 e 0, in cui 1 indica una cella che è possibile percorrere e 0 una cella che non è possibile percorrere. Inoltre, si assuma che non è possibile percorrere neanche le quattro celle adiacenti (sinistra, destra, sopra e sotto) ad una cella che contiene uno 0. Calcolare la lunghezza del più breve percorso possibile da qualsiasi cella nella prima colonna a qualsiasi cella nell'ultima colonna della matrice. L'obiettivo è evitare le celle contrassegnate con 0 e le loro quattro celle adiacenti (sinistra, destra, sopra e sotto). Le mosse possibili da una determinata cella sono lo spostamento di una cella adiacente a sinistra, destra, sopra o sotto (non sono consentiti spostamenti in diagonale).

  Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test con il corrispondente output atteso.

  Svolgimento:
  Soluzione 1:
  La soluzione fa uso della strategia di *backtracking*. L'algoritmo consiste nel costruire una soluzione (percorso) in modo incrementale, partendo da ogni cella percorribile della prima colonna ed aggiungendo ad ogni passo una cella percorribile adiacente alla precedente, fino ad arrivare ad uno dei due possibili casi limite:

  - l'ultima cella inserita nel percorso appartiene all'ultima colonna (nel qual caso, l'algoritmo proverà ad aggiornare il valore del percorso minimo con la lunghezza del percorso attuale);
  - l'ultima cella inserita nella soluzione non presenta celle adiacenti percorribili (oltre a quella precedente), dunque bisogna ritornare al passo precedente e provare in un'altra direzione.

  Il numero di mosse possibili ad ogni passo sono 4:

  1. proseguire verso destra;
  2. proseguire verso il basso;
  3. proseguire verso l'alto;
  4. proseguire verso sinistra;

  Prima di eseguire una di queste mosse, l'algoritmo controlla che la cella che si vuole raggiungere sia percorribile, ossia il suo valore deve essere pari a 1 e non dev'essere adiacente ad uno 0, e lo fa attraverso una funzione di CHECK. In alternativa, è possibile effettuare un pre-processing della matrice andando a sostituire tutti gli 1 delle celle non percorribili con degli 0, in modo tale da semplificare la funzione di CHECK. Tale trasformazione richiederebbe lo scorrimento dell'intera matrice, con una complessità pari a $\Theta(n \cdot m)$, che non andrebbe comunque ad intaccare la complessità dell'intero algoritmo. Tuttavia, in questa soluzione si è deciso di adottare la prima delle due opzioni, scegliendo di implementare una funzione di CHECK leggermente più complessa, evitando il pre-processing della matrice.

Per implementare la soluzione ci resta da fare un'ultima considerazione: date le 4 mosse possibili ad ogni step, l'algoritmo, così come descritto fin'ora, finirebbe in un loop, perché ad ogni passo finirebbe sempre per tornare nella cella precedente da cui si proviene, o comunque, più in generale, niente ci impedisce di tornare in celle già visitate nell'ambito del percorso attuale, e quindi rimanere bloccati in dei cicli infiniti.

Per evitare che ciò avvenga, l'algoritmo modificherà temporaneamente il valore delle celle già visitate nell'ambito del percorso che si sta attualmente valutando, ponendole pari a -1. Verrà poi ripristinato il valore originale della cella (1) quando questa sarà rimossa dal percorso che si sta valutando,ovvero durante la fase di *undo*. In questo modo ad ogni step della costruzione della soluzione ci saranno 3 tipi di celle da considerare non percorribili:

- le celle contenenti il valore 0;
- le celle contenenti il valore 1, ma adiacenti ad almeno un 0;
- le celle contenenti il valore -1.

Il valore -1 indica che la cella è già stata percorsa nel contesto della soluzione attuale.

Algoritmo:

```
1: function SHORTEST-PATH(A)
2:     dichiara min ← ∞
3:
4:     for i ← 1 to row[A] do
5:         if CHECK(i, 1, A) = 1 then
6:             STEP(i, 1, A, 1, &min)
7:             ▷ Con il simbolo "&" indichiamo l'indirizzo della variabile
8:         end if
9:     end for
10:
11:    if min = ∞ then
12:        min ← −1
13:    end if
14:
15:    return min
16: end function
17:
18: function STEP(i, j, A, length_att, pmin)
19:
20:    A[i][j] ← −1
21:
22:    if j = column[A] then
23:        if length_att < ∗pmin then
24:            ∗pmin ← length_att
25:            ▷ Con il simbolo "*" indichiamo la locazione di memoria associata a quell'indirizzo
26:        end if
27:    else
28:        if CHECK(i, j + 1, A) = 1 then
29:            STEP(i, j + 1, A, length_att + 1, pmin)
30:        end if
31:        if CHECK(i + 1, j, A) = 1 then
32:            STEP(i + 1, j, A, length_att + 1, pmin)
33:        end if
34:        if CHECK(i − 1, j, A) = 1 then
35:            STEP(i − 1, j, A, length_att + 1, pmin)
```

```
        end if
        if CHECK(i, j − 1, A) = 1 then
            STEP(i, j − 1, A, length_att + 1, pmin)
        end if
    end if

    A[i][j] ← 1
end function
```

La funzione di CHECK non è altro che una serie di *if* che effettuano i vari controlli sul fatto che:

- non siano superati i limiti della matrice;
- la cella che si vuole aggiungere al percorso sia percorribile.

La funzione di CHECK restituisce 1 se tutti i controlli sono superati, o 0 altrimenti.

Si noti che non si utilizza alcun vettore per memorizzare le celle che compongono il percorso che si sta valutando, perché:

1. l'informazione di quali celle sono state già percorse è memorizzata all'interno della matrice stessa, assegnando il valore -1 a tali celle;
2. la soluzione finale non richiede di conoscere quali celle compongono il percorso minimo, ma solo la sua lunghezza.

Ottimizzazioni:

Si può ridurre lo spazio di ricerca dell'algoritmo facendo una considerazione: il limite minimo per un percorso che parte dalla prima colonna di una matrice e termina all'ultima è uguale al numero di colonne della matrice. Ciò detto, se l'algoritmo trovasse, nel mezzo della sua esecuzione, un percorso che eguagli tale limite, allora potrebbe interrompere la sua ricerca, dato che saremmo sicuri che nessun altro percorso possa essere più breve di quest'ultimo.

É possibile estendere tale intuizione considerando che non appena l'algoritmo trovi un nuovo percorso valido nel corso della sua ricerca, la lunghezza di tale percorso costituisca una *baseline* per tutte le ricerche successive. Ossia, se nel corso delle successive esplorazioni dello spazio delle soluzioni, mi accorgo di aver ottenuto un sotto-percorso la cui lunghezza già eguaglia il minimo attuale, allora posso già tornare indietro e tagliare quel ramo di ricerca. Lo posso fare perché sono sicuro che, qualora continuassi ad esplorare quel sottoinsieme delle soluzioni, che origina da quel ramo, al massimo potrei ottenere percorsi validi la cui lunghezza sia maggiore, o al più uguale, alla *baseline* fissata.

Volendo sfruttare il più possibile quest'ultima considerazione, vorremmo che il nostro algoritmo trovi al più presto un percorso valido sufficientemente breve, in modo tale da ottenere subito una buona *baseline* permettendoci di troncare il prima possibile molti rami di ricerca inutili. Per far ciò adottiamo la strategia della *best-first search*, ossia l'ordine di esplorazione dei 4 percorsi possibili per ogni cella non è casuale, ma esploriamo prima i percorsi che tendono a restituirci risultati migliori (percorsi più brevi). Nel nostro caso, l'ordine di ricerca per riuscire ad ottenere prima percorsi più brevi partendo dalla prima colonna è il seguente:

1. esploro la cella adiacente a destra $(j + 1)$
2. esploro la cella adiacente in basso $(i + 1)$
3. esploro la cella adiacente in alto $(i - 1)$
4. esploro la cella adiacente a sinistra $(j - 1)$

In realtà scegliere di esplorare prima in alto o prima in basso non è incisivo per quanto riguarda le prestazioni dell'algoritmo.

In verità è possibile generalizzare ancor di più l'intuizione precedente per potare ulteriormente lo spazio di ricerca. In precedenza abbiamo detto di poter tagliare quei rami di ricerca che generano sotto-percorsi le cui lunghezze eguagliano il minimo attuale, fissato da un percorso valido trovato in precedenza. In realtà non è necessario che il sotto-percorso arrivi ad eguagliare perfettamente il valore della *baseline* attuale, ma è possibile interrompere la ricerca già da prima. Questo perché arrivati ad una cella (i, j) attraverso un sotto-percorso di lunghezza l, già sappiamo che il percorso valido che origina da tale cella, se esiste, avrà lunghezza almeno pari a:

$$length = l + (column[A] - j) \quad \text{con } 1 \leq j \leq column[A]$$

Tale lunghezza coincide col caso migliore per cui partendo dalla cella (i, j) si sia in grado di raggiungere l'ultima colonna spostandosi sempre verso destra ($j = j + 1$).

Per cui è possibile interrompere la ricerca non appena si verifica la condizione:

$$l + (column[A] - j) \geq min_{att} \quad \text{con } 1 \leq j \leq column[A] \tag{1}$$

Per quanto riguarda l'implementazione di tali ottimizzazioni, in realtà, è più complicato spiegarle che realizzarle. Per la strategia del *best-search first*, già in precedenza nella funzione STEP le 4 chiamate ricorsive erano state ordinate in modo da rispettare l'ordine di esplorazione descritto in precedenza: destra, sotto, sopra, sinistra. Per la strategia del *search-pruning* è sufficiente modificare leggermente la funzione di CHECK per includere la condizione trovata in precedenza. In particolare creeremo un nuova funzione di check (CHECK2) che andrà a richiamare la funzione di CHECK precedente (*wrapping*), e ne metterà il risultato in AND con la condizione (1).

Algoritmo:

```
1: function CHECK2(i, j, A, length_att, min)
2:     if CHECK(i, j, A) = 1  and length_att + column[A] − j < min then
3:         return 1
4:     else
5:         return 0
6:     end if
7: end function
```

Attenzione: si noti che tutte le ottimizzazioni riportate in precedenza hanno senso solo nel momento in cui l'algoritmo riesca a trovare un percorso valido (e anche sufficientemente breve) il prima possibile. Qualora la matrice in ingresso non presenti alcuna soluzione valida, non sarà possibile effettuare il *pruning* dello spazio di ricerca.

Soluzione 2:
Per il primo problema abbiamo elaborato una seconda soluzione che non fa uso della strategia di *backtracking*. Tale soluzione prevede di:

1. effettuare un *pre-processing* della matrice di input effettuando le seguenti modifiche:
   - ad ogni cella percorribile è assegnato il valore $\infty tuttelecellecontenentiilvalore0sonolasciateinaltera$
   - a tutte le celle contenenti il valore 1, ma non percorribili perché adiacenti ad almeno uno 0, è assegnato il valore -1
2. scorrere l'ultima colonna della matrice con un ciclo **for** assegnando il valore 1 a tutte le celle maggiori di zero (ovviamente $\infty > 0$);
3. per ogni cella modificata al passo 2, inserire gli indici della cella in una coda che memorizza le celle modificate al passo precedente dell'algoritmo;
4. iniziare un ciclo **while** che ad ogni passo estrae una coppia di indici dalla coda e richiama su tale cella la funzione UPDATE-CELLS. La funzione UPDATE-CELLS controlla se nel vicinato della cella passatale come input è presente una cella raggiungibile dalla cella attuale in meno passi di quelli memorizzati nella cella stessa (inizialmente ∞).$Senetrovauna, opidiuna, neaggiornailvaloreconvalore-cella-corrente+1eneaccodagliindicinella$
5. a questo punto non resta che calcolare il minimo elemento maggiore di zero presente nella prima colonna della radice (se tale elemento è $\infty significachenonesisteunpercorsovalido$).

La complessità di questa soluzione dovrebbe essere un $O(n \cdot m)$, perché ogni cella percorribile entra una sola volta nella coda.

- **Problema 2.2**
  Sia dato un insieme di n elementi, e si supponga che ogni elemento possa essere accoppiato con qualche altro elemento oppure può essere non accoppiato ("single"). Ogni elemento può essere accoppiato solo una volta. Si implementi un algoritmo per scoprire il numero totale di modi in cui gli n elementi possono restare single o possono essere accoppiati. Si utilizzi la programmazione dinamica per risolvere il problema.

  Ad esempio:
  Input : n = 3
  Output : 4

  Spiegazione:
  {1}, {2}, {3} : tutti gli elementi sono single
  {1}, {2, 3} : 2 e 3 sono accoppiati ma 1 è single.
  {1, 2}, {3} : 1 e 2 sono accoppiati, ma 3 è single.
  {1, 3}, {2} : 1 e 3 sono accoppiati, ma 2 è single.
  Le coppie {a, b} e {b, a} sono considerate come un'unica coppia.

  Svolgimento:

  5 STEPS

  1. **Definizione del sottoproblema:**

  $$DP(i) \;\; con\ i = 1, \ldots, n \tag{2}$$

  $DP(i)$ rappresenta il numero di modi in cui i elementi dell'insieme possono restare single o possono essere accoppiati (la dimensione del gruppo sarà al massimo pari a due).

  $$\#\, sottoproblemi = \Theta(n) \tag{3}$$

  2. **Scelte del sottoproblema:** Nessuna scelta

  $$tempo/sottoproblema = \Theta(1) \tag{4}$$

  3. **Ricorrenza + Memoization:**

  Per questo problema di programmazione dinamica, utilizziamo un approccio di tipo top-down.
  Ci sono due possibili scelte per l'i-esimo elemento:
  1) l'i-esimo elemento rimane single, richiamiamo DP(i – 1);
  2) l'i-esimo elemento si accoppia con i restanti i - 1 elementi, otteniamo così (i – 1)*DP(i – 2).

  Dunque possiamo scrivere la ricorrenza come:

  $$\begin{cases} DP(i) = DP(i-1) + DP(i-2) * (i-1) \;\; con\ i = 3, \ldots, n \\ DP(1) = 1 \\ DP(2) = 2 \end{cases}$$

  Per la Memoization utilizziamo un array di dimensione n, inizializzato in questo modo:

1) $mem[i] = 0$ *con* $i = 3, ..., n$;
2) $mem[1] = 1$ e $mem[2] = 2$ (casi base).
Ad ogni chiamata ricorsiva, i risultati vengono salvati in questo array, e all'occorrenza prelevati, evitando così di ripetere più volte gli stessi calcoli gia effettuati dalle precedenti chiamate. Ogni chiamata all'array costa $\Theta(1)$, cioè controlla semplicemente se DP(i) è già presente nell'array (ovvero se $array[i]$ è diverso da zero).

4. **Aciclicità:**

   Il grafo dei sottoproblemi è aciclico poichè nella ricorrenza l'indice i viene sempre decrementato.

$$T = \#\, sottoproblemi * tempo/sottoproblema = \Theta(n) * \Theta(1) = \Theta(n) \tag{5}$$

5. **Problema originario:**

   Il problema originario corrisponde ad un sottoproblema DP(n), quindi non c'è extra-time.

Algoritmo:

```
1: function DP(n, mem)
2:     if mem[n] = 0 then
3:         mem[n] ← DP(n − 1, mem) + (n − 1) * DP(n − 2, mem)
4:     end if
5:     return mem[n]
6: end function
```

- **Problema 2.3**
  Si consideri una matrice di 0 ed 1, in cui "1" indica "posizione occupata" e "0" indica "posizione libera". Si scriva un algoritmo per determinare la sottomatrice massima libera (ossia che contiene tutti 0). L'algoritmo deve riportare il numero di 0 di tale sottomatrice. Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test con il corrispondente output atteso

  Svolgimento:

  Tale problema può essere risolto utilizzando una funzione che calcola la massima area rettangolare all'interno di un istogramma. Inizialmente trasformiamo il nostro problema originario, scambiando tutti gli zero con gli uno e gli uno con gli zero. In questo modo, per ogni passo dell'algoritmo, è possibile vedere la nostra matrice come un istogramma. Fissata una riga i, base dell'istogramma, e un vettore sum che contiene informazioni ottenute dalle iterazioni precedenti, possiamo costruire per ogni j le barre dell'istogramma in questo modo:

  1) se $matrice[i][j]$ è diverso da zero, allora $sum[j] = sum[j] + matrice[i][j]$;
  2) se $matrice[i][j]$ è uguale a zero, $sum[j] = 0$;

  ovvero in questo modo ogni elemento del vettore, $sum[j]$, rappresenta l'altezza della barra j-esima dell'i-esimo istogramma. Quindi ad ogni iterazione costruiamo un nuovo istogramma del quale è possibile calcolare la sua massima area rettangolare. Confrontando tali aree al variare della base e quindi per ogni riga della matrice, riusciamo a ottenere la dimensione della sottomatrice massima contenente tutti gli elementi pari ad uno, che è equivalente al nostro problema originario.

Esempio:

1 0 1 1 0
0 1 0 1 1
1 1 1 1 0
0 1 1 0 1
1 1 1 1 1

$sum = 0\,0\,0\,0\,0,\ massima\ area = 0$

Step 0:
$sum = sum + riga_0 = 1\,0\,1\,1\,0,\ area = 2,\ massima\ area = 2$
Step 1:
$sum = sum + riga_1 = 0\,1\,0\,2\,1,\ area = 2,\ massima\ area = 2$
Step 2:
$sum = sum + riga_2 = 1\,2\,1\,3\,0,\ area = 4,\ massima\ area = 4$
Step 3:
$sum = sum + riga_3 = 0\,3\,2\,0\,1,\ area = 4,\ massima\ area = 4$
Step 4:
$sum = sum + riga_4 = 1\,4\,3\,1\,2,\ area = 6,\ massima\ area = 6$

Tale funzione, chiamata nel sorgente $max\_zero\_sottomat()$,ha una complesità $\Theta(n * m)$.

La massima area rettangolare all'interno di un istogramma è data da un rettangolo composto da un numero di barre contigue, le cui altezze sono fornite da un array. Per semplicità, supponiamo che tutte le barre abbiano la stessa larghezza e che tale larghezza sia pari ad uno.
Per ogni barra del vettore, i è la barra che ha l'altezza più piccola del rettangolo di cui vogliamo calcolarci l'area. Il rettangolo è ottenuto selezionando tra le barre contigue, a destra e sinistra, le barre che hanno un'altezza superiore o uguale a quella della barra i. In questo modo grazie all'indice destro e sinistro (base del rettangolo) possiamo calcolare l'area con la classica formula:

$A_i = array[i] * (indice\_destro - indice\_sinistro)\ dove\ i = 0, \ldots, n$

Di queste n aree deve essere selezionata la massima area per poter risolvere il problema. Per conoscere la i-esima area, dobbiamo conoscere l'indice della prima barra più piccola a sinistra dell'i-esima barra ($inidice\_sinistro$) e l'indice della prima barra più piccola a destra dell'i-esima barra($indice\_destro$). Attraversiamo tutte le barre del vettore da sinistra a destra e manteniamo uno stack di indici di barre. Gli indici delle barre possono essere aggiunti allo stack una sola volta. Un elemento viene estratto dallo stack appena si decide di aggiungere l'indice di una nuova barra che ha altezza inferiore rispetto ad uno già presente nello stack. Appena tale indice i viene estratto, calcoliamo la relativa area del rettangolo, dove l'altezza è l'altezza della barra estratta ($array[i]$), l'indice destro lo si ottiene dall'indice corrente e invece l'indice sinistro è l'indice dell'elemento precedente nello stack.

Tale funzione chiamata nel sorgente, $max\_area()$, ha una complesità $\Theta(m)$.

COMPLESSITA FINALE ALGORITMO: $\Theta(n * m)$