Algoritmi e Strutture Dati – 04/09/2020

Esercizio A1

È possibile risolvere il problema prima andando caso per caso per piccoli valori di a, e poi tracciando una legge generale per valori di a crescenti. Si utilizza il master theorem, versione base.

a	α	β	Caso	Complessità
2	$\log_2 2 = 1$	$\lceil 2/2 \rceil = 1$	2	$\Theta(n \log n)$
3	$\log_3 2 < 1$	$\lceil 2/3 \rceil = 2$	3	$\Theta(n^2)$
4	$\log_4 2 < 1$	$\lceil 4/2 \rceil = 2$	3	$\Theta(n^2)$
5	$\log_5 2 < 1$	$\lceil 5/2 \rceil = 3$	3	$\Theta(n^3)$

Poiché $\log_a 2 \leq 1$ per qualunque valore $a \geq 2$, è possibile vedere che $T(n) = \Theta(n \log n)$ per $a = 2$ e $T(n) = \Theta(n^{\lceil a/2 \rceil})$ per tutti gli altri casi.

Esercizio A2

È possibile risolvere il problema effettuando tre visite in ampiezza a partire da ciascuno dei nodi u, v, w e memorizzando le distanze da tali nodi in tre vettori di interi.
A questo punto, per ogni nodo si cerca il minimo della somma delle distanze e si restituisce il nodo corrispondente, che può essere uno dei nodi di partenza.
Il costo computazionale dell'algoritmo proposto è $O(n + m)$, corrispondente alle tre visite in ampiezza del grafo.

NODE meetingPoint(GRAPH G, NODE u, NODE v, NODE w)

```
int[] d_u = new int[1...G.n]
int[] d_v = new int[1...G.n]
int[] d_w = new int[1...G.n]
distance(G, u, d_u)
distance(G, v, d_v)
distance(G, w, d_w)
int min = +∞
NODE z
for i = 1 to n do
    if d_u[i] + d_v[i] + d_w[i] ≤ min then
        min = d_u[i] + d_v[i] + d_w[i]
        z = i
return z
```

Note aggiuntive L'idea di questo esercizio deriva dal testo "Selected papers on Fun & Games", di Donald Knuth. Nel capitolo 32 parla del gioco "Doublets", che richiede di trovare un percorso che unisca due parole, cambiando una lettera alla volta. Ad esempio, da "word" a "game": "word-ward-ware-dare-dame-game". Il nome Doublets è stato proposto da Lewis Carroll nel Natale del 1877. Nel capitolo 33, Knuth propone Triplets: si parte da tre parole e si cerca una parola "intermedia" che le unisca tutte e tre. Ovviamente, l'insieme delle parole e la possibilità di passare da una all'altra tramite un cambio di parola può essere rappresentata tramite un grafo non orientato.

Esercizio A3

Il problema può essere risolto in vari modi, con complessità che variano dall'assurdo all'ottimo.

Versione 1, Pythonista alla prime armi La prima versione viene proposta in Python, perché spesso e volontieri un uso poco accorto delle operazioni di slicing e delle primitive può far crescere la complessità.

```
def longestSingle(A):
  n = len(A)
  maxSoFar = 0
  for start in range(n):
    for dim in range(1,n-start+1):
      B = A[start:start+dim]
      duplicates = False
      for elem in B:
        if B.count(elem)>1:
          duplicates = True
      if not duplicates:
        maxSoFar = max(maxSoFar,dim)
```

```
  return maxSoFar
```

In questa versione, ci sono tre cicli di costo lineare annidati e una chiamata alla funzione count, che ha costo lineare, per un costo totale di $\Theta(n^4)$.

Versione 2, Pythonista un pochino più esperto Un pythonista un po' più esperto potrebbe conoscere strutture dati come `set` e scrivere codice leggermente più efficiente.

```
def longestSingle3(A):
  n = len(A)
  maxSoFar = 0
  for start in range(n):
    for dim in range(1,n-start+1):
      elements = set(A[start:start+dim])
      if len(elements)==dim:
        maxSoFar = max(maxSoFar,dim)
  return maxSoFar
```

In questa versione, ci sono due cicli annidati, all'interno dei quali il sottovettore compreso fra i e j viene trasformato in un insieme. Se l'insieme e il sottovettore da cui deriva hanno la stessa dimensione, allora non ci sono elementi duplicati e si incrementa la dimensione
Il costo della trasformazione da slice ad insieme è pari ad $O(n)$, quindi l'algoritmo ha costo $\Theta(n^3)$.

Versione 3, iniziamo a ragionare Ovviamente, possiamo renderci conto che utilizzando per bene la struttura dati Set ed evitando di ricostruire l'insieme ad ogni iterazione del ciclo interno, possiamo risparmiare un ulteriore ordine di grandezza. Da qui in poi, viene utilizzato lo pseudocodice.

```
int longestSingle2(int [][] A, int n)
    int maxSoFar = 0
    for start = 1 to n do
        Set S = Set()
        end = start
        while end ≤ n and not S.contains(A[end]) do
            S.insert(A[end])
            end = end + 1
        maxSoFar = max(maxSoFar, end − start)
    return maxSoFar
```

Il primo **for** cicla su tutti i possibili indici iniziali, mentre il secondo accresce l'insieme man mano che avanza l'indice finale. Notate che sia che si trovi un duplicato, che si sfori la dimensione del vettore, il ciclo **while** termina un elemento avanti a quello che vogliamo contare. Per questo motivo, calcoliamo $end - start$ e non il solito $end - start + 1$.
Assumiamo che l'insieme sia implementato tramite una tabella hash, e quindi le operazioni abbiano costo $O(1)$. La complessità è quindi $O(n^2)$ nel caso pessimo. Nel caso ottimo, un singolo valore ripetuto n volte, la funzione ha complessità $O(n)$.

Versione 4, complessità ottimale È possibile notare che tutte le volte che si incontra un duplicato, viene fatto avanzare *start* di 1. Ma se il duplicato si trova dopo, la stessa coppia di valori duplicati stopperà la crescita del sottovettore di nuovo, e di nuovo, fino a quando l'indice *start* non supererà tale duplicato.
Ad esempio, se $A = [1, 2, 3, 3, 1, 2, 3, 4]$, si considerano nell'ordine i sottovettori $[1]$, $[1, 2]$, $[1, 2, 3]$,$[1, 2, 3, 3]$; a questo punto si trova un duplicato e si considerano i vettori $[2]$, $[2, 3]$, $[2, 3, 3]$. Trovando la stessa coppia di duplicati, si riparte da $[3]$, $[3, 3]$ e solo successivamente si trova $[3]$, $[3, 1]$, $[3, 1, 2]$ etc. Avrebbe più senso far ripartire *start* dalla posizione successiva a quella in cui si trova il duplicato.

```
int longestSingle1(int [][] A, int n)
    Set S = Set()
    int maxSoFar = 0
    int start = 1
    int end = 1
    while end ≤ n do
        if S.contains(A[end]) then                          % Found duplicate
            while A[start] ≠ A[end] do                      % Remove elements before duplicate
                S.remove(A[start])
                start = start + 1
            start = start + 1                               % Start after duplicate
        else
            S.insert(A[end])                                % Add element
            maxSoFar = max(maxSoFar, end − start + 1)       % Update maximum, if needed
        end = end + 1
    return maxSoFar
```

Nonostante la presenza del doppio ciclo, la complessità computazionale è $\Theta(n)$. Ogni elemento viene inserito nell'insieme al massimo una volta, e ogni elememento viene rimosso dall'insieme al massimo una volta.

Versione 5, struttura dati più semplice Un ulteriore commento: è possibile sfruttare il fatto che i valori presenti nel vettore sono compresi da 1 a n e implementare l'insieme come un vettore di booleani, un approccio leggermente più efficiente ma che non cambia la complessità computazionale, che resta $\Theta(n)$.

```
int longestSingle1(int [][] A, int n)
    boolean [] S = new boolean[1 . . . n] = {false}
    int maxSoFar = 0
    int start = 1
    int end = 1
    while end ≤ n do
        if S[A[end]] then                                   % Found duplicate
            while A[start] ≠ A[end] do                      % Remove elements before duplicate
                S[A[start]] = false
                start = start + 1
            start = start + 1                               % Start after duplicate
        else
            S[A[end]] = true                                % Add element
            maxSoFar = max(maxSoFar, end − start + 1)       % Update maximum, if needed
        end = end + 1
    return maxSoFar
```

La complessità è uguale a quella precedente, i fattori moltiplicativi e l'occupazione di memoria leggermente inferiori.

Esercizio B1

Questo esercizio è simile ad un esercizio che era stato dato in passato. Utilizziamo programmazione dinamica. Sia $DP[i][r]$ il numero di modi con cui è possibile ottenere il valore r utilizzando i primi i valori.
$DP[i][r]$ può essere calcolato nel modo seguente:

$$DP[i][r] = \begin{cases} 1 & r = 0 \\ 0 & r > 0 \wedge i = 0 \\ DP[i-1][r] + DP[i-1][r - A[i]] & i > 0 \wedge r > 0 \wedge A[i] \leq r \\ DP[i-1][r] & i > 0 \wedge r > 0 \wedge A[i] > r \end{cases}$$

- Se $r = 0$, esiste un modo per ottenere il valore zero - non selezionare alcun elemento
- Se $r > 0$ e $i = 0$, ho esaurito i valori e non ho ancora raggiunto il valore 0
- Altrimenti, se $A[i] \geq r$ ho due possibilità, prendere e non prendere il valore. Se invece $A[i] < r$, devo ignorarlo.

Utilizziamo memoization per tradurre l'equazione ricorsiva in codice:

```
int countMenu(int[] A, int n, int k)
    int[] DP = new int[1...n][1...k] = {−1}
    return countRec(A, n, k, DP)
```

```
int countRec(int[] A, int i, int r, int[] DP)
    if r == 0 then
        return 1
    else if i == 0 then
        return 0
    else
        if DP[i][r] < 0 then
            DP[i][r] = countRec(A, i − 1, r, DP)
            if A[i] ≤ r then
                DP[i][r] = DP[i][r] + countRec(A, i − 1, r − A[i], DP)
        return DP[i][r]
```

La complessità è limitata superiormente dalla dimensione della matrice: $O(nr)$, pseudopolinomiale nella dimensione dell'input. Si noti che se più piatti hanno lo stesso costo, verranno contati separatamente (come è giusto che sia). Un'estensione dell'algoritmo che mostra i menù possibili dovrebbe ritornare ognuna di queste possibilità.
Si noti inoltre che per come è organizzato l'algoritmo, le scelte vengono effettuate nell'ordine inverso in cui sono memorizzati i piatti, e quindi non ci sono problemi di permutazioni.

Esercizio B2

Sia $DP[i][j]$ la lunghezza della più lunga sottostringa comune che termina in posizione i nella stringa P e nella posizione j nella stringa T. Il valore può essere calcolato tramite questa formula ricorsiva:

$$DP[i][j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ 0 & i > 0 \wedge j > 0 \wedge P[i] \neq T[j] \\ DP[i-1][j-1] + 1 & i > 0 \wedge j > 0 \wedge P[i] = T[j] \end{cases}$$

Si noti che al termine bisognerà cercare il massimo sull'intera matrice, non nella sola casella (m, n).

```
int LCSubstring(Item[] P, Item[] T, int n, int m)
    int[][] DP = new int[0...n][0...m] = {0}
    int maxSoFar = 0
    for i = 1 to n do
        for j = 1 to m do
            if P[i] == T[j] then
                DP[i][j] = DP[i − 1][j − 1] + 1
                maxSoFar = max(maxSoFar, DP[i][j])
            else
                DP[i][j] = 0
    return maxSoFar
```

Dovendo calcolare l'intera matrice, il costo computazionale è pari a $O(mn)$.

Informazioni aggiuntive Questo problema può essere risolto in tempo $O(m+n)$ utilizzando una struttura dati particolare, chiamata *generalized suffix tree*.

https://en.wikipedia.org/wiki/Longest_common_substring_problem

Esercizio B3

Si utilizza la tecnica backtrack utilizzando uno stack. In questa versione, partiamo dal fondo. Se l'elemento che si sta considerando è più piccolo del valore che stiamo considerando, abbiamo due possibilità: si può inserire oppure no nello stack. Se invece è più grande, viene saltato.
Nel caso di un vettore ordinato, ognuna delle 2^n sottosequenze deve essere stampata, con costo $\Theta(n)$. La complessità computazionale è quindi $O(n \cdot 2^n)$.

```
printIncreasing(int[] A, int n)
    STACK S = Stack()
    printRec(A,n,S)
```

```
printRec(int[] A, int i, STACK S)
    if i == 0 then
        print S                                              % LIFO order
    else
        if S.isEmpty() or A[i] < S.top() then
            S.push(A[i])
            printRec(A, i − 1, S)
            S.pop()
        printRec(A, i − 1, S)
```