Incontro 2024-04-30: programmazione dinamica (poldo)

Gli incontri avvengono sia in presenza che nella stanza Zoom:

https://univr.zoom.us/j/85817283103

E, quando disponibile, la loro registrazione è nel folder:

https://univr.cloud.panopto.eu/Panopto/Pages/Sessions/List.aspx?folderID=c4df8587-8f34-4732-8e51-b12d0108ea08

Poldo (massima sottosequenza crescente):

richiesta del problema: trovare una sottosequenza crescente di massima lunghezza entro una sequenza di interi data in input.

Esempio:

```
    7 11  8  5  3  9  1  2  6  4  10
    *     *     *       <-- soluzione non ammissibile in quanto 3 < 8
          *        *              *  <-- sottosequenza crescente (soluzione
ammissibile ma non ottima)
```

Osservazioni sul problema: il numero delle soluzioni ammissibili potrebbe benissimo essere esponenziale. Ad esempio, per la sequenza: 1 2 3 … n tutte le 2^n sottosequenze sono soluzioni ammissibili

quindi non possiamo puntare su un approccio di guessing (=provarle tutte, guessare il certificato). Ci serve entrare nella struttura del problema. (Per fare la quale cosa, non è in contraddizzione con quanto detto sopra, potrebbe aiutarci l'esercizio di saper solo contare quante sono le soluzioni ammissibili in tempo polinomiale)

Potrebbe bastare un approccio greedy?

Due possibili filosofie greedy per Poldo (assumiamo che il suo obiettivo sia mangiare il maggior numero possibile di panini):

1. appena mi presentano un panino che posso prendere (questo garantisce quantomeno l'ammissibilità della soluzione prodotta), io, quasi quasi, me lo magno:

   controesempio: 11 1 2 … n

2. potrei considerare di prendere (se posso) i panini partendo da quello più piccolo. (Nell'istanza controesemipo di cui sopra finirei proprio col pendere tutti i panini tranne il primo.) controesempio (al fatto che greedy trova sempre l'ottimo): 2 3 4 … (n-1) n 1

provare due approcci potrebbe non essere significativo per concludere che un approccio greedy non possa funzionare. E' più forte vedere se possiamo mettere in evidenza fenomeni di non-località (ossia che modificando l'istanza in un punto modifica la soluzione ottima anche in punti molto lontani da quello). (E' argomento efficace perchè l'idea del greedy è quella di un'euristica che sulla sola base di considerazioni di stampo locale mi sento di prendere una decisione definitiva quì ed ora. La non.località significa invece che io non posso limitarmi a considerare un orizzonte limitato nel decidere). Abbiamo ingegnerizzato il seguente esempio che evidenzia la necessità di non limitarsi ad un orizzonte limitato: (n+100) (n+101) (n+102) (n+1) (n+2) n (n-1) (n-2) … 3 2 1 (n+3) (n+4) …

Tutto questo per supportare il sospetto che sia necessario scomodare la tecnica della Programmazione Dinamica (se vogliamo mettere il problema in P)

Invece che ragionare in negativo, abbiamo degli elementi a favore della programazione dinamica? (ossia che suggeriscano che possa funzionare)

1. i panini sono disposti su una linea, uno struttura ricorsiva elementare cosa intendo dire: una fila indiana di formiche contiene un sacco di sotto-file-indiane di formiche. Ad esempio: un qualsiasi suffisso (o prefisso) di una linea è una linea. E anche qualsiasi intervallo di una linea/fila-indiana resta una linea/fila-indiana (se dovessi andar giù pesante nella definizione del contratto ricorsivo).
2. c'è la spia delle brise della PD? dentro una soluzione ottima ci sono, a guardarmi, soluzioni ottimi a sottoproblemi.

   Vediamo se ci pare che questo succeda. Prendiamo un esempio e cerchiamo di identificarne una soluzione ottima per poi analizzarla da questo punto di vista:

```
 7 11  8  5  3  9  1  2  6  4  10
 *     *        *               *  <-- ipotesi 1 (prendo il 7), e trovo una prima
soluzione ottima
                  *  *  *      *  <-- non prendo il 7, anche queste soono ottime,
valore ott=4
                  *  *     *   *  <-- ott
```

Ovviamente queste due soluzioni ottime sono soluzioni ottime anche di un'istanza che non offra gli elementi iniziali (7, 11, 8, 5, ...) che nessuna di esse (due) prende.

Ossia è soluzione ottima sui seguenti suffissi dell'istanza vera di partenza:

```
        7 11  8  5  3  9  1  2  6  4  10
          11  8  5  3  9  1  2  6  4  10
    su questo suffisso, a parte questa soluzione che appare non più valida (ma si è
persa davvero del tutto?)
    restano valide ed ottime tute le altre soluzioni ottime dell'istanza originale
              8  5  3  9  1  2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                 5  3  9  1  2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                    3  9  1  2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                       9  1  2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                          1  2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                          *  *  *      *
                          *  *     *   *
                             2  6  4  10  <-- restano valide ed ottime tutte quelle
di suffisso precedente
                             *  *      *
                             *     *   *
                                6  4  10  <-- tolto quanto fuoriesce dalle opt sol di
suffisso precedente, si ottengono sol valide ed ottime
                                *      *  <-- al prossimo passo questa perderà in
ottimalità (ma è persa del tutto?)
                                   *   *
                                   4  10
                                   *   *
                                      10
                                       *
```

mi piacciono i suffissi, perchè con un solo ditino riesco ad indicare a fatina ricorsina il suffisso/sotto-problema di sua pertinenza (ho squeezato al massimo l'interfaccia del contratto di Faust).

Ma in realtà abbiamo over-semplificato:

```
101 102 103 104 105 1 2 3 4
*     *   *   *   *
```

ma sul suffisso:

```
                    1 2 3 4
converrebbe prendere qualcosa piuttosto che niente, se il chiamante non mi dice nulla
di più (non mi lascia alcun messaggio nella cassaforte, come nel problema borse di
studio)

101 102 103 104 105 1 2 3 4
                    ^
                     quando arrivo quì, è importante che mi venga detto che elementi
così piccoli non possono più essere presi.
```

Proviamo quindi a definire un contratto che includa il passaggio di ulteriore informazione richiesta a fatina ricorsina per fare il suo lavoro ben coordinandosi alle scelte spese a monte dal chiamante:

- un ditino per dire quale sia il suffisso rimasto di pertinenza
- un ditino per dire il numero più grande già in precedenza preso

```
def opt_val(i,j):
    assert j <= i
    """ritorna il massimo numero di panini (=massima lunghezza di una sotto-sequenza
crescente) della sequenza suffisso panino[i:] sotto il vincolo di non includere
nessun panino di colesterolo < panino[j]"""
```

Esemplificazione:

```
n=11
panino=[-1, 7, 11, 8, 5, 3, 9, 1, 2, 6, 4, 10]
            ^   ^
            j   i
```

```
Quindi abbiamo scritto la seguente implementazione del contratto:
def opt_val(i,j):
    if i >= len(panino):
        return 0
    if panino[j] > panino[i]:
        return opt_val(i+1,j)
    return max( opt_val(i+1,j), 1 + opt_val(i+1,i) )
```

Lista della spesa di come procedere:

1. metterci della memoizzazione o riscriverlo come una PD
2. vestire la struttura perchè faccia qualcosa di più concreto come restituire una soluzione ottima invece che solo dirne il valore

Ma prima di fare queste cose, voglio proporre un contratto ricorsivo che con una domanda un pò sghemba riesce a poggiare su solamente n sotto-problemi. (Il numero attuale dei sotto-problemi, ossia di coppie (i,j), era O(n^2))

domanda(i) := dimmi la massima lunghezza di una sotto-sequenza crescente che prenda proprio l'i-esimo elemento come suo primo elemento.

```
def opt_val2(i):
    """ritorna il massimo numero di panini (=massima lunghezza di una sotto-sequenza
crescente) che riesco a mangiare se comincio la scorpacciata partendo col panino
panino[i]"""
    assert 0 <= i < len(panino)
```

```
    if i == len(panino) - 1:
        return 1
    best = 1 # mangio solo il panino i
    for j in range(i+1,len(panino)): # se mangio un secondo panino, sia esso j
        if panino[j] > panino[i]: # ok, posso mangiare j dopo i -filtro-
           best = max(best, 1 + opt_val2(j))
    return best
```

volgerlo in programmazione dinamica (conviene sempre provare a fare prima le cose a mano su un foglio di carta):

```
                    <----
       ?   ?  ?  ?  ?  2  4  3  2  2   1
panino=[7, 11, 8, 5, 3, 9, 1, 2, 6, 4, 10]
```

proposta di problema arcobaleno (colage,)

minimizzare il numero di fogli da impiegare per produrre un certo arcobaleno dato in in input.

```
input.txt
6
1 2 3 2 1 3
```

```
output.txt
4
```

(avendo in mente la seguente soluzione a 4 fogli colorati)

```
    3
  2 2 2
1 1 1 1 1 3
```