Commento al Laboratorio n. 10

Esercizio n. 1: **Collane e pietre preziose (versione 3)**
Ogni collana può iniziare con uno dei 4 tipi di gemma. Si considera a titolo esemplificativo il caso in cui la prima gemma è uno zaffiro (funzione `fZ`), gli altri si ricavano facilmente.

La programmazione dinamica top-down o ricorsione con memorizzazione o memoization opera secondo il paradigma divide et impera, decidendo quale gemma utilizzare per la posizione corrente e ricorrendo sulla successiva in base alla disponibilità di gemme e al soddisfacimento delle regole di composizione.

La condizione di terminazione è raggiunta quando non sono più disponibili gemme del tipo in esame.

Ogni sottoproblema è visto come una collana con `z` zaffiri, `r` rubini, `t` topazi e `s` smeraldi ancora disponibili. La soluzione al sottoproblema è la lunghezza della collana generata. La soluzione viene memorizzata in una cella di matrice quadridimensionale `Z[z][r][t][s]`. Prima di ricorrere si verifica se la soluzione al problema con `z` zaffiri, `r` rubini, `t` topazi e `s` smeraldi è già stata calcolata (`Z[z][r][t][s] != -1`). IN caso affermativo si ritorna tale soluzione, altrimenti si procede con la ricorsione. Viste le regole di composizione, uno zaffiro può essere solo seguito da un altro zaffiro o da un rubino. Si chiamano quindi le funzioni `fZ` e `FR` su un numero di zaffiri decrementato di 1, si confrontano i risultati `maxZ` e `maxR` e si sceglie il maggiore tra i 2, memorizzando in `Z` il valore incrementato di 1 per tener conto di aver scelto una gemma.
La funzione `solveM` alloca le 4 matrici quadridimensionali `Z`, `R`, `T` ed `S`, chiama singolarmente le funzioni `fZ`, `fR`, `fT` e `fS`, ne confronta i risultati e determina quale sia quello massimo.
Il `main` si occupa di leggere da file il numero di testset e la composizione del vettore delle gemme e di chiamare iterativamente su ogni testset la funzione `solveM`.

Esercizio n. 2: Corpo libero
Strutture dati: si definiscono le seguenti strutture dati:

- per i dati:
  - un tipo `elemento_t` come `struct` che contiene i campi dell'elemento
  - un tipo `diagonale_t` come `struct` che contiene i campi della diagonale, incluso un vettore `elementi` di indici che permettono di accedere alla collezione di elementi
- per le collezioni di dati:
  - un tipo `elementi_t` come `struct` che contiene un vettore di dati di tipo `elemento_t` e il numero di elementi disponibili
  - un tipo `diagonali_t` come `struct` che contiene un vettore di dati di tipo `diagonale_t`, il numero `nd` di diagonali valide effettivamente disponibili e la dimensione massima `maxd` del vettore allocato, per tracciare la crescita dinamica del vettore durante la fase di generazione di tutte le diagonali ammissibili.

Analisi del problema: la soluzione si articola in 3 fasi:

1. inizializzazione della collezione `D` delle diagonali, lettura degli elementi e loro memorizzazione nella collezione di dati `E` degli elementi
2. generazione delle diagonali a partire dagli elementi
3. generazione del programma ottimo a partire dalle diagonali disponibili.

Modelli:

- generazione delle diagonali a partire dagli elementi: si osservi che non è vietato esplicitamente ripetere lo stesso elemento in una diagonale. Si osservi inoltre che l'ordine con gli elementi compaiono nella diagonale conta, in quanto le specifiche impongono vincoli (ad esempio di entrata/uscita, precedenza, sequenza finale etc.). Il modello è quindi quello delle disposizioni ripetute di `MAXE` elementi presi al massimo a 5 a 5. Per limitare lo spazio di ricerca si impongono condizioni sulla discesa ricorsiva (pruning) per escludere di superare la massima difficoltà ammessa per una diagonale, per forzare la condizione di ingresso e quella di precedenza per il primo elemento della diagonale. La condizione di terminazione verifica sia se sono stati presi esattamente 5 elementi, ma anche se si può interrompere la costruzione della diagonale anticipatamente (con meno di 5 elementi) in quanto l'ultimo non prevede elementi che lo seguano e poiché, purché valida per composizione, anche diagonali più corte di 5 elementi sono comunque ammissibili. La soluzione corrente (`diag`, vettore di indici che permettono di accedere agli elementi che appartengono alla soluzione) viene memorizzata nella collezione di dati `D` delle diagonali posto che sia soddisfatto il vincolo sul numero di elementi acrobatici. Il vettore `vd` in `D` viene riallocato dinamicamente in funzione del numero di diagonali generate
- generazione del programma ottimo a partire dalle diagonali disponibili: si osservi che anche in questo caso non è vietato esplicitamente ripetere la stessa diagonale.La specifica secondo cui, se il ginnasta include un elemento finale di difficoltà 8 o superiore nell'ultima diagonale presentata in gara, il punteggio complessivo della diagonale viene moltiplicato per 1.5 sembra orientare verso un modello in cui conta l'ordine. In realtà basta verificare che vi sia una diagonale con elemento finale di difficoltà 8 o superiore e considerarla come ultima per usufruire del bonus moltiplicativo. Pertanto, a differenza del punto precedente, non conta l'ordine con cui si inseriscono le diagonali nel programma. Il modello sottostante è quindi quello delle combinazioni ripetute. L'output però non riporta le diagonali nell'ordine voluto. Il pruning dello spazio è realizzato subordinando la discesa ricorsiva alla verifica che non si ecceda la massima difficoltà del programma. La validità della soluzione è verificata nella condizione di terminazione.

Esercizio n. 3: Corpo libero (vers. greedy)

Analisi del problema

Si tratta di un problema in cui il numero e il tipo di vincoli preclude (o quanto meno scoraggia) una semplice strategia greedy con funzione di appetibilità "statica". Infatti, un elemento è accettabile/appetibile non in senso assoluto ma in funzione della sua posizione e in base agli altri elementi scelti. Pur essendo questo il caso tipico dei problemi di ricerca, le soluzioni greedy statiche prediligono i problemi in cui:

- prevalga il problema di ottimizzazione su quello di ricerca, cioè non sia difficile trovare una soluzione (ricerca) che rispetti i vincoli, mentre la strategia consista piuttosto nell'individuare (ottimizzazione) una soluzione ottima

- i vincoli di accettabilità siano semplici e non (o poco) dipendenti dall'ordine
- il criterio di appetibilità sia chiaro e calcolabile una tantum.

La strategia statica (semplice) di:

- ordinare gli elementi in base a un criterio di appetibilità (es. valore decrescente o crescente)
- percorrerli (una volta sola!) secondo l'ordine scelto
- selezionare gli elementi che rispettano i vincoli

pur se possibile, difficilmente troverebbe una soluzione, se non in casi estremamente semplici, proprio a causa dei (molti) vincoli: in pratica, è difficile trovare una soluzione ammissibile.

Notando che un elemento non accettabile al passo i potrebbe esserlo a un passo successivo (o precedente), è quindi preferibile una strategia greedy "dinamica", che ad ogni passo seleziona l'elemento i-esimo, rivalutando l'ottimo locale tra tutti gli elementi disponibili.

I ***vincoli*** sono di due tipi:

- locali (compatibilità tra direzioni di entrata e uscita di elementi), cioè verificabili direttamente quando si fa una scelta, confrontando un elemento con quelli adiacenti (vista la simmetria, è sufficiente il confronto col precedente)
- globali (almeno una diagonale con elemento acrobatico avanti, una indietro, almeno una con due elementi acrobatici consecutivi). Questi potrebbero solo essere verificati a posteriori, dopo aver completato il programma, con il rischio (molto elevato) che non siano rispettati.

Entrambi questi vincoli sono "dinamici", in quanto le scelte ammissibili, ad ogni passo, dipendono dalle precedenti (a livello di singola diagonale o di programma intero).

L'***appetibilità***, come in molte strategie greedy, può andare da criteri estremamente semplici ad altri più raffinati e dinamici. Va notato che, in questo problema, si possono considerare due tipi si appetibilità:

- finalizzata all'*ottimizzazione*. Un semplice criterio di appetibilità, in tale direzione, è il valore di un esercizio, in quanto il problema di ottimizzazione cerca il programma di valore più alto. Tale criterio rende questo problema simile al problema "classico" dello zainetto (knapsack) discreto (un esercizio va preso tutto, non in parte). Un criterio di ottimizzazione più complicato da gestire è il bonus, nel caso di ultimo esercizio a punteggio alto (difficile da gestire con strategia greedy).
- finalizzata alla *ricerca*, cioè a trovare (in modo diretto e senza tornare indietro su decisioni prese) una soluzione che rispetti i vincoli. In tale ambito, l'appetibilità deve indirizzare le scelte verso soluzioni con minor probabilità di fallimento (vincoli non rispettati). Nella soluzione proposta, si considerano come più critici i vincoli globali sugli esercizi acrobatici: per questi si proporrà un criterio di appetibilità. Non si proporrà invece una forma di appetibilità per la soddisfazione dei vincoli locali.

Euristiche: si ricordi che gli algoritmi si dicono *"non informati"* se non hanno conoscenza specifica del problema, *"informati"* altrimenti. Il corpus della conoscenza specifica di un problema, derivato da analisi, studio o esperienza, forma l'***euristica*** che può essere sfruttata per guidare la ricerca. Nel caso in esame:

- si consideri che un esercizio è *ripetibile* (visto dalla prospettiva dello zainetto, è come se un oggetto possa essere preso un numero arbitrario di volte). La ripetizione di un esercizio (non necessariamente in sequenza) può essere talora un vantaggio (se conviene in termini di

punteggio), talora uno svantaggio, se preclude, a causa dei vincoli, altri esercizi "appetibili". In concreto, è probabile, usando strategie greedy relativamente semplici, che un numero elevato di esercizi ripetuti porti a violare uno dei vincoli su esercizi acrobatici. La ripetizione tende inoltre a rendere meno equilibrata l'accettabilità in relazione ai vincoli (locali e globali): se un esercizio è ripetuto più volte, saranno più probabili, ad esempio come esercizio successivo, quelli compatibili (come direzione). Si può quindi pensare di introdurre un ulteriore criterio di appetibilità (oppure un vincolo, a seconda di come lo si voglia porre): privilegiare (o forzare) esercizi non ripetuti

- un ulteriore criterio *euristico* sfruttabile è il cambio di direzione. Siccome gli esercizi possono essere divisi in due sottoinsiemi disgiunti, quelli che iniziano in avanti e quelli che iniziano indietro, un esercizio che cambia direzione può essere considerato vantaggioso (appetibile) nel momento in cui non sia stato possibile trovare (localmente) un esercizio che soddisfi i vincoli (o un criterio di appetibilità).

La soluzione proposta

Si adotta (tra le varie possibili) una strategia a "livelli", che organizza più criteri di appetibilità, in modo dinamico, secondo uno schema prioritario:

- si cerca l'ottimo locale in base a un primo criterio (cercare di soddisfare i vincoli globali),
- se non lo si trova (o non si usa il primo criterio) si adotta un criterio di livello più basso (appetibilità basata su valore).
- Si usano ulteriori criteri euristici (di buon senso), ad esempio la non ripetizione di esercizi, orientati ad aumentare la probabilità di trovare una soluzione. Tali criteri derivano da un esame empirico/teorico del problema e sono anch'essi finalizzati ad aumentare la probabilità di successo dei problemi di ricerca e/o di ottimizzazione.

A) ***Appetibilità degli esercizi acrobatici***. Vista l'impossibilità di ritornare su scelta fatte in precedenza (il backtrack della ricorsione), si cerca di soddisfare innanzi tutto i vincoli di presenza di esercizi acrobatici:
   - come compromesso di alto livello, si sceglie di ripartire i tre vincoli su tre diagonali: una deve contenere un esercizio acrobatico in avanti, una diagonale deve contenerne uno all'indietro, la terza deve contenere due esercizi acrobatici. Sono sicuramente possibili altre strategie, in quanto i vincoli potrebbero essere soddisfatti da altre configurazioni, ma quella proposta costituisce un ragionevole compromesso che tiene conto dei tre vincoli sugli esercizi acrobatici
   - data una diagonale, il vincolo di avere uno (di un dato tipo) o due esercizio acrobatici in sequenza, viene considerato prioritario, in termini di appetibilità, rispetto al valore.

B) ***Appetibilità del valore***. Qualora non si debba (in quanto già soddisfatto a livello di diagonale) o non si possa (in quanto non soddisfacibile a livello di singola scelta) trovare un esercizio acrobatico, si attua una ricerca dell'elemento di massima appetibilità: visto il contesto e la dinamicità dei vincoli, si evita un ordinamento a priori a si considera accettabile una ricerca O(N) del massimo. Per quanto riguarda il criterio di appetibilità, si propongono tre alternative: valore assoluto (alto), valore relativo (alto) o scostamento (basso) rispetto a un valore medio ricavato dal massimo ammissibile per una diagonale, diviso per il numero massimo di esercizi.

Questo terzo criterio può essere considerato come il risultato uno studio più approfondito del problema.

C) ***Euristiche***. Si sono poi aggiunti ulteriori criteri di appetibilità (o vincoli) di buon senso:
- *ripetizioni*. Si evita di ripetere lo stesso esercizio due volte in una diagonale (pur se possibile, si tende a soluzioni troppo ripetitive spesso non valide). Si usa questo criterio come vincolo (lo si sarebbe potuto utilizzare come semplice appetibilità
- *cambio di direzione*. Qualora non si riesca a trovare un esercizio acrobatico (prioritario in termini di appetibilità), se ne accetta uno non acrobatico che modifichi la direzione di entrata del prossimo esercizio. Questo con lo scopo di cambiare completamente l'insieme di esercizi validi alla prossima scelta (tra i quali se ne potrebbero trovare di acrobatici).

BONUS. Si noti infine che, nella soluzione proposta NON si è utilizzato alcun criterio particolare orientato ad aumentare la probabilità di ottenere il bonus. Si fa semplicemente la correzione del punteggio finale, in caso di bonus raggiunto. Notando tuttavia che qualunque permutazione di tre diagonali è una soluzione valida del problema, si valuta il bonus in una qualunque delle tre diagonali, scegliendo (eventualmente) la migliore come ultima nel programma finale.

IN CONCLUSIONE
A seconda del gruppo di esercizi (letti da file) la soluzione può esistere o no. L'ottimo NON è garantito. Modificando le costanti DMAX e PMAX e/o il criterio di appetibilità scelto per il valore, si può passare da casi di soluzioni valide a soluzioni non esistenti.

Esercizio n. 4: Rete di elaboratori
Si osservino le seguenti differenze rispetto all'es. 4.3.3 proposto in *Algoritmi e programmazione in pratica:*
- il formato del file è diverso, in quanto consta solo di una sequenza di archi. Non sono quindi noti né il numero di vertici, né gli identificatori dei vertici. Si realizza quindi una funzione di lettura ad hoc `read_file`
- in fase di caricamento si genera una rappresentazione a matrice delle adiacenze e non a lista delle adiacenze. Si tiene conto (variabile `listgen`) se è stato richiesto di generare la rappresentazione a liste di adiacenza per poterla liberare in uscita dal `main`
- è richiesta una visualizzazione di vertici e archi in ordine alfabetico. A scopo si realizza la tabella di simboli come vettore ordinato
- è richiesta la verifica di completezza di un sottografo formato da 3 vertici noti
- non è richiesto il calcolo dei flussi i*ntra* e *inter*-rete.

Strutture dati:
- **Quasi ADT** `Item`: tipologia 3 con 2 campi stringhe `elab` e `net` (vettori di caratteri sovradimensionati) con funzioni di caricamento (`ITEMload`), visualizzazione (`ITEMstore`), creazione di item vuoto (`ITEMsetNull`), lettura della chiave che è il campo `elab` (`KEYscan`), estrazione della chiave (`KEYget`) e comparazione tra chiavi (`KEYcmp`)
- **ADT di I classe** `ST`**:** tabella di simboli esterna al grafo implementata come vettore ordinato di item con funzioni di inizializzazione (`STinit`), liberazione (`STfree`), calcolo della

dimensione (`STcount`), inserzione ordinata di un item (`STinsert`), ricerca dicotomica per chiave (`STsearch`), visualizzazione di un item con indice dato (`STdisplayByIndex`). L'implementazione a vettore permette naturalmente l'accesso ad un item dato il suo indice. L'ordinamento risponde alla richiesta di elencare i vertici ed i vertici ad essi adiacenti in ordine alfabetico

- **ADT di I classe** `Graph`**:** è un grafo non orientato e non pesato, con doppia memorizzazione come lista delle adiacenze e come matrice delle adiacenze con funzioni di inizializzazione (`GRAPHinit`) e liberazione (`GRAPHfree`). Non si propone un tipo per l'arco, che viene gestito direttamente nel grafo. La funzione esportata `GRAPHmat2list`genera la lista a partire dalla matrice delle adiacenze. Le altre funzioni operano sulla rappresentazione a matrice. L'ADT grafo esporta una funzione di inserzione di arco (`GRAPHinsertE`), di visualizzazione (`GRAPHstore`) in ordine alfabetico di vertice e dei vertici ad essi adiacenti e di controllo di 3-clique (`GRAPHcheck3clique`), banalmente realizzata verificando l'adiacenza delle 3 coppie di vertici. Si osservi come la verifica se 2 nodi sono adiacenti (funzione `adjacent`) abbia complessità O(1) con la rappresentazione a matrice. Essendo la tabella di simboli esterna al grafo, è compito del `main` fornire al grafo le informazioni sui vertici come interi.

Algoritmo: il `main` inizializza le strutture dati (grafo e tabella di simboli) e offre all'utente un menu di operazioni. Dalla tabella di simboli estrae le informazioni da passare alle funzioni del grafo. Le operazioni richieste sono realizzate direttamente dalle funzioni esportate dagli ADT.