Commento all'es. 1

L'esercizio presenta forti similitudini con il problema 6.5 (capitolo "problemi risolti") del libro "Ricorsione e problem-solving", in cui gli argomenti corrispondono alle categorie e le domande agli esercizi. Le ulteriori differenze consistono:

- nel fatto che l'esercizio 6.5 prevede di scegliere per ogni argomento un numero fisso di domande (mentre per ogni categoria si sceglie un numero compreso tra un valor minimo e un massimo di elementi)
- nel criterio di ottimalità: avvicinarsi a una difficoltà obiettivo nel caso delle domande, punteggio massimo, rispettando un criterio di difficoltà massima, nel caso degli esercizi
- nella eventuale ripetizione di un elemento (le domande non possono invece essere ripetute).

L'esercizio può essere risolto secondo più di uno schema/modello. Si tratta infatti di un caso in cui la soluzione è data dall'unione di sotto-insiemi di elementi delle varie categorie (argomenti in 6.5). Il problema può quindi essere risolto:

a) a due livelli, calcolando prima, per ogni categoria, dei sottoinsiemi (ammissibili) di elementi; successivamente, la soluzione viene calcolata mediante principio di moltiplicazione, scegliendo, per ogni categoria, uno dei sotto-insiemi precedentemente calcolati.
b) a un solo livello, mediante una funzione di visita ricorsiva dello spazio delle soluzioni, ad esempio mediante una variante del calcolo del powerset.

Si propongono, in linea con le strategie citate (casi a e b) tre soluzioni. Le prime due in linea col caso a, la terza con il caso b:

a) soluzioni basate sul doppio livello
    1. soluzione **a1** simile alla soluzione dell'esercizio 6.5 del libro "Ricorsione e problem-solving"
    2. soluzione **a2** basata su ADT di prima classe per rappresentare insiemi di elementi e insiemi di sotto-insiemi di elementi
b) soluzione **b** a un solo livello realizzato mediante variante del powerset con albero di ricorsione binario.

Per una spiegazione più dettagliata si suggerisce di seguire la videolezione n. 59.

Soluzione a1
Strutture dati:

- una struttura dati in grado di rappresentare categorie contenenti insiemi di elementi: il tipo `elemento` rappresenta un elemento, il tipo `categoria` una categoria, contenente un vettore di elementi, mentre il tipo `categorie` è la collezione di categorie
- a questi si aggiungono 2 strutture dati in grado di rappresentare:
    - gli insiemi di elementi di una data categoria (tipo `combinazione`, un vettore di interi)
    - la collezione di tutti gli insiemi di elementi (di **una** categoria) generati nella prima fase dell'algoritmo (tipo `combinazioniCat`)

- la collezione di tutti di tutti gli insiemi di elementi (di **tutte** le categorie) generati nella prima fase dell'algoritmo e utilizzati nella seconda come dati di partenza (tipo `combinazioni`).

L'algoritmo utilizza due funzioni ricorsive:

- `combRipetizione` genera, per ogni categoria, l'insieme delle combinazioni con ripetizione di elementi, con cardinalità compresa tra i valori minimi e massimi (di categoria) previsti. Per ogni combinazione vengono calcolati, e memorizzati insieme alla combinazione, i valori di difficoltà e di punteggio complessivi. Una combinazione viene rappresentata come vettore di interi (indici a elementi)
- `princMolt` applica il principio di moltiplicazione agli insiemi di combinazioni precedentemente generati, scegliendone uno per categoria.

Soluzione a2
La soluzione proposta, pur se simile alla precedente come strategia, si differenzia in termini di strutture dati, in quanto sfrutta due ADT, in grado di rappresentare insiemi di numeri interi (ADT SOL) e vettori di tali insiemi (ADT ARRAY). Gli interi vengono usati come indici, per far riferimento a elementi e a combinazioni (sottoinsiemi) di elementi (di una categoria). Un insieme di interi viene quindi utilizzato sia per rappresentare un insieme di elementi (interi che fungono da indici di singoli elementi) che un insieme di combinazioni di elementi (gli interi sono indici a combinazioni in un ARRAY). La soluzione proposta si caratterizza inoltre per l'utilizzo di wrapper che evitano il passaggio di più parametri alle funzioni ricorsive.

Soluzione b
La soluzione proposta utilizza una sola funzione ricorsiva (`powersetModificato`), che visita il powerset degli elementi (nella loro globalità) accettando al massimo una ripetizione per elemento (in quanto, mentre la prima ripetizione ha punteggio dimezzato, la seconda ripetizione risulterebbe a punteggio 0). Si applica poi pruning con filtri sulla difficoltà massima ammessa, nonché una forma semplificata di "branch and bound" che filtra (mediante pruning) le soluzioni già peggiori delle migliore trovata (quelle a cui l'aggiunta di un elemento farebbe superare il punteggio migliore già ottenuto).

Soluzione greedy
Si cerca di costruire una soluzione rispettando (almeno) i vincoli minimi per la sua validità. Sono proposte 4 alternative:

1. si ottimizza una categoria alla volta (fino a `min_i`): si cerca di soddisfare i vincoli di una categoria fino a `min_i` prima di passare alla successiva. Le categorie incontrate "più tardi" sono implicitamente svantaggiate, dato che hanno a disposizione solo la difficoltà residua non usata per soddisfare i vincoli sulle categorie che le precedono
2. si cerca di soddisfare i vincoli di una categoria (almeno) fino a `min_i` assegnando esercizi a rotazione (tecnica *Round Robin*) sulle categorie, aggiungendo un esercizio alla volta a ogni categoria. Una volta scelto un esercizio per la categoria `i`, si passa alla successiva (o alla 0, giunti all'ultima). In questo modo si tenta di ridurre lo svantaggio che caratterizza invece la precedente tecnica. Il ciclo sulle categorie è ripetuto finché c'è disponibilità di difficoltà e finché è possibile trovare una soluzione migliorante.

L'algoritmo usa una variabile `miss` che incrementa ogni volta che una certa categoria giunge a una configurazione stabile. A fronte di una modifica alla soluzione, `miss` viene riportato a 0, forzando di fatto un ulteriore check per ognuna delle categorie. Esiste ancora un bias rispetto all'ordinamento delle categorie, ma è meno marcato rispetto alla prima tecnica

3. Round Robin con priorità sulle categorie, aggiungendo un esercizio a quella con meno punti Per evitare che l'algoritmo rimanga incastrato nell'estrazione ripetuta di una categoria che non ammette altre modifiche la variabile `miss` viene affiancata da un vettore di flag lungo quanto il numero di categorie
4. Round Robin con priorità sulle categorie, aggiungendo un esercizio a quella con meno esercizi Per evitare che l'algoritmo rimanga incastrato nell'estrazione ripetuta di una categoria che non ammette altre modifiche la variabile `miss` viene affiancata da un vettore di flag lungo quanto il numero di categorie. Con questo approccio si punta a rispettare esattamente i vincoli sul minimo, senza provare a fare di meglio.

Commento all'es. 2

Si definisce un quasi ADT `Item` con le seguenti funzioni:

`Item ITEMscan(FILE *fp);` `void ITEMstore(FILE *fp, Item x);`	funzioni per lettura/scrittura di item da/su file
`Item ITEMnew(Key k, float p);`	funzioni di creazione di un nuovo item a partire da chiave e priorità
`void ITEMchange(Item *a, Item *b, float ratio);`	funzioni di modifica delle priorità di due item in base alle regole del gioco
`int ITEMzero(Item a);`	funzione di controllo se priorità può considerarsi nulla

L'item corrisponde alla tipologia 3 presentata a pag. 163 del testo *Puntatori e strutture dati dinamiche.*

Poiché la rimozione di un elemento dalla coda a priorità è fatta sulla base dell'identificatore, si definisce un tipo chiave:
`typedef char *Key;`
con le seguenti funzioni:

`Key KEYscan();`	funzione per lettura di chiave da tastiera
`Key KEYget(Item *x);`	funzione di estrazione della chiave da un item
`float GETprio(Item *x);`	funzione di estrazione della priorità da un item
`int KEYeq(Key k1, Key k2)`	funzione di controllo di uguaglianza tra

	chiavi

Si definisce un quasi ADT `PQ` con le seguenti funzioni:

`void PQinit();`
`int PQempty();`
`int PQsize();`
	funzioni standard di inizializzazione, controllo se vuoto e dimensione

`void PQload(FILE *fp);` `void PQstore(FILE *fp);`	funzioni di lettura/scrittura da/su file
`void PQinsert(Item x);`	funzioni di inserzione di un item nella coda a priorità
`void PQremove(Key k);` `Item PQreadMin();`	funzioni di estrazione di un item in base alla chiave e di lettura della chiave a priorità minima. L'estrazione della chiave a priorità minima si realizza con la chiamata in sequenza di `PQreadMin` e poi di `PQremove`
`void PQsort();`	funzione che visualizza la coda a priorità per priorità decrescenti. Non s ordina la lista, bensì si crea un vettore parallelo contenente gli stessi item e lo si ordina.

L'implementazione delle funzioni è evidente dal codice presentato come soluzione.

Commento all'es. 3

Gli ADT di I categoria per gli ingredienti e le ricette esportano sia il singolo ingrediente/ricetta, sia la collezione.

ADT di I categoria per gli ngredienti:

- la struttura dati wrapper `tabIngr_` contiene la collezione come numero di ingredienti e vettore `vettIng`, le cui celle sono di tipo `ingrediente`
- il tipo `ingrediente_` è una struttura con campi nome, calorie, prezzo, nonché 2 puntatori a testa e coda di una lista di nodi di tipo `elenco`. Questa lista serve a memorizzare le ricette in cui compare quell'ingrediente. Onde evitare riferimenti circolari, un ingrediente conosce le ricette di cui fa parte solo per nome. Nell'esercizio 2 del Lab. 8 in questa lista comparivano puntatori alle ricette. Si osservi che, se fosse possibile cambiare il nome di una specifica ricetta, sarebbe necessaria una verifica di consistenza, cambiando la ricetta in tutti gli elenchi in cui compare
- la collezione `ingredienti` ha visibilità dei dettagli di un certo ingrediente (sono moduli allo stesso livello gerarchico) e quindi può allocare il vettore `vettIng`

ADT di I categoria per le ricette:

- la struttura dati wrapper `tabRicette_` contiene il numero di ricette e 2 puntatori a testa e coda di una lista di nodi di tipo `ricetta_p`

- ogni nodo di tipo `ricetta_p` contiene, oltre al puntatore al prossimo nodo, i campi per il nome, il numero di ingrediente, le calorie, il tempo e il costo, nonché due puntatori a testa e coda di una lista di nodi di tipo `rIng`. Questa lista serve a memorizzare gli ingredienti che compaiono in quella ricetta. Onde evitare riferimenti circolari, una ricetta conosce le ricette di cui fa parte solo per nome. Nell'esercizio 2 del Lab. 8 in questa lista comparivano puntatori alle ricette.
- ogni nodo di tipo `rIng` contiene, oltre al puntatore al prossimo nodo, anche un puntatore ad un ingrediente, cioè ad una cella che fa parte del vettore degli ingredienti.

Funzioni: il main legge prima la collezione di ingredienti (`INGleggiCollezione`) e poi quella di ricette (`RICleggiCollezione`). La lettura della collezione degli ingredienti non presenta particolarità da commentare e si basa sull'iterazione sul singolo ingrediente (`INGleggi`). La lettura della collezione delle ricette si basa sull'iterazione sulla singola ricetta (`RICleggi`). Quando si legge una ricetta, per ogni ingrediente che vi compare, si identifica tramite una ricerca con `INGcerca` il suo puntatore nella collezione `ingredienti` e si aggiorna la lista degli ingredienti per tale ricetta. Tramite `INGaggiornaRicette` si aggiunge il nome della ricetta nell'ingrediente opportuno. In seguito si inserisce il nodo con le sue informazioni nella lista delle ricette di quell'ingrediente.

Le funzioni di `INGcerca`, `INGstampaCollezione`, `INGstampaRicette` sono semplici scansioni lineari di liste o di vettori. Le funzioni `INGstampa`, `INGgetPrezzo`, `INGgetCalorie` e `INGgetNome` sono banali.

Le funzioni di `RICcerca`, `RICstampaCollezione` e `RICstampa` sono semplici scansioni lineari di liste o di vettori. Le funzioni `RICgetPrezzo`, `RICgetCalorie` e `RICgetNome` sono banali. La funzione `RICcrea` richiama la `RICleggi` e procede poi ad una semplice inserzione in coda alla lista.