










Esercitazione di laboratorio n. 10

Esercizio svolto n. 0: antenne della rete di telefonia mobile

È dato un insieme di n città, disposte su una strada rettilinea, identificate con gli interi da 1 a n , ognuna caratterizzata dal numero di abitanti (migliaia, intero). In ogni città si può installare un'antenna della rete di telefonia mobile alla sola condizione che le città adiacenti (precedente e successiva, se esistono) non abbiano l'antenna. Ogni antenna copre solo la popolazione della città dove è posta.

							
antenne possibili	1	2	3	4	5	6	7
città							
abitanti	14	22	13	25	30	11	90

Con il paradigma della programmazione dinamica bottom-up, determinare il massimo numero di abitanti copribile rispettando la regola di installazione e la corrispondente disposizione delle antenne. Visti i vincoli, è evidente che non si potrà coprire tutta la popolazione di tutte le città

Svolgimento:

si tratta di un problema di ottimizzazione che potrebbe essere risolto identificando tutti i sottoinsiemi di antenne, valutando quelli che soddisfano la regola e, tra questi, quello ottimo. Il modello è quello del powerset.

In alternativa si propone una soluzione basata sul paradigma della **programmazione dinamica**. I dati sono memorizzati in un vettore di interi `val` di $n+1$ celle. La cella di indice 0 corrisponde alla città fittizia che non esiste e che non ha abitanti.

val	0	14	22	13	25	30	11	90
	0	1	2	3	4	5	6	7

Passo 1: applicabilità della programmazione dinamica

Ci si posiziona sulla città di indice k e si guarda all'indietro. Si osserva che è vera la seguente affermazione: la soluzione ottima del problema per la città di indice k corrisponde a uno dei seguenti 2 casi

- nella città k non c'è un'antenna: la soluzione ottima coincide con quella per le prime $k-1$ città
- nella città k c'è un'antenna: la soluzione ottima si ottiene dalla soluzione ottima per le prime $k-2$ città cui si aggiunge l'antenna nella città k .

Supponiamo di memorizzare in un vettore di interi `opt` di $n+1$ celle scandito da un indice k la soluzione ottima che si ottiene considerando le prime k antenne: `opt[0]=0` in quanto non ci sono né città, né abitanti, né antenne; `opt[1]=val[1]` in quanto c'è solamente la città di indice $i=1$ con i suoi abitanti `val[1]`. Per gli altri casi $1 < k \leq n$:



- è possibile piazzare un'antenna nella città k , quindi non è possibile piazzare un'antenna nella città $k-1$ che la precede, bensì nella città ancora prima $k-2$: $opt[k] = opt[k-2] + val[k]$
- non è possibile piazzare un'antenna nella città k , quindi è possibile piazzare un'antenna nella città $k-1$ che la precede: $opt[k] = opt[k-1]$.

Il problema per la città k -esima richiede la soluzione dei sottoproblemi per le città $(k-1)$ -esima o $(k-2)$ -esima. Se $opt[k-1]$ o $opt[k-2]$ non fossero massimi, si potrebbero trovare soluzioni $opt'[k-1] > opt[k-1]$ o $opt'[k-2] > opt[k-2]$ che contraddirebbero l'ipotesi di $opt[k]$ massimo. La programmazione dinamica è quindi applicabile.

Passo 2: soluzione ricorsiva

L'analisi precedente può essere riassunta con la seguente formulazione ricorsiva:

$$opt(k) = \begin{cases} 0 & k = 0 \\ val[1] & k = 1 \\ \max(opt(k-1), val[k] + opt(k-2)) & 1 < k \leq n \end{cases}$$

facilmente codificata in C come:

```
int solveR(int *val, int *opt, int n, int k) {
    if (k==0)
        return 0;
    if (k==1)
        return val[1];
    return max(solveR(val, opt, n, k-1), solveR(val, opt, n, k-2) + val[k]);
}

void solve(int *val, int n) {
    int *opt;
    opt = calloc((n+1), sizeof(int));
    printf("Recursive solution: ");
    printf("maximum population covered %d\n", solveR(val, opt, n, n));
}
```

Questa soluzione porta alla seguente equazione alle ricorrenze:

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

identica a quella vista lezione per i numeri di Fibonacci, dunque la soluzione ricorsiva ha complessità esponenziale.

Passo 3: soluzione con programmazione dinamica bottom-up (calcolo del valore della soluzione ottima)

Ispirandosi alla formulazione ricorsiva della soluzione, la si trasforma in forma iterativa:

- $opt[0]$ e $opt[1]$ sono noti a priori,
- per $2 \leq i \leq n$ $opt[i] = \max(opt[i-1], opt[i-2] + val[i])$.



```
void solveDP(int *val, int n) {
    int i, *opt;
    opt = calloc((n+1), sizeof(int));
    opt[1] = val[1];
    for (i=2; i<=n; i++) {
        if (opt[i-1] > opt[i-2]+val[i])
            opt[i] = opt[i-1];
        else
            opt[i] = opt[i-2] + val[i];
    }
    printf("Dynamic programming solution: ");
    printf("maximum population covered %d\n", opt[n]);
    displaySol(opt, val, n);
}
```

Passo 4: costruzione della soluzione ottima

La funzione `displaySol` costruisce e visualizza la soluzione (città dove si installa un'antenna). Essa utilizza un vettore di interi `sol` di $n+1$ elementi per registrare se l'elemento i -esimo appartiene o meno alla soluzione. La decisione viene presa in base al contenuto del vettore `opt` e viene costruita mediante una scansione da destra verso sinistra, in verso quindi opposto alla scansione con cui `opt` è stato riempito dalla funzione `solveDP`. Il criterio per assegnare 0 o 1 alla cella corrente di `sol` rispecchia quello usato in fase di risoluzione:

- `sol[1]` è assunto valere 1, salvo modificare questa scelta nel corso dell'iterazione successiva
- se `opt[i]==opt[i-1]` è certo che nella città i -esima non è stata piazzata un'antenna, quindi `sol[i]=0`, mentre non si può dire nulla di `sol[i-1]`. L'iterazione quindi prosegue sulla città $(i-1)$ -esima
- se `opt[i] == opt[i-2] + val[i]` è certo che:
 - nella città i -esima è stata piazzata un'antenna, quindi `sol[i]=1`
 - nella città $(i-1)$ -esima non è stata piazzata un'antenna, quindi `sol[i-1]=0`avendo preso una decisione sia per la città i -esima che per la città $(i-1)$ -esima, l'iterazione prosegue sulla città $(i-2)$ -esima.

```
void displaySol(int *opt, int *val, int n){
    int i, j, *sol;
    sol = calloc((n+1), sizeof(int));
    sol[1]=1;
    i=n;
    while (i>=2) {
        printf("i=%d\n", i);
        if (opt[i] == opt[i-1]){
            sol[i] = 0;
            i--;
        }
        else if (opt[i] == opt[i-2] + val[i]) {
            sol[i] = 1;
            sol[i-1] = 0;
            i -=2;
        }
    }
}
```



```
for (i=1; i<=n; i++)  
    if (sol[i])  
        printf("%d ", val[i]);  
printf("\n");  
}
```

Esercizio n. 1: Sequenza di attività (versione 2)

Si consideri la situazione introdotta nell'esercizio n.1 del laboratorio 8. Si proponga una soluzione al medesimo problema sfruttando il paradigma della programmazione dinamica.

Suggerimento:

- si ordinino le attività lungo una linea temporale. Si definisca il criterio di ordinamento in base al vincolo del problema (gli intervalli della soluzione non devono intersecarsi)
- ispirandosi alla soluzione con programmazione dinamica del problema della Longest Increasing Sequence, si costruiscano soluzioni parziali considerando solamente le attività fino all' i -esima nell'ordinamento di cui sopra, definendo opportunamente secondo quale criterio considerare le attività. L'estensione di soluzioni parziali con l'introduzione di una attività aggiuntiva può essere fatta sulla base della compatibilità tra la "nuova" i -esima attività e le soluzioni già note ai problemi con le $i-1$ considerate precedentemente
- si seguano i passi svolti nell'esercizio precedente (dimostrazione di applicabilità, calcolo ricorsivo del valore ottimo, calcolo con programmazione dinamica bottom-up del valore ottimo e della soluzione ottima).

Esercizio n.2: Collane e pietre preziose (versione 3)

Si risolva l'esercizio n.1 del laboratorio 8 mediante il paradigma della memoization. Si richiede soltanto di calcolare la lunghezza massima della collana compatibile con le gemme a disposizione e con le regole di composizione.

Suggerimento: affrontare il problema con la tecnica del divide et impera, osservando che una collana di lunghezza P può essere definita ricorsivamente come:

- la collana vuota, formata da $P=0$ gemme
- una gemma seguita da una collana di $P-1$ gemme.

Poiché vi sono 4 tipi di gemme Z, R, T, S, si scrivano 4 funzioni f_Z , f_R , f_T , f_S che calcolino la lunghezza della collana più lunga iniziante rispettivamente con uno zaffiro, un rubino, un topazio e uno smeraldo avendo a disposizione z zaffiri, r rubini, t topazi e s smeraldi. Note le regole di composizione delle collane, è possibile esprimere ricorsivamente il valore di una certa funzione f_X sulla base dei valori delle altre funzioni.

Si presti attenzione a definire adeguatamente i casi terminali di tali funzioni, onde evitare di ricorrere in porzioni non ammissibili dello spazio degli stati.

Si ricorda che il paradigma della memoization prevede di memorizzare le soluzioni dei sottoproblemi già risolti in opportune strutture dati da progettare e dimensionare e di riusare dette soluzioni qualora si incontrino di nuovo gli stessi sottoproblemi, limitando l'uso della ricorsione alla soluzione dei sottoproblemi non ancora risolti.



Esercizio n. 3: Gioco di ruolo (multi-file, con ADT)

[Esercizio guidato: si forniscono architettura dei moduli ed esempi di file .h]

A partire dal codice prodotto per l'esercizio n. 3 del laboratorio 8, lo si adatti così che sia il modulo `personaggi` sia il modulo `inventario` risultino ADT

La specifica “nessuna statistica può risultare minore di 1, indipendentemente dagli eventuali *malus* cumulativi dovuti alla scelta di equipaggiamento” può essere interpretata nelle seguenti 2 modalità, entrambe accettate:

- si impedisce la scelta di un equipaggiamento se il *malus* porta almeno una delle statistiche sotto il valore 1
- si ammette la scelta di *malus* che portano almeno una statistica a valori negativi o nulli, ma in questo caso questa statistica viene “mascherata” in fase di stampa, dove si stampa il valore fittizio 1, quando il valore è negativo o nullo.

Si tenga conto che esistono:

- un tipo di dato per un oggetto e un tipo di dato per un vettore di oggetti (`inventario`)
- un tipo di dato per un personaggio e un tipo di dato per una lista di personaggi
- un tipo di dato per un vettore di (riferimenti a) oggetti (`equipaggiamento`)

Si chiede di realizzare tutti i tipi di dato come ADT, utilizzando:

- la versione “quasi ADT” (`struct` visibile) per i tipi `personaggio` (`pg_t`) e oggetto (`inv_t`)
- la versione “ADT di I classe” per le collezioni, cioè il vettore di oggetti (`invArray_t`), la lista di personaggi (`pgList_t`) e gli equipaggiamenti (`equipArray_t`).

Per i riferimenti ad oggetti si evitino i puntatori, in quanto richiederebbero accesso a una struttura dati interna (all'ADT vettore di oggetti). Si utilizzino invece gli indici: ad un dato oggetto si fa quindi riferimento mediante un intero (progressivo a partire da 0) che ne identifica la posizione nel vettore dell'inventario.

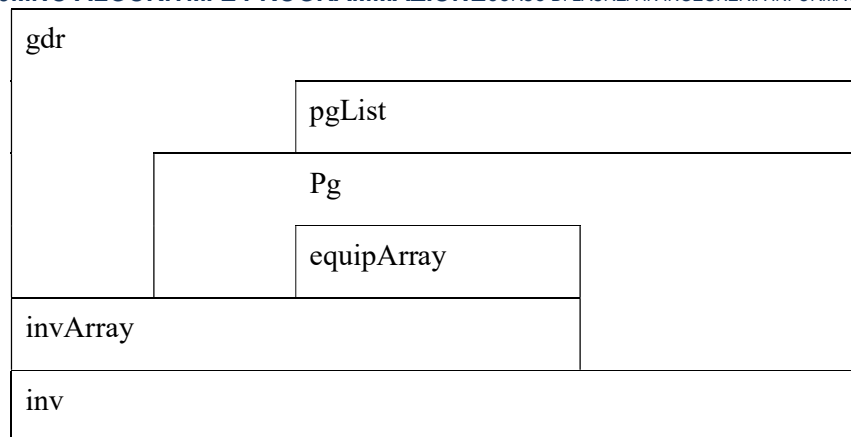
Pur potendo unificare l'ADT oggetto e il vettore inventario in un unico modulo, come pure l'ADT personaggio e l'ADT lista di personaggi, si consiglia di realizzare un modulo per ognuno degli ADT. Si realizzeranno quindi 5 file `.c` (`pg.c`, `inv.c`, `pgList.c`, `invArray.c`, `equipArray.c`) e 5 corrispondenti file `.h` per i moduli, più un `.c` per il client (ad esempio `gdr.c`).

I moduli `pg` ed `inv` realizzano tipi di dato composti per valore (sono quindi simili al tipo/ADT `Item` spesso usato in altri problemi): saranno ovviamente di dimensione ridotta.

I moduli per collezioni di dati dovranno fornire operazioni di creazione/distruzione, più eventuali operazioni di input o output, ricerca, modifica e/o cancellazione. Eventuali altre operazioni possono essere fornite qualora ritenute necessarie.

Architettura proposta:

La soluzione all'esercizio va realizzata secondo l'architettura dei moduli rappresentata nella seguente figura:



Lo schema mostra (dal basso in alto):

- gli elementi dell'inventario (modulo `inv`), un item quasi ADT, tipo `inv_t`, composto da 3 campi (di cui uno, un secondo tipo struct, `stat_t`, anch'esso quasi ADT) contiene le statistiche da leggere e aggiornare. Si consiglia di vedere `inv.c` come esempio di quasi ADT Item (non vuoto!) contenente operazioni elementari sui tipi di dato gestiti
- il vettore di elementi (`invArray`), realizzato come ADT vettore dinamico di elementi `inv_t`.
- il vettore di riferimenti (mediante indici) a elementi dell'inventario (`equipArray`) viene realizzato come semplice struct dinamica, contenente un vettore di interi di dimensione fissa (non allocato dinamicamente). Il modulo dipende da (è client di) `invArray`, solamente in quanto i riferimenti mediante indici sono relativi a un ADT di tale modulo
- il quasi ADT (un item) `pg_t` (modulo `pg`). Il modulo è client di `inv`, `invArray` e `equipArray`. Si noti che il quasi ADT contiene un campo ADT (`equip`) riferimento a un `equipArray_t`. Si tratta quindi di un quasi ADT di tipo 4 (in tal senso rappresenta un'eccezione rispetto alla prassi consigliata, non evitabile in base alle specifiche dell'esercizio), avente cioè un campo soggetto ad allocazione e deallocazione. L'allocazione (`equipArray_init`) viene gestita nella funzione di lettura da file (`pg_read`), mentre per la deallocazione si è predisposta la funzione `pg_clean`, che chiama semplicemente la `equipArray_free` (il tipo `pg_t` non va deallocato, in quanto quasi ADT)
- l'ADT `pgList_t` (modulo `pgList`) è un ulteriore ADT di prima classe, che realizza una lista di elemento del modulo `pg`
- il modulo principale (`gdr`) è client di `pgList`, `pg` e di `invArray`.

Si allegano i `.h` dei vari moduli e il `.c` di `gdr`. A scelta, si possono eventualmente modificare i nomi di tipi e funzioni, nonché definizioni di tipi e parametri delle funzioni (pur di rispettare l'architettura proposta e le richieste).

Questo non è l'unico schema realizzabile, ma deriva dall'aver effettuato certe scelte di modularità e di ripartizione delle operazioni tra moduli. Si consiglia di esaminare le dipendenze tra i moduli, nonché le scelte fatte nell'assegnare operazioni e funzioni ai singoli moduli.

SI noti che le funzioni che gestiscono i quasi ADT in alcuni casi ricevono e/o ritornano struct, in altri casi riferimenti (puntatori) a struct (ad esempio le funzioni di input/output da file sono state spesso predisposte in modo da ricevere puntatori alla struct che riceve o da cui si prendono i dati coinvolti nell'IO. Sono possibili altre scelte (es. le struct passate sempre per valore).



ATTENZIONE

Si ricorda che un ADT di I classe NASCONDE i dettagli interni di un dato: NON E' QUINDI POSSIBILE A UN CLIENT ACCEDERE A TALI DETTAGLI: non sarà possibile, quindi la modifica dell'equipaggiamento di un dato personaggio da parte di un client in modo diretto, cioè ottenendo il puntatore all'elemento in lista, per accedere ai campi da modificare. Il client dovrà quindi, ad esempio, chiamare una opportuna funzione (fornita dal modulo `pgList`) avente come parametri il codice di un personaggio. Tale funzione, all'interno, effettuerà una ricerca dell'atleta e ne modificherà l'esercizio selezionato NON in modo diretto, ma chiamando, ad esempio, una funzione (fornita dal modulo `equipArray`), che riceve come parametri il nome dell'oggetto di interesse e il tipo di operazione da eseguire. Quest'ultima funzione cerca l'oggetto e modifica lo stato della struttura dati di conseguenza.

**Valutazione: uno a scelta tra gli esercizi 1 e 2 e l'esercizio 3 saranno oggetto di valutazione
Scadenza: caricamento di quanto valutato: entro le 23:59 del 18/12/2020.**