Esercitazione di laboratorio n. 6

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

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							
città	1	2	3	4	5	6	7
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 posizioni sulla città di indice `k` e si guardi all'indietro. Si osservi 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` <=`n`:

- è possibile piazzare un'antenna nella città `k`, quindi non è può essere piazzata 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 citta `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:

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

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));
}
```

$T(n) = T(n-1) + T(n-2)$

Questa soluzione porta alla seguente equazione alle ricorrenze:

$T(n) = 2T(n-1)$

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

$T(n-1) = 2T(n-2)$

$T(n-2) = 2T(n-3)$

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

$2^2 T(n-2)$

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:

$2^3 (+1)$

- `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])`.

$T(n) = 2^n$ esp

$T(n) \Theta(2^n)$

0	1	2	3	4	5	6	7
0	14	22	13	25	30	11	10


```
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);
}
```

	0	1	2	3	4	5	6	7
opt	0	14	22	27	47	47	58	57

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;
    }
  }
```

Ritroso

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

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

Si consideri la situazione introdotta nell'esercizio n.1 del laboratorio 5. 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.3 del laboratorio 4 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 `fZ`, `fR`, `fT`, `fS` 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 `fX` sulla base dei valori delle altre funzioni.

Si presti attenzione a definire adeguatamente i casi terminali di tali funzioni, onde evitare di ricorre 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.

Nella funzione ricorsiva generale chiamare le 4 funzioni, che mi calcolano per ogni combinazione la lunghezza maggiore, salvo la soluzione in una matrice a 4 dimensioni con la combinazione di gemme in quel momento.

· Ricorsione solo nella funzione ricorsiva.
· Controllo presenza soluzione in matrice all'inizio della funzione ricorsiva
· Controllo sulle scelte in ognuna delle 4 funzioni
Alla fine salvo quella effettivamente di lunghezza maggiore

Esercizio n.3: Gioco di ruolo

Competenze: Vettori di struct, strutture dati dinamiche, vettori dinamici, riallocazione dinamica (Puntatori e strutture dati dinamiche 2.5.5, 3.3.3, 3.2.3). Programmazione multifile (Puntatori e strutture dati dinamiche 5.5)

Sia dato un file di testo (pg.txt), contenente i dettagli di alcuni personaggi di un gioco di ruolo, organizzato come segue:

- il numero di personaggi presenti nel file non è noto a priori
- i dettagli di ogni personaggio sono riportati in ragione di uno per riga. In ogni riga sono presenti tre stringhe quali un codice identificativo univoco, il nome del personaggio e la sua classe. Segue sulla stessa riga una sestupla a rappresentare le statistiche di base del personaggio, nella forma `<hp> <mp> <atk> <def> <mag> <spr>`. Ai fini dell'esercizio non è rilevante conoscere il significato dei campi della sestupla,
- il codice è nella forma `PGXXXX`, dove `X` rappresenta una cifra nell'intervallo `0-9`
- il nome e la classe di ogni personaggio sono rappresentati da una stringa, priva di spazi, di massimo 50 caratteri alfabetici (maiuscoli o minuscoli)
- le statistiche sono dei numeri interi positivi o nulli
- tutti i campi sono separati da uno o più spazi.

In un secondo file di testo (`inventario.txt`), sono memorizzati i dettagli di una serie di oggetti a cui i personaggi del gioco hanno accesso. Il file è organizzato come segue:

- sulla prima è presente il numero `O` di oggetti
- sulle `O` righe successive appaiono i dettagli di ogni oggetto disponibile
- ogni oggetto è caratterizzato da un nome, una tipologia e da una sestupla a rappresentare i modificatori alle statistiche base di un personaggio
- il nome e la tipologia sono rappresentati da una stringa, priva di spazi, di massimo 50 caratteri alfabetici (maiuscoli o minuscoli)
- i modificatori alle statistiche sono dei numeri interi (potenzialmente anche negativi), quindi possono essere visti come *bonus* (se positivi) o *malus* (se negativi).

stesse sestuple dei personaggi

Ogni personaggio può fare uso degli oggetti disponibili nell'inventario e comporre liberamente il proprio equipaggiamento, fino ad un massimo di otto elementi. 8 elementi

Ai fini dell'esercizio, si imposti la struttura dati in modo che sia coerente con la rappresentazione grafica proposta in figura (i nomi dei tipi e dei campi sono solo a titolo di esempio). Se necessario, si aggiungano tutti i campi addizionali ritenuti opportuni. Si presti particolare attenzione all'uso di strutture *wrapper* per la memorizzazione delle collezioni (per personaggi, oggetti e equipaggiamenti).

Si scriva un programma in C che permetta di:

- caricare in una lista l'elenco di personaggi
- caricare in un vettore di strutture, allocato dinamicamente, l'elenco di oggetti
- aggiungere un nuovo personaggio
- eliminare un personaggio
- aggiungere/rimuovere un oggetto dall'equipaggiamento di un personaggio
- calcolare le statistiche di un personaggio tenendo in considerazione i suoi parametri base e l'equipaggiamento corrente, applicando quindi i bonus e i malus dell'equipaggiamento stesso. Se una certa statistica risultasse inferiore a 0 in seguito all'applicazione dei malus, in fase di visualizzazione si stampi 0, per convenzione, anziché il valore negativo assunto dalla statistica stessa.

Si organizzi il codice in più moduli, quali:

- un modulo client contenente il main e l'interfaccia utente/menu
- un modulo per la gestione dei personaggi
- un modulo per la gestione dell'inventario.

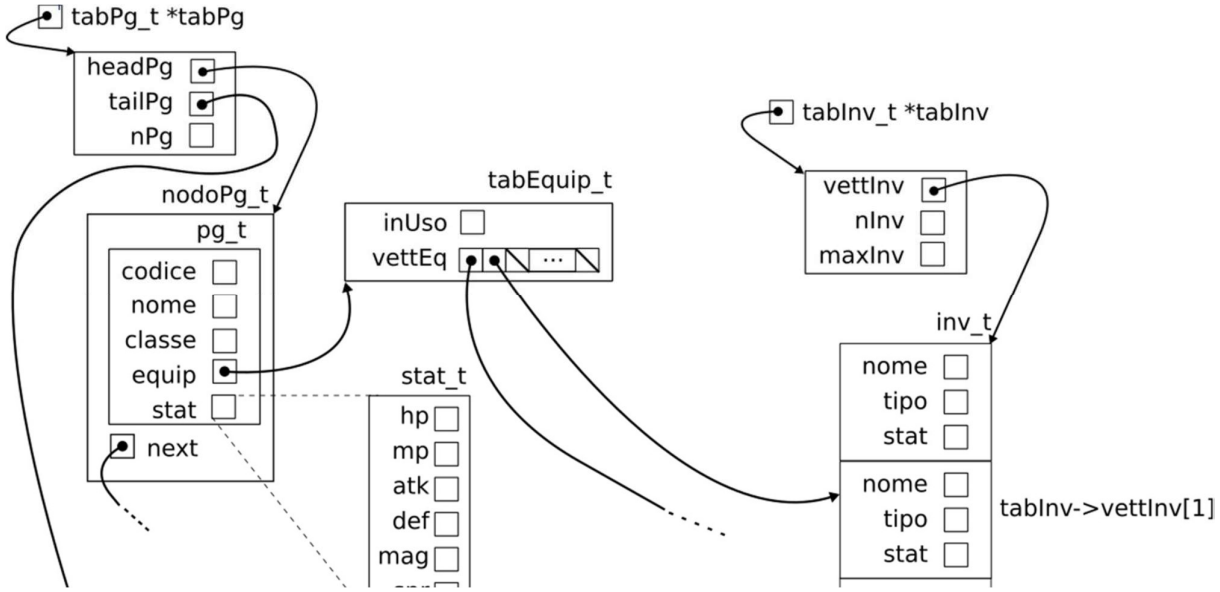
Il modulo per i personaggi deve fornire le funzionalità di:

- acquisizione da file delle informazioni dei personaggi, mantenendo la medesima struttura a lista richiesta nel laboratorio precedente
- inserimento/cancellazione di un personaggio
- ricerca per codice di un personaggio
- stampa dei dettagli di un personaggio e del relativo equipaggiamento, se presente
- modifica dell'equipaggiamento di un personaggio
    - aggiunta/rimozione di un oggetto.

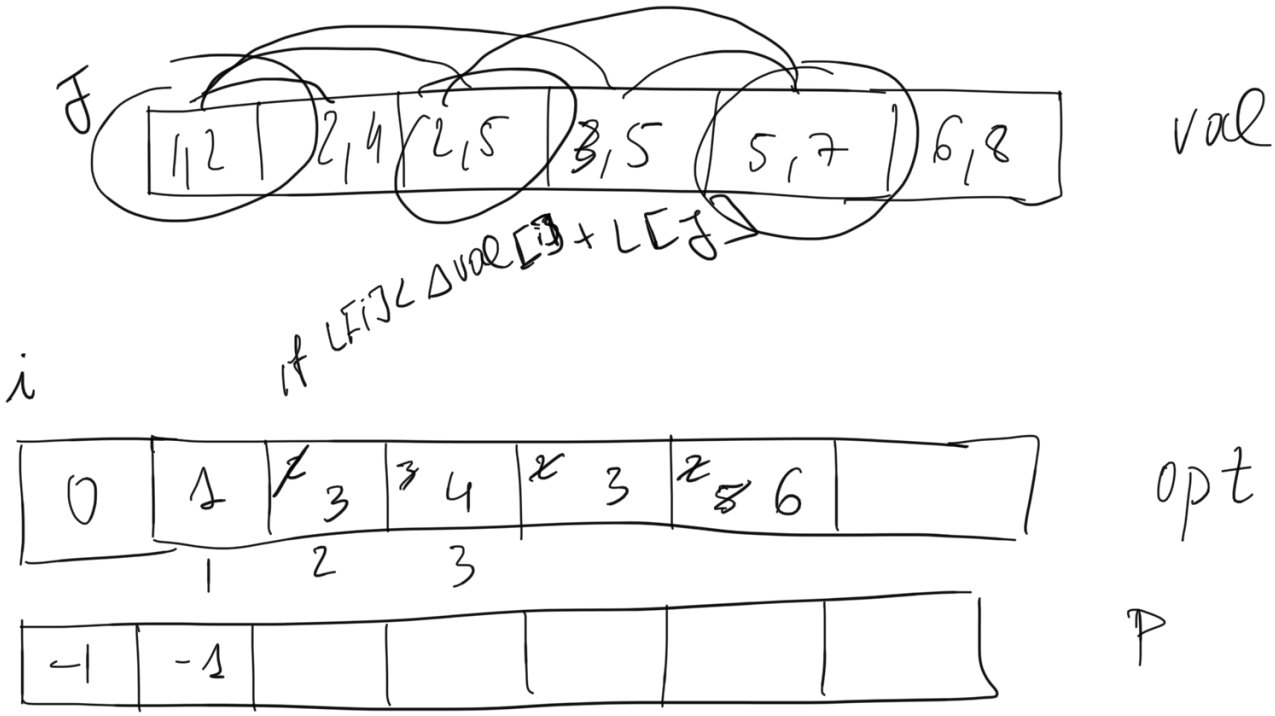
Il modulo dell'inventario deve fornire le funzionalità di:

- acquisizione da file delle informazioni relative agli oggetti disponibili, mantenendo la medesima struttura a vettore richiesta nel laboratorio precedente
- ricerca di un oggetto per nome
- stampa dei dettagli di un oggetto.

NB: il modulo personaggi è client del modulo inventario, in quanto ogni personaggio ha una collezione di (riferimenti a) dati contenuti nell'inventario.



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 16/12/2022.



- vettore opt per memorizzare il delta di ogni sequenza
- vettore P per memorizzare l'indice dell'elemento precedente
- last per memorizzare l'ultimo elemento (indice) della sequenza

<u>Passi</u>:

- Il valore minimo del Δ di una sequenza e' il Δ dell'elemento considerato
- dall'elemento corrente, guardo tutti gli elementi precedenti e controllo quale posso prendere e se posso se e' conveniente o meno

```
L[0] = 0        P[0] = -1
for (i=1; i<N; i++)
    L[i] = sval[i];
    P[i] = -1;
    for (j=0; j<i; j++)
        if ( no intersezione && (L[i] < sval[i]+L[j])

            L[i] = sval[i] + L[j];
            P[i] = j;
        if (ris < L[i])
            ris = L[i];
            last = i;
        }
```