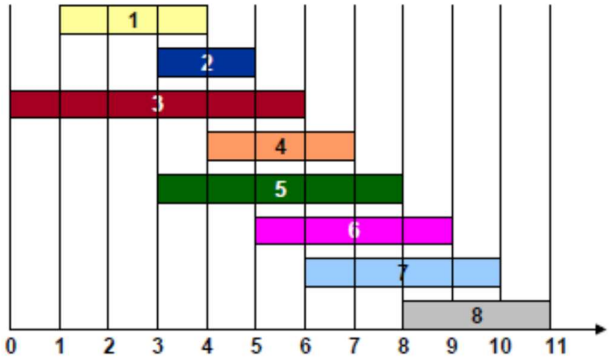
Commento al Laboratorio n. 6

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

Le `n` attività sono identificate da un indice `i` che inizia da 1. L'indice 0 indica l'attività 0 che è fittizia. Le attività sono memorizzate in un vettore di attività `v` (vedi es. 1 del Lab. 8) di `n+1` celle.

Il criterio di ordinamento è il tempo di fine crescente di ogni attività, conformemente a quanto fatto per la versione vista a lezione risolta con il paradigma greedy per il problema che mira a massimizzare il numero di attività (non la durata complessiva). Si usa il `MergeSort` per ordinare il vettore. Con riferimento al file di esempio `att1.txt`, il risultato di questo passo è riportato nella figura seguente:



Come secondo passo si identifica per ogni attività `i` quale è l'indice dell'attività che termina più tardi e che è compatibile con `i` e lo si memorizza in un vettore `q` di `n+1` celle, dove `q[0]=0` si riferisce all'attività fittizia 0. Per l'esempio di cui sopra il contenuto di `q` è:

0	0	0	0	1	0	2	3	5
0	1	2	3	4	5	6	7	8

Avere ordinato per tempo di fine crescente garantisce che tutte le attività che precedono l'attività che termina più tardi e che è compatibile con `i` siano a loro volta compatibili con `i`.

Passo 1: applicabilità

Si ipotizzi che `opt[i]` sia la soluzione ottima al problema in cui si considerano le attività da 1 a `i`:

- si ipotizzi che l'attività `i` faccia parte della soluzione. Il sottoproblema da risolvere è quello `q(i)`-esimo che prende in considerazione le attività da 1 a `q(i)`, quindi certamente compatibili con l'attività `i`-esima. Se la soluzione al problema `q(i)`-esimo non fosse ottima, se ne troverebbe una con valore `opt'[q(i)]` maggiore, che, sommato alla durata dell'attività `i`, porterebbe ad un valore `opt'[i]` > `opt[i]` contraddicendo l'ipotesi di `opt[i]` ottimo
- se l'attività `i` non fa parte della soluzione, `opt[i]=opt[i-1]` e se `opt[i-1]` non fosse ottimo non lo potrebbe neanche essere `opt[i]`.

Passo 2: soluzione ricorsiva divide et impera
L'analisi precedente può essere riassunta con la seguente formulazione ricorsiva:

$$opt(i) = \begin{cases} 0 & i = 0 \\ max(opt(i-1), f_i - s_i + opt(q(i))) & 1 \leq i \leq n \end{cases}$$

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]` è noto a priori,
- per $1 \leq i \leq n$ `opt[i] = max(opt[i-1], attDurata(v[i] + opt[q[i]])`.

Il valore della soluzione ottima è memorizzato in `opt[n]`.

Passo 4: costruzione della soluzione ottima
La funzione `displaySol` costruisce ricorsivamente e visualizza la soluzione ritracciando all'indietro (partendo da `pos=n`) quanto fatto per determinare il valore ottimo. La condizione di terminazione si raggiunge per `pos=0` e, trattandosi dell'attività fittizia, non si fa nulla, ma si ritorna.
Se è vera la condizione `attDurata(v[pos])+opt[q[pos]] >= opt[pos-1]` si ricorre per `pos=q[pos]` e al termine si stampa l'attività `v[pos]`, altrimenti si ricorre su `pos`-1.

Esercizio n. 2: **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. 3: Gioco di ruolo
Strutture dati: si definiscono le seguenti `struct`:

- per le statistiche una `struct stat_t` con i 6 campi interi indicati dalle specifiche
- per gli oggetti che formano l'equipaggiamento una `struct inv_t` avente come campi `nome` e `tipo` (stringhe allocate dinamicamente) e le statistiche
- per l'equipaggiamento una `struct tabEquip_t` avente un campo intero `inUso` e un vettore `vettEquip` di puntatori a oggetti di tipo `inv_t`
- per il personaggio una `struct pg_t` avente come campi `codice`, `nome` e `classe` (stringhe allocate dinamicamente), le statistiche di base e quelle date dall'equipaggiamento e un puntatore `equip` a una `struct tabEquip_t`.

Gli item delle collezioni di dati sono i personaggi di tipo `pg_t` e gli oggetti di tipo `inv_t`:

- per i personaggi la collezione è una lista realizzata come una `struct` wrapper di tipo `tabPg_t` contenente il numero corrente di personaggi e i puntatori a testa e coda della lista. Il nodo della lista contiene un personaggio di tipo `pg_t` e un puntatore di tipo `linkPg` al nodo successivo
- per gli oggetti la collezione è una `struct` wrapper di tipo `tabInv_t` contenente il numero corrente di oggetti, il vettore degli oggetti di tipo `inv_t` e il numero di oggetti `nInv`. Il campo `maxInv` riportato nella figura è un refuso.

Nel file `inv.h` sono state inserite le definizioni delle `struct` relative agli oggetti dell'inventario `stat_t` e `inv_t` e alla loro collezione `tabInv_t` e gli header delle funzioni che vi operano.
Nel file `inv.c` compaiono le implementazioni delle funzioni i cui header sono in `inv.h`: `leggiStat`, `stampaStat`, `leggiInv` e `stampaInv`, nonché quelle relative alla loro collezione `leggiTabInventario` e `stampaTabInv`.

Nel file `pg.h` sono state inserite le definizioni delle `struct` relative ai personaggi `pg_t`, alla loro collezione `tabPg_t` e all'equipaggiamento di ciascun personaggio `tabEquip_t` e gli header delle funzioni che vi operano.
Nel file `pg.c` compaiono le implementazioni delle funzioni i cui header sono in `pg.h`: `leggiPg`, `newNodoPg`, `inserisciInListaPg`, `leggiTabPg`, `stampaPg`, `stampaTabPg`, `ricercaCodice`, `ricercaCodiceRef`, `aggiungi`, `freeEquip`, `freePg`, `modificaTabEquip`, `aggiornaPgStatEquip`, `aggiornaPgEquip`, `elimina`.

Il menu nel `main` è basato su interi, nel quale è sufficiente un vettore di stringhe da visualizzare, chiedendo all'utente di specificare il numero corrispondente all'opzione scelta.

La gestione dei personaggi (lettura da file, inserimento in coda in lista, aggiunta, cancellazione, ricerca per codice con ritorno del personaggio, aggiornamento delle statistiche, etc) non presenta alcuna difficoltà concettuale, trattandosi di operazioni standard su liste.
La gestione degli oggetti dell'inventario (lettura da file, stampa) non presenta alcuna difficoltà concettuale.

L'aggiunta/rimozione un oggetto dall'equipaggiamento di un personaggio comporta la ricerca per codice dello stesso, di cui si ritorna il puntatore, nonché la modifica della `struct` di tipo `tabEquip_t` cui esso punta per rimuovere o aggiungere l'oggetto.