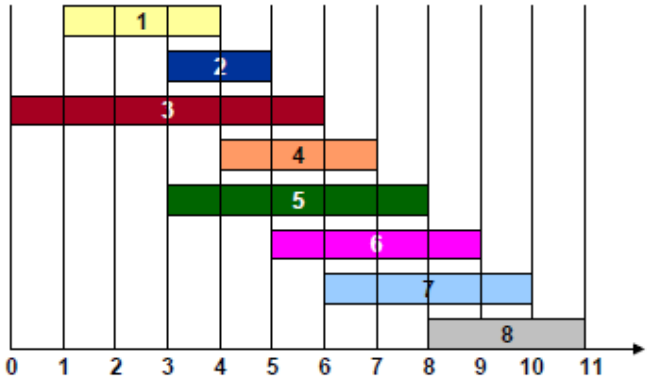
# *Commento al Laboratorio n. 9*

**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 do 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 |

Passo 1: applicabilità

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

- se l'attività `i` fa parte della soluzione, se la soluzione al problema (`i`-1)-esimo non fosse ottima, se ne troverebbe una con valore `opt'[i-1]` 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.

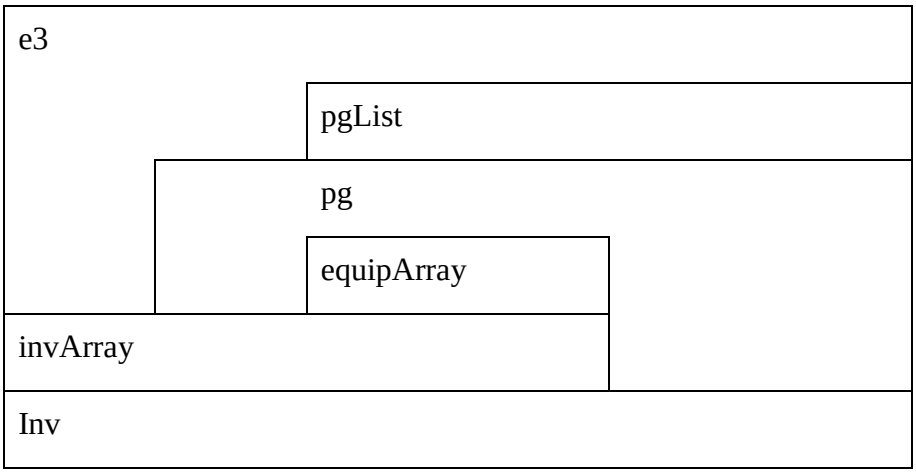
**Esercizio n.2**: **Collane e pietre preziose (versione 3)**
Ogni collana può iniziare con uno dei 4 tipi di gemma. Si considera a titolo esemplificativo il caso in cui la prima gemma è uno zaffiro (funzione `fZ`), gli altri si ricavano facilmente.

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 (multi-file, con ADT)**
La soluzione all'esercizio segue le strategie proposte per la realizzazione degli ADT, tenendo presente 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 moduli 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 (`e3`) è client di `pgList`, `pg` e di `invArray`.

Questo non è l'unico schema realizzabile, ma quello che deriva dall'aver effettuato certe scelte. 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).