Progetti di Programmazione Avanzata

Java

Si è sviluppato un progetto Java con l'obiettivo di gestire le informazioni riguardanti un centro commerciale. Il software implementa la logica di business di un ipotetico server contattabile per avere informazioni riguardanti le attività presenti all'interno del centro commerciale, ovvero il nome, l'orario di apertura e chiusura, l'affollamento e la valutazione. Le attività si dividono in negozi e servizi. L'applicazione permette di aggiungere nuove attività e nuovi parcheggi, modificare la valutazione e l'affollamento delle singole attività, ottenere la lista delle attività aperte, la lista dei negozi di un determinato tipo, i migliori ristoranti (che sono un tipo di servizio), l'affollamento medio del centro e di stampare a video le informazioni riguardanti i parcheggi.

Il progetto si divide in tre package:

- `attivita`
  contiene la classe astratta `Attivita.java`, ovvero la struttura dati base, questa presenta i campi nome, orarioApertura, orarioChiusura, affollamento e valutazione. Le due classi che la estendono sono `Negozio.java` e `Servizio.java`, entrambe aggiungono un campo enumerativo (TipoNegozio e TipoServizio rispettivamente) e la seconda un campo booleano che indica se è necessaria la prenotazione o meno. Tutte queste classi presentano i metodi get() e set() dei campi e il metodo toString(). Attivita, inoltre, implementa l'interfaccia comparable e definisce il metodo equals(). Infine, sono presenti classi accessorie per le eccezioni, gli enumerativi e un comparator (su Attivita).

- `centro_commerciale`
  contiene l'interfaccia dell'applicativo `CentroCommercialeIF.java`. L'interfaccia è implementata dalla classe `CentroCommerciale.java`, che contiene la lista di attività (campo listaAttivita) del centro commerciale, sulla quale lavora per fornire le funzionalità dell'applicativo indicate sopra (un metodo per ogni funzionalità, più un metodo accessorio privato). Anche in questo package sono presenti classi accessorie, ovvero un record (`Parcheggio.java`) e una eccezione.

- `test`
  contiene la classe `Test.java`, ovvero la classe con il main utilizzata per testare le funzionalità dell'applicativo. L'esecuzione di tale classe garantisce una copertura del codice pari al 88,7%.

Lo scopo del progetto è quello di testare alcune funzionalità base e avanzate del linguaggio di programmazione. Di seguito viene indicato dove, come e perché sono state implementate nel progetto:

- **Ereditarietà:** come già indicato, le classi Negozio e Servizio estendono Attivita, ereditandone quindi tutti i campi e i metodi. La classe Attivita è abstract per cui non può essere istanziata, gli unici oggetti possibili, quindi, saranno istanze delle due classi figlio. Negozio e Servizio aggiungono all'implementazione di Attivita solo uno e due campi rispettivamente e fanno overriding del solo metodo toString().

- **Sottotipazione:** in java l'ereditarietà coincide con la sottotipazione, per cui le classi Negozio e Servizio sono dei sottotipi di Attivita. Per questa ragione è possibile utilizzare Attivita come tipo di variabili, tipo restituito dai metodi, per la tipizzazione generica delle liste et cetera, anziché Negozio o Servizio. Inoltre, Attivita è sottotipo dell'interfaccia generica Comparable<Attivita>, per cui deve implementare il metodo comparable. Infine, la classe CentroCommerciale implementa l'interfaccia dell'applicativo, ovvero CentroCommercialeIF, e quindi ne implementa tutti i metodi.

- **Overloading:** viene fatto overloading del costruttore di Attivita, per cui sono presenti due costruttori dove il primo presenta come parametri tutti i campi della classe, il secondo invece non richiede affollamento e valutazione. L'implementazione del costruttore più semplice non è altro che la chiamata di quello più complesso con i valori 0 e 0 per i parametri affollamento e valutazione. Questa struttura viene replicata nei costruttori delle classi figlio Negozio e Servizio.

- **Singleton e parola chiave static:** per la classe CentroCommerciale è stato utilizzato il design pattern del Singleton, una best practice che permette di apprezzare l'utilizzo della parola chiave static. Il design pattern prevede l'utilizzo di un campo e un metodo static, il primo è l'unica istanza della classe, il secondo è il metodo che permette di creare la prima volta l'istanza e di restituirla se già stata creata.

- **Classi, campi e parametri final:** la parola chiave final indica l'immodificabilità di un elemento. È stata utilizzata per indicare che la classe CentroCommerciale non può essere estesa e per indicare che i campi TipoNegozio e TipoServizio (rispettivamente di Negozio e Servizio) una volta inizializzate nel costruttore non possono cambiare valore. Infine, sono stati indicati final i parametri formali dei metodi addAttivita() e addParcheggi() di CentroCommerciale per indicare che gli oggetti attuali passati a tali metodi non devono essere modificati, visto che lo scopo dei metodi è semplicemente aggiungere oggetti alle liste di CentroCommerciale.

- **Eccezioni:** le eccezioni sono state implementate per gestire eventuali input non corretti, ovvero valori errati dei campi di Attivita valutazione (che deve essere compreso tra 0 e 5) e affollamento (compreso tra 0 e 100) e l'inserimento di due Attivita con lo stesso nome. Queste tre classi eccezioni estendono direttamente Exception, e quindi devono essere per forza gestite da chi chiama metodi (o costruttori) che le lanciano.

- **Tipi enumerativi:** gli enumerativi sono stati inseriti per indicare il tipo dei negozi e dei servizi: TipoNegozio può assumere valore ABBILGIAMENTO, LIBRERIA, ELETTRONICA e ALIMENTARI mentre TipoServizio può essere PARRUCHIERE, RISTORANTE o LAVANDERIA.

- **Varargs:** i parametri formali dei metodi addAttivita(final Attivita… la) e addParcheggi(final Parcheggio… lp) di CentroCommerciale sono dei varargs, ovvero al momento della chiamata del metodo, questo accetta come parametri attuali una serie di Attivita (o Parcheggi), che viene poi trattata come un array. Questo permette di aggiungere Attivita e Parcheggi a CentroCommerciale molto più rapidamente.

- **Identificatore var:** l'identificatore var (che permette di non dover specificare il tipo di una variabile, senza però perdere la forte tipizzazione di Java) è stato utilizzato per alcune variabili interne di alcuni metodi di CentroCommerciale, come setValutazione() e getAttivitaAperte().

- **Optional:** la classe Optiona<T> permette di arginare il nullability problem, ovvero la gestione di metodi che possono restituire null. In questo progetto il metodo private getAttivitaPerNome() restituisce l'oggetto di tipo Attivita che ha come valore del campo nome uguale al parametro attuale del metodo. Se un tale oggetto non è presente viene restituito Optional.empty() anziché null. Analogamente il metodo getAffollamentoMedio() restituisce Optional.empty() se listaAttivita è vuota.

- **Streaming e lambda expression:** i metodi getNegoziPerTipo() e getMiglioriRistoranti() devono leggere listaAttivita e fornire in output una sottolista contenente solo elementi che rispettano determinate condizioni. Per questo tipo di operazioni gli streaming combinati con le lambda expression permettono di scrivere codice elegante, facilmente comprensibile e compatto. Lo stream rappresenta una sequenza di elementi sui cui eseguire operazioni intermedie o terminali, le operazioni utilizzate in questo progetto sono filter, sorted, limit. L'operazione sorted ordina la collezione secondo l'ordinamento definito da compareTo() oppure da un comparator. Nel metodo getNegoziPerTipo() l'ordinamento è quello definito da compareTo(), mentre in getMiglioriRistoranti() si utilizza il comparator ValutazioneComparator, che opera sul campo valutazione. L'operazione limit permette si selezionare i primi n elementi. L'operazione filter invece permette di applicare alla collezione delle lambda expression per filtrare gli elementi. Ad esempio, in questi metodi le lambda expression filtrano gli elementi della collezione in funzione della classe di appartenenza (attraverso **instanceof**) e del valore del campo TipoServizio e TipoNegozio.

- **Visibilità:** sono stati utilizzati diversi modificatori di visibilità per i campi e i metodi delle classi, ad esempio nella classe CentroCommerciale tutti i metodi sono pubblici tranne getAttivitaPerNome(), che essendo utilizzato solo all'interno della classe stessa è private, così come i campi. Anche il metodo getIstance() è private per rispettare la best practice del singleton. I campi delle classi figlio di Attivita invece sono friendly (default), ovvero visibili solo nel package, mentre i campi di Attivita sono protected (visibile nel package e alle classi figlio, che possono quindi essere spostate in altri package senza perderne l'accesso).

- **Utilizzo del JCF:** sono state utilizzate le funzionalità del Java Collection Framework per implementare la classe CentroCommerciale, che si basa sul campo listaAttività, di tipo List<Attivita> (interfaccia del JCF) e implementata da ArrayList<Attivita> (che estende List<Attivita> ed è basata su un array che rialloca dinamicamente se necessario). Inoltre, il JCF fornisce utili metodi che implementano algoritmi sulle collezioni, in questo progetto è stato utilizzato il metodo Collections.sort(), che ordina la collezione passata come parametro secondo l'ordinamento definito de compareTo().

- **Programmazione generica:** non è stato implementato nessun metodo o classe generica, ma come già indicato, sono state utilizzate classi generiche come ArrayList<T>, Comparable<T> e Optional<T>.

C++

Il progetto in C++ implementa le stesse funzionalità di quello in Java appena descritto, con alcune differenze, ovvero non sono stati gestiti i parcheggi e si è aggiunto il concetto di bar-tabacchi, che è sia un negozio che un servizio. Il progetto C++ è organizzato nello stesso modo del progetto Java, i due package `attivita` e `centro_commerciale` sono "diventati" due namespace con lo stesso nome.

Nel namespace `attivita` sono presenti i file .h e .c che implementano le classi Attivita, Negozio, Servizio e Bar. I campi delle prime tre sono gli stessi delle corrispondenti in Java, con la differenza che gli orari sono gestiti dalla struct Time, implementata appositamente, e che per gestire la stampa a video degli enumerativi è stato aggiunto un array di stringhe, enumToString[]. Tutte queste le classi presentano i metodi get() e set() necessari per accedere ai campi e il metodo toString(). Si sono quindi mantenuti gli enumerativi (anche se non in file a sé stanti), ma non le eccezioni. Situazioni di input non corretto per i parametri affollamento e valutazione sono state gestite utilizzando valori di default anziché quelli in input. L'aggiunta di un'attività con un nome già registrato attraverso il metodo della classe CentroCommerciale addAttivita(), invece, non viene eseguita e viene segnalato l'evento restituendo il valore false.

Nel namespace `centro_commerciale` sono presenti i file .h e .c della sola classe CentroCommerciale, che implementa gli stessi metodi pubblici della corrispettiva classe in Java (eccetto quelli legati ai parcheggi). Il campo più importante di questa classe è il puntatore al vector di puntatori di Attivita, chiamato listaAttivita. Risulta chiaro, quindi, che nel namespace `attivita` sia implementata la data structure dell'applicativo, mentre nel namespace `centro_commerciale` la logica, come nel progetto precedente.

Infine, il file `Test.cpp` implementa il metodo main utilizzato per testare l'applicazione.

Come per il progetto Java, lo scopo di questo progetto è quello di testare alcune funzionalità base e avanzate del linguaggio di programmazione. Di seguito viene indicato quindi dove, come e perché sono state implementate nel progetto:

- **File .h e .c:** le classi sono divise tra il file di intestazione (header) .h e di implementazione .cpp, nel primo vengono dichiarati i campi e i metodi e le loro visibilità, nel secondo vengono implementati (definiti) i metodi. Questo vale se i metodi non sono inline, caso descritto più avanti.

- **Ereditarietà:** con inheritance si intende il riutilizzo del codice di una classe base da parte di una sua classe derivata. In questo progetto la classe Attivita è di base per le classi Negozio e Servizio (che quindi ne ereditano il codice), che a loro volta sono di base per la classe Bar. Questo va a creare il problema del diamante. È inoltre possibile ereditare con diverse visibilità. Questi due aspetti sono descritti nei punti successivi.

- **Visibilità dell'ereditarietà e using:** in C++ è possibile indicare la visibilità con cui una classe derivata eredita dalla classe base. La visibilità può essere public, protected o private. Le classi Negozio e Servizio ereditano pubblicamente da Attivita, mentre Bar eredita pubblicamente da Servizio e privatamente da Negozio. Se la visibilità è public la classe derivata è sottotipo della classe base perché espone all'esterno il codice ereditato. Se invece l'ereditarietà è di visibilità minore è possibile che la classe derivata restringa l'interfaccia di quella base. La classe Bar comunque, pur ereditando privatamente da Negozio, ne espone pubblicamente il metodo getTipoNegozio() attraverso la parola chiave using.

- **Problema del diamante:** tale problema nasce dall'ereditarietà multipla, ovvero la possibilità di una classe derivata di avere più classi base. Dal momento che Bar eredita sia da Negozio che da Servizio (entrambe classi derivate di Attivita) è necessario evitare che in Bar ci sia duplicazione del codice di Attivita. Utilizzando la parola chiave virtual per indicare che Negozio e Servizio ereditano da Attivita, istanziando un Bar questo conterrà il codice di Negozio e il codice di Servizio, in cui è contenuta una sola istanza del codice di Attivita, che viene "condivisa".

- **Sottotipazione:** l'ereditarietà in C++ non coincide con la sottotipazione, infatti, come già indicato, è possibile ereditare con diverse visibilità. Per avere sottotipazione (ovvero la possibilità di utilizzare una classe derivata laddove è prevista la sua classe base) è necessario avere ereditarietà pubblica. In questo progetto quindi Negozio e Servizio sono sottotipi di Attivita mentre Bar è sottotipo di Servizio (e quindi di Attivita) ma non di Negozio.

- **Costruttori e liste di inizializzazione:** il C++ permette di ridurre notevolmente il codice necessario ad implementare i costruttori delle classi e di renderli più leggibili attraverso le liste di inizializzazione. Questa funzionalità permette di chiamare il costruttore della classe base, altri costruttori della classe stessa e di assegnare valore ai campi della classe subito dopo la firma del costruttore stesso e non nel suo corpo. Si è fatto ampio utilizzo di questa funzionalità per tutte le classi implementate.

- **Distruttori:** il distruttore permette di de-allocare la memoria (nello heap) utilizzata da un oggetto quando questo viene distrutto. La de-allocazione della memoria avviene attraverso la parola chiave delete, che, dato un puntatore, de-alloca la zona di memoria puntata nello heap. L'unico distruttore non vuoto è quello della classe CentroCommerciale, che chiama il delete su tutti i puntatori contenuti nel vector puntato da listaAttivita e infine anche sul puntatore al vector stesso. I distruttori devono essere dichiarati virtual.

- **Overloading:** è stato fatto overloading dei costruttori di Attivita, Negozio e Servizio. In tutti e tre i casi sono stati definiti due costruttori, la cui differenza è la presenza o l'assenza dei parametri affollamento e valutazione.

- **Overriding:** in C++ per avere overriding dei metodi di una classe derivata rispetto alla sua classe base è necessario indicare il metodo sovrascritto, nella classe base, come virtual. In questo modo, se si lavora con reference o puntatori, viene fatto binding dinamico. Nell'applicativo è stato definito virtual (ed è quindi possibile avere binding dinamico) del metodo toString() di Attivita, sovrascritto da Negozio, Servizio e Bar. Come già indicato è buona pratica rendere virtual anche i distruttori per avere una corretta de-allocazione della memoria.

- **Singleton e parola chiave static:** come nel caso del progetto Java, per la classe CentroCommerciale è stato utilizzato il design pattern del Singleton. Il design pattern prevede l'utilizzo di un campo e di un metodo static, il primo è il puntatore all'unica istanza della classe, il secondo è il metodo che permette di creare la prima volta l'istanza e di restituirla se già stata creata.

- **Passaggio dei parametri:** in C++ è possibile passare i parametri di metodi, funzioni e costruttori in tre modi diversi: per valore, per puntatore e per riferimento. Nel primo caso viene fatta una copia dell'oggetto nel record relativo alla chiamata, quindi le modifiche non sono persistenti. Nel secondo

caso si passa un puntatore all'oggetto, attraverso il puntatore è possibile modificare l'oggetto puntato. Il passaggio per riferimento, infine, "nasconde" il passaggio per puntatore, rendendolo meno macchinoso (si crea un alias dell'oggetto passato per riferimento nel metodo chiamato), quindi le modifiche all'oggetto sono persistenti perché riguardano l'oggetto passato dal chiamante. Nel progetto, ad esempio, viene fatto passaggio per valore dei parametri di tipo Time nei costruttori di Attivita. Viene passato il puntatore ad un oggetto di tipo Attivita nel metodo addAttivita() di CentroCommerciale. Infine, viene fatto passaggio per riferimento per passare un oggetto di tipo Time nel metodo getAttivitaAperte().

- **Parola chiave const:** la parola chiave const serve per indicare l'immodificabilità degli elementi indicati come tali in alcune parti del programma. Possono essere indicati come const, ad esempio, metodi, parametri e return values. Un metodo const non può modificare i campi della classe di appartenenza. Un parametro const non può essere modificato all'interno del metodo (o funzione) in cui è passato. Infine, un return value const non può venire modificato dal chiamante, per cui ha senso avere return values const solo in caso di return by reference. Nel progetto tutti i getter sono const, in più il metodo getNome() di Attivita restituisce per riferimento una stringa const. Infine, nella funzione getAttivitaAperte() di CentroCommerciale, viene passato per riferimento un oggetto const di tipo Time.

- **Inline functions:** per aumentare l'efficienza del programma è possibile indicare come inline metodi e funzioni di piccole dimensioni. A compile time la chiamata di una funzione inline viene sostituita dal corpo della funzione stessa, evitando l'overhead dovuto alla creazione sullo stack di un record per la chiamata della funzione. Una funzione in C++ è inline se dichiarata e definita all'interno della classe stessa oppure se nella classe viene soltanto dichiarata ma viene definita nell'header file. In questo secondo caso nella definizione deve essere utilizzata la parola chiave inline. Nel progetto sono inline tutti i metodi di tipo get e set.

- **Utilizzo di friend:** una classe può indicare quali altri classi (o metodi) possono accedere ai suoi campi e metodi, anche se privati, attraverso la parola chiave friend. Nel progetto in Attivita viene indicato come friend la classe CentroCommerciale, per cui nei metodi di CentroCommerciale è possibile accedere direttamente ai campi privati. Ad esempio, nel metodo getAttivitaAperte() si accede direttamente ai campi privati orarioApertura e orarioChiusura. In questo caso l'utilizzo di friend non porta in realtà vantaggi, visto che è possibile accedere a tali campi con i relativi metodi get, anzi, rompe il concetto di information hiding.

- **Default arguments:** nella definizione di un metodo o di una funzione è possibile assegnare un valore ai parametri. Tale valore viene utilizzato se non passato dal chiamante. Nel progetto, ad esempio, si utilizzano default argument nei metodi setValutazione() e setAffollamento() della classe CentroCommerciale rispettivamente per i parametri valutazione a affollamento.

- **Return by reference:** come già indicato, è possibile che una funzione (o un metodo) restituisca un riferimento ad un oggetto. Ciò permette di utilizzare tale funzione come left value in un assegnamento. Nell'applicazione il metodo getNome() di Attivita restituisce un riferimento a string.

- **Smart pointers:** l'utilizzo dei puntatori può portare a problemi come i dangling pointer (utilizzo del puntatore dopo la delete) o memory leak (mancata chiamata di delete e quindi spreco di memoria). Per evitare queste anomalie è possibile utilizzare gli smart pointer, particolari oggetti costruiti sulla base di un puntatore che gestiscono "automaticamente" la de-allocazione della memoria e l'accesso all'oggetto. Nel progetto viene utilizzata la classe generica shared_ptr<T>, che permette di avere più puntatori allo stesso oggetto nello heap e che tiene in memoria il numero di questi puntatori. I metodi che fanno utilizzo di questa classe sono getAttivitaAperte(), getNegoziPerTipo() e getMiglioriRistoranti(). In tutti e tre i casi si vuole creare un nuovo vector (sullo heap) di puntatori ad Attivita e restituirlo. Per evitare che chi chiama tali metodi debba anche ricordarsi di fare la delete si utilizza la classe shared_ptr<T>.

- **Overloading degli operatori:** in C++ è possibile ridefinire il significato degli operatori per le classi. Ad esempio, nel progetto, sono stati ridefiniti gli operatori > e < della struct Time così da velocizzare e rendere più chiaro il confronto tra due orari (ovvero due oggetti di tipo Time).

- **Tipi enumerativi:** come nell'applicativo Java, gli enumerativi sono stati inseriti per indicare il tipo dei negozi e dei servizi: TipoNegozio può assumere valore ABBILGIAMENTO, LIBRERIA, ELETTRONICA, ALIMENTARI e TABACCHI mentre TipoServizio può essere PARRUCHIERE, RISTORANTE o LAVANDERIA. A differenza di Java però è più macchinosa la stampa di un enumerativo. Nel progetto, oltre al campo enumerativo vero e proprio, è stato aggiunto, sia in Negozio che in Servizio, un array chiamato enumToString[] che contiene una lista di stringhe corrispondenti agli enumerativi e nello stesso ordine. In questo modo utilizzando enumToString[static_cast<int>(tipo)] è possibile ottenere la stringa corrispondente all'enumerativo tipo.

- **Identificatore auto:** coincide con l'identificatore var di Java e allo stesso modo permette di non dover specificare il tipo di una variabile, senza però perdere la forte tipizzazione. È stato utilizzato per alcune variabili interne di alcuni metodi di CentroCommerciale, come setValutazione() e getAttivitaAperte().

- **Visibilità degli elementi di una classe:** così come in Java, anche in C++ è possibile modificare la visibilità degli elementi di una classe. La visibilità di default per le classi è private (visibile solo all'interno della classe), per quanto riguarda le struct invece è public (visibile ovunque). La visibilità protected invece permette l'accesso anche alle classi derivate. Ad esempio, la classe Attivita presenta tutti i campi private (per l'information hiding), i costruttori protected (così che possano essere chiamati dalle classi derivate ma che non possa essere istanziato un oggetto di tipo Attivita) e i metodi pubblici.

- **Utilizzo di STL:** la Standard Template Library di C++ è una libreria contenente delle strutture dati e degli algoritmi per la manipolazione dei dati. Le classi di STL sono basate sulla programmazione generica. Nel progetto si è fatto grande utilizzo della classe vector<T> per la gestione delle liste di Attivita. Come ArrayList del JFC, vector<T> basa la sua implementazione su un array, che rialloca dinamicamente se necessario. È inoltre stato utilizzato il metodo sort di STL per ordinare le liste di puntatori secondo l'ordine definito dalle funzioni pointerCompare() e pointerCompareValutazione().

- **Iteratori:** le classi della libreria STL mettono a disposizione dei metodi che restituiscono degli iteratori, ovvero oggetti utili per accedere agli elementi di una lista ed iterare facilmente sulla lista stessa. Tali metodi sono begin() e end(), il primo restituisce un iteratore che punta al primo elemento della lista, il secondo all'ultimo. Nella classe CentroCommerciale si sono utilizzati gli iteratori in diverso metodi, tra i quali getMiglioriRistoranti().

- **Programmazione generica:** non è stato implementato nessun metodo o classe generica, ma come già indicato, sono state utilizzate le classi generiche vector<Attivita*> e la classe shared_ptr<vectort<Attivita*>>. Inoltre, sono stati utilizzati gli operatori static_cast<int> e dynamic_cast<Servizio*> per le conversioni di tipo.

Haskell

Si è sviluppato un applicativo senza alcuna utilità pratica ma con il solo obiettivo di testare le funzionalità del linguaggio di programmazione funzionale Haskell. Il programma è composto da una serie di funzioni, le principali sono:

- `msort l`: ordina la lista di elementi ordinabili `l`;
- `countLett c l`: restituisce una lista delle stesse dimensioni della lista di stringhe `l` dove l'i-esimo elemento è un intero che indica il numero di occorrenze del carattere `c` nell'i-esimo elemento della lista `l`;
- `eseguiOp f l`: restituisce la lista ottenuta applicando la funzione `f` su ogni intero della lista `l`;
- `printLett c n`: restituisce una stringa composta dalla ripetizione `n` volte del carattere `c`.

Inoltre, la funzione `msort` utilizza a sua volta le funzioni:

- `half l`: restituisce l'indice centrale (approssimato per eccesso) della lista `l`;
- `merge l1 l2`: prese le liste ordinate `l1` e `l2`, restituisce la lista ordinata composta da tutti gli elementi delle due liste.

Tali funzioni sono composte nella funzione main in modo che, una volta ottenuto l'input dall'utente, si produca un output come segue:

input1: 's'
input2: ["sassari", "si", "sesto"]
input3: 'z'
`countLett`: [3,1,2]
`eseguiOp`: [8,4,6]
`msort`: [4,6,8]
output (`printLett`): ["zzzz","zzzzzz","zzzzzzzz"]

Lo scopo del progetto è quello di testare alcune funzionalità del linguaggio di programmazione. Di seguito viene indicato dove, come e perché sono state implementate nel progetto:

- **Definizione tipo delle funzioni:** per ogni funzione definita è buona pratica indicarne anche il tipo, ad esempio la funzione `msort` ha tipo `Ord a => [a] -> [a]`, ovvero la funzione prende in ingresso una lista di elementi di tipo a, dove a è un tipo ordinabile, e restituisce una lista dello stesso tipo.

- **Ricorsione e tail recursion:** in Haskell le funzioni vengono spesso implementate tramite ricorsione, ad esempio la funzione `countLett` conta il numero di caratteri `c` del primo elemento della lista e concatena il risulato alla chiamata di sé stessa sul resto della lista. È possibile aumentare l'efficienza delle chiamate ricorsive implementando la tail recursion, ovvero facendo sì che l'ultima operazione svolta dalla funzione sia la chiamata ricorsiva. Nel programma la funzione `printLett c n` non è altro che la chiamata iniziale della funzione con tail recursion `printTail acc c n`, dove acc è il risultato intermedio passato da una chiamata ricorsiva alla successiva, necessario per implementare la tail recursion.

- **Curried functions:** in Haskell, per gestire funzioni con più input senza ricorrere alle tuple, è possibile definire il tipo di una funzione in modo che questa restituisca a sua volta una funzione (che a sua

volta può restituire una funzione e così via). Nel progetto tutte le funzioni con più di un argomento sono curried. Ad esempio, la funzione `printLett` è di tipo `Char->Int->String`, che coincide con `Char->(Int->String)`, ovvero è definita come una funzione che prende in ingresso un carattere e restituisce una funzione con ingresso un intero e uscita una stringa.

- **Higher order functions:** le funzioni possono avere in ingresso altre funzioni, in tal caso vengono definite di ordine superiore. Nel progetto la funzione `eseguiOp f l` (che semplicemente implementa la funzione standard map, ma solo per liste di interi) è di ordine superiore, infatti il suo tipo è: `(Int->Int) -> [Int] -> [Int]`, dove (Int->Int) è il tipo della funzione in ingresso.

- **Lambda expressions:** è possibile definire le funzioni direttamente dove devono essere utilizzate senza doverle nominare. Ad esempio, nel main, la funzione di input di `eseguiOp` è la lambda expression: `(\x -> x*2 + 2)`.

- **Pattern matching:** per definire una funzione è possibile indicare il suo comportamento in ogni caso di input possibile. In questo caso, quando la funzione viene chiamata, si utilizza la prima definizione che fa "match" tra quelle definite. Nel progetto per definire la funzione `msort` si indicano i tre casi di input: lista vuota, lista con un solo elemento e lista con uno o più elementi.

- **Wildcard:** nel pattern matching a volte un caso di input è definito solo da uno dei parametri della funzione, gli altri possono assumere qualsiasi valore. Per indicare ciò si utilizza la wildcard '_'. Ad esempio, per la funzione `countLett` il caso della lista vuota è definito: `countLett _ [] =[]`, ovvero se il secondo argomento è una lista vuota, l'output è una lista vuota a prescindere del valore del primo argomento.

- **Guardie e if then else:** per indicare comportamenti diversi di una funzione in funzione del suo input è possibile utilizzare i costrutti `if … then … else …` oppure le guardie (indicate con '`|`'). Ad esempio, si utilizza il primo costrutto in `merge` e il secondo in `printLett`.

- **Costruttore di liste:** in Haskell le liste vengono definite con il costruttore ':', che aggiunge un elemento in testa ad una lista. Ad esempio, è possibile costruire la lista [1,2,3] come segue: 1:[2,3], oppure 1:(2:(3:[])). Oppure è possibile costruire liste concatenandole, ad esempio: [1] ++ [2,3]. Entrambi i metodi sono utilizzati (il primo in `merge` e il secondo in `countLett`). Inoltre, il costruttore ':' può essere utilizzato nel pattern matching per indicare che una lista viene divisa tra testa e coda, ad esempio `merge (x:xs) (y:ys) = …` permette di avere le due liste in ingresso già divise in elemento di testa (x e y) e il resto della lista, ovvero la coda (xs e ys).

- **List comprehension e generator:** in Haskell è possibile definire le liste così come vengono definiti gli insiemi in matematica. Ad esempio, nella funzione `countLett` si definisce una lista come segue: `[1 | x <- s1, x == c]`, dove <- è il generatore ed indica che x rappresenta uno alla volta tutti gli elementi della lista s1.

- **let in e where:** all'interno della definizione di una funzione è possibile definire dei valori intermedi attraverso la parola chiave `where` (o in maniera analoga con `let … in …`). Il `let in`, ad esempio, è utilizzato in `msort` e il `where` in `countLett`.

- **Input/Output e main do:** per interagire con l'utente è necessario definire la funzione di main (quella eseguita al lancio dell'applicazione) uguale a do, dopodiché attraverso funzioni come `<-`, `putStrLn`, `print` e `read` è possibile gestire l'input e l'output con l'utente.