Università degli studi di Bergamo

Scuola di ingegneria
Corso di laurea magistrale in ingegneria informatica

Progetti per il corso di Programmazione avanzata

C++ e Haskell

Professor
Angelo Gargantini

Candidato
Matteo Locatelli
Matricola 1059210

Anno accademico 2021/2022

Sommario

1 C++ project

1.1 Introduzione

Il programma, realizzato con il linguaggio C++, consiste in un semplice sistema per gestire la prenotazione di pasti all'interno di una mensa accademica.
L'utente che fa uso di questa applicazione ha a disposizione diverse funzionalità tra cui visualizzare gli utenti salvati nel sistema, visualizzare i menu già creati e aggiungere nuove prenotazioni.
L'applicazione non dispone di un'interfaccia grafica: per questo motivo le stampe vengono visualizzate nella console di *Eclipse*, l'ambiente di programmazione integrato (*IDE*) utilizzato per questo progetto, e l'interazione con l'utente avviene tramite l'inserimento di numeri sempre nella console.

1.2 Funzionamento

Il funzionamento del programma è descritto da una serie di procedure implementate nella classe *Gestore*.
All'avvio del programma viene presentato un menu in cui si chiede all'utente di scegliere l'operazione da eseguire. La scelta viene effettuata inserendo un numero, corrispondente all'operazione che si vuole fare: si possono stampare le liste dei menu già presenti nel sistema, delle opzioni tra cui scegliere per personalizzare qualsiasi tipo di menu e degli utenti già registrati ed è inoltre possibile creare un nuovo menu. Quest'ultima operazione porta ad un nuovo menu in cui l'utente viene guidato dalle stampe a console nella personalizzazione del proprio pasto: è possibile scegliere la tipologia di menu (primo, secondo o completo, i cui prezzi sono fissati), la data e l'orario della prenotazione e, in base alla tipologia di menu selezionato, si fornisce all'utente la possibilità di scegliere tra i vari piatti registrati all'interno del sistema.
Inoltre è possibile registrare nel sistema nuovi utenti, come studenti o professori, qualora la persona che vuole effettuare la prenotazione non sia già presente nella lista degli utenti.
Una volta creata la prenotazione si viene riportati nel menu di avvio dell'applicazione da cui è possibile ripetere le operazioni elencate sopra.
Si vuole far notare che l'applicazione è stata progettata ponendo attenzione anche agli input che il sistema riceve: quando si digita un numero a console, questo invocherà nel programma una procedura che soddisfa tale input, ma solo se il numero scelto corrisponde effettivamente ad un'operazione. Esempi di questi controlli si possono avere in due casi:

- La data inserita per la prenotazione è uguale o precedente alla data odierna (intesa come data in cui si fa eseguire l'applicazione)
- Il numero in input inserito non corrisponde a nessuna scelta mostrata a console

In entrambi i casi la procedura che si stava eseguendo non ritorna nulla, viene lanciata un'eccezione e si riporta l'utente nel menu di avvio.

1.3 Diagramma delle classi

In questo paragrafo vengono descritte le entità principali del programma.
In *Figura 1* viene riportato il diagramma delle classi dell'applicazione (con la descrizione delle interfacce) dove è possibile notare come il software si sviluppi intorno a due categorie principali, gli utenti ed i menu, il tutto gestito da una classe denominata *Gestore*.
Per quanto riguarda gli utenti che si possono registrare all'interno dell'applicazione vi sono le classi *Studente* e *Professore*: entrambe queste classi ereditano in modo *public* dalla classe *Utente* e sono molto simili tra loro. Le differenze stanno nei campi statici privati *numStudenti* e *numProfessori*, che conteggiano rispettivamente il numero di studenti e di professori presenti nel software, e nella costruzione del codice in cui viene usata *S* per gli studenti e *P* per i professori. Questa distinzione è stata pensata in modo che studenti e professori vengano inseriti in liste diverse, ma per semplicità implementativa si è deciso di sfruttare una sola lista contenente tutti gli utenti registrati al sistema.

Per i menu, invece, la classe base *Menu* è ereditata in modo *public* dalle classi *MenuPrimo* e *MenuSecondo* le quali, a loro volta, sono ereditate in modo public dalla classe *MenuCompleto*. Questo modo di implementare queste classi ha dato origine ad una struttura a diamante, la quale porterebbe ad avere due istanze diverse di Menu all'interno degli oggetti *MenuCompleto* (una ereditata da *MenuPrimo* e l'altra da *MenuSecondo*). Per ovviare a questo problema è stata utilizzata l'ereditarietà *virtual* della classe *Menu* nelle classi *MenuPrimo* e *MenuSecondo*.

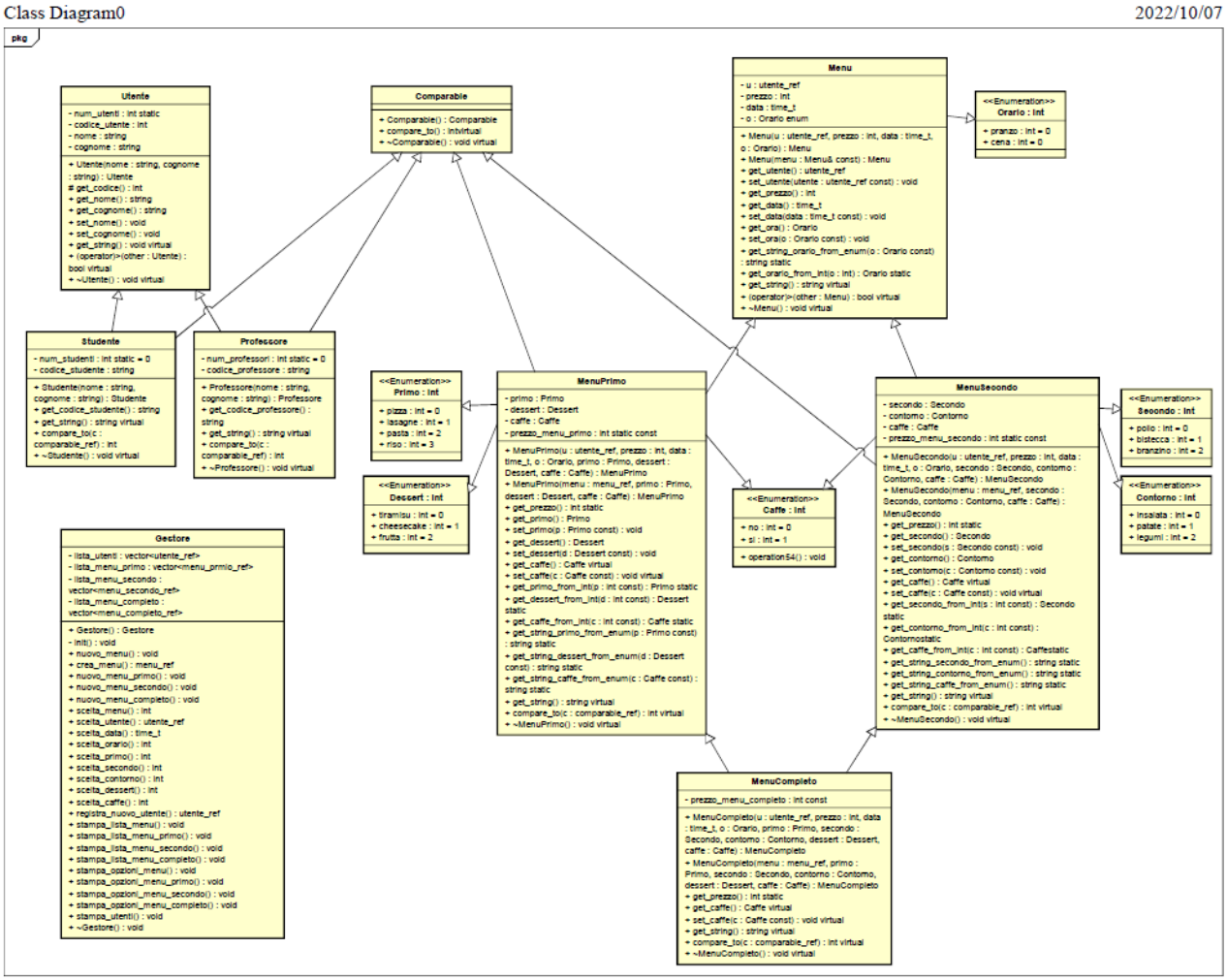

Figura 1: diagramma delle classi

1.4 Dettagli implementativi

Il punto d'ingresso dell'applicazione sviluppata è la funzione *main* presente nel file *PA_project_21_22.cpp*. Al suo interno viene creata l'istanza di Gestore che si occuperà del funzionamento dell'applicazione. Ciascuna classe è implementata in due file:

- Un header (.h) contenente l'interfaccia della classe
- Un file .cpp contenente l'implementazione dei metodi della classe in questione

Questo è stato fatto per rappresentare il concetto dell'astrazione tipico della programmazione ad oggetti. Inoltre il software usa diversi costrutti tipici del linguaggio e sfrutta la libreria offerta *Standard Template Library*. In questo paragrafo verranno mostrate le seguenti *features*.

1.4.1 Ereditarietà multipla

Gli utenti ed i menu sono rappresentati tramite una gerarchia di classi per permetterne la differenziazione. Per gli utenti la classe base è *Utente* che viene ereditata in modo *public* dalle classi *Studente* e *Professore*. Per i menu la classe base è *Menu*, la quale viene ereditata in modo *public virtual* dalle classi *MenuPrimo* e *MenuSecondo* (per risolvere la struttura a diamante), le quali a loro volta vengono ereditate in maniera *public* dalla classe *MenuCompleto*.
Da notare come nell'implementazione della classe *MenuCompleto*, per risolvere il *name clash* che si genera in relazione al membro di tipo *Caffe* (presente sia in *MenuPrimo* che in *MenuSecondo*), si è deciso di richiamare sempre esplicitamente i metodi presenti nella classe *MenuPrimo*.
Le classi *Studente*, *Professore*, *MenuPrimo* e *MenuSecondo*, inoltre, ereditano in modo *public* anche dalla classe *Comparable* la quale contiene il solo metodo *compare_to* utilizzato nell'applicazione per permettere ad un algoritmo di ordinamento di tipo *bubble sort* di ordinare gli utenti per cognome e le categorie di menu per data.
Va sottolineato che anche in questo caso *MenuPrimo* e *MenuSecondo* sfruttano l'ereditarietà *virtual* per risolvere una seconda struttura a diamante che si crea nel sistema con la classe *MenuCompleto*.

1.4.2 Copy constructor

All'interno della classe *Menu* è stato definito un *copy constructor* che copia in un nuovo oggetto di tipo *Menu* sia il puntatore che la zona di memoria associati all'oggetto *Menu* passato per riferimento.
L'idea di passare un oggetto *Menu* come riferimento viene poi propagata nelle sue sottoclassi che sono state dotate tutte di due costruttori: uno che riceve tutti i parametri necessari per costruire il menu (utente, data, orario e prezzo) e per personalizzarlo in base al tipo (primo, secondo, contorno, dessert e caffe) ed un altro che invece di costruire un menu da zero riceve in input un riferimento ad un oggetto di tipo *Menu*.

1.4.3 Distruttore Virtual

Tutti i distruttori delle classi che ammettono sottoclassi sono stati dichiarati *virtual* in modo che la distruzione di un oggetto puntato da un riferimento della superclasse richiami correttamente i distruttori delle sottoclassi.

1.4.4 Overloading, ridefinizione e overriding

In questo programma si sono sfruttati tutte le tipologie di ridefinizione di un metodo offerte da C++:

- *L'overloading*, presente nelle classi *Menu* ed *Utente*, le quali ridefiniscono con una propria segnatura l'operatore >, necessario per creare un ordinamento tra i menu e gli utenti, al fine di differenziare il comportamento dell'operatore in base al parametro passato.
- La ridefinizione, presente nelle classi *Menu*, *MenuPrimo*, *MenuSecondo* e *MenuCompleto* in relazione al metodo *get_prezzo()*: inoltre, avendo ciascun menu un suo prezzo fisso implementato come un campo statico, questi metodi sono stati dichiarati *static* e ciò impedisce di definirli come *virtual* per creare *overriding*.
- L'*overriding*, implementato definendo metodi *virtual* all'interno di alcune classi e delle relative sottoclassi in modo da sfruttare il polimorfismo offerto da C++ tramite l'uso di riferimenti ad oggetti.

1.4.5 STL: std::vector

Nell'applicazione è stato molto usato il *container vector* fornito dalla *Standard Template Library* di C++. Nella classe *Gestore*, infatti, vi sono ben quattro liste rappresentate con tale struttura dati: una per gli utenti e tre per i menu, una per categoria (primo, secondo, completo). Per aggiungere elementi alle relative

liste di oggetti è stato usato il metodo *push_back*, mentre per scorrere una lista ed eseguire operazioni su ciascun elemento di essa (come ad esempio la stampa a console), si sono sfruttati dei cicli *for*.

1.4.6 Smart Pointers

Per evitare un uso scorretto dei puntatori agli oggetti e facilitarne l'uso, evitando problemi quali *dangling pointers* e *memory leaks*, si è fatto largo uso di *smart pointers*.
In particolare sono stati usati gli *shared_ptr<…>*, i quali permettono proprietari multipli del puntatore *raw* sottostante che quindi può essere copiato e passato come parametro ai metodi. Questo tipo di *smart pointers* è stato usato per le classi *Menu* ed *Utente* e per tutte le relative sottoclassi definendo in ciascuna di esse un nuovo tipo (tramite la keyword *typedef*) come *shared_ptr<type>,* dove al posto di *type* è stato specificato il tipo a cui punta il puntatore incapsulato (avendo fatto ciò per ogni classe, *type* è stato sostituito con il nome della classe in cui questo nuovo tipo è stato definito).
È stato usato anche l'*unique_ptr<…>* nel file *PA_Project_21_22.cpp* al fine di creare uno smart pointer ad un oggetto *Gestore*. La differenza con gli *shared_ptr<…>* è che un *unique_ptr<…>* ammette un unico proprietario per il puntatore *raw* sottostante e ciò, nel caso di questo applciazione, permette di avere un solo puntatore ad un oggetto *Gestore* che si occupa del corretto funzionamento del programma.

1.4.7 Template

Al fine di implementare l'algoritmo di ordinamento *bubble sort* sia per i menu che per gli utenti, è stato definito un *header Sorting.h* contenente due metodi parametrici. Dovendo far riferimento ad oggetti diversi, quindi, questi metodi sono stati scritti sfruttando il meccanismo dei *template* di C++ che permette di renderli generici e di sfruttare il polimorfismo in compilazione.

1.4.8 Enum

All'interno dell'applicazione è stato molto sfruttato il costrutto *Enum*, soprattutto per la rappresentare le varie tipologie di primi, secondi, contorni e dessert presenti nel sistema e selezionabili dall'utente per personalizzare il proprio menu.
Per gestire l'uso dell'enumerazione all'interno delle classi *Menu*, *MenuPrimo* e *MenuSecondo* sono stati inoltre inseriti diversi metodi statici che:

- dato un intero ritornano un tipo enumerativo
- dato un enumerativo ritornano una stringa.

2 Haskell project

2.1 Introduzione

Il programma scritto con linguaggio Haskell consiste nell'implementazione di sei algoritmi di ordinamento di una lista di numeri. L'input del programma consiste nella selezione dell'algoritmo di ordinamento, mentre la lista deve essere descritta all'interno della funzione *main* in quanto la piattaforma su cui è stato sviluppato questo programma (*Replit*) non permette la generazione di liste di numeri casuali dato che non supporta la libreria *System.Random*.
Su tale input, inoltre, viene fatto un controllo affinchè si effettui una scelta tra quelle possibili.
Scopo di questo progetto, inoltre, è quello di mostrare a video il tempo di selezione dell'algoritmo e dell'esecuzione dell'ordinamento.

2.2 Descrizione degli algoritmi

2.2.1 Quick Sort

Si tratta di un algoritmo di ordinamento ricorsivo basato sul confronto e sul partizionamento della lista da ordinare in due sotto sequenze più piccole (da ordinare a loro volta) costituite una da tutti elementi minori e l'altra da tutti elementi maggiori di un dato elemento centrale chiamato pivot.

```
8   qsort :: Ord a => [a] -> [a]
9   qsort [] = []
10  qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
11                 where
12                    smaller = [a | a <- xs, a < x]
13                    larger = [b | b <- xs, b > x]
```

Figura 2: quick sort

Il passo base dell'algoritmo è presentato alla riga 9 dove se la lista è vuota viene considerata già ordinata.
Alla riga 10 si mostra come avviene l'ordinamento della lista non vuota: essa si ordina ponendo la testa della lista al centro tra due sotto sequenze ottenute ordinando gli elementi presenti nella coda della lista passata come parametro che sono più piccoli o più grandi della testa.
Alle righe 12 e 13 viene descritto come le liste *smaller* e *larger* sono costruite: *smaller* contiene tutti gli elementi minori del pivot e *larger* tutti gli elementi maggiori.

2.2.2 Merge Sort

È un algoritmo di ordinamento basato sul confronto che sfrutta un processo di risoluzione ricorsivo mediante la tecnica *Divide et Impera*.

- Si divide la collezione da ordinare in due partizioni più o meno della stessa grandezza;
- Si ordinano le due partizioni ricorsivamente;
- Si fondono le due partizioni ordinate.

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x < y     = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys

halve :: [a] -> ([a],[a])
halve xs = (take lhx xs, drop lhx xs)
           where lhx = length xs `div` 2

msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort  xs = merge (msort left) (msort right)
            where (left,right) = halve xs
```

Figura 3: merge sort

La divisione della lista avviene tramite la funzione *halve*: calcola la lunghezza della lista grazie alla funzione *length* e, successivamente, divide tale lista in due sotto sequenze all'incirca di uguale lunghezza con le funzioni *take* e *drop* che prendono, rispettivamente, i primi e gli ultimi *lhx* elementi dalla lista.
L'operazione di fusione viene fatta tramite la funzione *merge* che partendo da due liste ordinate produce un'unica lista ordinata. Le prime due equazioni (righe 17 e 18) gestiscono i casi base in cui una delle due liste è vuota, producendo come risultato l'altra lista. L'ultima equazione gestisce il caso in cui nessuna delle due liste da fondere sia vuota: si fa un'ulteriore distinzione per determinare quale tra i due elementi *x* e *y* in testa alle due liste è il più piccolo e quello diventerà il primo elemento della lista risultante. La coda è ottenuta dall'applicazione ricorsiva di *merge* sulle due liste, una delle quali è stata privata del primo elemento.
La realizzazione del merge sort con la funzione *msort* è poi immediata: distinti i casi base della lista vuota e della lista singoletto, si applica ricorsivamente *msort* alle sotto sequenze di *xs* create con la funzione *halve* e, una volta ordinate, si uniscono tramite la funzione *merge*.

2.2.3 Bubble Sort

Questo algoritmo di ordinamento basa il suo funzionamento sul confronto di elementi adiacenti all'interno di una lista e su un loro possibile scambio.

```haskell
bubbleSortImpl :: Int -> [Int] -> [Int]
bubbleSortImpl 0 xs = xs
bubbleSortImpl n xs = bubbleSortImpl (n - 1) (bubble xs)
                      where
                          bubble [] = []
                          bubble (x : []) = x : []
                          bubble (x : y : ys) = if x <= y
                              then x : (bubble (y : ys))
                              else y : (bubble (x : ys))

bsort :: [Int] -> [Int]
bsort xs = let n = length xs
                  in bubbleSortImpl n xs
```

Figura 4: bubble sort

L'algoritmo *bubble sort* viene implementato in primo luogo con una funzione ausiliaria *bsort* che chiama la vera implementazione tramite la funzione *bubbleSortImpl* che, oltre alla lista da ordinare, riceve in input anche la sua lunghezza.
A riga 34 si osserva il caso base: se la lista ha lunghezza *0* si ritorna tale lista.
Le righe successive, invece, descrivono il caso in cui la lunghezza della lista sia diversa da *0*: in questo caso si chiama ricorsivamente la funzione *bubbleSortImpl* diminuendo di *1* la sua lunghezza e passando come parametro la lista *xs* alla quale viene applicata la funzione *bubble*.
Quest'ultima funzione, descritta da riga 37 a riga 41, serve per confrontare i primi due elementi della lista e, in base al risultato del confronto, si chiama ricorsivamente la funzione *bubble* sulla stessa lista privata del primo o del secondo elemento. Così facendo l'elemento più grande presente in *xs* viene portato in fondo alla lista e, successivamente, si passa ad eseguire *bubbleSortImpl* con lunghezza diminuita di *1*.

2.2.4 Insertion Sort

È un algoritmo di ordinamento basato sul confronto molto semplice che consiste nel prelevare un elemento dalla lista da ordinare e di inserirlo nella posizione corretta al fine di ottenere la lista ordinata. Le due componenti chiave di questo algoritmo di ordinamento sono quindi la scansione della lista iniziale un elemento per volta e l'inserimento di un elemento in una lista ordinata.

```
48  insert :: Ord a => a -> [a] -> [a]
49  insert x [] = [x]
50  insert x (y:ys) | x < y     = x:y:ys
51                  | otherwise = y:(insert x ys)
52
53  isort :: Ord a => [a] -> [a]
54  isort [] = []
55  isort (x:xs) = insert x (isort xs)
```

Figura 5: insertion sort

La seconda componente, ausiliaria alla prima, è definita grazie alla funzione *insert* (è importante tenere presente l'assunzione che la lista all'interno della quale si inserisce l'elemento sia ordinata).
Il caso base della funzione descrive l'effetto dell'inserimento di un elemento *x* nella lista vuota: in tal caso, si restituisce la lista singoletto *[x]*.
Quando *x* viene inserito in una lista non vuota (della forma *y : ys*) occorre distinguere due casi per capire come inizierà la lista risultante.

- Se *x* <= *y*, allora *x* è l'elemento più piccolo e lo collochiamo in testa alla lista risultante, mentre la coda *y* : *ys* è già ordinata per ipotesi.
- Se *x* > *y*, allora *y* è l'elemento più piccolo e lo collochiamo in testa alla lista risultante, andando a inserire *x* ricorsivamente nella coda *ys*.

Implementare l'*insertion sort*, ora, consiste solo in una ricorsione:
Il caso base della ricorsione presente a riga 54 tratta la lista vuota, che è già ordinata.
Alla riga 55 si tratta il caso di una lista con testa *x* e coda *xs*: si deve ordinare *xs* con una applicazione ricorsiva della funzione *isort* e poi inserire x nel punto giusto della lista risultante.

2.2.5 Selection Sort

Si tratta di un algoritmo di ordinamento che opera confrontando gli elementi della lista e spostando il minimo tra quelli presenti all'interno di una lista ordinata.
Per l'implementazione di questo algoritmo si è fatto uso di due funzioni presenti nella libreria *Data.List*:

- *minimum*, che data una lista non vuota restituisce l'elemento più piccolo
- *delete*, che data una lista non vuota elimina da essa la prima occorrenza del parametro indicato

```
58  ssort :: Ord t => [t] -> [t]
59  ssort [] = []
60  ssort xs = let { x = minimum xs }
61             in  x : ssort (delete x xs)
```

Figura 6: selection sort

Le righe 60 e 61 spiegano il funzionamento di questo algoritmo: si prende il minimo elemento *x* dalla lista passata come parametro *xs*, lo si inserisce in testa alla lista ordinata e si chiama ricorsivamente l'algoritmo sulla lista *xs* privata della prima occorrenza del valore *x*.
Il passo base è descritto alla riga 59 dove se la lista passata come parametro è vuota si ritorna una lista priva di elementi e si inizia a ricostruire dal fondo la lista ordinata.

2.2.6 Permutation Sort

È un algoritmo di ordinamento molto inefficiente che consiste nel permutare casualmente gli elementi all'interno di una lista e verificare se essa risulta ordinata o meno.
Questo algoritmo richiede l'utilizzo della funzione *permutations* della libreria *Data.List*: questa funzione ritorna in una struttura dati tutte le permutazioni degli argomenti della lista passata come parametro.

```
64  sorted :: Ord a => [a] -> Bool
65  sorted (x:y:xs) = x <= y && sorted (y:xs)
66  sorted _        = True
67
68  psort :: Ord a => [a] -> [a]
69  psort = head . filter sorted . permutations
```

Figura 7: permutation sort

L'algoritmo opera nel seguente modo: data la lista delle permutazioni controlla uno ad uno se tale permutazione corrisponde alla lista ordinata o meno mediante l'algoritmo ricorsivo ausiliario *sorted*.
Alla riga 66 vi è il passo base: se la lista passata come parametro è vuota la funzione ritorna *true*.
Il passo ricorsivo, invece, presentato alla riga 65, confronta i primi due elementi della lista passata come parametro e chiama ricorsivamente l'algoritmo *sorted* sulla stessa lista privata del primo elemento.

2.3 Descrizione main

All'interno della funzione *main* è stato inserito un blocco *do* nel quale viene mostrata la lista che andrà ad essere ordinata e viene richiesto all'utente di inserire una serie di caratteri al fine di selezionare l'algoritmo di ordinamento. Se l'input corrisponde ad un certo algoritmo, il programma esegue l'ordinamento e mostra l'algoritmo usato, la lista ordinata ed il tempo richiesto per tali operazioni.
In particolare, quest'ultima operazione è stata fatta sfruttando le funzioni della libreria *Data.Time*:

- *getCurrentTime*, per memorizzare il tempo corrente nell'orologio del sistema in formato UTC all'interno di una variabile
- *diffUTCTime*, per fare la differenza tra due variabili contenenti ciascuna il tempo memorizzato con la funzione presentata sopra

Questo output è stato inserito per mostrare come per una lista di pochi numeri (anche solo una decina), i primi cinque algoritmi di ordinamento hanno tempi d'esecuzione che sono frazioni di secondo, mentre il *permutation sort*, che basa la sua riuscita sul caso, richiede molto tempo.