Esercizi Blocco 4

Luca Oliveri
luca.olivieri-1@unitn.it

Università di Trento — December 4, 2020

Introduzione

Questi esercizi vogliono spingere sui limiti di quanto avete imparato. Preparazione definitiva per l'esame.

4.1

SUP che dato un numero, controlla i numeri da zero al numero inserito comunicando all'utente l'appartenenze alle seguenti classificazioni numeriche, usando funzioni:

- Numeri Abbondanti: numeri più piccoli della somma dei loro divisori propri.
  Esempio: 12 essendo 1+2+3+4+6 = 16.
- Numeri Difettivi: numeri più piccoli della somma dei divisori propri.
  Esempio: 21 essendo 1+3+7 = 11.
- Numeri Abbondanti Primitivi: numeri Abbondanti i cui divisori propri sono tutti numeri Difettivi.
- Numeri Semiperfetti: numeri uguali alla somma di *alcuni* dei suoi divisori propri.
- Numeri Bizzarri: numeri Semiperfetti che non sono abbondanti.

Info: Parlando di *divisori propri* si vuole indicare tutti i divisori eccetto il numero stesso. È più facile confrontare la correttezza delle serie prodotte se i risultati vengono raggruppati per gruppo di appartenenza. Quindi prima tutti i numeri Abbondanti, poi Difettivi, eccetera.

4.2

SUP che prende da riga di comando una frase e chiama funzioni che svolgono le seguenti operazioni, stampando per ognuna il risultato:

- Trova la lunghezza della stringa senza usare funzioni di libreria
- Crea una nuova stringa che contiene tutti i caratteri della stringa originale, ma separati da un carattere a scelta dell'utente. *Esempio: "ciao" '@' diventa "c@i@a@o"*
- Restituisce una nuova stringa che l'invertita rispetto all'originale.
- Conta il numero di parole.
- Conta il numero di lettere, numero di cifre e numero di segni di punteggiatura nella frase. La funzione deve restituire `void`.
- Funzione che trova il carattere con più occorrenze e comunica al chiamante il carattere in questione e il numero di occorrenze.

4.3

Realizzare una libreria di funzioni in grado di gestire un array di strutture a lunghezza variabile.

Un array di puntatori a `struct` fornisce lo scheletro sul quale vengono mano a mano allocate le strutture. Se l'array allocato inizialmente si rivela insufficientemente lungo per contenere tutte le strutture di cui ha bisogno l'utente, la libreria alloca un nuovo array più capiente, copia i riferimenti delle strutture presenti in quello vecchio e lo distrugge a copia ultimata. Tutta questa operazione deve essere completamente trasparente all'utilizzatore, che semplicemente chiederà di aggiungere nuovi elementi.

L'unità di memorizzazione è una `struct` definita in un file header (`.h`) a sé stante. Sia il programma che la libreria andranno a riferirsi a questo file per avere un riferimento della struttura usata. In questo modo la libreria di funzioni rimane adattabile a diverse strutture, definibili riscrivendo il file di header a seconda della necessità. Per testare la libreria si consiglia di usare una struttura contenente un singolo `int`.

A seguire una bozza delle firme dei metodi da implementare ed un esempio di uso della libreria. Abbreviamo `VariableArray` con `VA`. `theStruct` è la struttura definita nel file header separato.

- `theStruct ** initVA(int numStartingElements)` → Funzione di partenza per utilizzare la libreria. Inizializza l'array usato per le successive chiamate.
- `theStruct ** addElementToVA(theStruct ** VA, theStruct * elementToAdd)` → Dato l'array di puntatori e la nuova struttura da aggiungere, aggiungi questa alla prima posizione libera nell'array. Se `VA` ha finito le posizioni disponibili, allora si occupa di allocarne uno nuovo e copiarvi tutti i riferimenti. Ricordarsi di deallocare quello vecchio. La funzione restituisce l'array, ricevuto in input o allocato nuovo a seconda di come si sono evolute le cose.
- `destroyVA(theStruct ** VA)` → Dealloca tutto, sia le strutture che l'array di puntatori.

main.cpp

```
// theStruct is a struct
// containing an integer called 'num'

theStruct ** VA = initVA(10);

// Adding elements to the VA
theStruct * s = new theStruct;
s->num = 45; // Or use rand in a for loop
VA = addToVA(VA, s);
// Repeat N times with N > 10 to test

for(int i = 0; i < N; i++) {
        cout << VA[i]->num << endl;
}

destroyVA(VA);
```

4.4

Contare le occorrenze delle parole in un testo, dove per parola si intende un insieme di caratteri delimitato da spazi. Quindi ad esempio in "*se non l'amore può salvarci*" consideriamo "*l'amore*" come parola unica.

Info: Basare il programma attorno ad una `struct` che al suo interno memorizzi una parola e un variabile di conteggio. Usare la libreria sviluppata all'esercizio precedente e stampare un elenco con le N parole più utilizzate nel testo. Analizzare testi lunghi usando l'apertura file vista in laboratorio.

4.5

Libreria di funzioni che modelli matrici usando le `struct`. La libreria dovrebbe fornire funzioni per creare e distruggere matrici, oltre che implementare funzioni per le operazioni di addizione, sottrazione e trasposizione. Fare dovuti controlli dimensionali per verificare che le operazioni tra le matrici coinvolte possano essere effettivamente svolte. Per i più temerari c'è la sfida del calcolo del determinante e decine di altre operazioni matriciali.

Info: Si consiglia che la struttura usi un array **monodimensionale** per contenere i valori della matrice e due interi che contengano i numeri di righe e colonne. È possibile usare anche un array bidimensionale, ma mentre può suonare più intuitivo, non è la soluzione tipicamente usata in librerie di algebra lineare.

4.6

Interpolazione lineare. Per *traiettoria* si intende l'evoluzione di una variabile nel tempo. Ad esempio la variabile può essere la posizione di una particella in un problema di fisica.

In generale le traiettorie sono composte da infiniti punti, quindi per poterle rappresentare è necessario discretizzarle. Il modo più semplice di discretizzare è rappresentare la traiettoria in punti equidistanziati nel tempo, quindi una volta stabilito un intervallo di tempo costante, possiamo descrivere la posizione della nostra particella (caso monodimensionale) di esempio con un array di numeri interi: `{3, 9, 1, -2, -10, -8, -4, -5, 2, 3, 7, -4, -6, 5, 9, 8}`. Usiamo questo array come riferimento per il nostro problema e ipotizziamo una discretizzazione con intervallo di $1s$, quindi essendo che il nostro array ha 16 elementi, conosciamo la traiettoria tra gli istanti di tempo 0 e 16.

Il programma ricostruisce la traiettoria partendo dai punti a nostra conoscenza e unendoli con delle rette. Possiamo quindi adesso avere una stima della traiettoria in qualsiasi punto tra 0 e 16. Il programma stampa il valori della traiettoria tra 0 e 16 a intervalli di $0.1s$. La stampa è divisa in due colonne, a sinistra il tempo e a destra il valore calcolato per quell'istante.

Info: Per risolvere il problema il programma deve innanzitutto essere in grado di ricavare l'equazione di una retta dati due punti. Valutare poi il valore l'equazione nei punti desiderati.

Possiamo creare un grafico che traccia l'evoluzione della posizione della particella nel tempo, sulle x abbiamo il tempo e sulle y abbiamo i valori dell'array. Qua un esempio di grafico con interpolazione lineare. Per verificare la correttezza dell'algoritmo scritto, incollare il risultato del programma in un foglio di calcolo e aggiungere al grafico, con un colore diverso, i punti calcolati. Dovreste ottenere una situazione come quella nella foto del link. Può essere comodo scrivere il risultato su un file anziché in console.

Volendo fare una considerazione di carattere generale, l'interpolazione lineare è il modello di interpolazione più semplice. La derivata prima è discontinua e questo può essere un problema in molte applicazioni. Un modello più sofisticato è l'interpolazione polinomiale, ma la sua implementazione è complessa, richiedendo l'uso di una libreria per gestire l'algebra lineare. Qua un esempio di grafico.

Esercizi `CodeStepByStep`

- `sumCubes`
- `sumOfSquares`
- `moveToEnd`
- `printBinary`
- `sameDashes`