Relazione del Progetto
Algoritmi e Strutture Dati

Università degli Studi di Torino
Baviello Francesco e Derosas Elena

a.a. 2023/24

Esercizio 1:
QuickSort e MergeSort con tipi generici in C

Scelte Implementative Generali

- Utilizzo di strutture dati dinamiche, nello specifico un array dinamico, per rendere la struttura indipendente dal numero di elementi presenti nel CSV passatogli come parametro. La dimensione parte da un singolo elemento e non è prevista una funzione di *resizeDown*, dato che nella nostra implementazione la dimensione può solo aumentare. La funzione di *ResizeUp*, invece, raddoppia la dimensione attuale del vettore, rendendo così la complessità ammortizzata degli inserimenti appartenente a $O(1)$.
- Creazione di una `struct` per mantenere coerenza fra i diversi valori che compongono una singola riga del CSV. Comprende quattro campi:
  - `long` per l'identificativo della riga;
  - `char*` per il primo campo stringa del CSV;
  - `int` per il secondo campo del CSV;
  - `double` per il terzo campo del CSV.

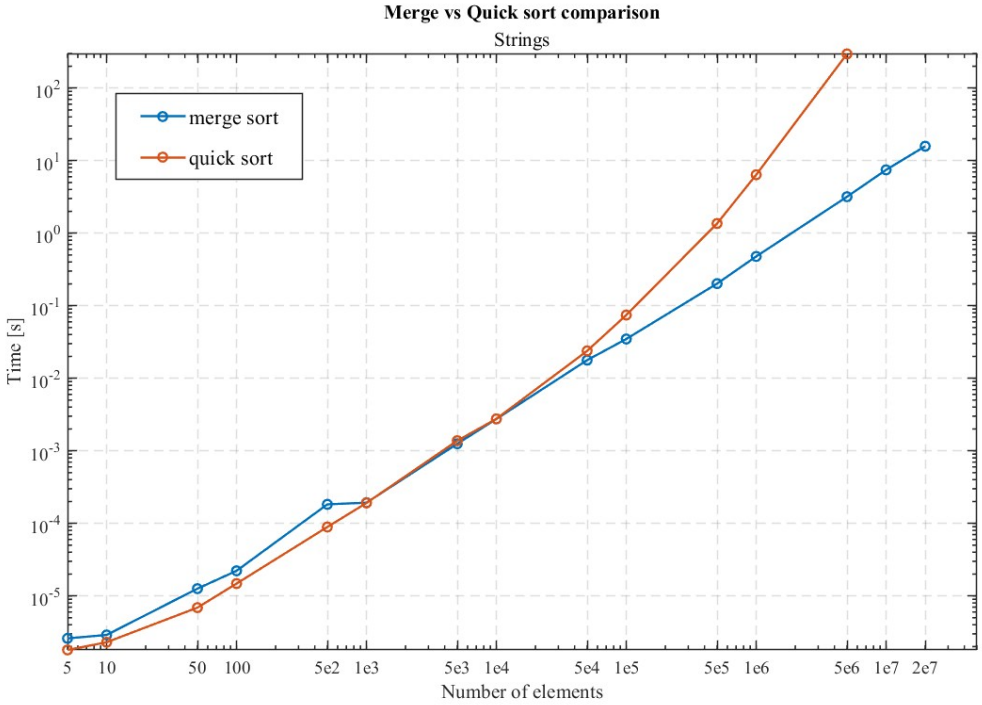
Scelte Implementative QuickSort

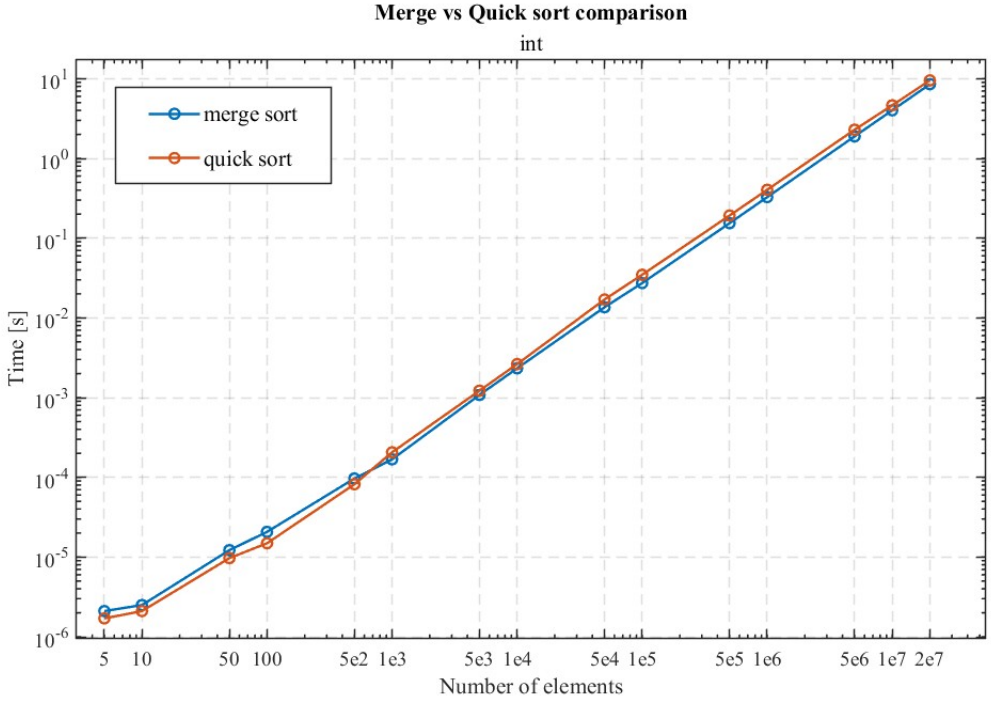
- Creazione di un *wrapper* per tenere traccia degli indici di inizio e fine dei sotto-vettori durante le chiamate ricorsive.
- *Partition*: usato il primo elemento del vettore come *Pivot*. Tale valore è implicito nell'esecuzione, in quanto tale scelta permette di evitare la dichiarazione di una variabile ridondante. Il valore ritornato è il punto utilizzato per partizionare il vettore per le successive chiamate ricorsive.

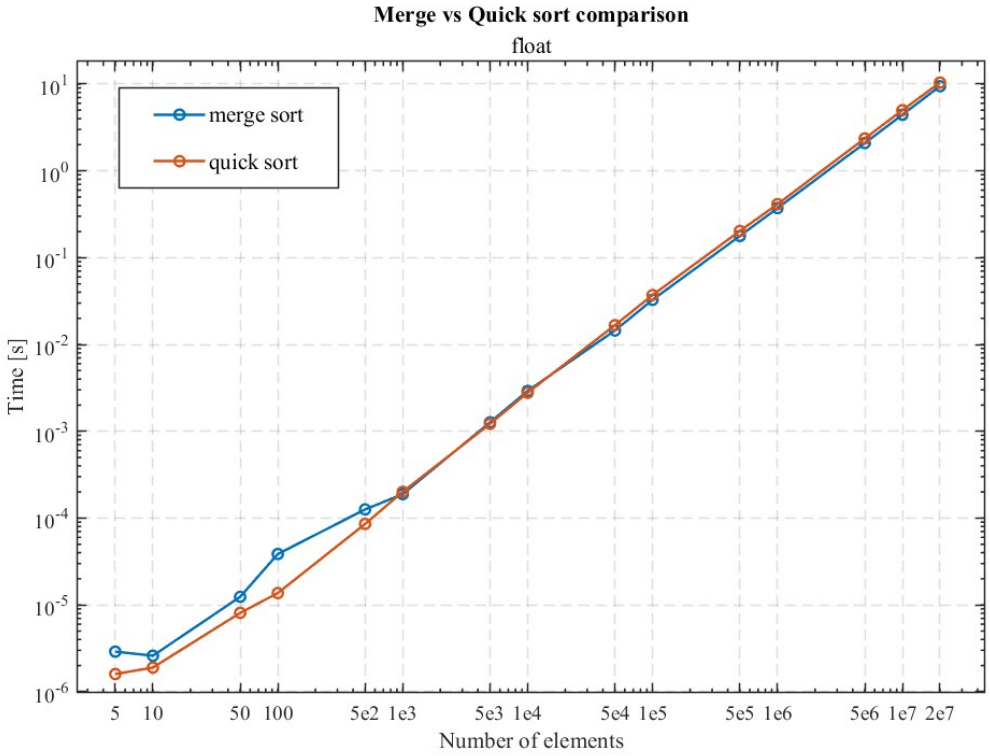
Scelte Implementative MergeSort

- Creazione di un *wrapper* per tenere traccia degli indici di inizio e fine dei sotto-vettori durante le chiamate ricorsive.
- Esecuzione: seguito lo pseudocodice fornito implementando *Divide et Impera*.

Tempistiche







n	string		int		float	
elementi	MergeSort	QuickSort	MergeSort	QuickSort	MergeSort	QuickSort
5	2,6e-6	1,8e-6	2,1e-6	1,7e-6	2,9e-6	1,6e-6
10	2,9e-6	2,3e-6	2,5e-6	2,1e-6	2,6e-6	1,9e-6
50	1,26e-5	6,9e-6	1,22e-5	6,7e-6	1,24e-5	8,1e-6
100	2,22e-5	1,48e-5	2,07e-5	1,49e-5	3,86e-5	1,37e-5
500	1,83e-4	8,92e-5	9,67e-5	8,13e-5	1,26e-4	8,57e-5
1e3	1,93e-4	1,91e-4	1,68e-4	2,06e-4	1,89e-	2,01e-4
5e3	1,25e-3	1,38e-3	108e-3	1,22e-3	1,27e-3	1,22e-3
1e4	2,75e-3	2,76e-3	2,33e-3	1,22e-3	2,92e-3	2,78e-3
5e4	1,78e-2	2,39e-2	1,36e-2	1,7e-2	1,45e-2	1,67e-2
1e5	3,48e-2	7,44e-2	2,73e-2	3,47e-2	3,27e-2	3,71e-2
5e5	2,02e-1	1,36	1,54e-1	1,92e-1	1,78e-1	2,03e-1
1e6	4,78e-1	6,39	3,27e-1	4,04e-1	3,72e-1	4,13e-1
5e6	3,18	296	1,89	2,29	2,09	2,36
1e7	7,47	fallimento	4,01	4,63	4,43	5
2e7	15,8	fallimento	8,53	9,52	9,45	10,4

I tempi sono misurati in secondi.
Specifiche della macchina: Virtual machine Oracle Virtualbox con Linux Ubuntu 64 bit, 8Gb RAM ddr4, 4 processori AMD Rysen5 3600

Si noti che per pochi elementi, fino a circa 500 elementi, il QuickSort si è rivelato più veloce del MergeSort. Probabilmente ciò dipende, oltre che dalle caratteristiche proprie degli algoritmi, anche dal vettore preso in considerazione.
Si osservi che il QuickSort per le stringhe fallisce[1] superati i 7 milioni di elementi, mentre il MergeSort termina l'esecuzione anche per dimensioni molto maggiori in tempi contenuti. Questo perché la complessità del QuickSort dipende enormemente dalla struttura degli elementi, variando da $O(N \log N)$ a $O(N^2)$. Probabilmente gioca un ruolo non indifferente anche la struttura per la comparazione delle stringhe, che causa ulteriore dilatazione temporale.

[1]Per specifiche del progetto si considera fallimento un tempo di esecuzione che superi i 600 secondi (10 minuti) di solo ordinamento.

Esercizio 2: Edit Distance in C

Teoria di Base

Nelle due stringhe si contano il numero di cancellazioni ed inserimenti necessari per passare dalla prima stringa alla seconda. Durante l'esecuzione la lunghezza della stringa si può solo ridurre e solo dall'inizio (da sinistra verso destra). La lunghezza delle due stringhe è parametro sufficiente per stabilire la loro composizione. Questa proprietà permetterà di implementare la memoizzazione nella parte di programmazione dinamica. La ricorsione termina nel momento in cui almeno una delle due stringhe risulta avere lunghezza uguale a zero. Il valore ritornato è la minima combinazione trovata di inserimenti e cancellazioni.

Programmazione Non Dinamica

Senza memoizzazione dobbiamo eseguire tre chiamate ricorsive per ogni esecuzione della funzione. Queste numerosissime ricorsioni calcolano molte volte le stesse coppie di stringhe, provocando una complessità esponenziale.

Programmazione Dinamica

Considero n = length(s1) e m = length(s2). Per implementare la memoizzazione si sfrutta la proprietà delle stringhe analizzata precedentemente. Questa permette di salvare le distanze già calcolate in una matrice di dimensioni $n \times m$ effettuando il pruning dell'albero di ricorsione, potando ad ogni nodo già incontrato, che diventa una foglia.
In questo modo si ottiene un duplice effetto:

- Da un lato si porta la complessità spaziale a $O(n \times m)$;
- Dall'altro si riduce la complessità temporale da essere esponenziale a $O(\min(n, m)^2)$ nel caso peggiore.

Esercizio 3: Heap Minimo in Java

Scelte Implementative

- `ArrayList`, usata per la realizzazione dell'array dinamico che implementa lo heap minimo. Per trasporre lo heap si usano le caratteristiche proprie della ADT: `parent(i) = (i-1)/2`, `leftChild(i) = (i*2)+1` e infine `rightChild(i) = (i+1)*2`.
- `HashMap`, usata per poter implementare una ricerca nel vettore iniziale che appartenesse a $O(1)$. Contiene tutti e soli gli elementi attualmente presenti nella `ArrayList` usata per lo heap.
- Utilizzo di `Nelem` per ottimizzare la gestione dinamica della `ArrayList`. In questo modo si sacrifica della complessità spaziale per ridurre il numero di *resize* effettuate in caso di numerose estrazioni e reinserimenti.
- Tramite controlli appositi, si effettuano solo ed esclusivamente le chiamate ad *Heapify* se necessario:
  - *HeapifyUp* se e solo se l'elemento attuale non è la radice e se viola le proprietà dello heap minimo;
  - *HeapifyDown* se e solo se l'elemento attuale non è una foglia, ovvero se `i < (N/2)-1`.

Esercizio 4: Creazione di un Grafo e Algoritmo di Prim

Graph

Scelte Implementative:

- Usata una `LinkedList` per implementare le liste di adiacenza dei nodi all'interno del grafo. Queste liste sono associate ad ogni nodo mediante una `HashMap`, in modo tale da avere la ricerca di una lista di adiacenza appartenente a $O(1)$. Nella `HashMap` si utilizzano i nodi come chiave (*key*) e le intere `LinkedList` come valore (*value*).
- L'utilizzo delle liste di adiacenza permette di eliminare le ridondanze di informazioni in termini spaziali provocate dalle matrici di adiacenza, e consente di conformare l'implementazione dei metodi astratti ai requisiti di complessità richiesti dalla consegna.

Algoritmo di Prim

Scelte Implementative:

- La necessità di tenere traccia dei nodi visitati è stata soddisfatta grazie ad un `HashSet`, che permette di effettuare ricerche con complessità appartenente a $O(1)$ senza necessitare di coppie chiave-valore.
- La foresta finale è salvata mediante `LinkedList` di archi (*Edge*), e la ricerca del minimo arco fra i nodi connessi è implementata tramite `PriorityQueue` mediante lo heap minimo creato per il terzo esercizio.