Progetto Algoritmi e Strutture Dati - giugno 2025

Fusar Bassini Leonardo - 24865A

June 7, 2025

Abstract

Il presente documento descrive la progettazione e l'implementazione di un software in linguaggio C per l'aggregazione delle preferenze e il clustering di individui, con applicazioni nella gestione del bilancio comunale. Il problema affronta la complessità derivante da un elevato numero di abitanti e progetti, rendendo impraticabile la gestione manuale. Vengono implementati tre metodi di aggregazione delle preferenze — Plurality System, Metodo di Borda e Metodo di Condorcet — per produrre un ordine di preferenza di gruppo tra i progetti. Parallelamente, viene sviluppato un algoritmo di clustering greedy per raggruppare gli individui con preferenze simili. La relazione dettaglia l'architettura del codice, le scelte di strutture dati e algoritmi adottate, con un'analisi approfondita della complessità computazionale di ogni fase. Vengono inoltre illustrate le istruzioni per l'uso, i codici di errore gestiti e le avvertenze generali relative all'esecuzione del programma. Maggiori informazioni sul taglio generale dato al progetto sono nel readme. **Le sezioni 2 e 5**, come motivato nel readme, contengono per lo più **dettagli tecnici** ma che ho ritenuto potessero dimostrare maggiore consapevolezza del codice nel contesto d'uso, sono perciò **saltabili** qualora non siano invece ritenute di interesse.

1 Introduzione

Il problema richiede la progettazione di algoritmi per ordinare un insieme di progetti al fine di allocare razionalmente e democraticamente il bilancio comunale di un comune, assegnando i fondi al primo progetto non ancora avviato, secondo un ordine prestabilito. Data l'elevata complessità dovuta al numero di abitanti (nell'ordine delle migliaia) e progetti (nell'ordine delle decine o centinaia), la gestione manuale risulta impraticabile. Poiché i metodi di aggregazione delle preferenze producono risultati potenzialmente discordanti, si intende implementare tre approcci principali (plurality, Borda e Condorcet) e un algoritmo di clustering per raggruppare gli abitanti in base alle loro preferenze.

Dati in ingresso:

- Un insieme I di n individui, identificati dal codice fiscale.
- Un insieme P di p progetti, identificati univocamente dal nome.
- Per ogni individuo $i \in I$, una preferenza descritta da un ordine debole su P, ovvero una relazione binaria riflessiva, completa e transitiva, ma non necessariamente antisimmetrica. Scriviamo $q \preceq_i q'$ se i preferisce debolmente q a q', indicando una preferenza stretta ($q \prec_i q'$) o indifferenza ($q \sim_i q'$).

Obiettivo: Implementare algoritmi che producano un ordine di preferenza di gruppo per i progetti e raggruppino gli individui in k gruppi con preferenze simili, al fine di facilitare discussioni ristrette.

Output dell'algoritmo: L'output si presenta in 4 parti:

1. **Plurality System:** Per ogni progetto $q \in P$, si calcola il numero di individui che lo considerano il loro preferito (ossia, è il primo nella loro preferenza):

   $$V(q) = |\{i \in I : q \text{ è primo nella preferenza di } i\}|.$$

   Si stampa l'elenco dei progetti ordinati per valori decrescenti di $V(q)$. In caso di parità, i progetti sono considerati indifferenti e ordinati alfabeticamente. Si include il numero totale di voti per ciascun progetto.

2. **Metodo di Borda:** Per ogni individuo $i \in I$ e progetto $q \in P$, si assegna un punteggio
$$B_i(q) = |\{q' \in P : q \preceq_i q'\}|,$$
e si calcola il punteggio totale $B(q) = \sum_{i \in I} B_i(q)$. Si stampa l'elenco dei progetti ordinati per valori decrescenti di $B(q)$. In caso di parità, i progetti sono considerati indifferenti e ordinati alfabeticamente. Si include il punteggio totale per ciascun progetto.

3. **Metodo di Condorcet:** Per ogni coppia di progetti $q, q' \in P$, si confronta il numero di individui che preferiscono debolmente q a q':
$$q \preceq q' \iff |\{i \in I : q \preceq_i q'\}| \geq |\{i \in I : q' \preceq_i q\}|.$$
Si stampa l'elenco dei progetti, evidenziando le preferenze di gruppo e identificando i sottoinsiemi massimali di progetti vicendevolmente raggiungibili tramite catene di preferenze (componenti Condorcet), ordinati in un ordine totale. Si include il numero di individui che supportano ciascuna preferenza.

4. **Clustering delle preferenze:** Si raggruppano gli individui in k gruppi, identificando k individui mediani che minimizzino la somma delle distanze:
$$f(x) = \sum_{j \in I} \min_{i \in x} d(i, j),$$
dove $d(i, j) = \sum_{q \in P} |B_i(q) - B_j(q)|$ è la distanza basata sui punteggi di Borda. Si stampa l'elenco dei k individui mediani, il numero di individui in ciascun gruppo e la distanza media tra gli individui e il loro mediano più vicino.

Una buona referenza per iniziare approfondire il tema della decisione di una scelta ottimale, che dà peraltro vita a svariati paradossi, può essere [Sen70]

2 Istruzioni per l'uso, codici di errore e avvertenze

2.1 Istruzioni per l'uso

(Questa sezione, come precedentemente motivato, contiene per lo più dettagli tecnici non richiesti per il progetto. E' stata inserita per un duplice scopo, da una parte può mostrare di saper contestualizzare in un ambiente applicato le competenze acquisite, e d'altra parte va a completare ulteriormente il progetto in vista di una **possibile** inserzione in un portfolio lavorativo.)
Il programma, dato un corretto file di ingresso, restituisce l'ordinamento dei progetti secondo i metodi di aggregazione delle preferenze (plurality system, metodo di Borda e metodo di Condorcet) e i risultati del clustering greedy degli individui in k gruppi, come specificato nella descrizione del problema. L'output è strutturato in quattro parti: l'elenco dei progetti ordinati per il plurality system con i relativi voti, l'elenco per il metodo di Borda con i punteggi totali, l'elenco per il metodo di Condorcet con i sottoinsiemi massimali di progetti, e l'elenco dei k individui mediani con il numero di individui per cluster e la distanza media al mediano.

Un file di ingresso è da considerarsi corretto se è nel formato seguente:

```
n p k
codice_fiscale_1 prog_1 flag_1 prog_2 flag_2 ... prog_p
codice_fiscale_2 prog_1 flag_1 prog_2 flag_2 ... prog_p
...
codice_fiscale_n prog_1 flag_1 prog_2 flag_2 ... prog_p
```

Dove:

- n è il numero di individui, p il numero di progetti, k il numero di cluster ($k \leq n$).
- Ogni `codice_fiscale_i` è una stringa di `CF_LENGTH` caratteri, univoca.
- `prog_1 prog_2 ... prog_p` è una sequenza di p nomi di progetti (stringhe di lunghezza massima `p`, univoche), uguale per ogni cittadino, che rappresenta l'ordine debole delle preferenze.
- `flag_1 flag_2 ... flag_(p-1)` sono simboli `<` (preferenza stretta) o `=` (indifferenza) tra progetti consecutivi.

2.2 Struttura del codice

Per migliorare la leggibilità, la modularità e la manutenibilità del codice, il progetto è stato organizzato suddividendo le funzionalità principali in diverse librerie (file `.h` per le dichiarazioni e `.c` per le implementazioni). Questo approccio facilita la comprensione del flusso logico del programma e permette di riutilizzare o modificare singole componenti senza impattare l'intero sistema.

Le librerie principali sono le seguenti:

- `prog_elezioni.c`: Contiene la funzione principale `main()` dell'algoritmo. Gestisce l'input da linea di comando, coordina le chiamate alle funzioni delle altre librerie per l'acquisizione, l'elaborazione e la deallocazione dei dati, e produce l'output finale.
- `data_manager.h` (`data_manager.c`): Questa libreria è il cuore della gestione dei dati. Definisce le strutture fondamentali (`cittadino`, `contatore`, `componente`, `ParametriInput`) e i tipi di puntatori associati (`vint`, `vchar`). Contiene inoltre funzioni essenziali per l'allocazione e la deallocazione dinamica di memoria per le principali strutture dati, la lettura dei dati dal file di input, e l'inizializzazione dei contatori per i metodi di aggregazione.
- `sort.h` (`sort.c`): Implementa gli algoritmi di ordinamento basati sul *Quicksort* per diverse tipologie di dati utilizzate nel progetto. Nello specifico, include funzioni per ordinare alfabeticamente i nomi dei progetti, ordinare i contatori dei metodi (per punteggio numerico e poi alfabeticamente), e ordinare le componenti del clustering. L'ordinamento è cruciale per la presentazione dei risultati e per l'efficienza di alcuni algoritmi.
- `metodi.h` (`metodi.c`): Questa libreria incapsula l'implementazione dei tre metodi di aggregazione delle preferenze (Plurality, Borda e Condorcet) e dell'algoritmo di clustering greedy. Per ciascun metodo, sono presenti funzioni per l'aggregazione dei dati dalle preferenze degli individui e funzioni per l'elaborazione finale e la stampa dei risultati.
- `grafo.h, listaarchi.h`: Queste librerie contengono le strutture e le procedure di base per quanto è servito nella costruzione, manipolazione e costruzione di grafi e cammini.

Tutte le procedure implementate nelle librerie sono richiamate e orchestrate all'interno del `main`. Per evitare ridondanze, le complessità computazionali e le scelte di implementazione specifiche di ogni funzione saranno dettagliate nelle sezioni successive, al momento della descrizione del loro effettivo utilizzo e funzionamento.

2.3 Codici di errore

Gli errori gestiti dal programma sono i seguenti, con i relativi messaggi di errore stampati a terminale. Non è garantita la stabilità del programma su altri errori, aspetto che richiede ulteriori miglioramenti.

- Errore nell'apertura del file

  Messaggio di errore: `"ERRORE: File non trovato o non disponibile, nome file cercato: "nomefile" "`

  Descrizione: Si verifica quando il programma non riesce a trovare o ad accedere al file specificato.

  Comportamento: Il programma termina con `exit(EXIT_FAILURE)`.

  Complessità spaziale: $O(1)$, poiché l'errore si verifica prima di completare l'allocazione delle strutture principali.

  Complessità temporale: $O(1)$, in quanto l'errore è rilevato durante il tentativo di apertura del file e il programma si interrompe immediatamente.

- Errore di allocazione di memoria

  Messaggio di errore: `"ERRORE: Errore nell'allocazione del vettore di puntatori a stringhe"` o `"ERRORE: Errore nell'allocazione del vettore di caratteri"` (per l'array dei progetti), oppure `"ERRORE: Errore nell'allocazione della struttura cittadino"` (per gli individui).

Descrizione: Si verifica quando un'operazione di allocazione dinamica di memoria (es. `malloc` o `calloc`) fallisce a causa dell'esaurimento della memoria disponibile.

Comportamento: Il programma termina con `exit(EXIT_FAILURE)`. La memoria precedentemente allocata viene deallocata in base alla fase in cui si verifica l'errore.

Complessità spaziale: $O(1)$, poiché l'errore si verifica prima di completare l'allocazione delle strutture principali.

Complessità temporale: $O(1)$, in quanto l'errore è rilevato durante l'allocazione e il programma si interrompe immediatamente.

- Errore nel formato della prima riga

  Messaggio di errore: `"ERRORE: Errore nel formato della prima riga: valori n, p, k non validi"`.

  Descrizione: Si verifica se la prima riga del file non contiene tre interi positivi n, p, k, con la condizione aggiuntiva che $k \leq n$.

  Comportamento: Il programma termina dopo aver letto la prima riga.

  Complessità spaziale: $O(1)$, poiché nessuna struttura significativa è allocata in questa fase.

  Complessità temporale: $O(1)$, in quanto l'errore è rilevato durante la lettura della prima riga.

- Errore nel formato di righe cittadino

  Messaggio di errore: `"ERRORE: Errore nel formato della riga cittadino: codice fiscale o preferenze non validi"`.

  Descrizione: Si verifica se una riga dedicata a un individuo non rispetta il formato atteso, ad esempio se il codice fiscale non è di `CF_LENGTH` caratteri, o se la sequenza di preferenze non contiene p nomi di progetti seguiti da $p-1$ flag (`<` o `=`).

  Comportamento: Il programma termina durante la lettura della riga incriminata.

  Complessità spaziale: $O(p \cdot \mathrm{p} + i \cdot p)$, dove i è il numero di righe cittadino lette correttamente prima dell'errore (considerando le allocazioni intermedie).

  Complessità temporale: $O(p \log p + i \cdot p \cdot \mathrm{p})$, dove $O(p \log p)$ deriva dall'ordinamento iniziale del vettore dei progetti e $O(i \cdot p \cdot \mathrm{p})$ dalla lettura delle righe cittadino fino all'errore.

- Errore nell'inserimento del progetto

  Messaggio di errore: `"ERRORE: Errore nell'inserimento del progetto: nome non valido o duplicato"`.

  Descrizione: Si verifica se un nome di progetto nel file di input è vuoto, supera la lunghezza massima `p`, o non è univoco rispetto ai progetti precedentemente letti.

  Comportamento: Il programma termina durante la lettura dei nomi dei progetti.

  Complessità spaziale: $O(j \cdot \mathrm{p})$, dove j è il numero di progetti letti correttamente prima dell'errore.

  Complessità temporale: $O(j \cdot \mathrm{p} \cdot \log j)$, per la lettura e il controllo di unicità (tramite ricerca binaria in un array ordinato) dei nomi dei progetti fino all'errore.

2.4 Avvertenze

Il programma è progettato per garantire affidabilità e semplicità d'uso, mantenendo una struttura che faciliti future modifiche e manutenzione. Le seguenti avvertenze derivano dalla natura del software in C e dalla sua interazione con il sistema operativo, e non dall'impossibilità di aspettarsi una corretta aderenza alle regole nelle organizzazioni.

Lettura del file di ingresso: La lettura dei dati avviene all'inizio dell'esecuzione e può richiedere tempo significativo, soprattutto per file di grandi dimensioni. Si raccomanda di non modificare, cancellare o spostare il file durante l'esecuzione per evitare errori o comportamenti imprevedibili.

Controllo preliminare dei dati: Per file con grossi volumi di dati, si consiglia di verificarne la correttezza prima di avviare il programma. Parte dell'elaborazione, come l'assegnazione dei nomi dei progetti agli indici, avviene durante la lettura per ridurre i requisiti di memoria. Errori verso la fine di file di grandi dimensioni possono comportare perdite di tempo significative.

Gestione della memoria: Il programma alloca dinamicamente la memoria sull'heap per gestire insiemi di dati di dimensioni arbitrarie, evitando problematiche di **stack overflow** e migliorando la flessibilità. Tuttavia, si raccomanda di non interrompere forzatamente l'esecuzione, specialmente su sistemi operativi datati o con scarsa implementazione della virtualizzazione, per prevenire **memory leaks** o instabilità del sistema.

3 Il modello

Per modellizzare il problema, ho scelto di definire una struttura fondamentale per rappresentare gli individui e le loro preferenze, insieme a un meccanismo per gestire i nomi dei progetti in modo efficiente (definite nella libreria `gestione_dati.h`):

1. **Struct cittadino**: Memorizza le informazioni di un individuo, identificate dal codice fiscale (una stringa di `CF_LENGTH` caratteri) e dalla sua preferenza sui progetti. La preferenza è rappresentata come un vettore dinamico di interi (`int*`) che codifica l'ordine debole sui p progetti. Ogni progetto è identificato da un indice unico in $\{0, 1, \ldots, p-1\}$, e il vettore contiene gli indici dei progetti ordinati secondo la preferenza dell'individuo, con un vettore ausiliario dinamico di interi (`int*`) che specifica se tra due progetti consecutivi vi è una preferenza stretta (1 per $\prec$) o indifferenza (0 per $\sim$).[1] Questa struttura consente di rappresentare in modo compatto l'ordine debole di ciascun individuo, occupando $O(p \log p)$ spazio per gli indici e i flag di indifferenza.

2. **Gestione dei nomi dei progetti**: A ogni progetto viene assegnato un indice univoco. L'ordinamento lessicografico dei nomi dei progetti viene mantenuto anche nell'ordinamento numerico dei loro indici. La scelta di associare un indice a ogni nome progetto permette di ridurre la memoria occupata nelle memorizzazioni di tale informazione all'interno delle `struct cittadino`. Questo non è solo chiaramente più efficiente in ogni caso (l'informazione richiesta per salvare un nome progetto sarebbe in principio $O(\mathrm{p})$, mentre per un indice è $O(\log_2 p)$), ma anche ottimale, in quanto $\log_2 p$ è la minima informazione necessaria a discriminare p possibilità [Knu98].

   - **Efficienza temporale**: L'assegnazione di un array di indici crea una struttura astratta che associa a ogni progetto un puntatore, indicizzato in un vettore ordinato. Ciò elimina la necessità di utilizzare `strcmp` per confrontare i nomi dei progetti nelle operazioni di preferenza, riducendo il costo dei confronti da $O(\mathrm{p})$ a $O(1)$. Questo è particolarmente vantaggioso per operazioni come il calcolo delle distanze nel clustering o il confronto delle preferenze nel metodo di Condorcet.
   - **Lettura dati efficiente**: La creazione della struttura e la lettura dei dati, descritte nella sezione successiva, possono subire un lieve rallentamento dovuto all'allocazione e all'ordinamento iniziale del vettore dei progetti. Tuttavia, l'ordinamento preliminare del vettore consente ricerche efficienti (ad esempio, binarie in $O(\log p)$), rendendo il costo trascurabile rispetto ai benefici.
   - **Efficienza spaziale**: La struttura dei nomi dei progetti occupa al più $O(p \cdot \mathrm{p})$ spazio, poiché memorizza solo i nomi dei p progetti una volta, mentre le preferenze degli individui usano indici, riducendo la memoria complessiva.
   - **Flessibilità nell'ordinamento**: Ordinare gli elenchi (ad esempio, individui per cluster o progetti per punteggi) è semplificato dallo scambio di puntatori, senza modificare le strutture sottostanti, preservando i riferimenti reciproci.

[1] La scelta di utilizzare indici numerici per i progetti, anziché i loro nomi, riduce significativamente la memoria occupata. Memorizzare i nomi dei progetti per ogni individuo richiederebbe $O(np \cdot \mathrm{p})$ spazio, dove p è la lunghezza massima del nome di un progetto. Assegnando a ogni progetto un indice unico, la memoria si riduce a $O(np \log p)$, poiché ogni indice può essere rappresentato con $\lceil \log_2 p \rceil$ bit. Inoltre, questa rappresentazione facilita l'accesso e il confronto delle preferenze in tempo costante per progetto.

- **Natura statica dei dati acquisiti**: Poiché tutti i dati sono acquisiti all'inizio dal file di input, non è necessaria una gestione dinamica della memoria come nelle liste concatenate durante l'elaborazione, rendendo i vettori una scelta più efficiente per l'accesso.

Un problema di particolare rilievo affrontato nella progettazione del software è la scelta della struttura dati in cui salvare l'informazione che colleghi ogni progetto al suo indice. Di seguito, si analizzano alcune possibili alternative, confrontate in una tabella che ne valuta le prestazioni in termini di spazio occupato, tempi di accesso e capacità di mantenere l'ordine lessicografico.

Struttura	**S.O.**	**N → I**	**I → N**	**Stabilità**[2]	**O.L.**
Array non ordinato	$O(p \cdot \text{max\_str\_len})$	$O(p \cdot \text{max\_str\_len})$	$O(1)$		×
Array ordinato	$O(p \cdot \text{max\_str\_len})$	$O(\log p \cdot \text{max\_str\_len})$	$O(1)$		
Albero binario	$O(p \cdot \text{max\_str\_len})$	$O(\log p \cdot \text{max\_str\_len})$	$O(\log p)$	×	
Hash map	$O(p \cdot \text{max\_str\_len})$	$O(\text{max\_str\_len})$	$O(1)$	×	×

Table 1: Confronto delle strutture dati per il mapping progetto-indice.
S.O.: Spazio Occupato, I: Indice, N:Nome progetto, O.L.: Ordinamento Lessicografico, X: Non Garantita

4 Analisi del Programma

4.1 Commenti di Carattere Generale

Seguono alcune note di carattere generale sulle scelte metodologiche e di implementazione adottate nella stesura del codice. Queste sono valide nella sua interezza e, pertanto, è utile sottolinearle in generale.

Struttura dell'Algoritmo

La funzione `main` è composta da una sequenza di funzioni fondamentali, pensate nello stile di programmazione top-down, che gestiscono le fasi di acquisizione, elaborazione e deallocazione dei dati. Procederò, dunque, con la descrizione dell'algoritmo descrivendo in ordine di comparsa queste funzioni principali. Per ciascuna di esse, verranno illustrate brevemente le scelte fatte in merito alle strutture dati e agli algoritmi utilizzati, e verrà svolto un calcolo dell'ordine di complessità computazionale, considerando il caso peggiore e fornendo una stima dell'ordine asintotico. Al termine dell'analisi di ogni sezione, verrà presentata una sintesi dei vari costi per calcolare il costo temporale e spaziale totali dell'algoritmo.

Analisi della Complessità Computazionale

Per quanto concerne le analisi dei costi temporali e spaziali, ho deciso di adottare un criterio di costo uniforme. Ometterò di analizzare nello specifico alcune operazioni considerate di **costo costante ($O(1)$)** ricorrenti.

Per il **costo temporale**, si considerano a costo costante: confronti semplici non iterati (es. `if(a > b)`), assegnazioni di variabili, operazioni aritmetiche e logiche su interi o double (es. `&&`, `||`), accesso a elementi di un vettore tramite indice della posizione, e chiamate a funzioni standard della libreria (es. `printf`, `fprintf`, `fscanf`, `exit`).

Per il **costo spaziale**, si considerano a costo costante: allocazione di memoria per variabili statiche (come le costanti definite con `#define`), e il passaggio di parametri interi o puntatori a funzioni non ricorsive.

È stata considerata **limitata la lunghezza delle stringhe di caratteri** che costituiscono i nomi dei progetti (`p`) e i codici fiscali (`cf_L`). Volendo tenerne traccia, detta L la lunghezza massima in caratteri di tali stringhe, ogni operazione che coinvolge la lettura o la scrittura di queste stringi comporterebbe un costo aggiuntivo $O(L)$, sia in termini di spazio che di tempo, per le operazioni di copia o confronto (es. `strcmp`).

Allocazione della Memoria

Per l'allocazione della memoria di variabili non statiche, ho utilizzato prevalentemente la procedura `calloc()`. Nonostante un costo temporale superiore rispetto a `malloc()` (che è $O(1)$ in molti casi, mentre `calloc()` è $\Theta(N)$ per inizializzare N byte), ho preferito una maggiore robustezza e chiarezza, garantendo l'inizializzazione a zero di tutti i campi di memoria allocati. Questa scelta è particolarmente vantaggiosa per le variabili su cui vengono eseguiti controlli di nullità o valori iniziali in altre parti del codice. Ho comunque utilizzato `malloc()` quando l'inizializzazione a zero non si rendeva necessaria, ottimizzando le performance.

Passaggio di Variabili a Funzioni

Per quanto possibile, ho cercato di passare in argomento a tutte le funzioni esterne al `main` solamente **puntatori**. Questo permette di limitare il costo associato a tale passaggio a un $O(1)$, evitando la copia locale di grandi quantità di dati e migliorando l'efficienza complessiva. L'unica eccezione sono gli interi e altri tipi primitivi, che per le loro dimensioni ridotte sono equivalenti ai puntatori in termini di costo spaziale e temporale per il passaggio in argomento a funzioni.

Ordinamenti

Tutti gli algoritmi di ordinamento utilizzati sono basati sul **Quicksort**. Questa scelta è stata fatta per la sua **ottimale complessità temporale media di** $O(N \log N)$, che lo rende estremamente efficiente nella maggior parte dei casi pratici. Sebbene il caso peggiore del Quicksort sia $O(N^2)$, nell'implementazione si utilizzano tecniche che mitigano la probabilità di incappare in tale scenario. A differenza di MergeSort (un'alternativa ottimale anche nel caso peggiore), Quicksort è un algoritmo di ordinamento **in-loco**, il che significa che richiede una memoria ausiliaria minima, tipicamente $O(\log N)$ nello stack per le chiamate ricorsive [CLRS09]. [2][2]

—

4.2 Analisi Dettagliata

Questa sezione descrive le prime fasi di esecuzione del programma, focalizzandosi sull'acquisizione dei dati in ingresso e la loro strutturazione in memoria.

4.2.1 Handle_arguments

La funzione `handle_arguments` si occupa di leggere dalla riga di comando il nome del file contenente i dati di input. Questo nome viene poi salvato in una stringa, che viene passata per indirizzo alla funzione successiva.

Questa operazione ha una complessità temporale $\Theta(1)$ e una complessità spaziale $\Theta(1)$, considerando costante la lunghezza massima consentita per il nome del file di input.

4.2.2 Fetch_all_data

La funzione `fetch_all_data` è responsabile dell'apertura del file di input e del caricamento di tutti i dati in esso contenuti nelle opportune strutture definite nella libreria `data_management.h`. In particolare, la funzione apre il file e lo scorre interamente una sola volta, salvando le informazioni essenziali: il numero di individui (n), il numero di progetti (p), e il numero di cluster (k). Successivamente, procede alla lettura e memorizzazione dei nomi univoci dei progetti e, infine, dei dati di preferenza per ciascun individuo. Al termine della lettura del file, tutte le strutture dati principali risultano inizializzate e contengono i loro valori definitivi.

[2]La stabilità sugli input si riferisce alla capacità della struttura di gestire input variabili (es. albero binario molto sbilanciato, che avrebbe quindi caratteristiche simili a quelle di una lista, o collisioni nel caso della hash map) senza richiedere modifiche significative al codice o causare errori.

4.2.3 Scelte di Algoritmi e Strutture Dati in CaricaDati

Nella fase di caricamento dei dati, sono state adottate diverse scelte di design per ottimizzare sia l'efficienza spaziale che quella temporale:

Allocazione Efficiente delle Preferenze Individuali: Per ciascun individuo, le sue preferenze sui p progetti e i $p-1$ flag di indifferenza/preferenza stretta vengono allocati come vettori dinamici all'interno della `struct citizen`. Sebbene la dimensione massima di questi vettori sia nota (p e $p-1$), l'allocazione dinamica per ciascuna struttura `citizen` è preferibile a un array statico di dimensioni fisse **nel caso in cui la dimensione p fosse variabile tra diverse esecuzioni del programma o per supportare un'architettura più flessibile. Se p è costante per tutti i cittadini e noto a tempo di compilazione, un array statico potrebbe offrire migliori prestazioni in termini di accesso alla memoria e overhead.** Questo previene sprechi di memoria nel caso in cui la dimensione p fosse variabile o molto grande. **Ordinamento Preliminare dei Nomi dei Progetti**: Non appena completata la lettura e il salvataggio dei nomi univoci dei progetti, l'elenco viene ordinato alfabeticamente utilizzando l'algoritmo Quicksort. Questa scelta è cruciale perché, come discusso nella sezione "Il Modello", permette di assegnare a ogni progetto un indice numerico basato sul suo ordine lessicografico. Tale ordinamento preliminare riduce drasticamente il costo delle future operazioni di ricerca dei progetti (passando da una ricerca lineare $\Theta(p \cdot l)$ a una ricerca binaria $\Theta(\log p \cdot l)$), un'operazione che altrimenti verrebbe ripetuta $n \cdot p$ volte durante la lettura delle preferenze di ogni cittadino. **Lettura a Singolo Passaggio**: L'intera logica di caricamento del file è stata progettata per effettuare un singolo passaggio (una sola lettura completa) sul file di input. Questo riduce significativamente i tempi di I/O, che possono essere molto elevati per file di grandi dimensioni. Durante questo unico passaggio, vengono estratti tutti i parametri iniziali (n, p, k), i nomi dei progetti, e le preferenze dettagliate di ogni individuo. Ogni informazione viene processata e inserita nelle strutture dati pertinenti in tempo reale, minimizzando la necessità di scorrimenti multipli o di ricaricare sezioni del file.

4.2.4 Calcolo della Complessità Computazionale

Di seguito viene analizzata la complessità temporale e spaziale della procedura `CaricaDati`, considerando il caso medio e adottando il criterio di costo uniforme precedentemente definito.

Complessità Temporale (Θ_2):

- Apertura del file di input: $\Theta(1)$.
- Lettura dei parametri iniziali n, p, k: $\Theta(1)$.
- Allocazione del vettore di puntatori ai nomi dei progetti (`char **p`$_{names}$`list`) :$\Theta(1)$.
- `Allocazione e lettura dei nomi dei` p `progetti:`
  - `Allocazione di memoria per ogni singolo nome di progetto (stringa di l caratteri):` $\Theta(p \cdot l)$.
  - `Lettura di ciascun nome di progetto:` $p \times \Theta(l)$, `quindi` $\Theta(p \cdot l)$.
  - `Totale:` $\Theta(p \cdot l)$.
- `Ordinamento dei nomi dei progetti (Quicksort):` `per` p `progetti, la complessità media è` $\Theta(p \log p)$. `Considerando che i confronti tra stringhe hanno costo` $\Theta(l)$, `il costo è` $\Theta(p \log p \cdot l)$.
- `Allocazione del vettore di puntatori alle strutture citizen (citizen **c`$_{data}$`)` :$\Theta(1)$.
- `Lettura dei dati per ogni individuo (`n `iterazioni):`
  - `Per ciascun individuo (`$i = 0, \ldots, n-1$`):`
    - `Allocazione della struttura citizen:` $\Theta(1)$.
    - `Lettura del codice fiscale:` $\Theta($`cf_L`$)$.
    - `Allocazione dei vettori projs (`p `interi) e op (`$p-1$ `interi):` $\Theta(p)$.

* Per ciascun progetto nella preferenza dell'individuo ($j = 0, \ldots, p-1$):
  · Lettura del nome del progetto: $\Theta(l)$.
  · Ricerca binaria del progetto nell'elenco ordinato $p_names_list per ottenere il suo indice: $\Theta(\log p \cdot l)$.
  · Lettura del flag (< o =): $\Theta(1)$.
  · Assegnazione dell'indice e del flag nel citizen: $\Theta(1)$.
* Costo per singolo cittadino: $\Theta(\text{cf\_L} + p + p \cdot (l + \log p \cdot l + 1))$. Questo semplifica a $\Theta(p \cdot l \log p)$.

– Totale per tutti gli individui: $n \times \Theta(p \cdot l \log p) = \Theta(n \cdot p \cdot l \cdot \log p)$.

Il costo temporale totale medio della procedura `CaricaDati` è la somma di tutti questi contributi. Poiché l'ultimo termine domina gli altri per n, p, l sufficientemente grandi, si ha:

$$\Theta_2 = \Theta(1) + \Theta(p \cdot l) + \Theta(p \log p \cdot l) + \Theta(n \cdot p \cdot l \cdot \log p)$$

$$\Theta_2 = \Theta(n \cdot p \cdot l \cdot \log p)$$

L'analisi viene completata dall'osservazione che cf_L = 16.

Complessità Spaziale (S_2):

Vettore di puntatori ai nomi dei progetti (`p_names_list`): $\Theta(p)$ per i puntatori, più $\Theta(p \cdot l)$ per i dati effettivi. Totale: $\Theta(p \cdot l)$. Vettore di puntatori alle strutture `citizen` (`c_data`): $\Theta(n)$ per i puntatori. Memoria per ogni struttura `citizen`:

- – Campo `code` (array fisso): $\Theta(\text{cf\_L})$.
  – Vettore `projs` (dinamico): $\Theta(p)$ per gli interi.
  – Vettore `op` (dinamico): $\Theta(p)$ per gli interi.
  – Totale per una `citizen`: $\Theta(\text{cf\_L} + p)$.

- Memoria totale per tutte le strutture `citizen`: $n \times \Theta(\text{cf\_L} + p) = \Theta(n \cdot (\text{cf\_L} + p))$.
- Memoria dello stack per Quicksort: $\Theta(\log p)$ (profondità media della ricorsione).

Il costo spaziale totale della procedura `CaricaDati` è la somma di questi contributi. I termini dominanti sono la memoria per i nomi dei progetti e per le strutture `citizen`:

$$S_2 = \Theta(p \cdot l + n \cdot (\text{cf\_L} + p)) + \Theta(\log p)$$

$$S_2 = \Theta(p \cdot l + n \cdot p)$$

Anche qui, l'analisi viene completata dall'osservazione che cf_L = 16.

—

4.2.5 DeallocaMemoria

Una gestione accurata della memoria è cruciale in linguaggio C per prevenire **memory leaks** e garantire la stabilità del programma, specialmente in applicazioni che trattano grandi volumi di dati dinamici. La funzione `DeallocaMemoria`, situata in `data_management.c`, è stata progettata per rilasciare sistematicamente tutta la memoria allocata dinamicamente sull'heap durante l'esecuzione del programma. Questo approccio modulare assicura che ogni blocco di memoria allocato, come i nomi dei progetti, le strutture dei cittadini (con i loro vettori di preferenze e flag), e le strutture ausiliarie temporanee (es. contatori, matrici di Condorcet, cluster), venga correttamente liberato al termine delle operazioni o in caso di errore.

L'implementazione prevede una serie di chiamate alla funzione `free()` per ogni puntatore precedentemente allocato. La deallocazione avviene in ordine inverso rispetto all'allocazione, assicurando che i "blocchi" di memoria più interni (es. i vettori `projs` e `op` all'interno di ogni `citizen`) siano liberati prima dei puntatori che li contenevano (es. il puntatore alla `struct citizen` stessa), e così via, fino alle strutture principali che contengono gli insiemi di dati. In caso di errore durante l'esecuzione, il programma invoca una funzione di deallocazione parziale che libera solo la memoria già allocata fino a quel punto.

Complessità Computazionale ($\Theta_{\text{deallocazione}}$, $S_{\text{deallocazione}}$):

Temporale ($\Theta_{\text{deallocazione}}$): La deallocazione della memoria coinvolge lo scorrimento di tutte le strutture dati dinamiche e la chiamata a `free()` per ogni blocco. Questo include:

- – Liberazione dei nomi dei p progetti: $\Theta(p \cdot l)$.
  – Liberazione delle strutture per gli n cittadini (inclusi i loro vettori interni di preferenze): $\Theta(n \cdot p)$.
  – Liberazione di eventuali strutture ausiliarie (contatori, matrici, strutture per il clustering): dipendente dalle dimensioni delle strutture, ma generalmente legato a $\Theta(p)$, $\Theta(p^2)$, o $\Theta(n^2)$.
  – **Costo Totale**: $\Theta_{\text{deallocazione}} = \Theta(n \cdot p + p \cdot l + \max(p^2, n^2))$.

- **Spaziale ($S_{\text{deallocazione}}$)**: La deallocazione di per sé non alloca nuova memoria significativa oltre a quella dello stack per le chiamate a funzione, quindi la complessità spaziale è $\Theta(1)$.

—

4.2.6 Sistema di Pluralità

Algoritmo Nel main, per un riutilizzo futuro, alloco un vettore di p puntatori a strutture contatore. Inizialmente, azzero le loro variabili di conteggio e inserisco gli indici da 0 a $p-1$ nella variabile di indice. Ogni contatore è, quindi, associato a un progetto specifico. Successivamente, per ciascun individuo, identifico i progetti da lui maggiormente preferiti: il primo progetto elencato e tutti quelli a cui è indifferente, se presenti. Per l'indice di ciascuno di questi progetti, incremento la variabile di conteggio del contatore corrispondente. Infine, utilizzo una variante del Quicksort per ordinare il vettore di strutture contatore. L'ordinamento è in ordine decrescente per conteggio e, in caso di parità, in ordine crescente per indice, garantendo così un ordinamento alfabetico per i progetti con lo stesso numero di voti. Concludo stampando la soluzione, scorrendo il vettore e riassociando a ogni indice il nome del progetto corrispondente.

Complessità Computazionale ($\Theta_{\text{plurality}}$, $S_{\text{plurality}}$):

Temporale ($\Theta_{\text{plurality}}$):

- – Allocazione e indicizzazione dei contatori: $\Theta(p)$.
  – Aumento del conteggio per ogni individuo: $\Theta(n \cdot p)$.
  – Quicksort sul vettore di `struct contatore`: $\Theta(p \log p \cdot l)$.
  – Stampa della soluzione (scorrendo il vettore e riassociando i nomi): $\Theta(p \cdot l)$ (dove l è la lunghezza massima del nome del progetto).
  – **Costo Totale**: $\Theta_{\text{plurality}} = \Theta(n \cdot p + p \log p \cdot l + p \cdot l)$. Per valori significativi, questa si riduce a $\Theta(n \cdot p + p \log p \cdot l)$.

- **Spaziale ($S_{\text{plurality}}$)**:
  La struttura dei contatori: $\Theta(p)$. Spazio ausiliario per il Quicksort: $\Theta(\log p)$.
  Costo Totale: $S_{\text{plurality}} = \Theta(p + \log p)$. Questa si riduce a $\Theta(p)$.

—

4.2.7 Metodo di Borda

Algoritmo Nel main, per supportare l'algoritmo di clustering successivo, creo una matrice di n righe e p colonne. Questa matrice funge da contatore delle preferenze dei progetti: gli indici delle colonne da 0 a $p-1$ corrispondono ai progetti ordinati alfabeticamente, mentre gli indici delle righe da 0 a $n-1$ corrispondono a ogni individuo. Per ogni individuo i, scorro i suoi vettori di preferenze e operatori in parallelo, sommando i punteggi assegnati ai singoli

progetti j e memorizzandoli nella posizione (i, j) della matrice. Successivamente, per elaborare e riordinare i dati della matrice di Borda, riutilizzo il vettore di puntatori a contatore allocato per il Sistema di Pluralità. Questa scelta, che comporta un azzeramento e una nuova indicizzazione delle sue variabili, ha un costo lineare $\Theta(p)$, equivalente all'allocazione di un nuovo vettore, ma permette un risparmio di spazio. Nelle variabili di conteggio dei contatori, memorizzo la somma dei punteggi di Borda di ogni progetto, ottenuta sommando sulle colonne della matrice. Infine, applico la variante del Quicksort per ordinare il vettore di strutture contatore in ordine decrescente per punteggio e, in caso di parità, crescente per indice. Procedo poi a stampare la soluzione scorrendo il vettore.

Complessità Computazionale (Θ_{borda}, S_{borda}):

Temporale (Θ_{borda}):

- Allocazione della matrice: $\Theta(n \cdot p)$.
- Scorrimento dei vettori e somma dei punti nella matrice: $\Theta(n \cdot p)$.
- Azzeramento e indicizzazione dei contatori: $\Theta(p)$.
- Somma sulle colonne della matrice per ottenere i punteggi Borda totali: $\Theta(n \cdot p)$.
- Quicksort sul vettore di `struct contatore`: $\Theta(p \log p \cdot l)$.
- Stampa della soluzione (scorrendo il vettore e riassociando i nomi): $\Theta(p \cdot l)$.
- **Costo Totale**: $\Theta_{\text{borda}} = \Theta(n \cdot p + p \log p \cdot l + p \cdot l)$. Per valori significativi, questa si riduce a $\Theta(n \cdot p + p \log p \cdot l)$.

- **Spaziale (S_{borda})**:
  La matrice di Borda: $\Theta(n \cdot p)$. Spazio ausiliario per il Quicksort: $\Theta(\log p)$.
  Costo Totale: $S_{\text{borda}} = \Theta(n \cdot p + \log p)$. Questa si riduce a $\Theta(n \cdot p)$.

4.2.8 Metodo di Condorcet

Il Metodo di Condorcet confronta ogni coppia di progetti (q, q'). Per ogni coppia, si determina quanti individui preferiscono q a q' e quanti preferiscono q' a q. Il progetto che vince più "duelli" è il vincitore di Condorcet. Se esiste un candidato che batte tutti gli altri in confronti a coppie, è il vincitore di Condorcet. Spesso, si possono verificare paradossi (cicli di preferenze) che non permettono di identificare un singolo vincitore. In tal caso, si identificano componenti di Condorcet (sottoinsiemi massimali di progetti reciprocamente raggiungibili).

Scelte di Implementazione: L'implementazione di Condorcet è la più complessa tra i metodi di aggregazione. Richiede una matrice di confronti a coppie per memorizzare i risultati dei duelli tra progetti.

(a) **Matrice di Confronto**: Viene creata una matrice $p \times p$ (`int **condorcet_matrix`) dove ogni cella (i, j) memorizza il numero di individui che preferiscono il progetto con indice i al progetto con indice j. L'inizializzazione della matrice a zero ha costo $\Theta(p^2)$.

(b) **Calcolo dei Duelli**: Si scorre l'array dei cittadini (n iterazioni). Per ogni cittadino, si analizza la sua preferenza. Poiché la preferenza è un ordine debole, per ogni coppia di progetti (q_a, q_b) nella preferenza dell'individuo, si determina se $q_a \prec_i q_b$ o $q_a \sim_i q_b$. Per tutti i p progetti, si considerano $\Theta(p^2)$ coppie. Per un individuo, il costo per popolare i suoi contributi alla matrice di confronto è $\Theta(p^2)$, poiché si devono considerare tutte le coppie di progetti. In totale, il riempimento della matrice ha costo $\Theta(n \cdot p^2)$.

(c) **Identificazione del Vincitore / Componenti Condorcet**: Una volta riempita la matrice, si cerca un eventuale vincitore di Condorcet (un progetto che vince contro tutti gli altri). Se non esiste, si procede a identificare i sottogruppi massimali di progetti in cui tutti i membri sono reciprocamente raggiungibili (componenti fortemente connesse in un grafo di preferenze). Questo può essere fatto costruendo un grafo orientato dove i nodi sono i progetti e un arco da A a B esiste se A batte B o c'è parità. Le componenti fortemente

connesse (Strongly Connected Components – SCC) possono essere trovate usando algoritmi come Tarjan o Kosaraju, che hanno complessità $\Theta(V + E)$ dove $V = p$ ed $E = p^2$, quindi $\Theta(p^2)$.

(d) **Ordinamento e Stampa**: Le componenti vengono poi ordinate in un ordine topologico. I progetti all'interno di ogni componente possono essere ordinati alfabeticamente. La stampa avviene per ogni componente e per le relazioni tra esse.

Complessità Computazionale ($\Theta_3^{\text{condorcet}}$, $S_3^{\text{condorcet}}$):

Temporale ($\Theta_3^{\text{condorcet}}$):

- – Inizializzazione matrice: $\Theta(p^2)$.
  – Calcolo duelli (riempimento matrice): $\Theta(n \cdot p^2)$.
  – Ricerca vincitore / SCC: $\Theta(p^2)$.
  – Ordinamento e stampa: $\Theta(p^2 \cdot l)$.
  – **Costo Totale**: $\Theta_3^{\text{condorcet}} = \Theta(n \cdot p^2 + p^2 \cdot l)$. Poiché n può essere molto maggiore di p, il termine $n \cdot p^2$ domina.
- **Spaziale ($S_3^{\text{condorcet}}$)**:
  – Matrice di confronto: $\Theta(p^2)$.
  – Strutture dati per SCC (stack di ricorsione, array visitati): $\Theta(p)$.
  – **Costo Totale**: $S_3^{\text{condorcet}} = \Theta(p^2)$.

—

4.2.9 Algoritmo di Clustering K-Medians

Approccio Algoritmico: Identificazione di Cluster tramite Euristica Greedy Data la complessità intrinseca nell'identificazione dei cluster, che non può essere risolta con algoritmi a complessità polinomiale, si ricorre a un'**euristica greedy**. Questo metodo mira a selezionare la soluzione localmente ottimale in ogni passaggio iterativo, basandosi su una funzione obiettivo predefinita.

Il processo inizia con un insieme vuoto di **mediani**, indicato con X. Iterativamente, viene identificato l'elemento $i \notin X$ che minimizza la funzione $\sum_{j \in I} \min_{k \in X \cup \{i\}} d(j, k)$, dove $d(j, k)$ rappresenta la distanza tra gli individui j e k. L'elemento i viene successivamente aggiunto a X. L'algoritmo si conclude quando X contiene un numero predeterminato di k mediani.

— Preparazione dei Dati e Ottimizzazioni Inizialmente, viene costruita una **matrice di distanze** D, una matrice $n \times n$ simmetrica con elementi nulli sulla diagonale, derivata dalla matrice di Borda. Ogni elemento D_{ij} quantifica la dissimilarità tra l'individuo i e l'individuo j. Gli indici di riga e colonna (da 0 a $n - 1$) sono mappati agli indici di un vettore di puntatori agli individui.

Per ottimizzare il calcolo delle somme delle distanze, si impiega un **vettore di aggregazione**, allocato dinamicamente con dimensione n. Questa scelta strategica evita riallocazioni iterative del vettore, generando un significativo risparmio di tempo. Sebbene le porzioni finali del vettore possano contenere valori non più rilevanti a causa della progressiva riduzione delle somme da registrare, ciò non compromette la precisione del processo, poiché tali elementi non vengono più utilizzati. Questa metodologia accelera l'esecuzione eliminando costi di riallocazione di $\Theta(n - s)$ a ogni passo, dove s indica il numero di mediani già identificati.

— Miglioramenti Computazionali Aggiuntivi Un ulteriore miglioramento computazionale è ottenuto attraverso l'introduzione di un vettore di n puntatori a **componente**. Questa implementazione riduce la complessità della ricerca delle distanze minime. Invece di confrontare s valori per ciascuno degli $n - s$ individui a ogni iterazione (per $s > 2$), le distanze minime per ogni componente vengono aggiornate dinamicamente. Questo riduce l'operazione a un singolo confronto tra due valori per gli $n - s$ individui. Similmente, si rivela vantaggioso assegnare ogni individuo al mediano correntemente più prossimo in ogni fase, aggiornando la variabile associata alla componente. Questo approccio evita confronti

aggiuntivi di $\Theta(k \cdot (n-k))$ che si verificherebbero posticipando le assegnazioni alla fine della fase di ricerca dei mediani.

La determinazione del minimo tra le somme delle distanze minime, calcolata ad ogni passo dell'algoritmo greedy, è realizzata tramite uno scorrimento lineare del vettore di aggregazione, supportato da una variabile ausiliaria per il confronto. Questa implementazione è preferita alla creazione di un min-heap ausiliario ad ogni iterazione. Sebbene un heap offrirebbe un aggiornamento in $\Theta(\log n)$, la sua creazione iniziale richiederebbe $\Theta(n)$ e un aggiornamento efficace sarebbe impraticabile dato che tutti i valori delle somme sono soggetti a modifica. Di conseguenza, la soluzione adottata bilancia al meglio l'efficienza temporale con un ridotto consumo di spazio ausiliario ($O(1)$ rispetto a $\Theta(n)$).

Infine, una variante del **Quicksort** viene impiegata per ordinare il vettore di strutture componente: prima in base all'indice dei rappresentanti (mediani) in ordine crescente, e poi, per parità, in base agli indici degli elementi all'interno delle singole classi. La soluzione risultante, che include le somme delle distanze, viene successivamente presentata scorrendo il vettore ordinato.

Complessità Computazionale Le seguenti stime delineano la complessità temporale e spaziale dell'algoritmo di clustering K-Medians.

Complessità Temporale:

- * Allocazione della matrice delle distanze: $\Theta(n^2)$.
  * Calcolo delle distanze: $\Theta(n^2 \cdot p)$.
  * Allocazione e inizializzazione del vettore di `struct componente`: $\Theta(n)$.
  * Allocazione e inizializzazione del vettore di aggregazione (ex `struct contatore`): $\Theta(n)$.
  * Procedura greedy (iterazioni k volte): k iterazioni, ognuna richiede di trovare il miglior mediano. La ricerca del minimo tra le somme delle distanze minime è $\Theta(n)$ per ogni iterazione. L'aggiornamento delle distanze minime per gli $n - r$ individui rimanenti richiede $O(n)$ per iterazione. Dunque, la fase greedy è $\Theta(k \cdot n)$.
  * Quicksort sul vettore di `struct componente`: $\Theta(n \log n)$.
  * Stampa della soluzione: $\Theta(n \cdot l)$.
  * Deallocazione del vettore di `struct componente`: $\Theta(n)$.
  * Deallocazione del vettore di aggregazione: $\Theta(n)$.
  * **Complessità Temporale Totale:** $\Theta(n^2 \cdot p + k \cdot n + n \log n + n \cdot l)$. Il termine dominante è $\Theta(n^2 \cdot p)$.

- **Complessità Spaziale:**
  Matrice delle distanze: $\Theta(n^2)$. Vettore di aggregazione: $\Theta(n)$. Vettore di `struct componente`: $\Theta(n)$. Spazio ausiliario per la ricerca del minimo: $O(1)$. Spazio ausiliario per il Quicksort: $\Theta(\log n)$. Vettore ausiliario per le somme per la stampa: $\Theta(k)$. **Complessità Spaziale Totale:** $\Theta(n^2)$.

4.3 Sintesi dei Costi Complessivi

In questa sezione viene fornita una sintesi della complessità computazionale totale, sia in termini di tempo che di spazio, dell'intero algoritmo, combinando i costi delle diverse fasi descritte. I costi sono presentati considerando il caso medio e il criterio di costo uniforme.

Riepilogo Complessità Temporale (Θ_{totale}): La complessità temporale complessiva dell'algoritmo è data dalla somma delle complessità di ciascuna fase principale. I termini dominanti sono quelli che crescono più rapidamente al crescere delle dimensioni dell'input n (numero di individui) e p (numero di progetti).

Come si evince dalla tabella, il **costo temporale complessivo** dell'algoritmo è dominato dal termine più grande tra i vari algoritmi. Considerando che n e p sono indipendenti e l è la

Table 2: Riepilogo delle Complessità Temporali

Fase dell'Algoritmo	**Complessità Temporale**
InterpretaLineaComando	$\Theta(1)$
CaricaDati	$\Theta(n \cdot p \cdot l \cdot \log p)$
Sistema di Pluralità	$\Theta(n \cdot p + p \log p \cdot l)$
Metodo di Borda	$\Theta(n \cdot p + p \log p \cdot l)$
Metodo di Condorcet	$\Theta(n \cdot p^2 + p^2 \cdot l)$
Algoritmo di Clustering K-Medians	$\Theta(n^2 \cdot p + k \cdot n + n \log n + n \cdot l)$
DeallocaMemoria	$\Theta(n \cdot p + p \cdot l + \max(p^2, n^2))$
Complessità Temporale Totale	$\mathbf{\Theta}(\max(\mathbf{n \cdot p \cdot l \cdot \log p, n \cdot p^2, n^2 \cdot p}))$

lunghezza massima del nome del progetto: In generale, il costo complessivo è il massimo tra i costi delle singole fasi: $\mathbf{\Theta}(\max(\mathbf{n \cdot p \cdot l \cdot \log p, n \cdot p^2, n^2 \cdot p}))$, ragionevolmente $\mathbf{\Theta}(\mathbf{n^2 \cdot p}))$.

Riepilogo Complessità Spaziale (S_{totale}): La complessità spaziale complessiva rappresenta la memoria massima richiesta dall'algoritmo durante la sua esecuzione.

Table 3: Riepilogo delle Complessità Spaziali

Fase dell'Algoritmo	**Complessità Spaziale**
InterpretaLineaComando	$\Theta(1)$
CaricaDati	$\Theta(p \cdot l + n \cdot p)$
Sistema di Pluralità	$\Theta(p)$
Metodo di Borda	$\Theta(n \cdot p)$
Metodo di Condorcet	$\Theta(p^2)$
Algoritmo di Clustering K-Medians	$\Theta(n^2)$
DeallocaMemoria	$\Theta(1)$
Complessità Spaziale Totale	$\mathbf{\Theta}(\max(\mathbf{p \cdot l + n \cdot p, p^2, n^2}))$

Dalla tabella si osserva che il **costo spaziale complessivo** è dominato dal termine più grande tra i costi spaziali delle singole fasi: $\mathbf{\Theta}(\max(\mathbf{p \cdot l + n \cdot p, p^2, n^2}))$, ragionevolmente $\mathbf{\Theta}(\mathbf{n^2}))$.

5 Considerazioni Future e Miglioramenti

(Questa sezione, come precedentemente motivato, contiene per lo più dettagli tecnici non richiesti per il progetto. E' stata inserita per un duplice scopo, da una parte può mostrare di saper contestualizzare in un ambiente applicato le competenze acquisite, e d'altra parte va a completare ulteriormente il progetto in vista di una **possibile** inserzione in un portfolio lavorativo.)
Il presente progetto ha fornito una solida base per l'aggregazione delle preferenze e il clustering in contesti di scelta collettiva. Tuttavia, esistono diverse aree in cui il software potrebbe essere migliorato e esteso per aumentarne la robustezza, l'efficienza e la versatilità.

* **Gestione degli Errori Avanzata**: Attualmente, il programma gestisce errori di base relativi all'input e all'allocazione della memoria. Un miglioramento futuro potrebbe includere la validazione più granulare dei dati (es. controllo che tutti i nomi dei progetti nelle preferenze siano tra quelli inizialmente dichiarati, o che i flag siano solo '¡' o '='). Si potrebbe anche implementare un meccanismo di recupero più elegante degli errori, che permetta di continuare l'esecuzione ignorando righe malformate e loggando gli errori, anziché terminare bruscamente.

– **Ottimizzazione del Clustering**: L'algoritmo greedy per il clustering, sebbene efficace, non garantisce l'ottimalità globale e la sua complessità $O(I \cdot n^2 \cdot p)$ nel caso peggiore può essere elevata per grandi dataset. L'esplorazione di algoritmi di clustering più avanzati o euristici (es. k-medoids con sampling, o l'uso di tecniche di ottimizzazione come simulated annealing o genetic algorithms) potrebbe ridurre il tempo di esecuzione e migliorare la qualità dei cluster.

- **Interfaccia Utente e Visualizzazione**: L'attuale output è testuale. Per un'applicazione più user-friendly, si potrebbe sviluppare un'interfaccia grafica (GUI) che permetta di caricare file, visualizzare i risultati (es. grafici per Condorcet, cluster visualizzati in uno spazio ridotto), e personalizzare i parametri di input (k).
- **Metodi di Aggregazione Aggiuntivi**: Il framework è estendibile per includere altri metodi di aggregazione delle preferenze (es. Metodo di Coombs, Voto Approvazione, o approcci basati su modelli probabilistici) per fornire un'analisi ancora più completa.
- **Gestione di Dataset su Disco**: Per dataset estremamente grandi che non rientrano interamente in memoria, si potrebbe considerare l'implementazione di tecniche di gestione dei dati su disco, riducendo la necessità di caricare tutto in RAM e potenzialmente ottimizzando le operazioni di I/O.
- **Parallelizzazione**: Alcune fasi del calcolo, in particolare il riempimento della matrice di Condorcet o il calcolo delle distanze nel clustering, potrebbero beneficiare della parallelizzazione su sistemi multi-core, riducendo il tempo di esecuzione effettivo.

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[Sen70] Amartya K. Sen. *Collective Choice and Social Welfare*. Holden-Day, San Francisco, 1970.