FRANCESCO ROLLO s273802

Problema 1

Per risolvere il problema, **ho adottato la seguente strategia**:

- nel primo passo scansiono ogni elemento della matrice di partenza alla ricerca di una vocale. Il controllo viene effettuato in un'opportuna funzione booleana **isvowel().** Se viene trovata una vocale, le coordinate i e j del carattere vengono passate come parametri, insieme alla matrice e alle sue dimensioni, alla funzione **change()** che trasforma l'intera riga e l'intera colonna della matrice in un carattere speciale arbitrariamente scelto (**' * '**).
- nel secondo passo, dopo aver calcolato le dimensioni della matrice risultante (n° righe meno il n° di vocali, idem per le colonne), alloco la nuova matrice e tramite una scansione della matrice di partenza copio ogni carattere diverso da **' * '** sequenzialmente nella nuova matrice.

Il codice in allegato presenta una parziale differenza rispetto a quello consegnato:

1. nel codice di partenza per calcolare le nuove dimensioni della matrice risultante ho chiamato una funzione *getdim()* per contare le vocali trovate in modo tale da poter sottrarre questo conteggio alle dimensioni di partenza. Nel codice in allegato ho rimosso questa funzione poiché per contare le vocali bastava di fatto aggiornare un counter direttamente nel primo passo della soluzione.
   Dal momento che l'intera riga e colonna vengono trasformate in '*', se in precedenza era presente una vocale in corrispondenza della stessa riga o colonna questa vocale non verrà rilevata e non contribuirà all'aggiornamento del counter.
2. la copia dei valori nella nuova matrice è stata completata poiché mancante.

Problema 2

La struttura dati dell'esercizio prevede un ADT I classe così definito:

- il file header "***LIST.h***" contiene una dichiarazione di LIST come puntatore ad una **struct list** non ancora definita.
- il file "***LIST.c***" contiene la definizione delle struct per il nodo della lista e per il wrapper. Nella *struct list* si specifica il puntatore alla testa della lista, invece nella *struct node* la chiave come intero, un puntatore all'elemento successivo e uno a quello precedente trattandosi di una lista doppio linkata.

Per risolvere il problema, **ho adottato la seguente strategia**:

- tramite un ciclo *for* itero ogni nodo della lista e ad ogni passo controllo se il nodo è un potenziale candidato da eliminare tramite la funzione booleana **check()** che, ricevuti in input la sua chiave e i due estremi, verifica l'appartenenza o meno all'intervallo [a,b].
- In caso affermativo viene effettuata la chiamata alla funzione **listDelKey()** la quale riceve in ingresso il puntatore al nodo precedente e al nodo stesso. La funzione fa utilizzo di un nodo di appoggio *p* a cui viene copiato il puntatore del nodo successivo, agganciando al *prev* di p il nodo precedente del nodo da rimuovere e al next di

quest'ultimo il nodo p. Il nodo "cancellato" verrà poi eliminato in memoria tramite una **free().**

Il codice in allegato presenta una leggera differenza rispetto a quello consegnato:

1. nel codice aggiornato ho modificato la funzione listDelKey() da void a link per poter di volta in volta assegnare al nodo temporaneo x del ciclo in *f()* il nuovo nodo p. Inoltre ho gestito la rimozione in testa direttamente tramite il nodo p controllando comunque, come nel codice di partenza, che il nodo prev in input non sia NULL e quindi non agganciare nulla a esso.
2. nella funzione f() ho aggiornato x con il suo precedente ritornato dalla listDelKey() per non saltare il nodo agganciato gestendo meglio il caso della testa con un flag.

Problema 3

La soluzione al problema fa utilizzo di un **algoritmo** basato sul modello delle **combinazioni ripetute.**
Il problema richiede di individuare quanti pezzi di ogni tipo è possibile produrre entro un tempo T per minimizzare il delta tra S e la somma dei valori dei pezzi prodotti. Di conseguenza nella visita dello spazio delle soluzioni è possibile utilizzare più di una volta ogni pezzo e il suo ordine non conta. Le strutture dati utilizzate sono:

- una ***struct prods_t*** di P pezzi che contiene per ogni tipo le chiavi valore e tempo di produzione.
- i vettori ***sol*** e ***bestsol*** per salvare temporaneamente la soluzione parziale e la soluzione ottima calcolata ad ogni passo. Il contenuto di entrambi è settato a -1. Inoltre si fa utilizzo del vettore ***val*** di indici per lavorare direttamente sugli indici della struct prods che identificano sequenzialmente ogni pezzo.

Nella **strategia utilizzata** si combina ogni volta una possibile soluzione e nella condizione di terminazione della funzione ricorsiva ***comb_ripet()*** si effettua una chiamata alla funzione ***check()*** per verificare il vincolo del T massimo e il delta più appetibile. In caso affermativo si aggiorna bestsol con la soluzione corrente.
La funzione ***check()*** riceve in input la soluzione generata, la sua dimensione, il vettore val, i vincoli T e S, e un intero bestval passato come riferimento. Iterando sulla soluzione si calcola il valore raggiunto con il tempo impiegato e si verifica che innanzitutto venga rispettato il vincolo su T e poi eventualmente si aggiorna bestval se si è trovato un delta migliore di quello precedente.

Modifiche:
Nel codice in allegato si è optato per aggiungere un output leggibile all'utente. Nella modifica si conta tramite un vettore occ_prods le occorrenze di ogni tipo di pezzo nella bestsol e si stampa con un accesso in *prods[bestsol[i]]* le quantità da produrre per ogni pezzo.

N.B. Nel codice di partenza si è scelto un valore max arbitrario per esplorare una grandezza finita di soluzioni che rispettano i vincoli definiti dal problema. In realtà il max lo si può identificare più precisamente dividendo T con il tipo di pezzo che ha il tempo di produzione più piccolo. Difatti entro un tempo T potrò produrre al massimo max pezzi del tipo a minor tempo di produzione. Nell'esplorazione della soluzione non si può che trovare un numero uguale o inferiore a *max* di pezzi prodotti che rispettano i vincoli richiesti.