domini: rs, utente, prodotto, integer, real

rs composto da un insieme di utenti, un insieme di prodotti e un insieme
di elementi composti da un utente, prodotto e un intero 1 =< i =< 5:
R(U), R(P), R(G)

creaRS() -> rs
registraUtente(rs, utente) -> rs
aggiungiProdotto(rs, prodoto) -> rs
gradisce(rs, utente, prodotto, integer) -> rs
gradimenti(rs, prodotto) -> integer
mediaGradimenti(rs, prodotto) -> real
gradimentiSimili(rs,utente,utente) -> integer

creaRS() = R PRE: \
              POST: R(U), R(P), R(G) = vuoto

registraUtente(R, u) = R'
              PRE: u non app R(U)
              POST: R' = <R(U) U {u}, R(P), R(G)>

aggiungiProdotto(R, p) = R'
              PRE: p non app R(P)
              POST: R' = <R(U), R(P) U {p}, R(G)>

gradisce(R,u,p,g) = R'
              PRE: u app R(U), p app R(P), 1 =< g =< 5
              POST: Se non esiste g' = <u,p,x> con x= 1,5 app G
                    Allora R' = <R(U), R(P) U {p}, R(G) U {g}>
                    Altrimenti R' = <R(U), R(P) U {p}, R(G) / {g'} U {g}>

gradimenti(R, p) = i
              PRE: p app R(P)
              POST: i = numero di elementi di R(G) dove il generico elemento
              x = <u', p', g'> con p' = p

mediaGradimenti(R, p) = r
              PRE: p app R(P)
              POST: r = somma dei g' degli elementi di R(G) dove il generico elemento
              x = <u', p', g'> con p' = p / gradimenti(R,p)

gradimentiSimili(R, u1, u2) = i
              PRE: u1,u2 app R(U)
              POST: i = numero di coppie <x,y> app R(G) dove
              x = <u',p',g'>, y = <u'',p'',g''> con
              u' = u1, u'' = u2, p' = p'', g' e g '' qualsiasi

```
class Gradimento{
public:
        string utente;
        string prodotto;
        int gradimento;
}

class Recommender{
public:
        void creaRS();
        void registraUtente(string u);
        void aggiungiProdotto(string p);
        void gradisce(string u, string p, int g);
        int gradimenti(string p);
        double mediaGradimenti(string p);
        int gradimentiSimili(string u1, string u2);
```

```
private:
        List<string> utenti;
        List<string> prodotti;
        List<Gradimento> gradimenti;
}


cancNodo(Grafo, nodo) -> Grafo
cancArco(Grafo, nodo, nodo) -> Grafo

cancNodo(G, n) = G'
                PRE: G=(N,A), n app N, non esiste a app A tale che
                a = <m,n>
                POST: G' = (N', A) con N' = N \ {n}

cancArco(G,n,m) = G'
                PRE: G=(N,A), n,m app N, <n,m> app A
                POST: G' = (N, A') con A' = A / {<n,m>}


BFS(Grafo G, Nodo u){
        CreaCoda(Q)
        InCoda(u,Q)
        WHILE NOT CodaVuota(Q){
                u = LeggiCoda(Q)
                FuoriCoda(Q)
                esamina u e settalo come "visitato"
                FOR tutti i nodi v app A(u){
                        esamina l'arco (u,v)
                        IF v non è visitato e non app Q
                        InCoda(v,Q)
                }
        }
}
```

In questa realizzazione utilizziamo la struttura Coda per memorizzare volta per volta i nodi adiacenti di u.
La struttura coda è necessaria in quanto abbiamo bisogno di una struttura che ci permetta di analizzare i nodi nell'ordine di adiacenza dal nodo di origine

Grafi rappresentati tramite matrice di adiacenza:
Si utilizza una matrice M di dimensione NxN, dove N è il numero di nodi presenti nel grafo. Se (i,j) app A, allora l'elemento della matrice [eij] = 1 mentre se (i,j) non app A [eij] = 0.
In caso di grafo non orientato allora avremo una matrice simmetrica, cioè [eij]=[eji]=1 se (i,j) app A.
Nel caso gli archi siano pesati, è possibile sostituire il peso dell'arco al posto di 1 mentre se (i,j) non app A allora [eij]= inf.
Questa realizzazione presenta il vantaggio di essere molto semplice da realizzare e inoltre è molto facile generare l'insieme dei nodi adiacenti di un dato nodo i semplicemente scansionando la riga i alla ricerca delle colonne settate a 1.
Questa realizzazione è svantaggiosa per lo spreco di memoria in caso di matrice sparsa.

Grafi rappresentati tramite matrice di incidenza:
Si utilizza una matrice NxA, dove N è il numero di nodi e A il numero di archi del grafo. Ogni riga della matrice rappresenta un nodo, mentre ogni colonna un preciso arco.
Se il nodo i è collegato al nodo j attraverso il nodo k allora [eik] = -1 se l'arco esce da i e [ejk] = 1 se l'arco entra in j (nel caso di grafo orientat…
allora [eik] = 0.
Nel caso di grafo non orientato avremo [eik] = [ejk] = 1.
Lo svantaggio di questa rappresentazione è la difficoltà nel costruire l'insieme di adiacenza di un nodo i.
Infatti dovremmo scansionare la riga i alla ricerca delle colonne k tale che [eik] = -1 e successivamente scansionare le restanti righe j alla ricerca della riga per

cui la colonna k è impostata a 1 [ejk] = 1

Una collisione si verifica quando una funzione hash h produce lo stesso valore per due o più chiavi differenti: h(k1) = h(k2), il dizionario quindi potrebbe tentare di scrivere entrambi gli elementi rappresentati da k1 e k2 nella stessa posizione.
Per gestire le collisioni si possono utilizzare vari metodi:
scansione lineare: se la posizione puntata da h(k2) è già occupata da h(k1) allora si procede a scansionare la posizione successiva nel dizionario, se questa è vuota si memorizza l'elemento altrimenti si continua finchè non si trova una posizione vuota oppure non si deduce che il dizionario è pieno.
Questa tecnica tende a favorire la creazione di agglomerati.
scansione quadratica: come la scansione lineare ma il passo tra una posizione e l'altra è variabile, riducendo cosi il rischio di agglomerati.
Hash doppio: nel caso la chiave k2 generi una collisione si provvede ad un utilizzare una seconda funzione di hash che provvede ad assegnare una nuova posizione all'interno del …
Il rischio di agglomerati è basso, dipende dalla bontà della seconda funzione hash.
Hash aperto con liste di trabocco: Ad ogni posizione del dizionario viene assegnata una lis…
quale vengono memorizzati tutti gli elementi le cui chiavi generano una collisione con la p…
della lista nel dizionario.

Problema di ricerca:
Sia P un problema di ricerca, P={I,S,R,S U {vuoto}, qRic}
In un problema di ricerca siamo interessati a ritrovare una qualunque delle soluzioni per l'istanza i del problema P oppure siamo interessati a sapere se i non possiede alcuna soluzione, per questo introduciamo il simbolo "vuoto" nello spazio di output che indica l'assenza di soluzioni.
qric è la regola che definisce, in base a R, la relazione Rqric che contiene tutti e soli i seguenti elementi: ogni coppia contenuta in R e una coppia <i,"vuoto"> per ogni istanza del problema che non ha soluzione.

La tecnica greedy è una tecnica del paradigma generativo, quindi permette di generare una soluzione senza esplorare lo spazio di ricerca.
Si applica a problemi di ottimizzazione e richiede che la costruzione di un elemento dello spazio di ricerca avvenga in stadi attraverso i seguenti criteri:
Ad ogni stadio i , per la componente i-esima, tra tutti i valori ammissibili viene scelto il migliore rispetto ad un determinato criterio.
Una volta scelto il valore che soddisfa la componente i-esima si passa al prossimo stadio senza più tornare indietro.
Quindi si presuppone che l'algoritmo disponga di un metodo per organizzare in stadi la costruzione di un elemento dello spazio di ricerca. Ad ogni passo viene quindi scelta la soluzione migliore per lo stadio i-esimo, questa strategia non sempre porta alla costruzione della soluzione ottima, dipende dalla caratteristica del problema, cioè deve avere due caratteristiche:
La proprietà della scelta greedy: Ci assicura che è possibile ottenere una soluzione ottima globale prendendo decisioni che sono ottimi locali per ogni stadio.
Presenta una sottostruttura ottima: La soluzione ottima del problema contiene al suo interno soluzione ottime per i suoi sottoproblemi.

insSottoAlbero(Nodo, Albero, Albero) -> Albero
insPrimoSottoAlbero(Nodo,Albero, Albero) -> Albero

insPrimoSottoALbero(n,T, T') -> T''
        PRE: n app T, T e T' non vuoti
        POST: T'' è il nuovo albero ottenuto inserendo il nodo m, radice di T', come figlio del nodo n.

insSottoAlbero(n, T, T') -> T''
        PRE: T=(N,A), n app N, T e T' non vuoti
        POST:T'' è il nuovo albero ottenuto aggiungendo la radice m di T' come succFratello di n

La rappresentazione di un albero n-ario come vettore di padri è ottenuta
numerando ogni nodo dell'albero con un intero che rappresenta la sua posizione
nel vettore. Ogni elemento contiene un cursore alla posizione del padre occupata
nel vettore. Questa rappresentazione è molto semplice e permette di esaminare
con semplicità l'albero partendo dalle foglie e risalendo fino alla radice.
Tuttavia gestire la rimozione e l'inserimento di sottoalberi diventa molto complesso.
La rappresentazione con lista di figli invece associa ad ogni nodo, che viene
rappresentato come elemento di un vettore, una lista di riferimenti a ciascun figlio
del nodo. Questa realizzazione permette di superare il vincolo di non poter
sapere quanti figli ha ogni nodo attraverso l'utilizzo di una struttura dinamica.
La rappresentazione mediante cursori permette di rappresentare i nodi all'interno
di un vettore dove ogni elemento contiene l'etichetta del nodo, un cursore al suo primo fi…
e un cursore al suo fratello successivo secondo la relazione d'ordine.

```
preVisita(Albero T, Nodo a)
        Esamina il nodo a
        IF NOT foglia(T,a)
                Nodo b = primoFiglio(a)
                preVisita(T, b)
                WHILE NOT ultimoFratello(b)
                        b = succFratello(b)
                        preVisita(T, b)
                END
        END
END

postVisita(Albero T, Nodo a)
        IF NOT foglia(T,a)
                Nodo b = primoFiglio(a)
                postVisita(T, b)
                WHILE NOT ultimoFratello(b)
                        b = succFratello(b)
                        postVisita(T, b)
                END
        END
        Esamina il nodo a
END

inVisita per i = 1
inVisita(Albero T, Nodo a)
        IF NOT foglia(T,a)
                Nodo b = primoFiglio(a)
                inVisita(T, b)
                Esamina il nodo a
                WHILE NOT ultimoFratello(b)
                        b = succFratello(b)
                        inVisita(T, b)
                END
        END
END
```

Una pila può essere rappresentata molto semplicemente tramite vettore memorizzando
un cursore alla testa della pila, cioè all'ultimo elemento inserito in fondo al vettore.
In questo modo sia le letture che le scritture nella pila avvengono con tempo costante,
una limitazione di questa rappresentazione è quella di dover fissare un limite massimo
agli elementi che è possibile inserire nella pila.
Gestire una coda con un vettore invece risulta più problematico in quanto gli inserimenti
avvengono in fondo al vettore mentre le letture dalla cima, quindi è necessario spostare
ogni volta gli elementi. Oppure rappresentare la coda come un vettore circolare che va da
0 a maxlung -1, si considera la posizione 0 come la successiva di maxlung-1 e si
memorizzano dei cursori che puntano alle estremita' della coda, in questo modo si risolve
il problema di dover spostare gli elementi.

La tecnica di backtracking fa parte del paradigma selettivo, quindi genera una soluzione
al problema analizzando lo spazio di ricerca. L'algoritmo di backtracking suddivide
la generazione di un elemento dello spazio di ricerca in stadi, ad ogni stadio viene

scelto un componente che contribuisce a formare una soluzione parziale al problema. La caratteristica del backtracking è che è possibile che la soluzione parziale scelta sia fallimentare, ovvero arrivati ad un certo stadio non contribuisca più a generare un elemento dello spazio di ricerca, in questo caso si fa un passo indietro e si ritorna alla componente precedente, cercando un altra
E' possibile visualizzare la costruzione degli elementi dello spazio di ricerca come un alb…
la radice rappresenta una soluzione parziale fittizia mentre tutti i nodi all'interno di livello j > 0 rappresentano tutte le possibili soluzioni parziali per cui sono state scelte le prime j componenti ed hanno tanti figli quanti sono i possibili modi di costruire una soluzione parziale con j+1 componenti.
Ogni foglia dell'albero di ricerca rappresenta un elemento dello spazio di ricerca.
Per un albero di ricerca è necessario stabilire un metodo per rappresentare ogni soluzione parziale, ossia i nodi degli alberi.
Un metodo per stabilire se una soluzione parziale viola i vincoli imposti dal problema.
Una funzione di ammissibilità che equivale a TRUE quando un nodo interno non rispetta i vincoli del problema o quando un nodo foglia non rappresenta una soluzione ammissibile per il problema

La tecnica di backtracking, applicata ad un problema di ottimizzazione, cerca la prima soluzione ammissibile per il problema e la memorizza come ottimo attuale, poi torna indietro e tenta altre strade per cercare nuove soluzioni e aggiornare la variabile ottimo attuale se la nuova soluzione risulta migliore, continuando quindi a generare nuovi elementi dello spazio di ricerca finquando è possibile.
La tecnica greedy invece genera la soluzione ottima costruendo soluzioni ottime locali ai sottoproblemi che compongono il problema senza mai tornare indietro. Questo, a dispetto della tecnica del backtracking è molto più veloce ma non garantisce che la soluzione trovata sia la ottima globale, dipende dalle caratteristiche del problema.

La tecnica enumerativa fa parte del paradigma selettivo, si basa sulla sistematica ispezione di tutto lo spazio di ricerca e quindi garantisce una terminazione se questo è finito. Per un problema di ricerca ci si fermerà una volta trovata la prima soluzione ammissibile mentre per uno di ottimizzazione bisognerà controllare tutte le soluzioni ammissibili o comunque continuare fino ad esaurire lo spazio di ricerca.
Per garantire la visita sistematica, allo spazio di ricerca è associato una relazione d'ord… totale sui suoi elementi in modo da stabilire un metodo per individuare il primo elemento dello spazio di ricerca da cui avviare l'ispezione, un metodo che consenta, dato un elemento dello spazio di ricerca, di individuarne il successivo, un metodo che verifichi se tutti gli elementi dello spazio di ricerca sono stati analizzati.

La tecnica del divide-et-impera deriva dal principio di decomposizione induttiva, è necessario disporre di:
Una relazione d'ordinamento sulle istanze del problema basato sulla dimensione dell'input
Un metodo di risoluzione diretto di tutte le istanze del problema che non superano un certo limite massimo nell'input
Un metodo per suddividere i dati di input di un problema in diverse parti, di dimensione minore di quella principale e che siano dati di input di una nuova istanza dello stesso pro…
Un meccanismo per comporre le soluzioni delle istanze generate dalla suddivisione e generare quindi la soluzione al problema originario.

Passi:
1. se l'input ha dimensione inferiore a k, applica un metodo diretto per generare una soluz…
2. (divide)altrimenti suddividi l'input dell'istanza in parti di dimensioni più piccole del…
3. Esegui ricorsivamente 1 e 2 sugli input individuati al passo precedente
4. (impera)Componi la soluzione all'istanza originaria del problema componendo le soluzioni… istanze generate dalla suddivisione dell'input