Il container perfetto

Il container perfetto

In questo articolo parleremo di *containers* e di *complessità computazionale.*
Daremo anche uno sguardo ad *MVC* applicato al framework grafico *Qt*.
Ci renderemo conto che le scelte che facciamo quando progettiamo un software possono essere determinanti per le prestazioni dei nostri programmi.
Persino oggi, anche avendo a disposizione una potenza di calcolo che era semplicemente impensabile in passato, dobbiamo essere consapevoli delle scelte che facciamo.
Ho deciso di trattare l'argomento principale di questo articolo, ovvero: la scelta di un *container*, affrontandolo mediante una *GUI* per *desktop* completa che utilizza *Qt*.
I concetti espressi, ad ogni modo, sono validi per *qualunque* software, indipendentemente dal linguaggio e dalle tecnologie impiegate.
Per chi fosse interessato a visionare o provare il codice dell'esempio che utilizzeremo, lo può trovare al mio repository di GitHub: AddressBook.

due chiacchere su QT

Qt è un framework grafico cross platform e open source.
Storicamente è stato molto utilizzato in ambito desktop anche se da qualche anno estende il supporto alle applicazioni mobile.
La struttura interna di *Qt* e le sue *API* pubbliche appaiono come una complessa gerarchia di classi *C++*.

qt C++ EXTENSIONS

In realtà occorre fare una precisazione: il *C++* con cui si scrivono le interfacce grafiche per *Qt* rappresenta

un’*estensione non standard* del linguaggio.
Significa che quando scriviamo codice per *Qt*, abbiamo a disposizione nel linguaggio alcune *keyword* non interpretabili direttamente dal compilatore *C++*.
Bisogna dire che purtroppo, ereditiamo anche alcune limitazioni, a volte *tricky,* come per esempio l’impossibilità di definire classi *template* per i tipi derivati da QObject.
Quando compiliamo con la suite *Qt*, dietro le quinte, stiamo avviando una *toolchain.*
Il codice originale, per prima cosa, viene processato dal compilatore *moc* in codice *C++ vero* e solo successivamente, passato al compilatore *C++*.

una questione pratica

Perchè gli ingegneri di *Qt* operarono questa scelta?
Non sarebbe stato meglio per il programmatore che usa *Qt,* scrivere direttamente codice *C++* senza estensioni?
Il problema è di tipo pratico: nel *C++* di *Qt* i vari componenti inviano e ricevono *eventi* in forma *asincrona* utilizzando un meccanismo chiamato: *signals* e *slots*.
D’altronde le interfacce grafiche lavorano principalmente per *eventi* e “*signal* e *slot*” è l’infrastruttura di *Qt* per la gestione di questo aspetto.
Fornire all’utente *Qt* direttamente un’interfaccia *C++* per gestire un’infrastruttura del genere deve essere sembrato eccessivamente complicato e verboso.
La scelta è stata quindi, per la nostra comodità, l’introduzione di due *keyword non C++*: signalse slotscon cui marcare rispettivamente i *metodi* che rappresentassero un *segnale* e quelli per *ricevere* quel *segnale*.
Vediamo un piccolissimo esempio di una classe Counter minimale presa proprio dalla guida di Qt per spiegare i *signals* e gli *slots*:

```
#include <QObject> class Counter : public QObject
{ Q_OBJECT public: Counter() { m_value = 0; } int value()
const { return m_value; } public slots: void setValue(int
value); signals: void valueChanged(int newValue); private: int
```

```
m_value;
};
```

Questa classe possiede un metodo “segnale” e un metodo “slot”. Emettere quel “segnale” è molto semplice e si realizza mediante un’altra keyword: emit:

```
void Counter::setValue(int value)
{ if (value != m_value) { m_value = value; emit valueChanged(value); }
}
```

A questo punto possiamo *connettere* un *signal* di un’istanza di Counter con uno *slot* di un’altra istanza di Counter mediante il metodo connect:

```
Counter a, b;
QObject::connect(&a,   &Counter::valueChanged,   &b, &Counter::setValue); a.setValue(12); // a.value() == 12, b.value() == 12
b.setValue(48); // a.value() == 12, b.value() == 48
```

Come possiamo vedere il codice per gestire l’invio e la ricezione di un segnale che trasporta un’informazione è tutto sommato abbastanza contenuto.
Se avessimo dovuto gestire tutto questo direttamente in *C++*, credetemi, non ce la saremmo cavata così facilmente.
Ad ogni modo, chiunque può farsi un’idea dando un’occhiata al codice *C++* prodotto dal compilatore *moc*.

MVC

MVC o *model view controller* è uno dei più *vetusti* pattern architetturali utilizzati per l’implementazione di interfacce grafiche utente: se ne cominciò a discutere sin dagli anni 70.
Tuttavia, era una *gran* bella idea e ancora oggi rappresenta *de facto* lo standard di riferimento per quando si vuole implementare una buona GUI scalabile e mantenibile nel tempo.
In *estremissima* sintesi *MVC* fornisce le linee guida per una corretta separazione ed interazione dei ruoli tra *modello*, *controllo* e *vista*.

Il modello è la parte più nobile dei tre attori: rappresenta i nostri dati nella loro forma più *pura* e fornisce dei metodi per essere *interrogato*, *popolato* ed *aggiornato*.
Il *controllo* è quella parte della nostra applicazione preposta ad interagire con il *modello* per quanto concerne il popolamento e l'aggiornamento dei dati in forma interattiva.
Un *modello* nella realtà, può cambiare il suo stato indipendentemente da fattori diretti comandati dall'utente.
Pensiamo per esempio ad una *chat*, nella quale il *modello* riceve messaggi esterni dal nostro interlocutore.
Il *controllo* consente anche di modificare la *vista* per modificare il modo con cui i dati del *modello* sono presentati.
Nella forma più pura di *MVC*, la *vista* non può interagire direttamente con il *modello*.
Quello che sicuramente è *precluso* alla *vista* è la modifica del *modello* da cui attinge i dati.
La vista è un compomente "*stupido*", non deve sapere *nulla* della sorgente da cui attinge i dati.
Tutti i framework grafici contemporanei sia per desktop che per mobile supportano *MVC* in una qualche forma ed ovviamente *Qt* non fa eccezione.

MVC secondo Qt

Qt implementa *MVC* in maniera molto elegante e offre tutta una serie di componenti pronti all'uso che possono essere combinati tra loro per ottenere i comportamenti più svariati.
In particolare *Qt* offre già tre tipologie di *views* già pronte che effettuano il *rendering* di modelli che rappresentano:

Naturalmente, se il nostro modello non è coperto da una di queste tipologie, possiamo implementare la nostra vista estendendo: QAbstractItemView.
Ad ognuna di queste viste possiamo agganciare il nostro modello semplicemente invocando il metodo: setModel.
Naturalmente la vista non sa nulla di come è fatto il modello, quello che la vista vede è solo un'interfaccia del tutto astratta (ricordate? la vista è "*stupida*"!).

È tutto qui: una volta impostato il modello sulla nostra vista, quest'ultima si preoccuperà di *interrogare* il modello sottostante per effettuare il *rendering* dei dati che ha senso siano visualizzati in un determinato momento.
Per esempio, se la nostra applicazione mostra la vista di una tabella in un database e la finestra è dimensionata di modo che si vedano solo 5 righe, la vista è abbastanza intelligente da chiedere al modello sottostante le *sole* righe visibili in quel momento.
Quello che è richiesto al nostro modello è che derivi la classe QAbstractItemModel.
Naturalmente *Qt* fornisce anche modelli più specializzati rispetto a *QAbstractItemModel* e potrebbe essere una buona idea partire da quelli se buona parte delle funzionalità che ci servono sono già state implementate.
Al nostro modello si richiede anche che sia un *buon* modello: se la vista richiede una determinata riga, questa operazione non *dovrebbe*, nei limiti del possibile, generare *ulteriore* lavoro inutile.
Inoltre il nostro modello dovebbe essere organizzato in modo tale da reperire l'insieme dei dati interessanti nel modo più *efficiente* possibile.
Cominciamo pertanto, a dare un'occhiata al vero tema di questo articolo.

Address book

AddressBook è un'applicazione grafica *Qt didattica.*
Mostra cioè, come due differenti *modelli*, che alimentano una *medesima* vista e forniscono i *medesimi* dati, differiscono invece, in maniera significativa nelle loro *prestazioni*.
Ma che cosa fa esattamente *AddressBook*?
AddressBook è come una *rubrica*: legge un file di testo che contiene *n righe*, ed associa ogni *parola* di ogni riga con la riga stessa.
Quando poi, l'utente comincia a digitare una *parola* in una *input-box*, vengono mostrate in una vista tutte le righe che

contengono quella *parola*.
Non occorre che la parola sia completa: infatti, come quando cerchiamo un nome nella rubrica del nostro cellulare, digitare per esempio: “*giu*”, mostra tutti i nostri contatti per i quali il nome o il cognome cominciano per “*giu*”.
Se per esempio il file con cui alimentiamo il nostro modello contenesse l’incipit dell’*Eneide*:

Arma virumque cano, Troiae qui primus ab oris Italiam fato profugus Laviniaque venit

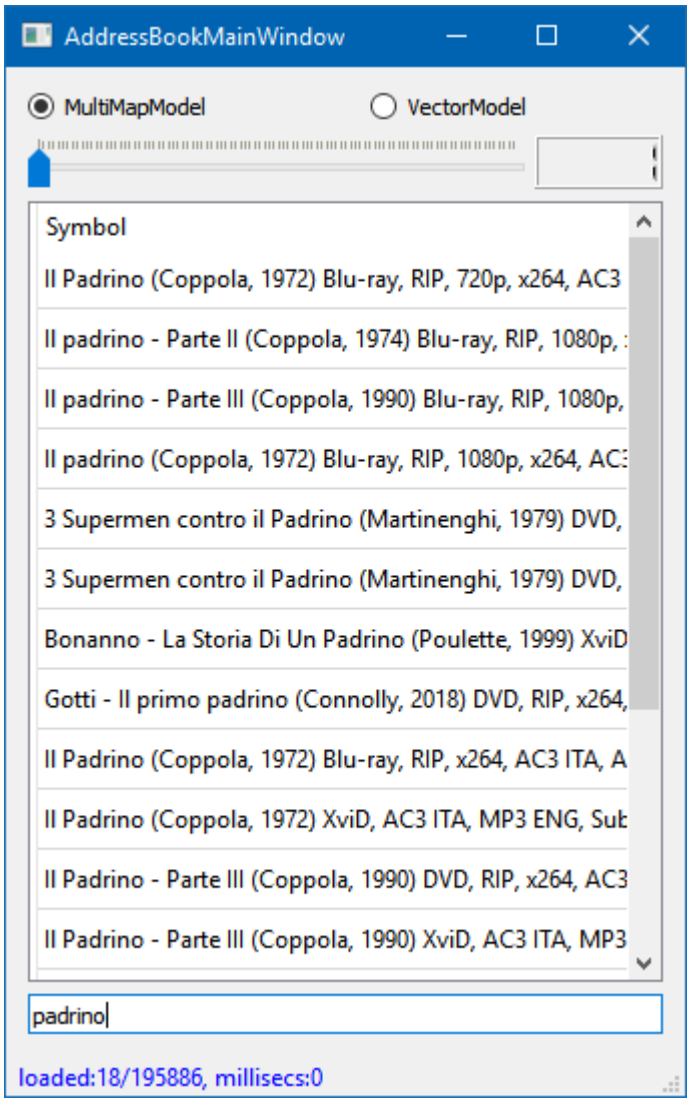
Il programma effettuerebbe la *tokenizzazione* della riga in:

Arma virumque cano, Troiae qui primus ab oris Italiam fato profugus Laviniaque venit

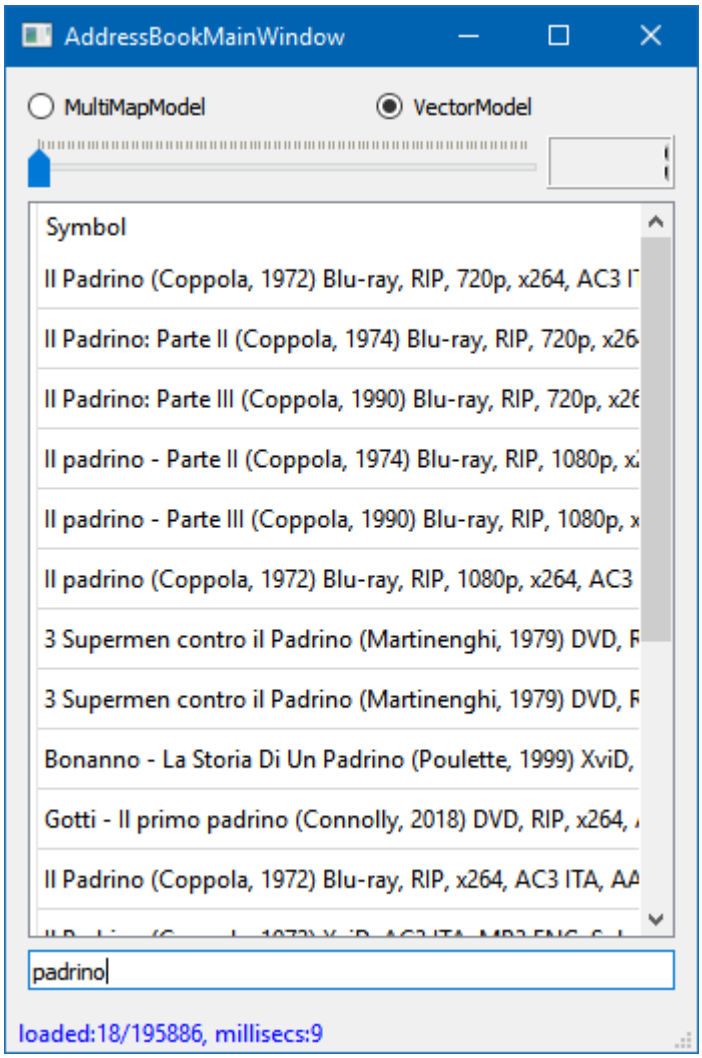
ed associerebbe ogni *token* alla riga stessa.
Per cui, digitare ad esempio: “*pri*” nella input-box, farebbe *selezionare* al modello anche il venerando incipit virgiliano.

Address book in azione

Una volta avviato, *AddressBook* mostra una *vista* vuota.
Se digitiamo qualcosa nella *input-box* in basso potremmo vedere popolarsi la *vista* con tutte le righe che contengono almeno un *token* che comincia con la parola digitata:



Come potete notare, nelle mie prove ho deciso di utilizzare un file di testo, preso in rete che contiene un bel listone – 195886 righe – di film nei più svariati formati.
Sono sicuro che il possessore del file ha in casa un gran bel *NAS* e ha deciso di convertirsi tutti i suoi legalissimi *Blu-ray* e i suoi legalissimi *DVD* nei formati più disparati.
Potete notare come digitare la parola *padrino* nel *controller input-box*, popola la *vista* con le righe del *modello* che la contengono.
Notiamo anche che ci sono due controlli di tipo *check-box* in alto denominati *MultiMapModel* e *VectorModel.* Se proviamo a selezionare *VectorModel* e ripetiamo l'inserimento della parola padrino otteniamo *quasi* esattamente lo stesso risultato:



Cosa *non* è cambiato?
Come prima, l'aver digitato la parola *padrino,* mostra nella *vista* le righe del *modello* che la contengono ovvero le 18 righe di prima.
Notiamo che però è cambiato l'*ordinamento* delle righe mostrate dalla vista.
Nella *status bar* in basso, inoltre, è cambiato l'*elapsed* espresso in millisecondi, relativo al **tempo impiegato** per caricare la porzione del *modello* richiesto.
Se prima occorreva un tempo ***t*** *submillisecond* per generare la porzione di modello richiesta, adesso ne servono addirittura 9!

UN BUON MODELLO

In che cosa differiscono *MultiMapModel* e *VectorModel?*
Perchè è lecito aspettarsi che per l'esperienza d'uso che

vogliamo fornire ad *AddressBook*, *MultiMapModel* sia senza ombra di dubbio migliore di *VectorModel*?
La risposta è tutto sommato semplice ma apre a parecchie considerazioni, vediamo pertanto le definizioni di *MultiMapModel* e di *VectorModel*:

MultiMapModel

```
class AddressBookMultiMapModel : public AbstractBookModel {
public:
... private: std::multimap<std::string, AddressBookModelEntry>
symb_map_;
};
```

VECTORModel

```
class AddressBookVectorModel : public AbstractBookModel {
public:
...    private:    std::vector<std::pair<std::string,
AddressBookModelEntry>> symb_array_;
};
```

Tralasciamo i dettagli dei metodi di queste due classi e concentriamoci sulle differenti *rappresentazioni* del nostro modello.
In AddressBookMultiMapModelstiamo utilizzando una std::multimapuno dei container associativi della standar library mentre in AddressBookVectorModelutilizziamo un *random access* contanier: std::vector.
Giusto per essere espliciti: la multimappa associa una std::string, che contiene il *token* che funge da *chiave*, con un AddressBookModelEntry, che altro non è che un wrapper ad un'altra stringa che contiene l'intera riga del file.
La scelta di un wrapper è motivata dal fatto che in futuro potremmo voler aggiungere anche altre informazioni, fosse anche solo il colore con il quale vogliamo visualizzare il carattere della riga.

I containers MAP & MUltimap

Una *mappa* ed una *multimappa* sono containers *associativi*, nel

senso che associano una *chiave* ad un *singolo valore* (*mappa*) o a *n valori* (*multimappa*).
Recuperare un *elemento*, inteso come coppia: *[chiave, valore]*, da una *mappa* (o gli *elementi* dalle *multimappe*) conoscendo la *chiave* è molto semplice:

```
//the same code works for std::map also
std::multimap<char,int> mymm;
mymm.insert (std::make_pair('x',10));
std::cout << "x => " << mymm.find('x')->second << '\n';
```

L'ORDINAMENTO CHE FA LA DIFFERENZA

Le *mappe* e le *multimappe* della *standard library* sono *containers associativi ordinati.*
Dalla definizione leggiamo:

> Internally, the elements in a multimap are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

Che significa?
Significa che gli *elementi* delle *mappe* e delle *multimappe* sono mantenuti *ordinati* dal container seguendo il *criterio di ordinamento* definito per le *chiavi*.
L'importante, se non *fondamentale* vantaggio rispetto per esempio ad un *array*, è che effettuare un'*enumerazione* degli elementi di una std::multimapavverrà secondo l'ordinamento prestabilito per le *chiavi*.
L'ordinamento avviene per costruzione ad ogni *inserimento* o *rimozione* di un elemento, dato che, internamente, una mappa è implementata mediante un *albero binario bilanciato*.
Esistono ovviamente anche containers associativi che non si basano su alberi binari bilanciati ma bensì su *indici,* calcolati come il risultato di funzioni *hash* applicate alle *chiavi*.
Tuttavia, i containers basati su *hashtables* non possiedono *nessuna* nozione di ordinamento sugli elementi.

multimap: Complessità computazionale

Quanto costa calcolare il set di elementi corrispondenti ad una *chiave parziale* con una multimappa?
Inizialmente, dobbiamo capire quanto *costa* raggiungere il punto dell'*albero bilanciato* che rappresenta la *prima chiave* che non sia *strettamente minore* della chiave parziale offerta.
Successivamente, dobbiamo capire quanto costa *enumerare* tutte le chiavi *successive* alla nostra che siano ammissibili per il *set*.
Visualizziamo cosa dovrebbe succedere con un esempio.
Questa volta usiamo la prima strofa del capolavoro di Gabriele D'Annunzio: La pioggia nel pineto:

Taci. Su le soglie
del bosco non odo
parole che dici
umane; ma odo
parole più nuove
che parlano gocciole e foglie
lontane.
Ascolta. Piove
dalle nuvole sparse.
Piove su le tamerici
salmastre ed arse,
piove su i pini
scagliosi ed irti,
piove su i mirti
divini,
su le ginestre fulgenti
di fiori accolti,
su i ginepri folti
di coccole aulenti,
piove su i nostri volti
silvani,
piove su le nostre mani
ignude,
su i nostri vestimenti
leggieri,
su i freschi pensieri
che l'anima schiude

novella,
su la favola bella
che ieri
t'illuse, che oggi m'illude,
o Ermione.

che succede se digitiamo la chiave parziale: “*pio*” ?

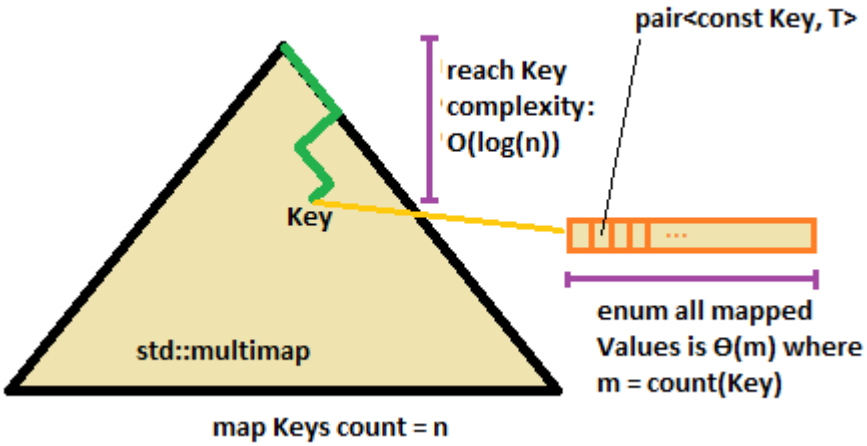
```
// la chiave parziale digitata è: "pio"
// un elemento nella multimappa è una coppia: [Key, Value]
// il primo elemento nella mappa la cui chiave NON è precedente a "pio" è: ["piove", "Ascolta. Piove"].
// partiamo da quello e enumeriamo fintanto che la chiave dell'elemento comincia per: "pio".
// il risultato è che il set di elementi selezionati sarà: ...
["piove", "Ascolta. Piove"]
["piove", "Piove su le tamerici"]
["piove", "piove su i pini"]
["piove", "piove su i mirti"]
["piove", "piove su i nostri volti"]
["piove", "piove su le nostre mani"]
... // prendiamo a questo punto il Value di ogni elemento e lo visualizziamo nella vista.
```

Il codice di AddressBook che fa questo mestiere è:

```
//When we digit something in the input-box this method of AddressBookMultiMapModel is called.
//This method computes the set of lines we are interest to.
void AddressBookMultiMapModel::offerKey(const QString &key, std::vector<AddressBookModelEntry *> &cur_symbols)
{ auto kstr = key.toStdString(); std::transform(kstr.begin(), kstr.end(), kstr.begin(), ::tolower); for(auto it = symb_map_.lower_bound(kstr); it != symb_map_.end(); ++it) { if(it->first.find(kstr) == 0) { cur_symbols.push_back(&it->second); } else { break; } }
}
```

Questo codice usa il metodo lower_bounddi std::multimap, che data una chiave *Key,* restituisce un *iteratore* al primo elemento la cui *chiave* non è considerata *minore* di *Key*.
Ottenuto l’iteratore, poichè sappiamo che enumerare i

successivi elementi avverrà in maniera *ordinata*, possiamo selezionare ogni successiva chiave che inizi *esattamente* per *Key*.
Nel momento in cui, durante l'enumerazione, questo non sia più vero, possiamo tranquillamente terminare il calcolo del *set*, perchè è *impossibile* che nella multimappa esistano altre chiavi che *soddisfino* la nostra *Key*.
Proviamo a visualizzare schematicamente quello che dovrebbe succedere:



Siamo in grado a questo punto dare una stima della *complessità computazionale* richiesta per calcolare il *set* di elementi da mostrare nella vista di AddressBook:

T(n, m) = ***O(log(n)) + θ(m)***
with n = multimap.size()
with m = multimap.count(Key), m <= n

Possiamo notare che la funzione di complessità *T* è a due variabili e che il peso di gran lunga maggiore è determinato dalla variabile *m* perchè è associata ad una funzione lineare di *notazione θ (theta).*
All'atto pratico significa che se nella nostra multimappa ci sono *m* elementi che soddisfano la chiave *Key*, allora li dobbiamo *per forza* enumerare *tutti*.
Quindi, più sono questi elementi, più tempo impiegheremo per calcolare il nostro set.

Il caso *degenere* avviene quando $m = n$, ovvero quando *tutti* gli elementi della multimappa soddisfano la chiave *Key*.
In questo caso non noteremmo differenze prestazionali tra AddressBookMultiMapModele AddressBookVectorModel.

Vector

A questo punto dovrebbe essere chiaro perchè AddressBookVectorModelche si basa su un semplice std::vector, nella pratica, non può far meglio di AddressBookMultiMapModel.
La complessità computazionale per calcolare il set di elementi associati a *Key,* con un *vector* non ordinato, rimane inchiodata a:

$T(n) = \boldsymbol{\theta(n)}$

Non importa se nel *vector*, per esempio, esistesse un solo elemento a soddisfare la chiave *Key*.
Il vettore lo dovremmo necessariamente scorrere tutto quanto per essere sicuri di controllare tutti gli elementi. Non possiamo fare di meglio.

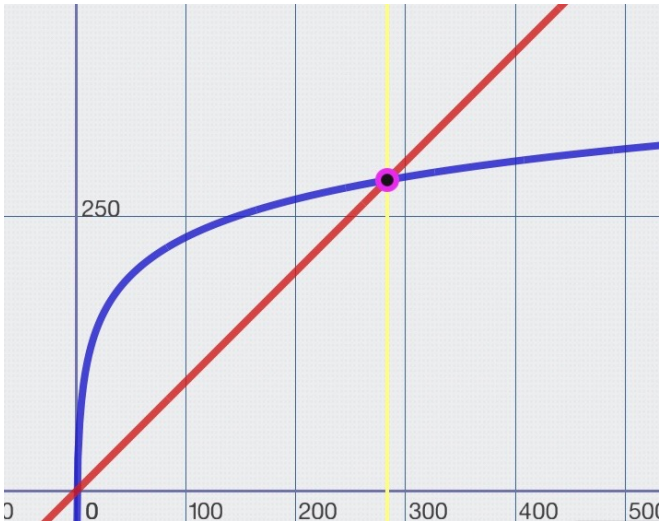
Qualche parola sulle costanti k

In questa nostra disquisizione sulle complessità abbiamo del tutto tralasciato il ruolo delle *costanti K* che andrebbero applicate *a fattore* delle funzioni *T*.
Abbiamo cioè fatto un'analisi computazionale prendendo in considerazione solo le complessità *teoriche pure* dei container coinvolti.
Le *costanti K* danno una misura della *bontà* dell'*implementazione* di un algoritmo.
Per esempio se l'implementazione della multimappa fosse parecchio *scadente* potremmo ipotizzare, per il nostro problema, una funzione di complessità *T* di questo tipo:

$T(n, m) = \boldsymbol{O(26log(n)) + \theta(12m)}$

In questo caso, dove ipotizziamo invece nel caso di vector: $K = 1$ potremmo affermare che per qualche $n < h$ l'implementazione

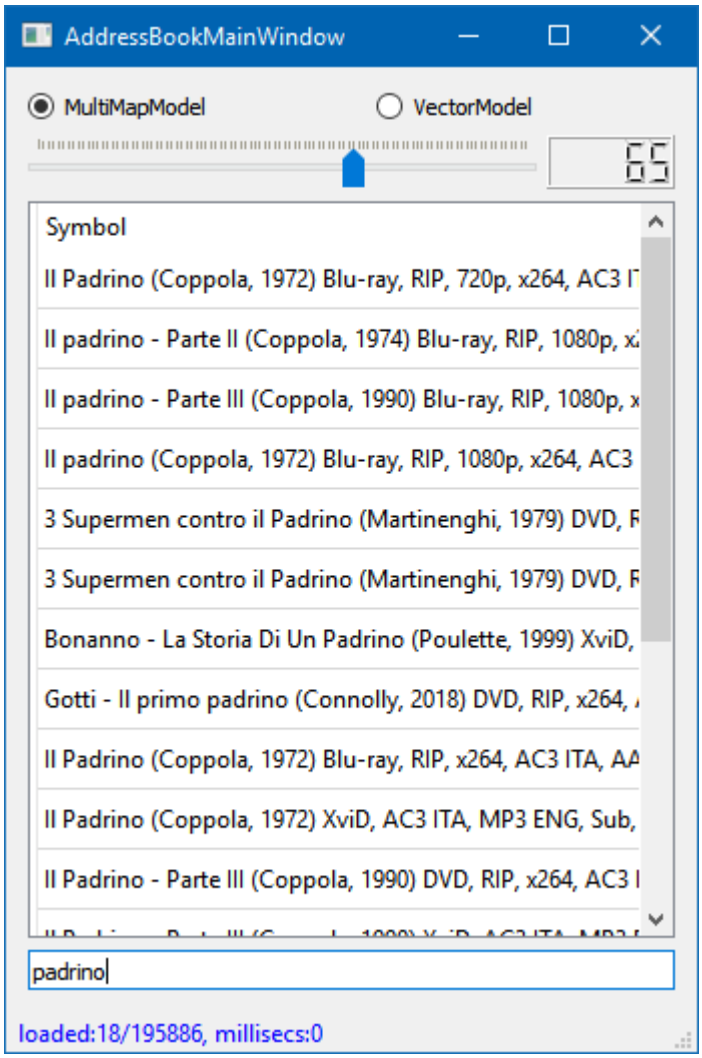
con *vector* sarebbe migliore rispetto a quella della *multimappa*.
Nel caso di *m = 1,* in cui cioè il set di elementi associati a *Key* è 1, graficamente avremmo una situazione del genere:



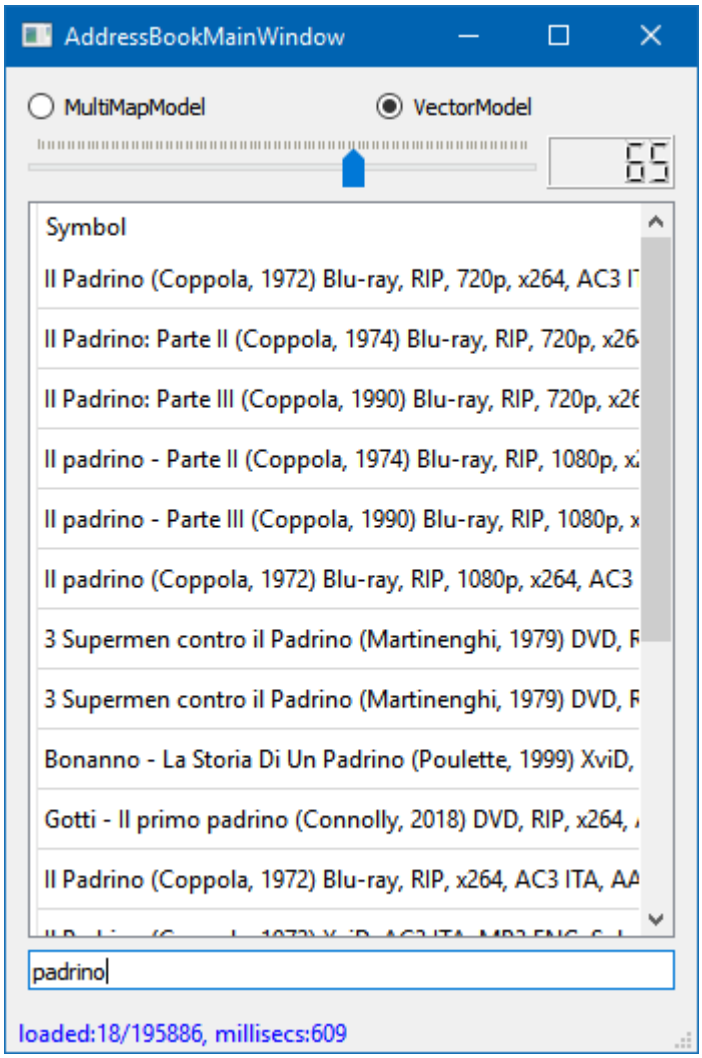
A causa delle alte costanti *K* applicate alla funzione logaritmica esiste certamente un *h*, che rappresenta la soluzione di *T(multimap) = T(vector)* per la quale le dimensioni dei container con *n < h* rendono la soluzione con *vector* più appetibile.
Ma anche con *K irragionevolmente* alti come in questo caso, *asintoticamente* non ci sarebbe gara: una *T logaritmica* vincerebbe a man bassa su una *T lineare*.
È per questo motivo che quando si effettua una analisi computazionale in prima battuta si tralasciano le costanti *K*.

Address book e le costanti k

Possiamo, in ogni caso *simulare* **alte** costanti *K* nel nostro esempio AddressBook.



Possiamo agire sullo *slider* e impostare per esempio un $K = 65$, ma poichè in questo caso
$m = 18$, quindi molto piccolo, ancora otteniamo un **t** *submillisecond*.
La parte *logaritmica* di T ci salva, perchè non è molto sensibile ad un K elevato.
Con *vector* invece non siamo altrettanto fortunati:



In questo caso stiamo scaricando $K = 65$ interamente sopra una funzione *lineare*.
Come vedete il risultato è ben diverso dal caso di $K = 1$.

conclusioni

Sembra che siamo arrivati alla fine.. *se Dio vuole!*
Ho titolato questo articolo: “*Il container perfetto*”, ma quale è quindi questo *container* perfetto?
Semplice: *non esiste*.
O meglio, non esiste un container *buono per tutti i problemi*.
Ogni *container* tipicamente è *progettato* per fare *bene* determinate cose e fare *peggio* altre.
Tuttavia, deve essere l’*esperienza* e la *sensibilità* di noi: *software developers* che ci deve far capire quando una soluzione calza bene per il nostro problema e quando invece dovremmo valutare se esistono metodi migliori.

Vi ringrazio davvero tanto per l’attenzione.
A presto!

https://italiancoders.it/il-container-perfetto/