TECNICHE DI PROGRAMMAZIONE

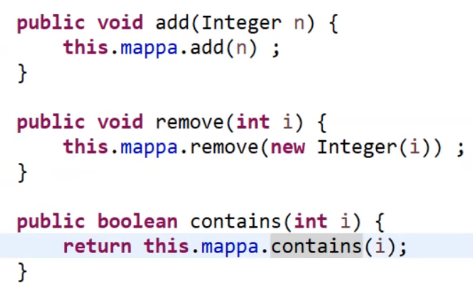
08/04/19

CONTINUAZIONE ESERCIZIO ‘QUADRATO MAGICO’.

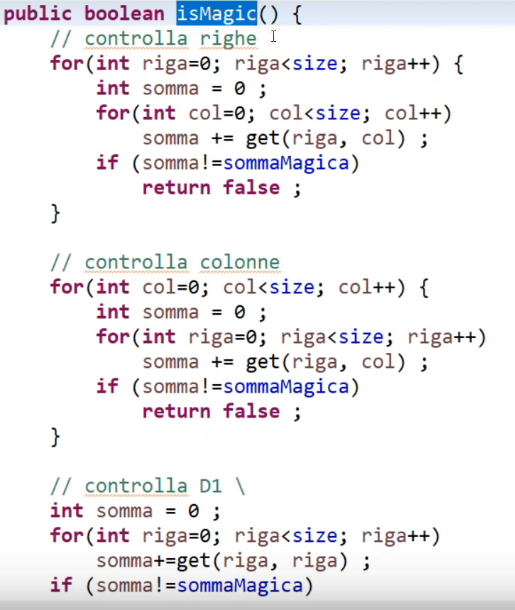
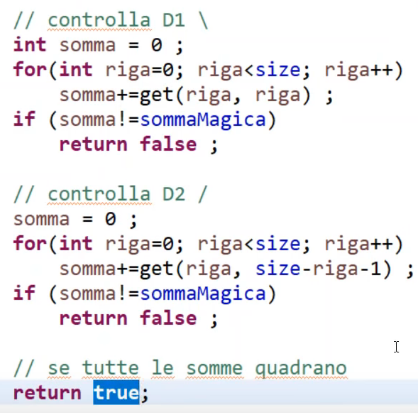
Il professore ha terminato l’esercizio per fatti suoi implementando alcune righe di codice.

Ha aggiunto il metodo isMagic() che permetteva appunto di controllare se il Quadrato finale era magico.

Ha inoltre implementato i metodi ‘add’, ‘contains’ e ‘remove’ nella classe Quadrato.



L’unica cosa un po’ più difficile è stato implementare appunto il metodo isMagic(). E’ un metodo noioso in quanto bisogna controllare riga per riga se la loro somma è uguale alla ‘sommaMagica’ (variabile); stessa cosa per le colonne e per le diagonali.

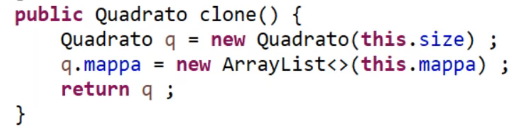
 

Niente di particolarmente difficile da spiegare.

Una cosa importante da ricordarsi è che quando salvo la soluzione devo fare un clone del quadrato ottenuto, altrimenti rischiamo di salvare riferimenti dello stesso oggetto che poi saarà vuoto al termine della ricorsione.



Di conseguenza dobbiamo creare il metodo clone nella classe Quadrato.



Il programma finito dal prof. si può prendere tranquillamente da GitHub forkandolo. Quando faccio il run del programma ottengo tutti i quadrati magici di lato 3 il circa 224 millisecondi.

Se si cambia la lunghezza del lato da 3 a 4. Il numero di soluzioni e il tempo impiegato a trovarle aumenta spropositatamente (in 134 ore ha trovato 308 risultati e non erano ancora tutti).

Un prima problema è dovuto dal fatto che, quando lancio il programma, esso utilizza solamente 1 degli 8 core che il computer offre. Per ovviare a questo, si potrebbe pensare di avviare il programma 8 volte e ognuno di questi 8 controlla una coppia di numeri (fino a 16 numeri se scelgo 4 come lato).

Bisogna quindi cercare di Parallelizzare l’algoritmo per riuscire a sfruttare la potenza di calcolo.

Con la ricorsione questo è abbastanza facile, in quanto posso per esempio fare i primi due livelli di ricorsione e poi dal 3 in poi faccio partire un processo parallelo che se ne occuperà.

La domanda che sorge spontanea è: “Dove viene speso il tempo di esecuzione?”.  
Per far questo, usiamo il programma ‘Java Mission Control’, un programma che fa parte del pacchetto JDK e che quindi dovremmo già avere. Esso è una console di amministrazione della macchina virtuale JAVA e ci permette di andare ad osservare cosa fa la macchina virtuale e analizzarlo.

Se apriamo questo programma, ci dice che ci sono in esecuzione 3 macchine virtuali.Una che sta facendo girare Eclipse, una che sta facendo girare Mission Control e una che sta facendo girare ‘TestQuadrato’. Ci possiamo collegare a quest’ultima per vedere cosa sta facendo. Possiamo registrare quello che sta succedendo. Se attivo il “Flight Recorder”, registro per una certa durata, il Mission Control osserva cosa fa il programma e lo registra. Successivamente posso quindi salvare la registrazione su un file che potrò in seguito analizzare.

Il professore ha già analizzato quanto tempo la macchina ha speso in determinate azioni. L’ 84% del tempo speso dal programma è stato utilizzato nel metodo ‘cerca’ della classe CercaQuadrato. Il metodo isMagic che sembrava molto ingombrante a livello di tempo, in realtà è irrilevante.

Se analizziamo il nostro metodo ‘remove’ (che ci occupa il 12% del tempo), possiamo notare che in realtà questo metodo, per eliminare il numero inserito poco prima, scandisce di nuovo tutta la lista, inutilmente.

Per risolvere questo problema basterebbe dire di eliminare direttamente l’ultimo numero, senza dover per forza scandire tutta li lista.

Un altro metodo per recuperare tempo e sicuramente quello di capire quando una soluzione parziale è già sbagliata. Nel caso di un quadrato con lato 4, ogni 4 caselle posso controllare se la somma della riga sia il numero magico. Se così non fosse escludo la soluzione a prescindere senza andare avanti e perdere tempo.

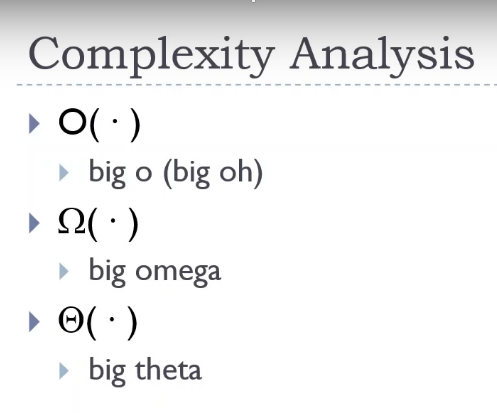
Quindi modificando l’algoritmo si può ridurre notevolmente il tempo di attesa.

Col tempo, la potenza dei computer è aumentata esponenzialmente e ciò ci ha permesso di risolvere problemi che in precedenza erano impensabili.

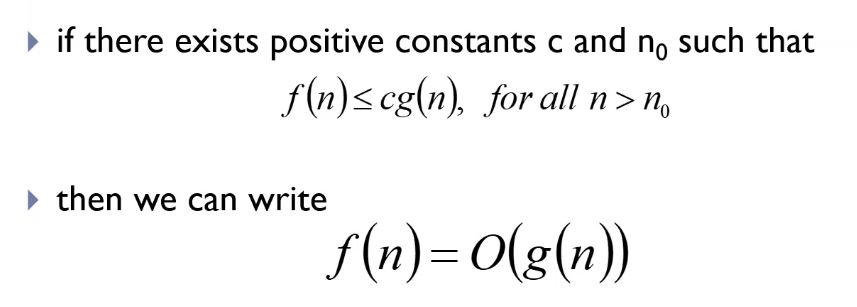
La maggior parte degli algoritmi dipendono dalla dimensione dell’input in entrata. Immaginiamo di avere una funzione T(n) che mi rappresenta il tempo di esecuzione di una generica istanza di dimensione n (nel nostro caso ‘n’ era la lunghezza del lato).

Prendiamo come esempio il ‘bubble sort’, cioè l’algoritmo per ordinare un vettore. Come aumenta il tempo di esecuzione di questo algoritmo all’aumentare della dimensione?

Nel caso del bubble sort, l’algoritmo richiede (n^2-n) / 2 passaggi. Possiamo ben vedere che basta aumentare n di poco che il numero di passaggi aumenta notevolmente.



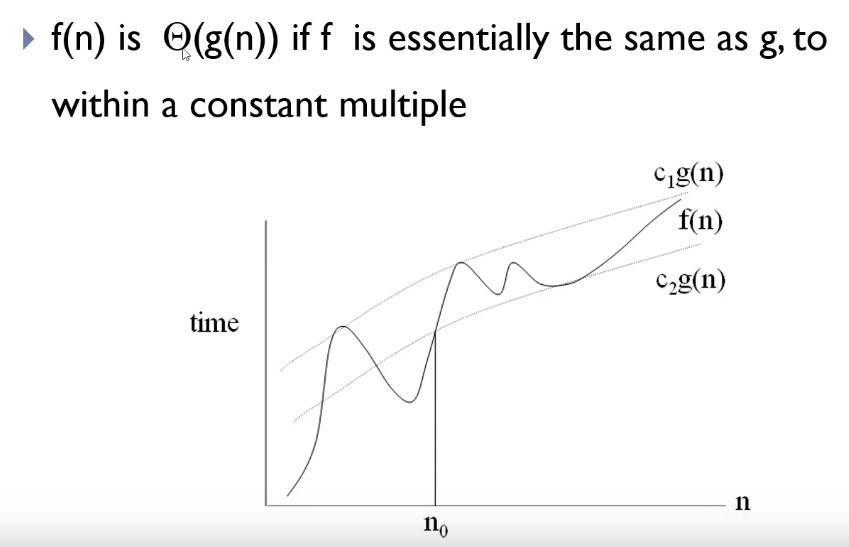
‘O grande’ è il limite massimo sul tempo di esecuzione. F(n) è un O grande di g se f cresce al massimo veloce come g. Vuol dire cioè che esistono un c ed un n0 tali per cui la mia funzione asintoticamente è dominata dalla funzione g.



Affianchiamo all’upperbound un lowebound. Definiamo quindi ‘Omega Grande’ che è esattamente l’opposto di ‘O Grande’. Se la nostra funzione domina un’altra funzione, allora diciamo che la nostra funzione è un ‘Omega Grande’ dell’altra funzione.

Se io riesco a trovare una funzione che sia contemporaneamente upperbound e lowebound della nostra funzione f, allora la nostra funzione del tempo può anche essere molto complicata, ma so esattamente in che binario cresce.

Quindi l’ultima definizione che introduciamo è ‘Teta Grande’ che è l’unione di ‘Omega Grande’ o ‘O Grande’.



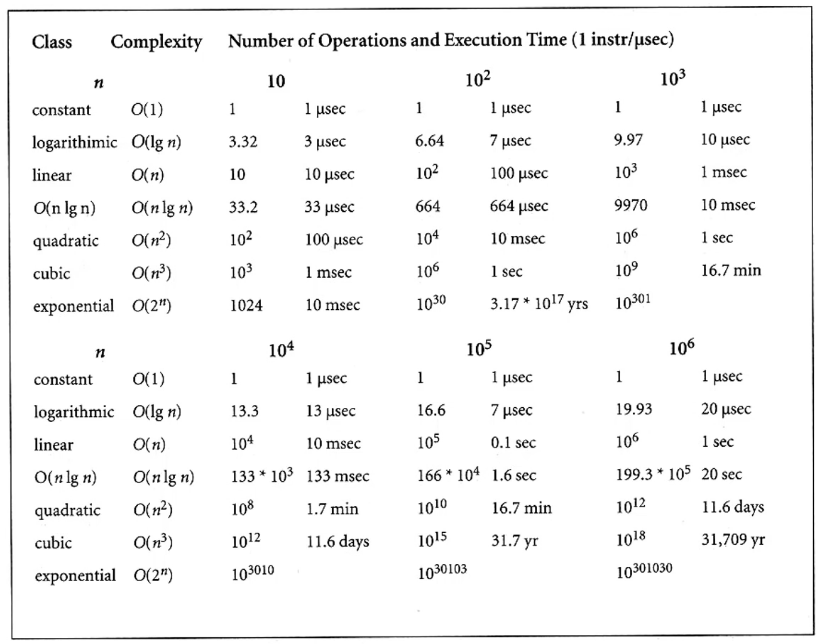
Se ci troviamo in questo casso possiamo dire che f è proprio dello stesso ordine di g. F(n), invece, è O grande di g se si trova sempre sotto, mentre è Omega grande se si trova sempre sopra.

Di solito, l’Omega Grande ci dà la soluzione migliore, perché so che sotto di un tot non posso scendere.

L’O Grande ci dà invece il caso peggiore, perché ci dice che più di un tot io non posso arrivare.

Teta Grande invece ci dà il caso normale, la crescita tendenziale effettiva.

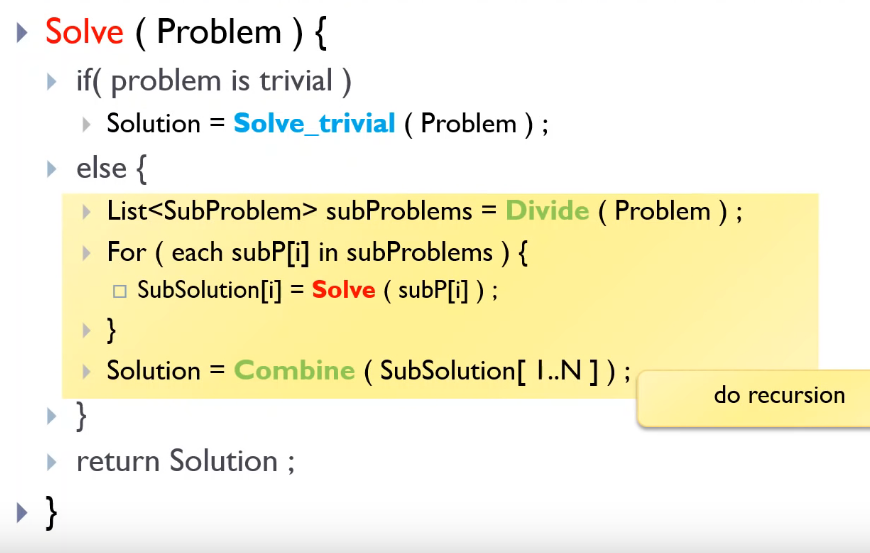
Bisognerebbe cercare sempre di mantenere la dimensione degli algoritmi sul lineare, logaritmico al fine di riuscire a risolvere problemi di dimensione più grande.



Ci sono 5 regole che ci permettono di approssimare il nostro Tempo di Esecuzione.

1. Ogni istruzione singola del programma ha una complessità costante (per esempio 3+2, println…)
2. Va ignorato il fatto che istruzioni diverse hanno tempi di esecuzioni diversi (sempre nel costante).
3. Nel caso di if-else, calcolo la complessità computazionale del ramo if, poi quella del ramo else e poi prendo il peggiore.
4. Se ho una sequenza di istruzioni, la complessità è data dalla somma dei passi delle singole istruzioni (ricordandosi dell’O grande che vince).
5. Nei cicli se ho k istruzioni all’interno del ciclo, la complessità del ciclo totale è data dalle n iterazioni moltiplicato le k istruzioni. Due cicli annidiati danno complessità totale uguale a n^2.

Ma allora da dove salta fuori l’esponenziale? Dalla ricorsione e non solo.



Schema generale ricorsione.

Dividiamo il problema in sottoproblemi.

A: fattore di decomposizione (branching), cioè dato un problema, in quanti sottoproblemi lo suddivido?

B: quanto ciascuno dei sottoproblemi è piu piccolo del problema di partenza?

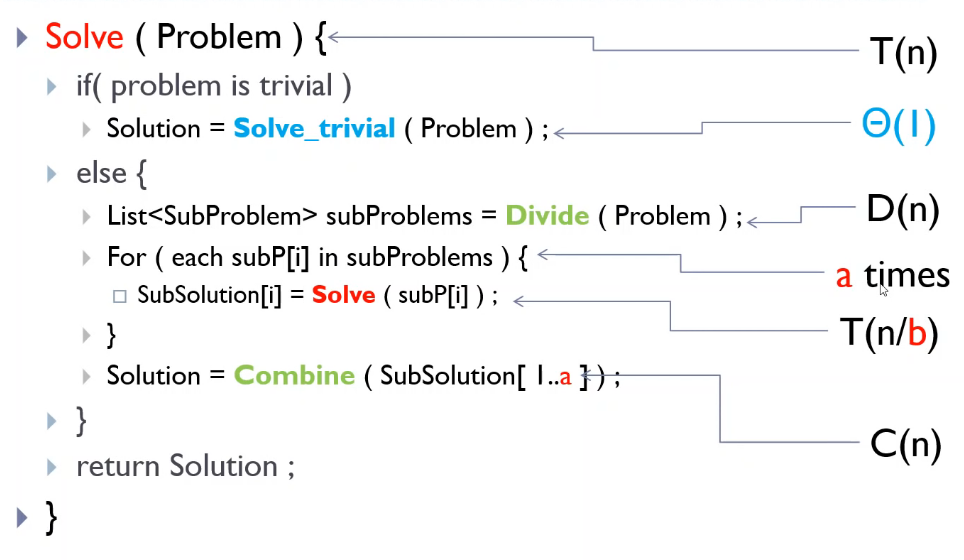
N: dimesione del problema di partenza.

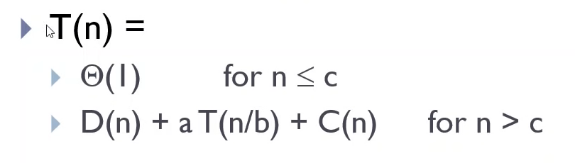
T(n): complessità del nostro metodo che risolve il problema.

Teta(1): risolve la soluzione nel caso terminale.

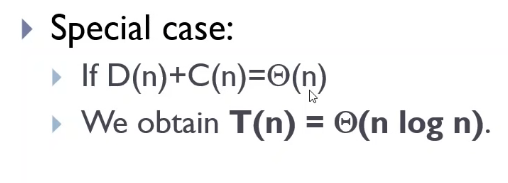
D(n): complessità della divisione.

C(n): complessità della ricombinazione.





Quindi la complessità (non teniamo conto del caso terminale in cui è uguale a Teta) è uguale ad una somma di fattori in cui compare ancora T(n/b). Questo tira fuori un esponenziale.



Se siamo fortunati, la ricombinazione e la divisione hanno complessità lineare, e quindi la complessità del problema sarà nlog(n). Negli altri casi no.

Nel nostro caso del Quadrato Magico noi abbiamo considerato tutte le possibili permutazioni, quindi avremo 16! Soluzioni. Ma la nostra n non è sedici ma quattro.

Abbiamo una complessità che è il fattoriale del numero delle caselle del quadrato.

Quindi, ogni volta che noi creiamo una procedura ricorsiva, dobbiamo cercare di ridurre i fattori nella formula dell’immagine sopra. O cerco di ridurre ‘A’ o ‘B’. Dobbiamo sempre cercare di avere un A più piccolo di B per avvicinarci al lineare. Se A incomincia a diventare più grande di B, ci avviciniamo sempre di più all’esponenziale e ciò non va bene.

FINE.