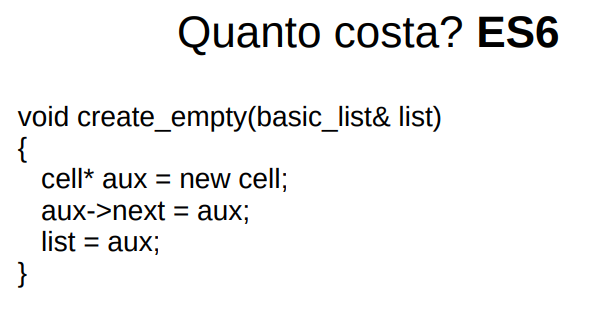
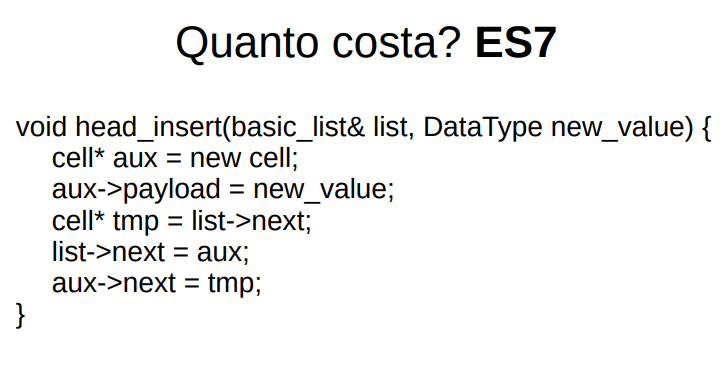
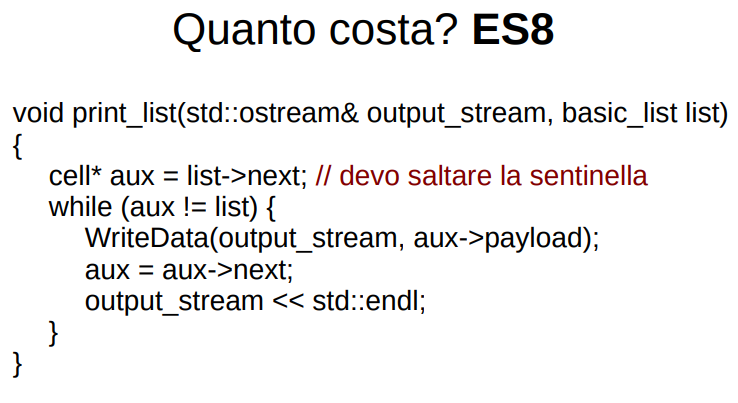


La complessità risulta essere **THETA( n^2 )**, ma ciò, rispetto a due cicli normali (ES 4) con i=j=0 in partenza, non deriva da una moltiplicazione “pura” n\*n = n^2 ma bensì dallo sviluppo (dimostrato per induzione) della sommatoria da k=0 a n di k (= [n(n+1)]/2) che sviluppata presenta il termine dominante n^2. Svolgiamo infatti n+(n-1)+(n-2)+...1 cicli.

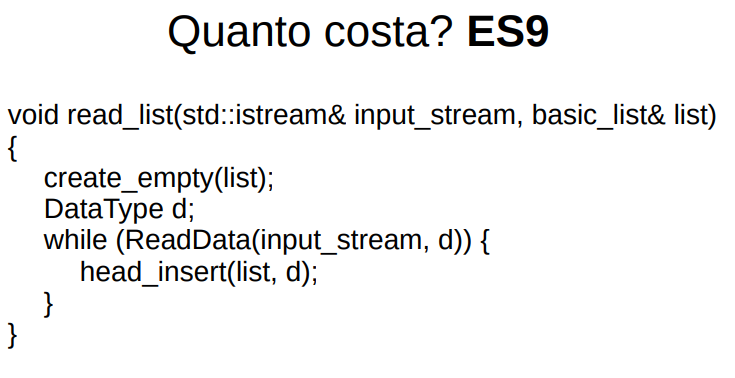


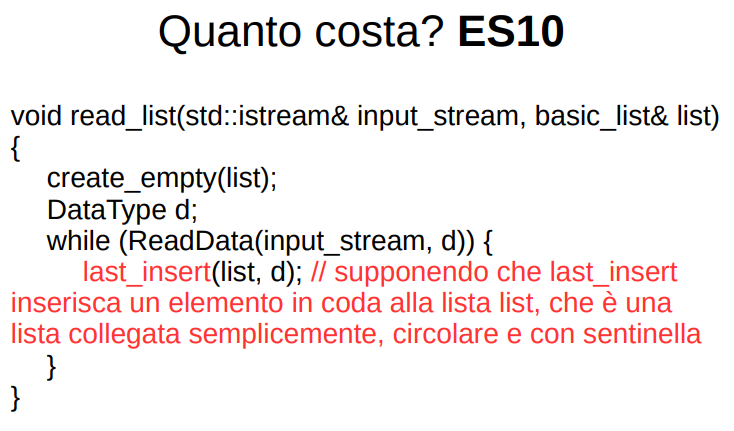
La complessità risulta **THETA(1)** in maniera triviale.

La complessità risulta **THETA(1)** in maniera triviale.

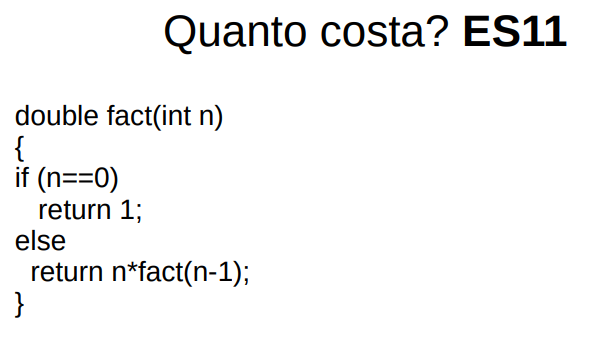


Stampare una lista, in questo caso circolare come si evince da aux != list, comporta ovviamente stampare i suoi n elementi (considerando costante WriteData). **THETA(n)**.

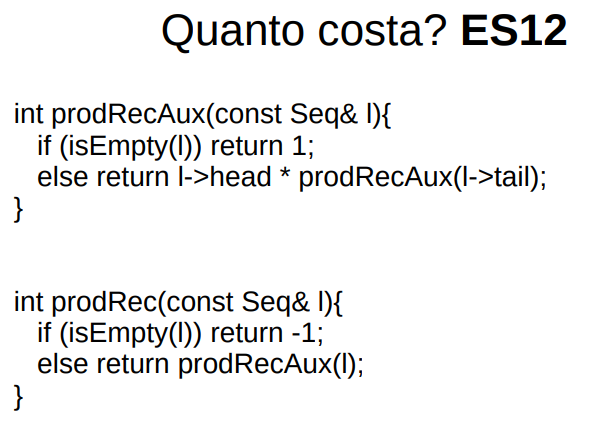


create\_empty e ReadData hanno THETA(1), dunque la complessità è data dalla iterazione su ReadData degli n elementi presenti nello stream, che vengono inseriti in list. **THETA(n)**.

Rispetto ad ES9, devo inserire in CODA e non in TESTA. Dato che non mi trovo su una lista doppiamente collegata circolare con sentinella, ma la lista è SEMPLICEMENTE collegata; non posso sfruttare il puntatore a prev per inserire in coda con THETA(1), ma dovrò inserire proporzionalmente, man mano, agli elementi di n già inseriti (devo scorrere la lista) → risulta una sommatoria da k=0 a n di k (= [n(n+1)]/2) che sviluppata presenta il termine dominante n^2. La complessità risulta dunque essere in conclusione **THETA(n^2)**.

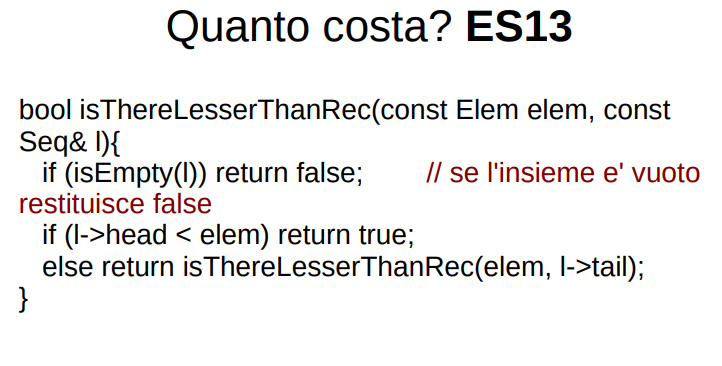


Svolgo un’operazione di moltiplicazione per n volte. La complessità risulta quindi banalmente **THETA(n)**. Tuttavia in questo caso risulta utile annotare che, in caso di numeri grandi, il fattoriale potrebbe diventare “ingestibile”. Non ce ne occupiamo.



La complessità della funzione non deve spaventare. Ci interessa solo che venga effettuata n volte la moltiplicazione su tutti gli elementi di l (produttoria). Ergo **THETA(n)**.

Dovevo riempire questo spazio vuoto, mi dava fastido.

****

Incontriamo un algoritmo con caso migliore e caso peggiore. Questo perché se già il primo elemento della lista è minore di elem, ritorno true e l’algoritmo termina con **THETA(1)**.

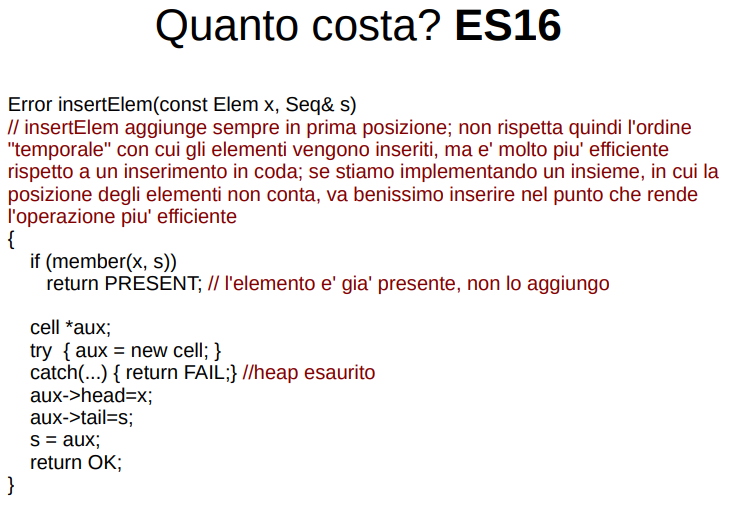
Se invece tutti gli elementi risultano essere maggiori di elem (non esiste x<elem) allora avrò come caso peggiore **THETA(n)** dopo aver scandito tutta la lista. Dunque il caso migliore corrisponde al ritorno di true, mentre il caso peggiore al ritorno di false (arrivo ad isEmpty).



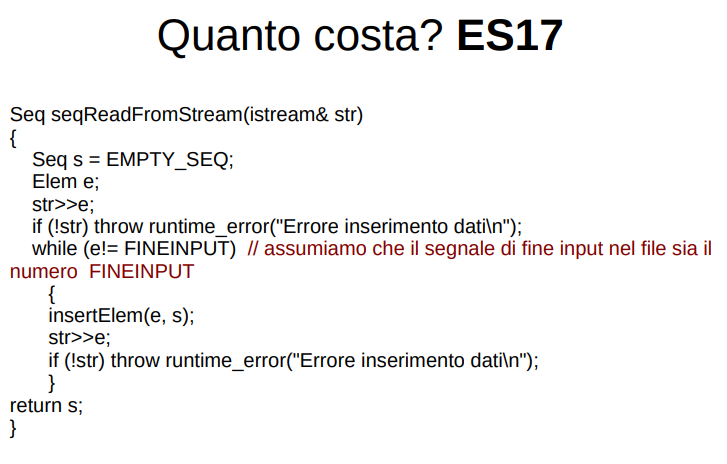
Si rimanda all’ES 13. Avremo dunque caso migliore **THETA(1)**, ma questa volta quando il return finale è un false (opposto di ES13), e caso peggiore **THETA(n)** quando il return finale è un true. La funzione, come già visto ad IP e rifacendosi al corso di Logica, sfrutta le leggi di DeMorgan per i quantificatori (esistenziale ed universale).



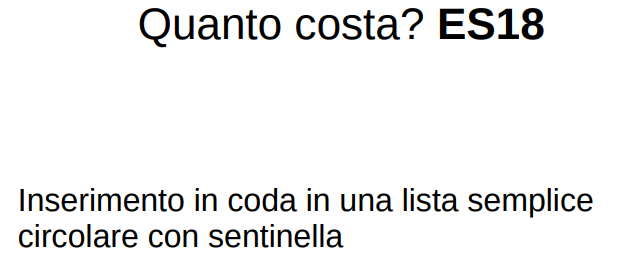
Anche in questo caso abbiamo due casi. Per la precisione nel caso migliore (l’elemento che stiamo cercando è il primo) avremo complessità **THETA(1)**, nel caso peggiore dobbiamo ovviamente scansionare la lista, ottenendo **THETA(n)**.



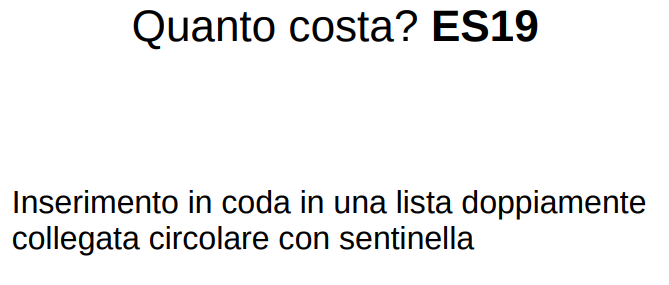
Tutto ciò che accade dopo ha complessità costante, dunque tutto dipende dalla funzione member (descritta nell’ES15) → **THETA(1)** nel caso migliore e **THETA(n)** nel peggiore.



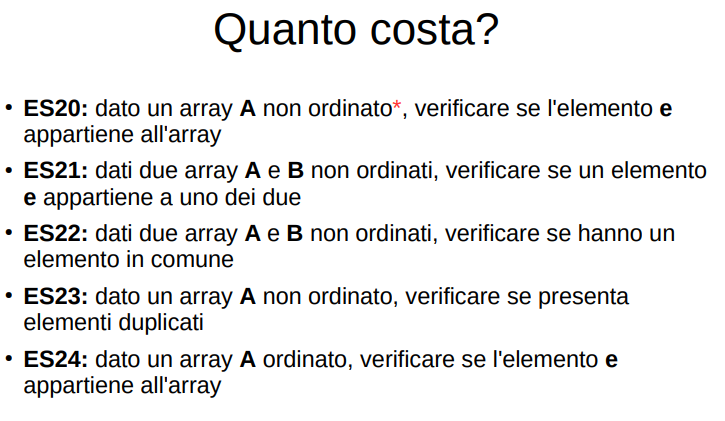
Partiamo con l’individuare gli unici due nodi importanti per il nostro calcolo: il ciclo sugli elementi letti e la funzione insertElem (ES16). Sapendo che la funzione insertElem ha complessità THETA(1) nel caso migliore, ne deduciamo che di conseguenza (ripetendo questa operazione n volte per via del ciclo) la complessità nel caso migliore è **THETA(n)**. Invece, nel caso peggiore, inseriamo gli elementi mano a mano, proporzionalmente alla lunghezza della lista che cresce ad ogni inserimento. Deduciamo dunque una sommatoria, da cui → **THETA(n^2)** nel caso peggiore.



Abbiamo **THETA(n)** (scansione della lista per inserire in coda, non avendo ->prev).



Siamo giunti al caso tanto citato. La complessità è **THETA(1)** perchè possiamo balzare direttamente all’ultimo elemento, un po’ come il gatto della prof.Mascardi quando la cena è pronta. Okay, questo era cringe, ma sto scrivendo da 3 ore e sono stanco.



ES20: In modo triviale, se troviamo subito l’elemento: **THETA(1)** (caso migliore). Se invece l’elemento non è presente abbiamo **THETA(n)** (caso peggiore).

ES21: Se troviamo subito l’elemento (1a posizione del 1o array): **THETA(1)** (caso migliore). Se invece l’elemento non è presente in nessuno dei due array, devo scandirli entrambi, ottenendo una complessità in funzione sia di n sia di m (due dimensioni dei due array differenti; non ipotizzando che abbiano la stessa dim.) → **THETA(n+m)** (caso peggiore).

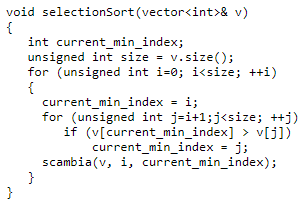
ES22: Se troviamo subito l’elemento in comune tra i due array, al primo controllo incrociato: **THETA(1)** (caso migliore). Se invece i due array (dim n ed m) non hanno elementi in comune, li devo scorrere entrambi. Si ipotizza un algoritmo “dummy” in cui sono semplicemente presenti due cicli annidati che scandiscono → **THETA(n\*m)** (caso peggiore).

ES23: Se troviamo subito l’elemento duplicato: **THETA(1)** (caso migliore). Se invece l’elemento non è presente abbiamo **THETA(n^2)** (caso peggiore). Questo n^2 può derivare da due cicli annidati (controllo ogni elemento per ogni elemento) → n\*n ; oppure da un algoritmo più “furbo” in cui a ogni iterazione esterna escludo il sottoarray di sinistra dalla ricerca (di sicuro lì non c’è un duplicato: pensaci!) → in questo caso avrò una sommatoria, che pur essendo migliore di un puro n\*n degenera in THETA(n^2). Purtroppo, è la vita.

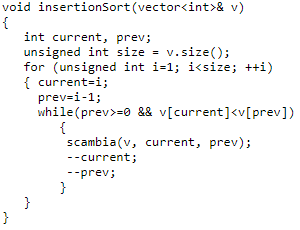
ES24: Non sappiamo ancora farlo, ma giunge voce che la complessità della ricerca binaria è logaritmica. Per ora non ci fidiamo.



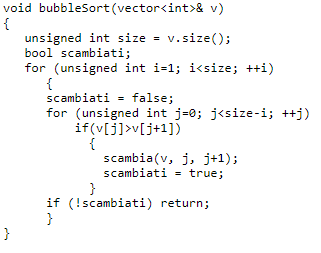
Se ce la fa il gatto, puoi farcela anche te. Tieni duro bro.



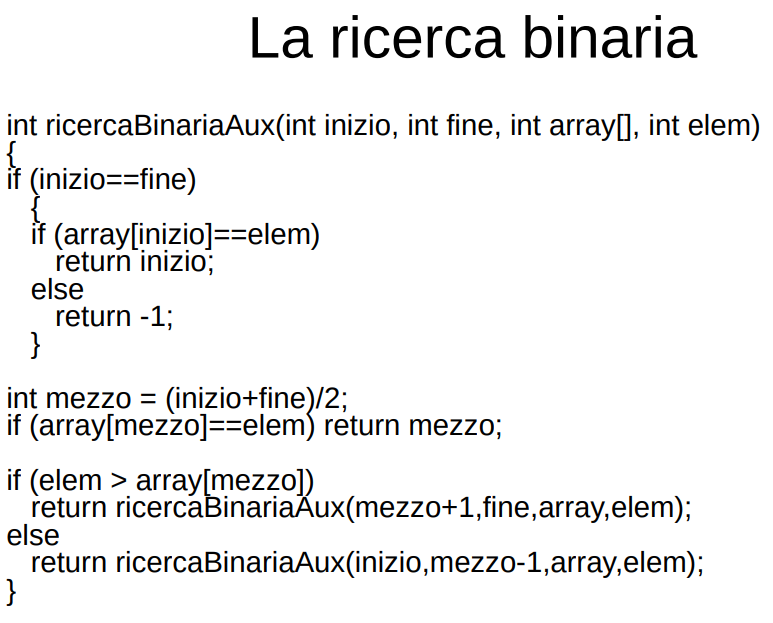
Il selection sort non è adattivo. Ne risulta che in ogni caso la complessità è **THETA(n^2)**. Questo perchè non essendoci un’ottimizzazione nel caso di sottosequenze già ordinate, il selection effettua sempre e comunque tutte le iterazioni dei cicli. (deriva da sommatoria)

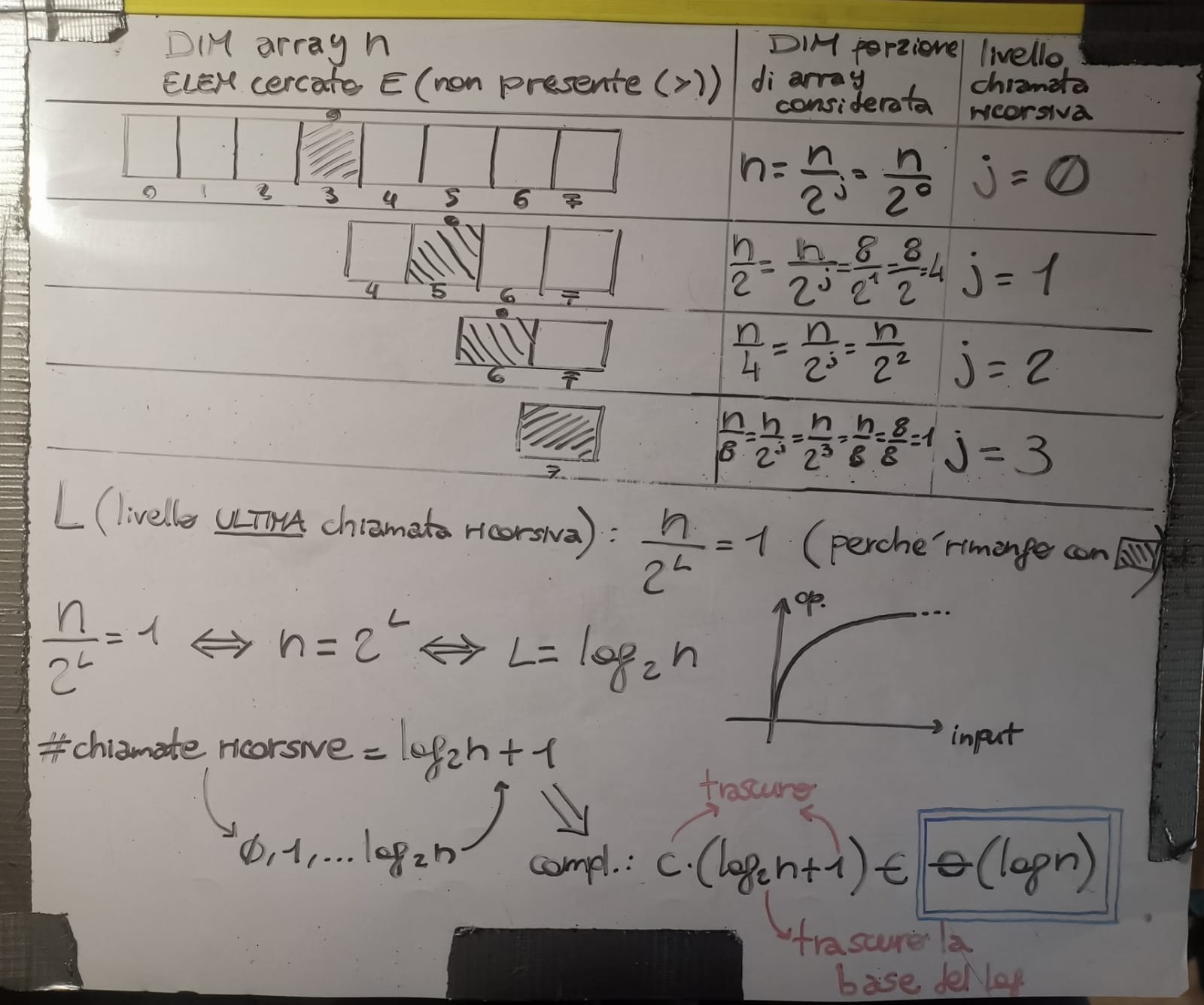


Laddove il caso peggiore, come nel selectionSort da sommatoria, risulta **THETA(n^2)**, in questo caso l’insertionSort è adattivo, e dunque si ha un caso migliore con **THETA(n)**; questo per costruzione interna dell’algoritmo (il confronto a sinistra, se “fallisce” fa fermare l’iterazione: non è necessario che vengano effettuati ulteriori confronti → n confronti e nessuno scambio). Il selection sort effettuava comunque sia confronti sia scambi.



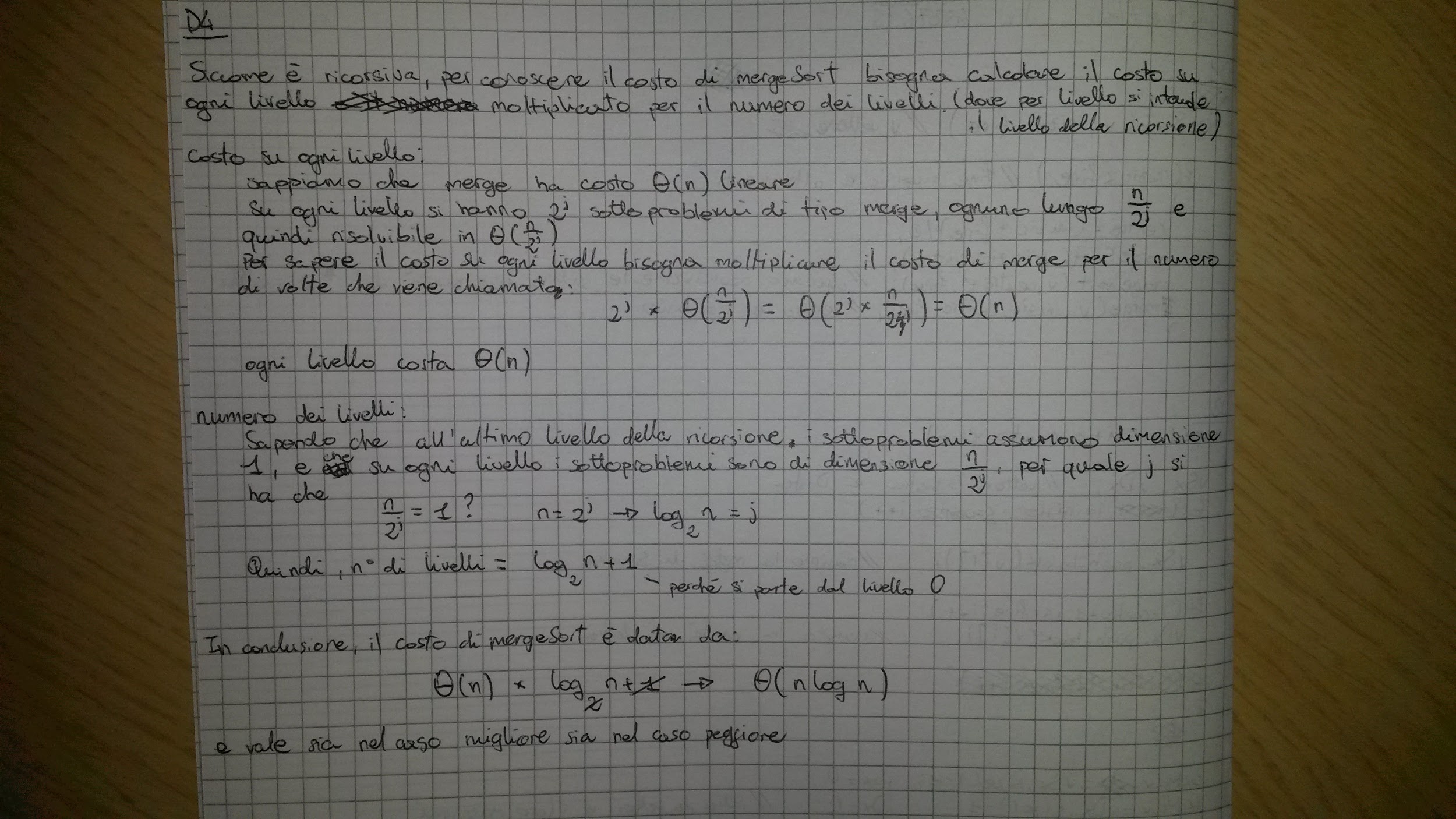
Il bubble sort classico avrebbe come UNICA complessità THETA(n^2), ma grazie all’utilizzo del flag booleano “scambiati” otteniamo come caso migliore **THETA(n)**: se l’array è già ordinato infatti alla prima iterazione esterna completa (n) il flag sarà ancora false perchè nessuno scambio sarà stato effettuato, portando il programma alla terminazione. Il caso peggiore, in maniera triviale, rimane **THETA(n^2)** come gli altri due algormti quadratici.





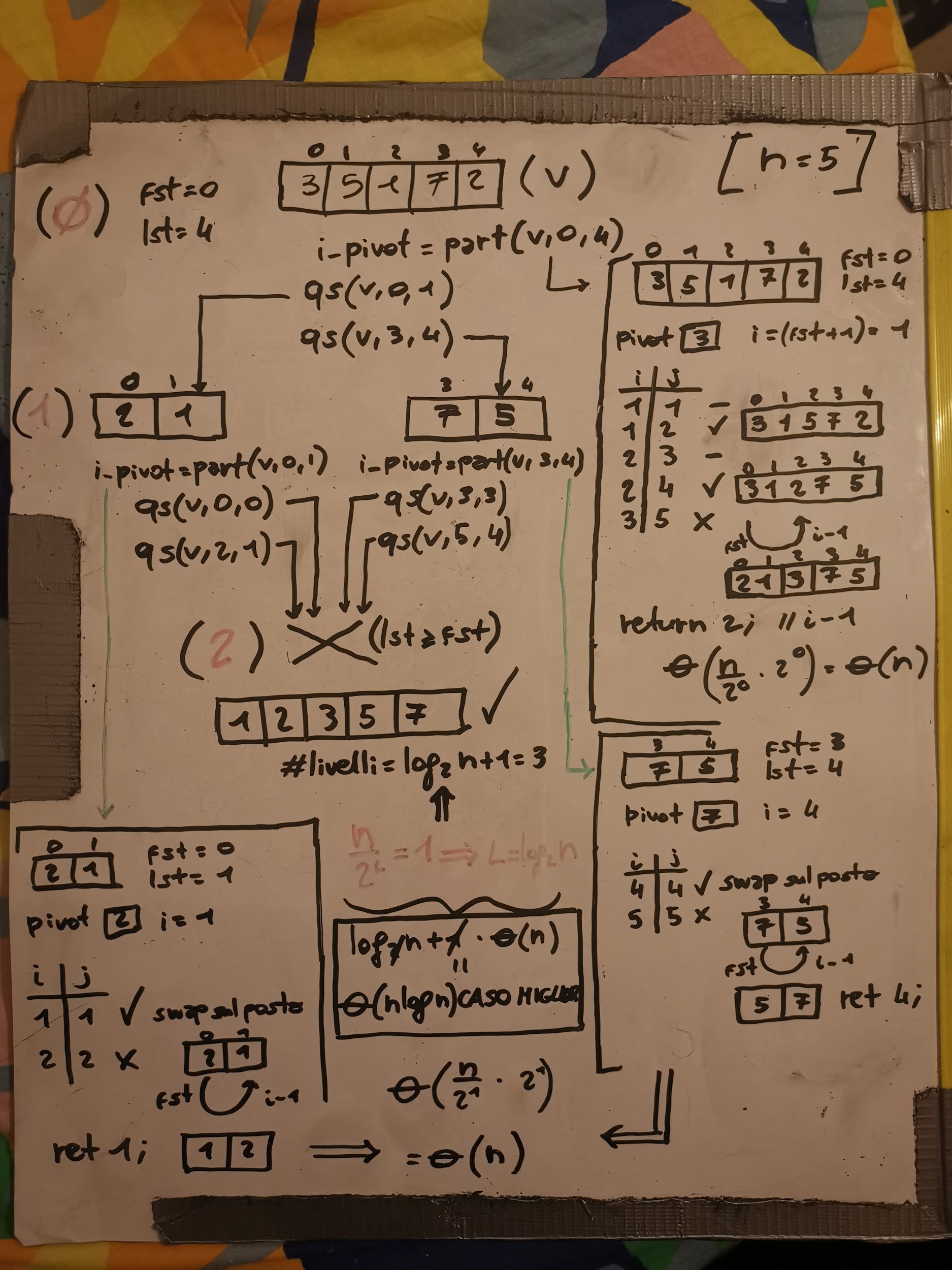
**MERGE SORT**

**Non metto il codice perchè è troppo lungo**



GIUSTA ma migliorabile, come?

**QUICK SORT**



Si fa riferimento ad un quick sort con algoritmo IN PLACE.

Nella foto sopra osserviamo la complessità nel caso migliore di QUICK SORT, ossia quello in cui “magicamente” vengono scelti come PIVOT sempre gli elementi mediani (ossia gli elementi che, se ordinassimo la sequenza, risulterebbero nella posizione n/2 (+/- se dispari)). In questo specifico caso, essendo gli elementi pochissimi, basta che solo la prima volta venga scelto il mediano, e nei casi successivi sarà triviale (su 2 elementi, per forza, viene scelto il mediano). La sequenza è stata dunque divisa a metà ad ogni livello dell’albero della ricorsione, ottenendo un calcolo della complessità logicamente equivalente a quello del Merge Sort (nel suo unico caso): **THETA(n\*log(n)).**

Il caso medio non è stato calcolato, perchè richiederebbe conoscenze statistiche e probabilistiche. La sua complessità permane **THETA(n\*log(n))** (ecco evidenti i limiti della notazione THETA!).

Prima di riportare la complessità del caso peggiore, consideriamo una metodologia di scelta del pivot diversa da quella banale (prima posizione), ossia la scelta randomica del pivot. Importante dunque notare che la scelta del pivot non è direttamente collegata alla complessità che otterremo, anche se può ovviamente influenzarla positivamente o meno.

*Scegliendo il pivot a caso tra gli elementi disponibili, si spera di sceglierlo “abbastanza buono” “abbastanza spesso”*

*● Abbastanza buono: un pivot che partizioni l'array in modo tale che 25% degli elementi sia < del pivot e 75% sia >=, è sufficiente per restare nella complessità Θ(n log n) [non lo dimostriamo]*

*● Abbastanza spesso: metà degli elementi dà una partizione 25%-75% o anche migliore*

Riportando ora la complessità nel caso peggiore (**THETA(n^2)**) ci accorgiamo che Quick Sort può apparentemente essere “peggiore” di Merge, ma per un insieme di considerazioni statistiche e spaziali (Merge usa strutture d’appoggio), in realtà non è così!

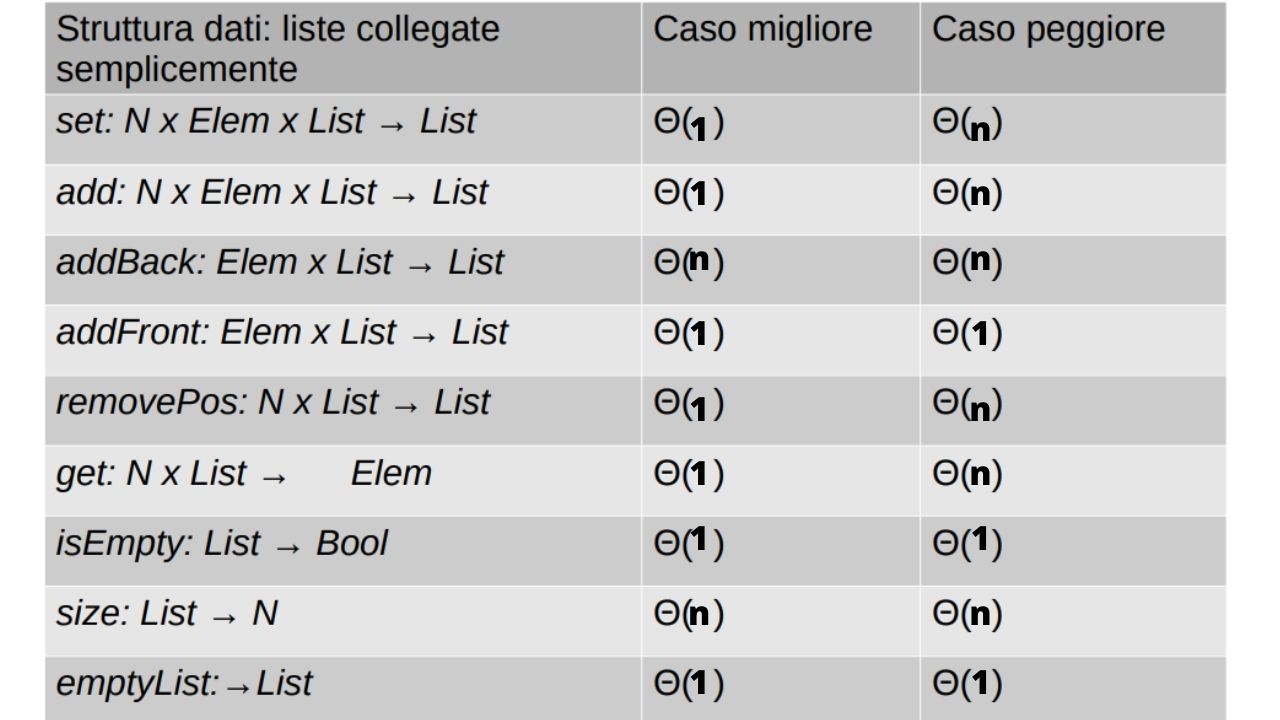
La complessità di n^2 si può ottenere, in generale, se ad ogni chiamata viene scelto come pivot il minimo o massimo elemento della sequenza, andando a generare due chiamate ricorsive, di cui una su 0 elementi (nulla) e un’altra su n-1 (si vanno così a perdere le agognate proprietà del Divide Et Impera) → otteniamo dunque una sommatoria da 1 ad n che ci riconduce ad una complessità quadratica!

Un ESEMPIO può essere un QuickSort con scelta banale del Pivot chiamato su un array già ordinato.

**TDD: LISTE; con liste “dll”**



**TDD: LISTE; con liste semplici**



L’unica differenza di complessità evidente, tra implementazioni del TDD Lista con liste semplici o liste dll, è la complessità di addBack, che nel caso delle dll può essere compiuta in modo costante grazie alla possibilità di passare subito all’ultimo elemento.

**TDD: LISTE; con array dinamico **

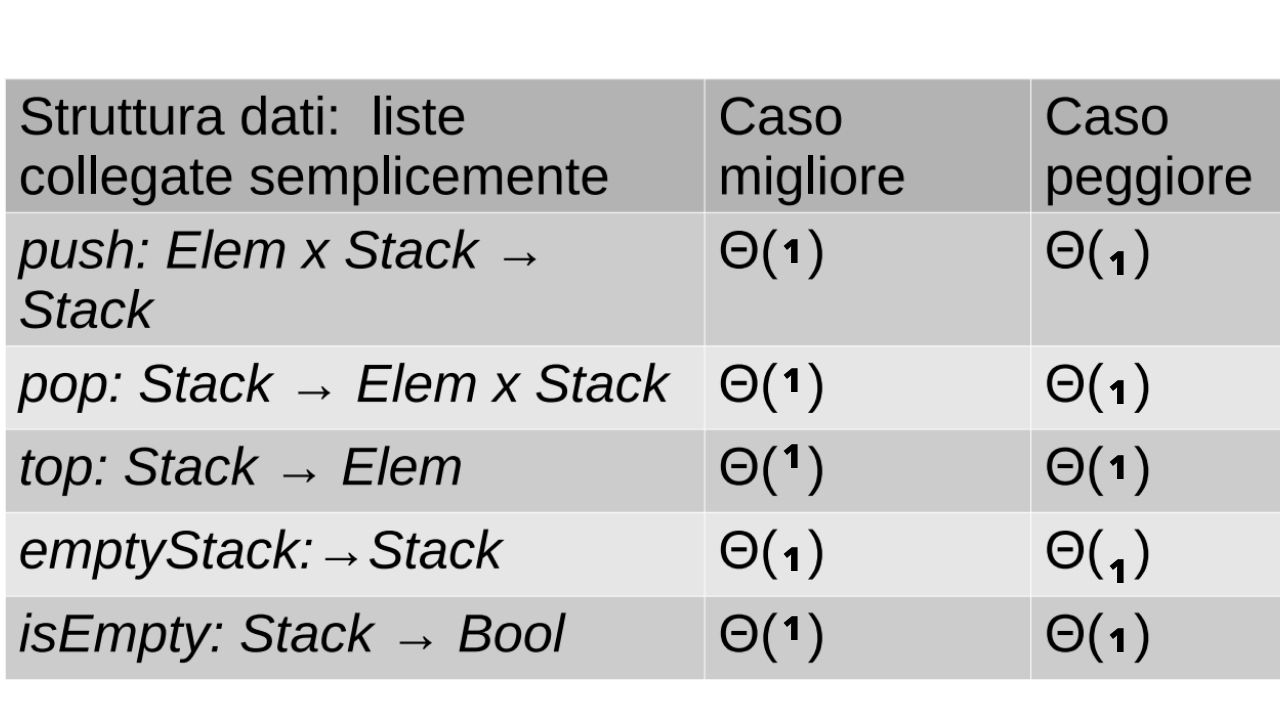
Degne di nota sono le funzioni add, addBack, addFront (legate profondamente tra di loro, anche perchè a livello implementativo addBack e addFront richiamano semplicemente add con degli specifici parametri (sono casi limite di add)) e removePos:

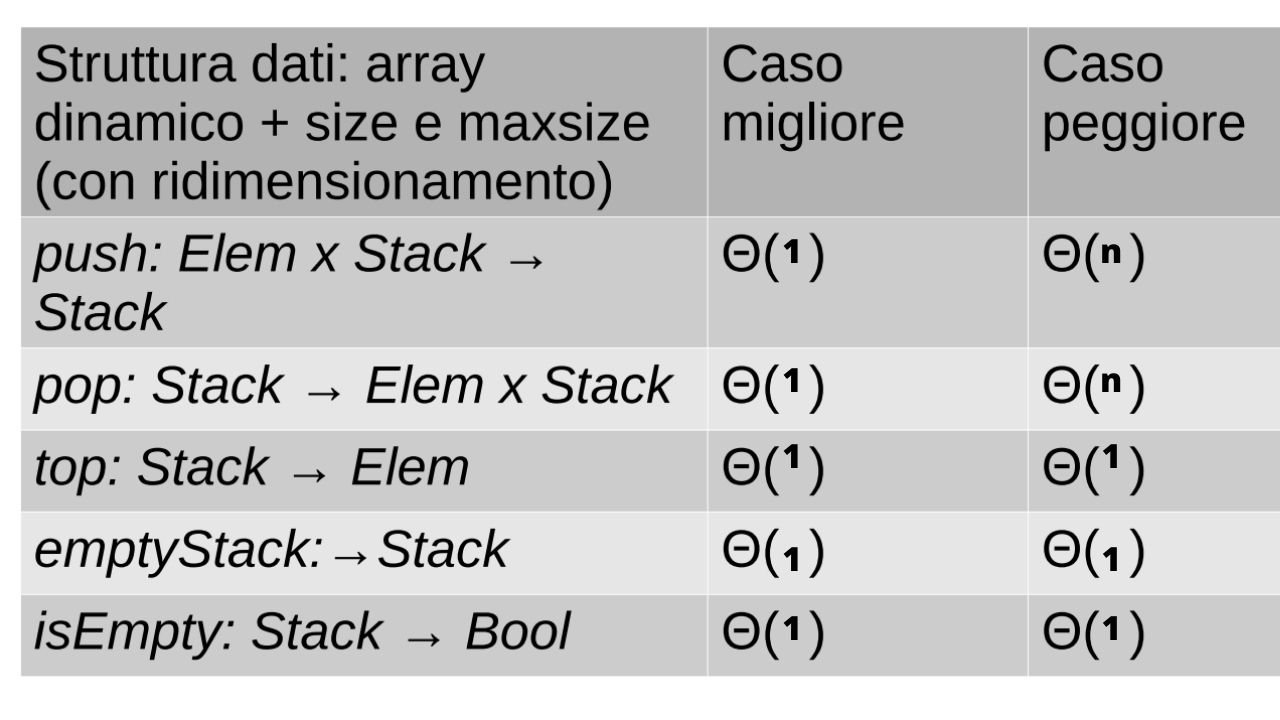
Partendo dalla più “semplice”, la **addFront**, notiamo che in OGNI caso dovremo shiftare a destra tutti gli elementi, quindi THETA(n) in tutti i casi. La **addBack**, invece, ottiene una complessità di THETA(1) quando non bisogna ridimensionare l’array perchè non è pieno (quindi viene aggiunto semplicemente l’elem senza shift e ciò non dipende da n), mentre se l’array è pieno dovremo per forza operare sul resizing, ottenendo THETA(n). Dato che la add racchiude i casi limite delle precedenti, abbiamo la distinzione tra caso migliore (quello di una add in ultima posizione senza resize, corrispondente al caso migliore della addBack) e tra caso peggiore. → THETA(1) e THETA(n).

La **removePos** ha THETA(1) quando rimuove in ultima pos (nessuno shift richiesto) con array da non ridimensionare → “stesso caso del caso migliore di addBack”

**TDD: STACK; con liste “dll”**



**TDD: STACK; con liste semplici**

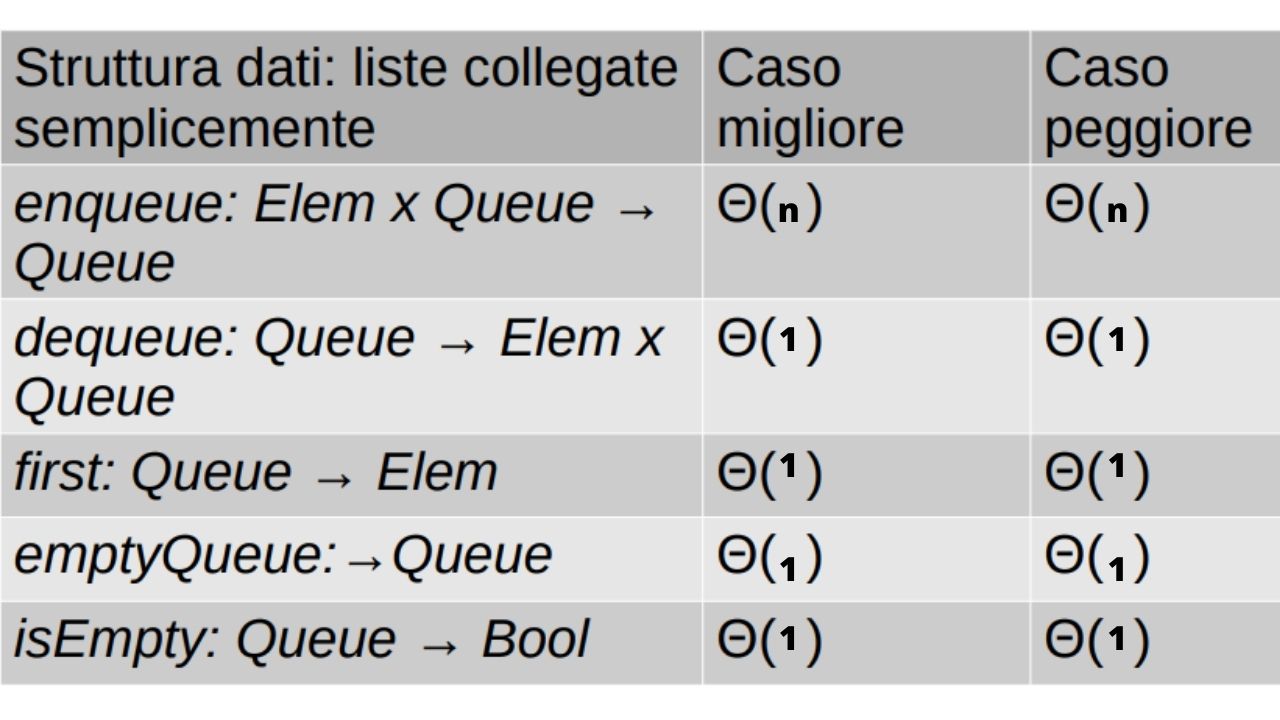
**TDD: STACK; con array dinamico**

Per lo STACK/PILA (LIFO), se assumiamo un’implementazione ottimale, otteniamo sempre una complessità THETA(1) tranne che per push e pop con struttura dati: array dinamico;

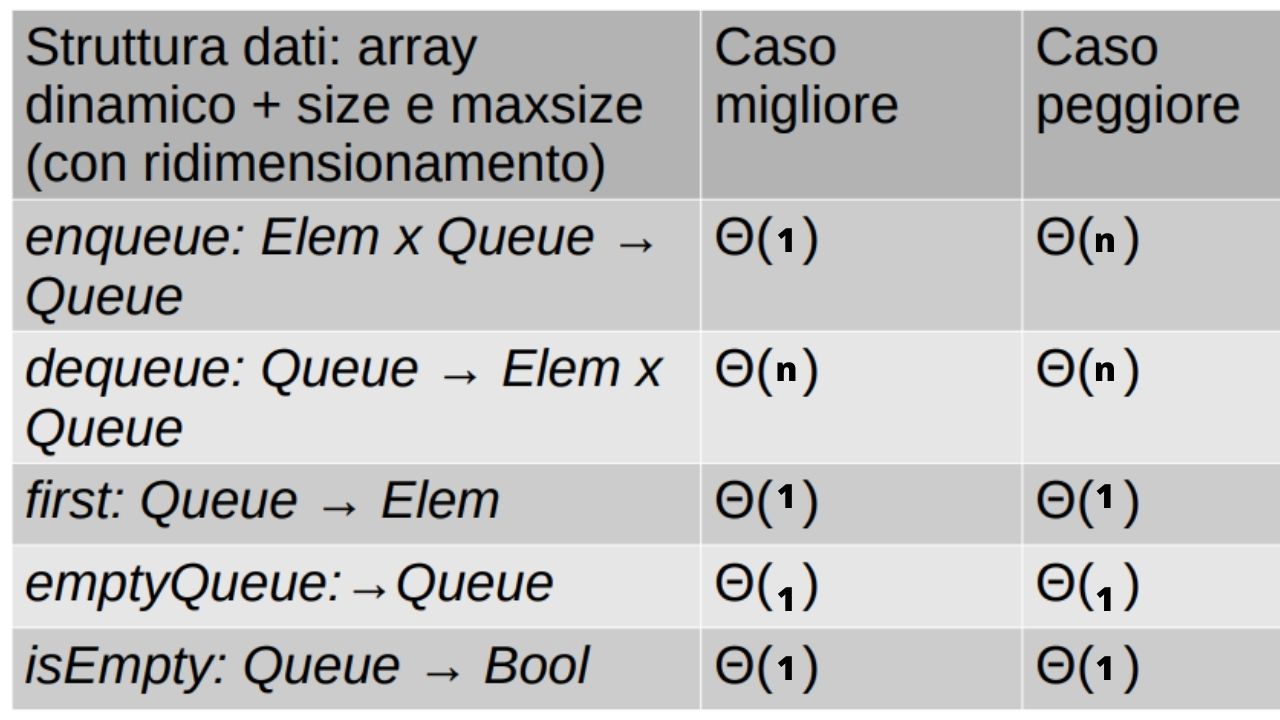
questo perchè potrebbe servire un resize.

**TDD: QUEUE; con liste “dll”**



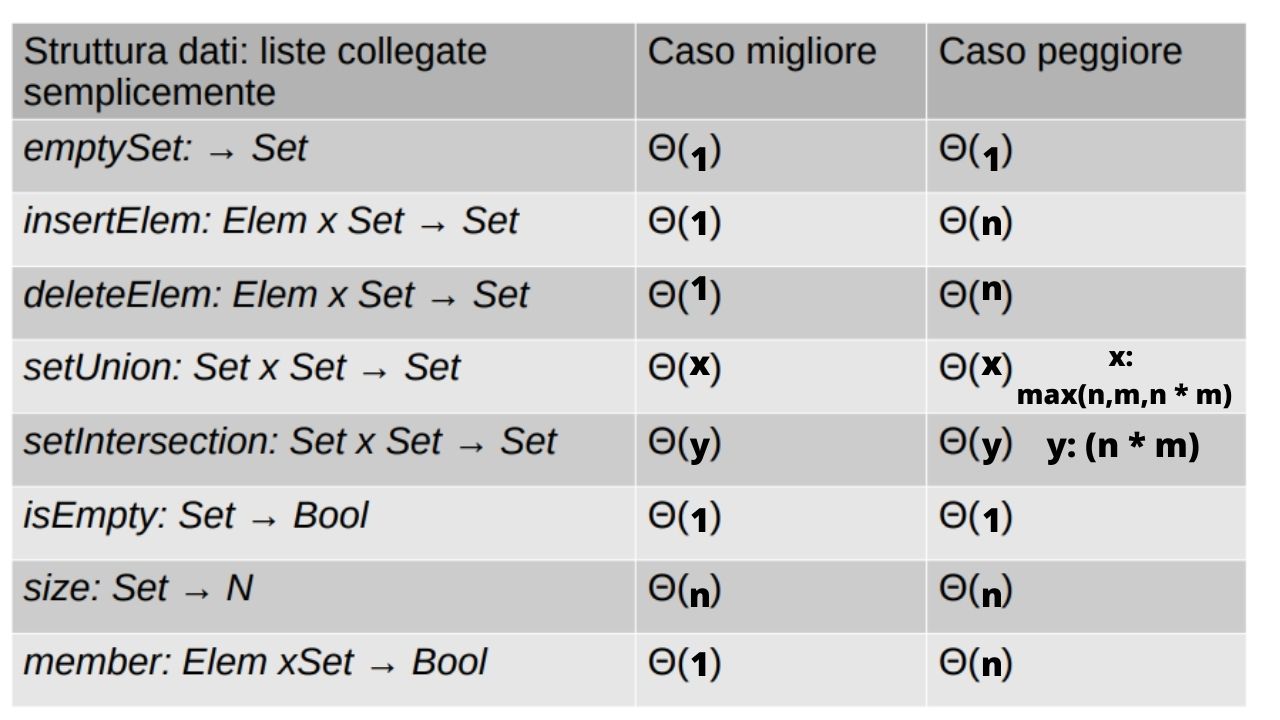
**TDD: QUEUE; con liste semplici**

Se scegliamo un verso “da dx verso sx” (deq←[][][]←enq) abbiamo sì una enqueue con THETA(n) perchè dobbiamo scorrere la lista, ma così sia dequeue sia first sono in THETA(1). Se scegliamo il verso opposto enqueue è THETA(1) MA sia dequeue sia first diventano THETA(n): “conviene questa scelta”, a differenza degli array in cui è indifferente.

**TDD:QUEUE; con array dinamico**

Possiamo scegliere di rendere efficiente la enqueue o la dequeue (la first non cambia perchè abbiamo l’accesso diretto). Per scelta coerente con il verso delle liste semplici si sceglie un verso “da dx verso sx” (deq←[][][]←enq), ottenendo una enqueue efficiente grazie al posizionamento dell’elemento in ultima posizione, che non richiede shift (però comunque con caso peggiore THETA(n) nel caso del resize) e una dequeue non efficiente (con shift necessario di tutti gli elementi).

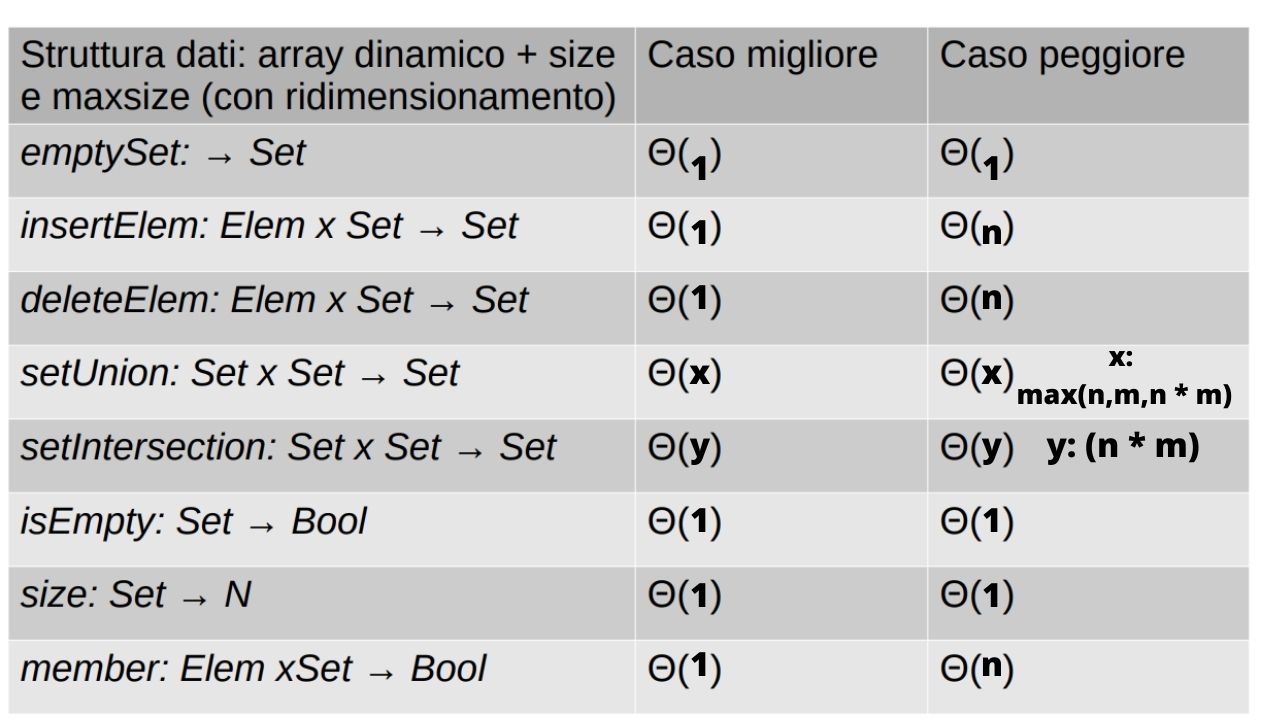
**TDD: SET**

****

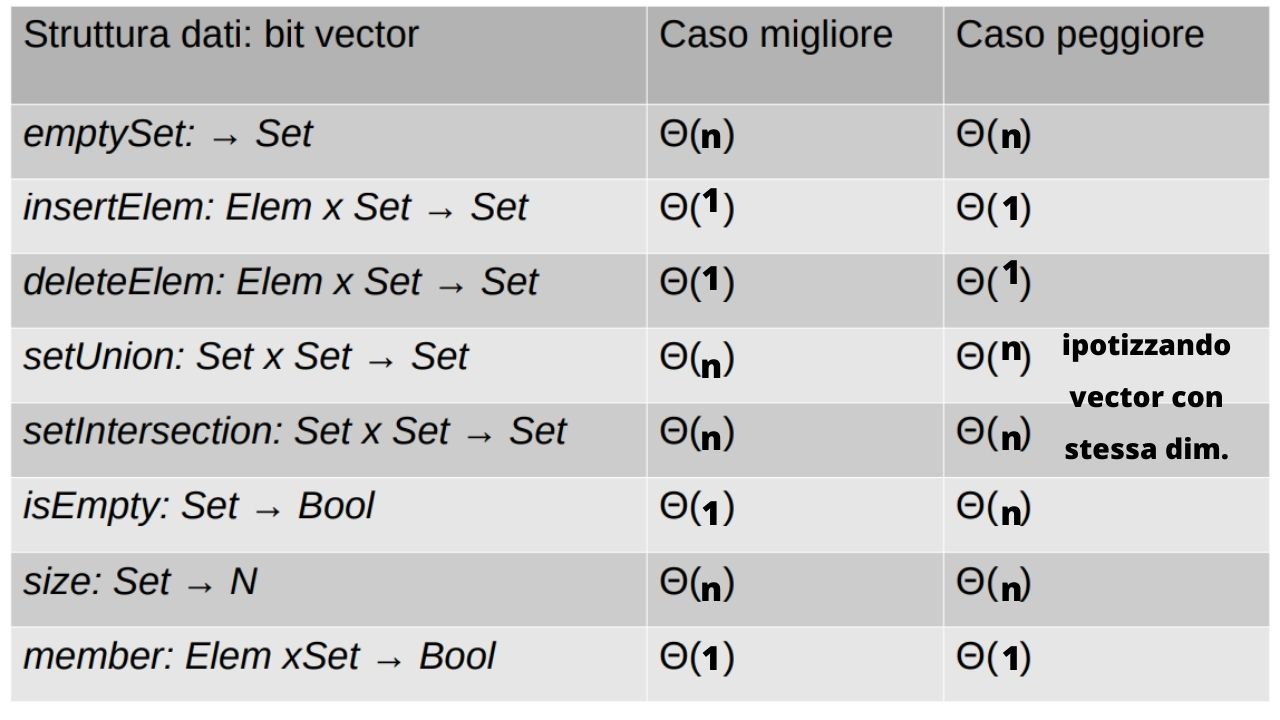
**setUnion**: se entrambi gli insiemi sono pieni, varrà n\*m (unione tra i due), se entrambi sono vuoti varrà ovviamente 0, se n è pieno ed m è vuoto, viene copiato n → THETA(n); viceversa se n è vuoto ed m è pieno, viene copiato m → THETA(m). Discrimino perchè in questi due casi n\*m vale 0 per via della moltiplicazione!

**setIntersection**: se uno (o entrambi) gli insiemi sono vuoti, ovviamente non viene copiato nulla (per definizione) → è giusto n\*m senza dover discriminare come nella Union.

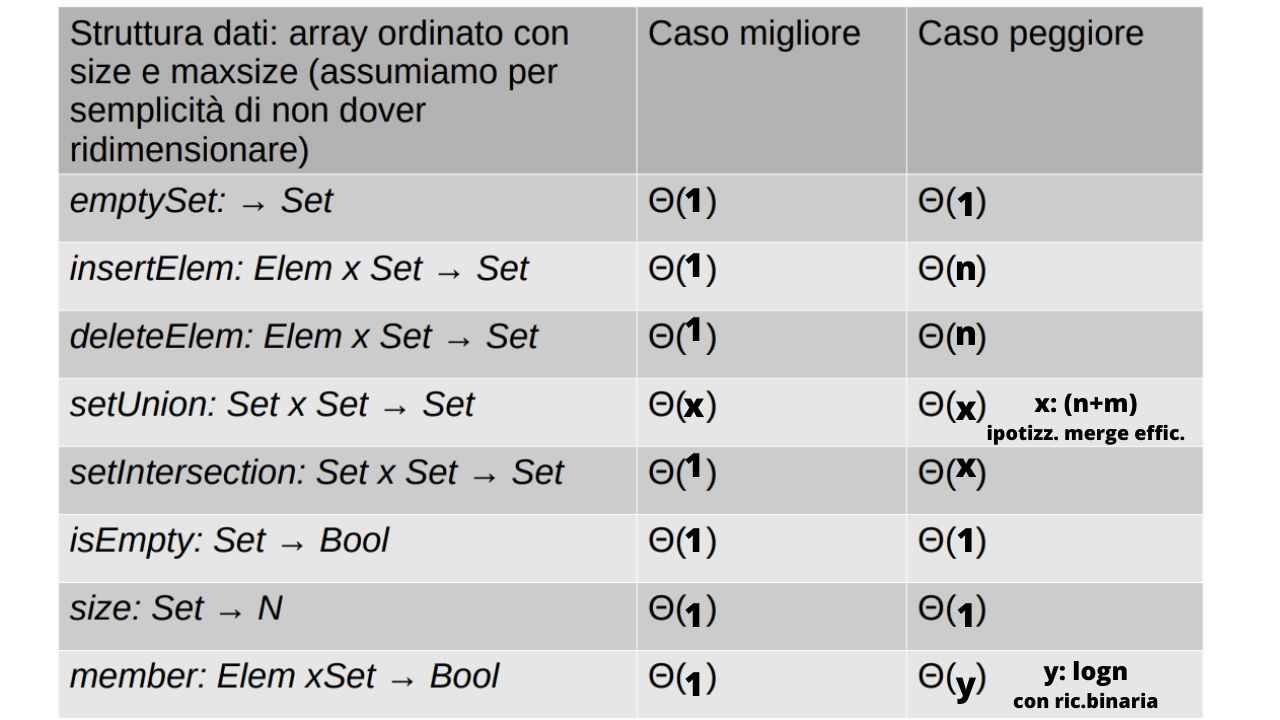
Tutte le complessità della tabella risultano uguali per le liste **dll** (con sentinella e circolari)!



...Invece con gli array varia solamente la size, che ovviamente in questo caso è costante.

(...)

(…)Molte complessità variano coi bit vector: la costante emptySet ha una complessità THETA(n) poichè il bit vector vuoto ha senso se riempito comunque con dei valori; insert, delete e member sono costanti poichè basta effettuare l’accesso diretto; isEmpty invece peggiora e assume una complessità THETA(n) per la necessità di controllare che ogni valore sia false; setUnion e setIntersection (ipotizzando che i due vector operandi abbiano la stessa dim) hanno complessità THETA(n) per via dell’operazione di AND/OR che va effettuata su ogni cella i-esima dei due corrispettivi array (prova ad immaginare un AND/OR bit a bit). Nel caso di due dim diverse la complessità si complica.

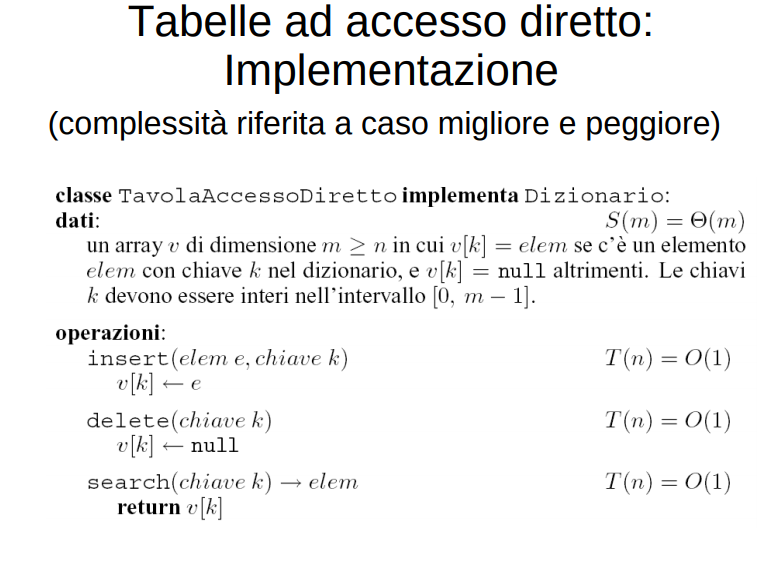


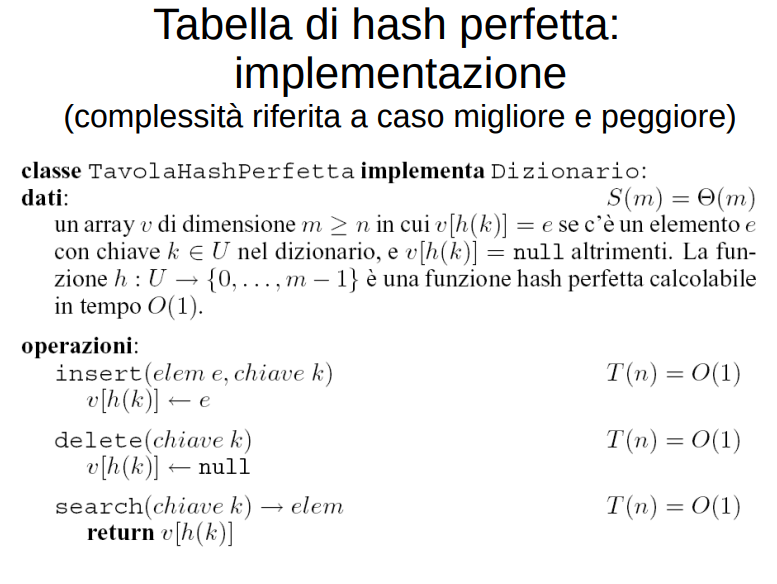
Rispetto ai normali array non ordinati, a livello di complessità THETA (in realtà abbiamo altre ottimizzazioni “non immediatamente visibilI”) cambiano due funzioni:

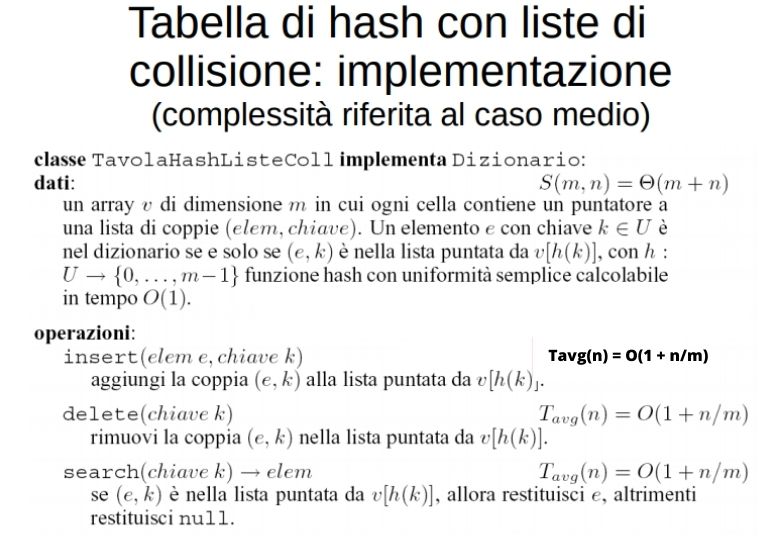
**setUnion**: THETA(n+m) ipotizzando l’uso di un algoritmo di tipo merge (due puntatori che si spostano a sx e dx inserendo gli elementi (ordinati) in un terzo array d’appoggio): dobbiamo dunque scorrere sicuramente l’array1 (n) e l’array2 (m) = THETA(n+m). Senza merge otteniamo una complessità decisamente diversa.

**setIntersection**: Ipotizzando anche qui un algoritmo merge (questa volta scorrendo due puntatori e aumentando entrambi se i due elementi (sx e dx) sono uguali (inserendo), o aumentando solo il minore se diversi); THETA(1) nel caso migliore (intersect vuota) se immaginiamo una ottimizzazione con un controllo sull’ultimo elemento (se < di Elem, inserisco suibito Elem in fondo) oppure direttam. se uno dei due è vuoto → come la insert.

**TDD: DIZIONARIO**

****

****

****

Per Hash implementata con tabelle ad accesso diretto, e anche per Hash perfetta, abbiamo complessità di THETA(m) (con m: dimensione effettiva tabella) per la createEmpty. Però abbiamo complessità di THETA(1) per tutte le altre operazioni. Nel caso della tabella ad accesso diretto, tuttavia, troviamo degli svantaggi evidenti (v. slide).

Per Hash non perfetta implementata con liste di collisione (ipotizzando caso medio e con funzione con uniformità semplice) abbiamo complessità THETA(m) per la createEmpty e complessità di THETA(1+ loadFactor) ~= THETA(loadFactor).

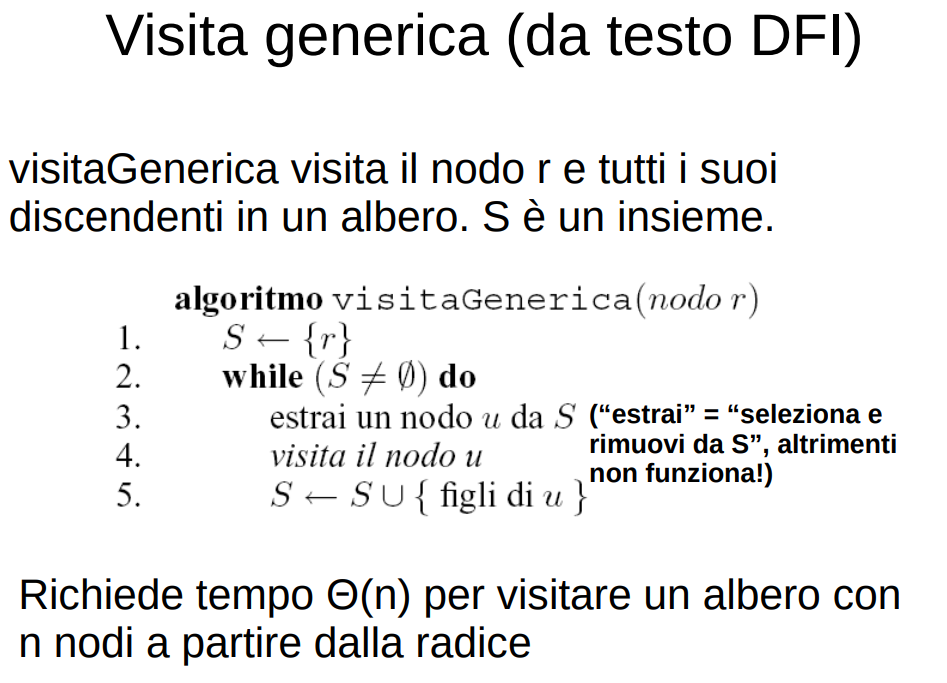
A livello spaziale abbiamo THETA(m+n), che anche se all’apparenza sembra peggiore del THETA(m) dell’Hash implementata con tabelle ad accesso diretto, è difatti migliore perchè l’m considerato è decisamente inferiore (in linea teorica).

NB: attenzione; |U|, m ed n sono tre cose in generale differenti!

**TDD DIZIONARIO (o SET) CON BST** (binary search tree): v.slide per maggiori dettagli

In questo caso abbiamo, per ogni operazione (search, insert, delete) complessità THETA(1) nel caso migliore (trova, inserisce, cancella subito dopo la radice) e THETA(h) nel peggiore (con h: altezza dell’albero, che approssima bene i vari casi) → ricorda perchè no THETA(n)!

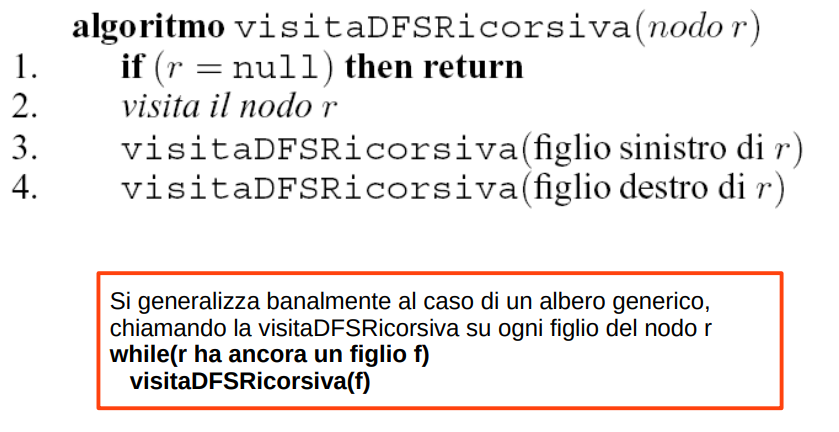
**TDD: TREE**



La visita con Set non è sistematica (non deterministica).

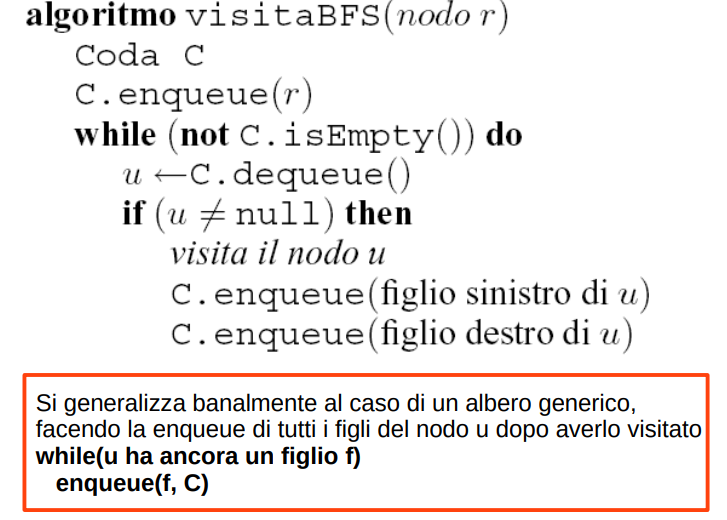
Sostanzialmente copia in un Set la radice, per poi cancellarla e “leggerla”, a questo punto nell’insieme S mette S (vuoto) U figli di u (radice): ora il set ha due elementi. Al prossimo ciclo estrae un elemento x, per poi inserire al suo posto i suoi n figli.

Schematizzando: S = {root} → S = {f1, f2, .. fn} → S = {f11, .. f1n, f2, .. fn}



La visita in profondità (DFS) è sistematica (intrinsecamente ricorsiva, anche se sulle slide si può trovare una sua versione iterariva (simile alla BFS ma con le pile).

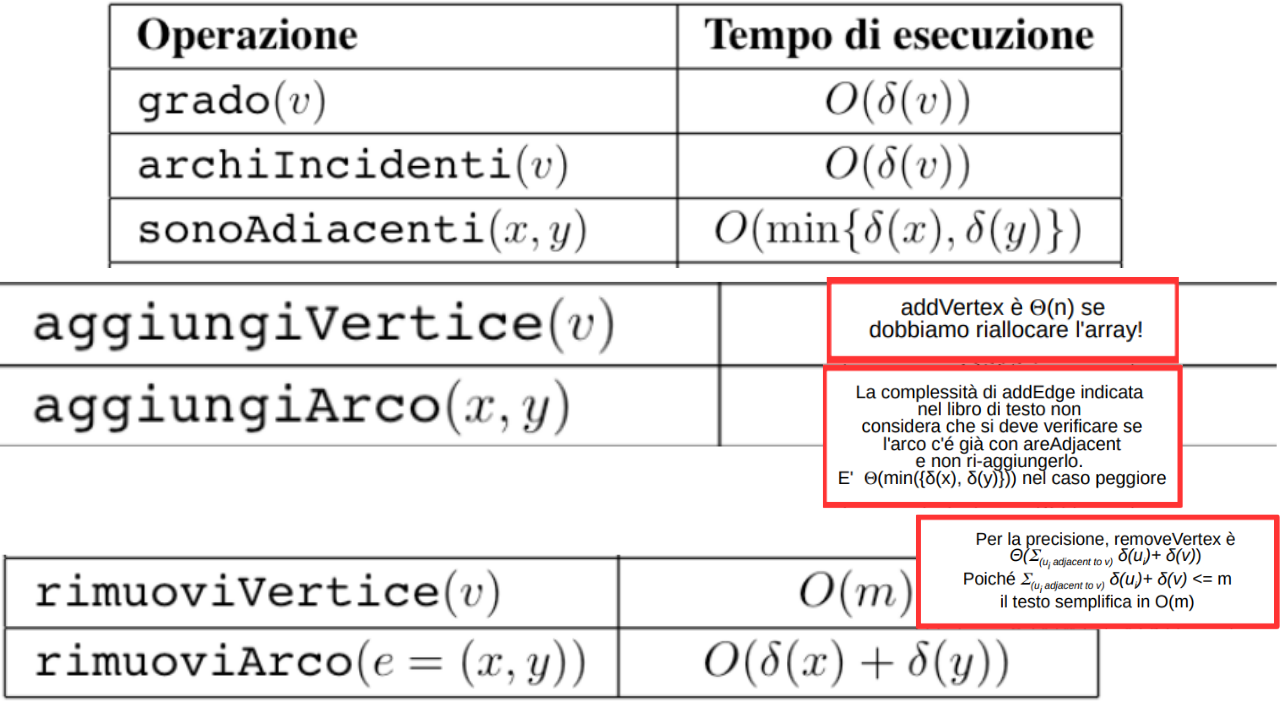
Della DFS esistono 3 varianti: preordine (root, sx, dx), postordine(sx, dx, root) e (anche se non è applicabile agli alberi generici) simmetrica (sx, root, dx).



La visita in ampiezza (BFS) è sistematica (iterativa).

[v. slide e podcast per esempi ed esercizi su tutte le visite]

**TDD GRAFO non orientato e non etichettato con liste di adiacenza con array**

****

Per **degree**(v, G) si accede in tempo costante ad A[v] e si visita la lista di adiacenza sommandone gli elementi → ritorno il grado di v → complessità sempre THETA(grado(v)).

Per **incidentEdges**(v, G) stesso discorso di degree ma ritorno una lista con gli archi.

Per **areAdjacent**(u, v, G), ipotizzo una scansione in paralello delle liste di adiacenza A[u] ed A[v] (sono certo di fermarmi quando finisce la più corta: se nella più corta delle due, supponiamo che sia la lista di u, trovo v, allora sono adiacenti, sennò no): T(min(g(u),g(v)) nel caso peggiore e THETA(1) nel migliore (trovo subito l’adiacenza).

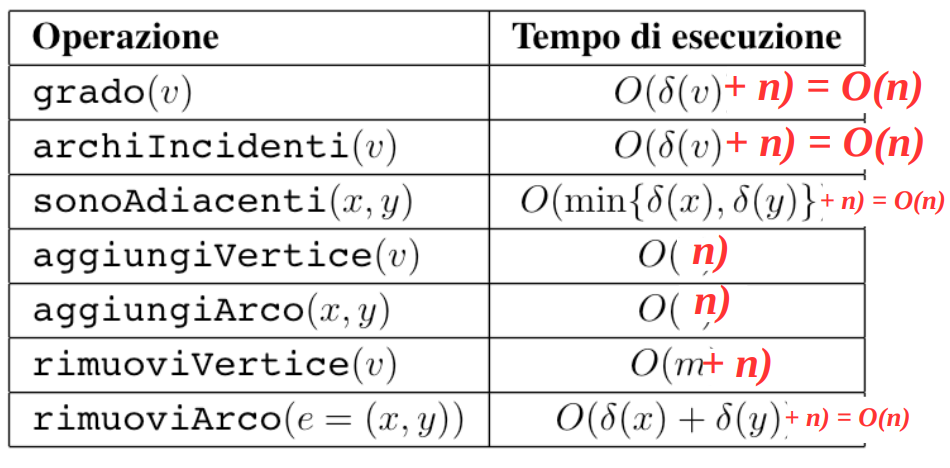
Per la **addVertex**(v, G) sembra logico un THETA(1).. tuttavia vale solo nel caso migliore, poichè l’implementazione con array sz e maxsz impone la necessità della riallocazione: nel caso peggiore quindi T(n). Inoltre questa implementazione ha un vincolo forte sul non poter decidere arbitrariamente le etichette dei vertici (se dopo 3 voglio aggiungere 20? Bratta!)

Per la **addEdge**(u, v, G)avremmo THETA(1) poichè il controllo di esistenza dei vertici è costante (sono < size?) e l’inserimento dell’arco in testa alla lista di u e di v pure.. MA dobbiamo prima verificare che l’arco non esista già tramite la areAdjacent: la complessità dipende dunque da essa (nel caso peggiore T(min(g(u),g(v)) → v.slide per ult. dettagli

Per la **removeVertex**(v, G) dobbiamo prima verificare che il vertice esita (basta controllare che sia < size dell’array: t costante). Poi bisogna scorrere la sua lista di adiacenza e PER OGNI arco accedere alla sua lista per eliminare l’occorrenza parallela di v (e poi eliminare l'arco) (quindi avremo sempre THETA(grado(x)), che coincide col caso migliore nel caso di non esistenza di nodi adiacenti, o comunque di presenza di v in prima posizione nelle liste legate ad essi). Nel caso peggiore otteniamo THETA(sommatoria del grado dei vertici adiacenti a v + grado di v), generalizzabile a THETA(2m) = THETA(m) poichè nel caso sovrastimato abbiamo la visita di TUTTE le liste di adiacenza e perlatro fino in fondo (passo effettivamente per tutti e 2m gli archi → approssimando per THETA: THETA(m)).

Per **removeEdge**(u, v, G) devo controllare che i vertici esistano (tempo costante guardando se sono < size dell’array), rimuovere u dalla lista di v e rimuovere v dalla lista u. A seconda delle posizioni ho caso migliore THETA(1) e caso peggiore THETA(grado(u) + grado(v))

**TDD GRAFO non orientato e non etichettato con liste di adiacenza con lista**

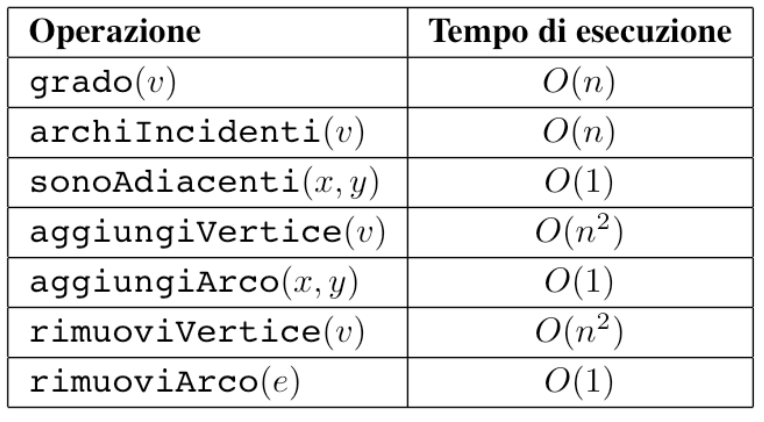
****

Perdendo l’accesso diretto alla lista di vertici devo aggiungere nei casi peggiori un addendo +n ad ogni operazione: questo (a parte la rimuoviVertice che tra poco affrontiamo) ci porta ad avere una complessità nei casi peggiori sempre di THETA(n) (questo perchè n è sempre >= al grado di v (che per natura è uguale al max a (n-1) nel caso abbia tutti i nodi a sè adiacenti meno il cappio), dunque possiamo maggiorare il grado con n). Nel caso di **rimuoviVertice** si deve cambiare l’implementazione (non conviene scorrere A[v] e per ogni suo arco visitare la lista associata a quel vertice adiacente, perchè ogni volta bisognerebbe comunque ricercare il vertice nella lista verticale) → conviene dunque scansionare esaustivamente tutte le altre liste, rimuovendo ogni occorrenza trovata di v (oltre al dover rimuovere la lista in A[v]). Ottengo dunque sempre T(n+m) perchè la m non è semplificabile (m (#archi) può variare tra 0 (non minorabile con n) se il grafo è completamente sconnesso e tra n(n-1)/2 (ordine n^2) se il grafo è completo (neanche quindi maggiorabile con n).

I casi migliori (tranne la rimuoviVertice in cui coincide col caso peggiore T(n+m) li eredito in generale dalla implementazione con array, dato che semplicemente vado ad “eliminare” l’addendo n nel caso di accesso “fortuito” a v in prima pos (o al max a v ed u nelle prime due pos)). Nella addEdge ad esempio nel caso migliore devo trovare le 2 liste nelle prime 2 pos.

*NB*: In aggiungiVertice la n è data però dalla scansione della lista verticale, non dalla realloc

**TDD GRAFO non orientato e non etichettato con matrice di adiacenza**

(...)

Per **degree**(i, G) abbiamo sempre THETA(n) per la scansione della riga o colonna (essendoci simmetria riga o colonna sono equivalenti a questo fine)

Per **incidentEdges**(i, G) stesso discorso di degree ma ritorno una lista con gli archi.

Per **areAdjacent**(i, j, G) basta accedere ad M [ i ] [ j ] (o [ j ] [ i ]) : THETA(1) in ogni caso

**addVertex**(i, G) e **removeVertex**(i, G) *possono richiedere una realloc e quindi avere una complessità* T(n^2) *[complicazione: se rimuovo il vertice con etichetta 3 e “comprimo” tutta la matrice di conseguenza, il vertice che aveva etichetta 4 finisce in posizione 3 e tutti i vertici successivi “perdono una posizione”... quindi “cambiano etichetta”. Devo tener conto di questa ridenominazione. Più sensato marcare un vertice come cancellato e lasciarlo nella matrice].* Nella add la complessità nel caso migliore è T(1) perchè se aggiungo un vertice che è già “allocato” la aggiunta è solo logica (l’utente vede la finestra di utilizzo della matrice aumentare +1). Nella removeVertex potrei semplicemente (v. complicazione) marcare il vertice come cancellato e non reallocare. Nota bene: essendo non orientati, per ogni valore a 1 che trovo nella posizione (i, x) devo anche rimuovere (x, i) per preservare la simmetria (equivale a mettere a 0 sia la riga i sia la colonna i, ma in quel modo dovrei scorrerle sempre entrambe.). Il caso migliore è dunque T(n) in cui scorro solo la riga (o colonna) e trovo solo degli zeri. Tuttavia anche scorrendo entrambe riga e colonna per mettere tutto a 0, si avrebbe caso migliore T(2n) = T(n).

NB: una ottimizzazione di removeVertex (che però “scaricherebbe il barile” alla addVertex in succesiva sede) sarebbe quella di marcare semplicemente il vertice come “unused” (ad esempio ponendo a 1 la sua diagonale); in questo modo si eviterebbe di dover porre a 0 l’insieme delle celle a 1 (lo dovrà però fare la add).

Per **addEdge**(i, j, G) **e removeEdge**(i, j, G) basta cambiare in 1/0 il valore in M [ i ] [ j ] e anche in M [ j ] [ i ] per via della simmetria che va preservata (è non orientato): T(1)

→ *non si è fatta la complessità delle visite (v. slides)*

**TDD CODA CON PRIORITA’**

| **struttura dati→**  **operazioni** | **liste semplici ORDINATE** | **heap (albero binario quasi completo con v > dei figli)** |
| --- | --- | --- |
| **findMax(q) : elem** | T(1) | T(1) |
| **insert(e, k, q) : q’** | T(1) - T(n) | T(1) - T(logn) = T(h) |
| **deleteMax(q) : q’** | T(1) | T(1) - T(logn) = T(h) |

La **findMax** ha sempre T(1) perchè basta accedere al primo elemento della lista/array(root).

La **insert** ha due casi peggiori diversi: nelle liste semplici ordinate bisogna scorrere fino alla fine la lista per trovare il punto in cui inserire (T(n)), mentre nello heap inserisco l’elemento in ultima posizione (prima a sx all’ultimo livello) e faccio risalire l’elemento fino alla pos giusta con scambi successivi: alla peggio effettuo logn = h scambi (fino alla radice).

La **deleteMax** è costante nelle liste ordinate (semplicemente rimuovo il primo elemento (root)) mentre non lo è nello heap: non posso solo togliere la radice (perderei la struttura dell’albero) quindi devo sostituire l’ultimo elemento con la radice per poi spostarlo in basso fino alla pos corretta (scambiandolo, se minore di uno o entrambi i figli, col figlio maggiore):

ottengo dunque T(logn) nel caso peggiore (scambi fino all’ultimo liv,). → v.slide per dettagli