
Algoritmi

Appunti di Giorgio Anzalone

Sommario

Come usufruire di questi appunti.....	7
Introduzione	7
Definizione di Algoritmo.....	8
Spiegazione dell'Insertion Sort.....	9
Random Access Machine	12
Cos'è il tempo computazionale?	13
Best Case → L'array è già ordinato.....	15
Lower Case → L'array è inversamente ordinato	15
Tasso di crescita.....	17
Notazione Asintotica	17
Notazione Big-Oh	18
Caso Upper Bound "O"	18
Caso Lower Bound " Ω "	19
Notazione Little-oh	21
O per Upper Bound non asintoticamente stretti.....	21
ω per Lower Bound non asintoticamente stretti.....	22
Algoritmi di ordinamento	24
Algoritmi Ricorsivi → Quick Sort e Merge Sort.....	29
Le differenze sostanziali tra Quick Sort e Merge sort sono che:	32
Ricorsione	33
Serie di Fibonacci.....	35
Problema dello zaino	38
Caso 1	38
Caso 2.....	39
Equazione di Ricorrenza	42
Analisi del Merge Sort	43
Analisi del Merge Sort con il metodo dell'albero di Ricorsione.	43

Analisi del Merge Sort con il metodo di sostituzione	46
La coda con priorità	50
Cosa fa Heapify?	53
Codice Algoritmo Heapify	54
Complessità di Heapify	54
Estrazione di un elemento / Delete	57
Inserimento	60
Codice di Inserimento	60
Decrease	60
Codice Decrease	60
Rappresentazione Astratta e Rappresentazione fisica	61
Codice:	63
Come creo un Heap?	64
Metodo Master [Esperto]	68
Esempio Importante: Heapify	72
Algoritmi Counting Sort e Heap Sort	74
HeapSort	74
Counting Sort	75
Spiegazione Riassuntiva Counting Sort	83
La Ricerca – Tabelle Hash	84
Problema → Collisioni e Soluzione	89
Come progettare una buona funzione Hash?	92
Metodo della DIVISIONE	93
Metodo della MOLTIPLICAZIONE	95
Hashing Universale Semplice	95
Introduzione alle Tabelle Hash → Prof. Viola	97
Ricerca Tabella Concatenazione	101
Ricerca Senza Successo	101
Ricerca CON successo	102

Digressione Sui grafi	103
Come si rappresenta un grafo?	103
Indirizzamento Aperto	105
Sequenze di scansioni.....	106
L’Inserimento.....	107
La ricerca.....	108
Nota.....	109
Ragioniamo sulle Sequenze di Scansioni.....	110
Hashing Uniforme → Diverso dall’Hashing Uniforme Semplice	111
Strategia Lineare	112
La legge di attrazione dei corpi ma nelle tabelle Hash.....	113
Analisi dell’efficienza dell’indirizzamento aperto.....	117
Analisi computazionale della ricerca SENZA Successo	117
Analisi computazionale della ricerca con Successo.....	120
Analisi degli algoritmi di ricerca nelle tabelle Hash ad indirizzamento Aperto.....	122
Analisi della ricerca senza successo [Indirizzamento Aperto]	123
Analisi della ricerca CON successo [Indirizzamento Aperto].....	125
Riassuntone tabelle Hash	126
Quantum Computing.....	131
Entanglement.....	133
Come faccio a misurare un Qbit?	134
Albero Rosso Nero	135
Rotazione Destra → Right Rotate	140
Funzione di inserimento → RB – INSERT – FIXUP.....	141
Caso 1 → Zio Nero	142
Caso 2 → Zio Rosso	144
La cancellazione di un Albero Rosso-Nero.....	146
Analisi Alberi Rosso-Neri – Prof. Viola	154
Problemi di ottimizzazione	156

Programmazione Dinamica	158
Ottimizzazione.....	158
ROD-CUTTING	159
Pseudo Codice → Rod-Cutting(n).....	160
Problema Rod-Cutting.....	162
Moltiplicare una sequenza di Matrici.....	165
Analisi del problema Matrix Chain Multiplication.....	168
Percorsi minimi o massimi	169
Longest Common Subsequence (LCS)	169
Riassunto LCS → Fatto Online	171
Definizione Ricorsiva di LCS	173
PSEUDO CODICE PER LCS	173
Introduzione all'approccio Greedy.....	175
Approccio Greedy	176
Cosa sono gli Algoritmi Greedy?	176
Problema dello Zaino.....	177
Elementi della strategia Greedy	178
Proprietà di scelte Greedy.....	178
Proprietà di scelta Greedy per il problema Activity Selection	179
Algoritmi di Compressione di HUFFMAN	180
Analisi temporale dell'algoritmo di Huffman.....	185
Correttezza dell'Huffman	186
Spiegazione Viola.....	187
Sottostruttura Ottima e Approccio Greedy nel Problema dello Zaino	191
B-Alberi.....	193
I Grafi	194
Rappresentazione con Liste di Adiacenza.....	195
Matrici di Adiacenza.....	196
BFS	197

Cammini minimi	205
Accenno DFS.....	205
DFS	207
Ordinamento Topologico.....	218
Componenti Fortemente Connesse	220
Algoritmo Topological Sort	221
Correttezza di Topological Sort	222
Cammini Minimi da Sorgente Unica.....	223
Sottostruttura ottima di un cammino minimo.....	223
Lemma 1 + Dimostrazione	224
Archivi di peso negativo.....	224
I cicli.....	224
Rappresentazione dei cammini minimi	225
Riassuntazzo	229
Problema del Single-Source Shortest Path	230
Algoritmo di Bellman-Ford	231
Correttezza dell'Algoritmo Bellman Ford.....	232
Algoritmo di Bellman-Ford.....	234
Algoritmo di Dijkstra.....	235
Come funziona questo algoritmo?.....	235
Quanto è veloce l'algoritmo di Dijkstra?.....	237
All Pairs Shortest-Path.....	238
Cammini minimi e moltiplicazioni tra matrici.....	240
Proprietà dei cammini minimi e procedura Relax	241
Proprietà dei cammini minimi e dei rilassamenti	241

Come usufruire di questi appunti

Questi appunti sono una parte FONDAMENTALE della spiegazione, sarà la sacra guida per lo studio di algoritmi, dato che qui sarà complicato scrivere dimostrazioni etc... creerò un file One Note in cui scriveremo le dimostrazioni o tutte quelle informazioni non scrivibili in maniera facile su word se non facendo immagini su immagini. Da questo File avremo un riferimento alla parte del file One Note da leggere.

Introduzione

Algoritmi sarà una materia fondamentale densa di nozioni e insegnamenti. **Il nostro scopo sarà capire come nasce un algoritmo, come creare un algoritmo, come scegliere se un algoritmo è più o meno efficiente di un altro e tanto altro ancora.**

Impareremo infatti a studiare la correttezza e le performance degli algoritmi. Lo studio degli algoritmi è fondamentale, per uno stesso problema, infatti, è possibile avere più algoritmi e quindi più soluzioni, una soluzione potrà essere più o meno performante di un altro algoritmo.

Cos'è un algoritmo?

In maniera informale possiamo dire che un algoritmo è un modo per imparare a fare qualcosa. Facciamo l'esempio del contare, fin dalle elementari qualcuno ci ha insegnato a contare, inizialmente abbiamo imparato a contare usando le nostre dita, successivamente ci hanno insegnato a contare in colonna. Imparare a contare in colonna o con l'utilizzo delle dita sono algoritmi, e sappiamo bene che l'algoritmo di contare in colonna è più efficiente perché più veloce e preciso di contare con le dita.

Nella pratica?

Nella pratica quando creiamo un algoritmo dobbiamo tener conto della macchina su cui dobbiamo realizzarlo, dobbiamo tener conto il tipo di architettura del processore o in generale il tipo di macchina. **Questo è troppo dispendioso allora ci creiamo un modello ideale,** ovvero una sorta di modello comune a tutte le macchine di qualsiasi tipologia in modo quindi da risolvere il problema di portabilità tra un'architettura ed un'altra.

Definizione di Algoritmo

L'algoritmo è una procedura che prende in input un determinato valore o più valori e seguendo una serie di passaggi lo trasformerà in output. L'algoritmo di base ha lo scopo di definire una sequenza di passi che ci permettano di partire da un input e di arrivare ad un output atteso, un output che vogliamo avere. Un tipo problema è il problema dell'ordinamento, partiamo da una sequenza di n interi, dopo una serie di determinati passaggi dovremmo avere una serie di n numeri ma ordinati per cui $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$.

➔ L'ordinamento è un'operazione importante nell'informatica usata ampiamente come operazione intermedia.

Un algoritmo si dice corretto se effettivamente risolve il problema fornendo effettivamente l'output di nostro interesse partendo da un determinato input. Un algoritmo errato potrebbe non terminare mai fornendo un risultato errato.

Con lo studio degli algoritmi possiamo risolvere una grandissima quantità di problemi, analizzando non solo il problema stesso ma anche la soluzione, e tra le soluzioni quella più corretta ed efficace possibile. A volte scegliere un'operazione che risolve il problema e possibilmente scegliere la soluzione migliore in assoluto a volte noteremo essere anche IMPOSSIBILE.

Studieremo moltissime strutture dati ➔ una struttura dati è modo per conservare un determinato insieme di informazioni e organizzare i dati relativi a queste informazioni nel miglior modo possibile in modo da migliorare gli accessi e la fruizione di queste informazioni conservate.

A volte per alcuni problemi non esisterà nemmeno un possibile algoritmo risolutivo, starà a noi provare a trovarne uno, ad analizzare una nostra soluzione, quello che dovremo imparare sarà analizzare questi algoritmi per capire se effettivamente sono buoni.

Problemi Difficili

- **Ci concentreremo sullo studio di algoritmi efficienti ➔ vi sono vari modi per considerare un algoritmo efficiente**, uno tra queste unità di misura dell'efficienza di un algoritmo è proprio il tempo.
- **Esistono problemi complessi chiamati NP-Hard, sono problematiche di cui effettivamente non abbiamo trovato una soluzione.** Trovare una soluzione per un problema NP-Hard indirettamente vuole dire trovare una soluzione a tutti i problemi

NP-Hard. **Questi problemi sono difficili da risolvere perché solitamente la loro risoluzione ottimale chiede un tempo smisurato per l'esecuzione.**

- A volte un piccolo cambiamento sul problema può comportare enormi modifiche alla risoluzione.
- I problemi NP-Hard sono più comuni di quanto possiamo pensare → Problema del commesso viaggiatore.
- Limiti fisici: sono un altro problema di base dato che a volte soluzioni ipotizzate per problemi NP-Hard richiedono tantissimo tempo.

Efficienza

Uno tra gli aspetti più importanti degli algoritmi ma NON il solo è la **velocità**. La scelta di un buon algoritmo rispetto un altro impatta tantissimo sulla velocità di esecuzione dell'intero processo. **Studieremo ogni algoritmo considerando la loro crescita asintotica**, ovvero come cresce l'esecuzione al crescere degli elementi in input. Facciamo un esempio drastico analizzando anche il contesto → *immaginiamo di avere 2 computer, Computer A e Computer B. Il computer A è un computer mostruosamente veloce circa 1000 volte più veloce del computer B, diciamo questo perché il PC A è come se fosse una Ferrari, riesce ad eseguire 10 miliardi di operazioni al secondo, mentre il PC B riesce ad eseguire solo 10 milioni di operazioni al secondo. Ipotizziamo che entrambi i PC devono riordinare un array da 10 milioni di numeri.*

- *Il PC Ferrari A usa l'insertion sort che studieremo avere un tempo pari a $c2 * n^2$, dove $c2$ che è una costante è poco rilevante se paragonato alla crescita di n^2 .*
- *Il PC Basic B usa il Merge Sort che studieremo avere un tempo pari a $c1 * n * \log n$, dove $c1$ è una costante poco rilevante nel calcolo.*

Risultato è che il PC B finirà molto prima del PC A, oserei dire MOLTO prima di quella sazietà del PC A.

- ➔ PC A con l'algoritmo Insertion Sort impiega 5,5 ore
- ➔ PC B con l'algoritmo Merge Sort impiega 20 minuti nonostante fosse 1000 volte più lento del PC A.

Spiegazione dell'Insertion Sort

Pseudo Codice

```

INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Inserisce A[j] nella sequenza ordinata A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key

```

Studiamo un tipico problema ovvero quello dell'ordinamento. Abbiamo una sequenza in input di N numeri, e un determinato Output atteso degli stessi N numeri ma che seguono una determinata logica, ipotizziamo che vogliamo ordinarli in maniera crescente.

Immagina di essere in piazza con altri vecchi e di star giocando a carte tra una Briscola e l'altra e tra una Moretti ghiacciata e l'altra. Prendiamo una carta alla volta dal tavolo e la inseriamo in maniera ordinata nel nostro mazzo confrontando questa carta con quelle che già abbiamo in mano. Quando tutti gli input sono finiti (quindi quando tutte le carte che potevamo prendere sono state prese) ci fermiamo ottenendo in mano un output composto da tutte le carte pescate e ordinate.

Spiegazione generale dell'algoritmo

- **Logica:** l'elemento corrente, trovato con l'indice J identifica la "Carta Corrente" che viene inserita nel nostro Pool relativo alle carte ordinate. Quello che faremo nel pratico sarà pescare una carta e confrontarla con quelle che già avevamo in mano. Trovata la posizione corretta di questa carta, questa verrà inserita e tutte le altre carte (se vi sono) che erano state messe prima e che si trovano dopo la carta inserita verranno spostate di una cella verso destra in modo da mantenere ancora l'ordinamento.
- **$A[1, j-1]$:** costituisce il Pool di carte che abbiamo in mano e che dovranno essere ordinate
- **$A[j+1 \dots n]$:** corrispondono alla pila delle carte che si trovano ancora sul tavolo.
- Riflessione: in effetti riflettendoci all'interno dell'insertion sort c'è qualcosa di sempre vero, ovvero **una proprietà logica che resta sempre vera** e che ci potrà aiutare a ragionare sulla correttezza dell'algoritmo. Nel nostro caso sappiamo che SEMPRE gli elementi che fanno parte di $A[1, j-1]$ sono elementi sempre ordinati. Abbiamo la certezza che per la conoscenza di quel ciclo e degli elementi che abbiamo analizzato il pool $A[1, j-1]$ è sempre vero. Questa proprietà logica sempre vera ad ogni ciclo viene chiamata **Invariante di Ciclo**.
- Definiamo 3 aspetti fondamentali:
 - o **Inizializzazione:** è vera prima della prima interazione del ciclo. Prima di entrare nel while o nel ciclo possiamo dire che l'invariante di Ciclo è vera?

- **Conservazione:** Se è vera prima della prima interazione del ciclo e resta vera anche dopo prima interazione del ciclo, proprio come in matematica quando si parla di induzione, allora resterà vera l'invariante di Ciclo anche per tutti i possibili cicli successivi.
 - Di base quindi se l'invariante di ciclo è vera prima del primo ciclo e resta vera dopo la prima esecuzione, resterà vera per tutte le altre esecuzioni successive [similare all'induzione matematica].
- **Conclusione:** la conclusione ci permette di interrompere l'induzione matematica sull'invariante di Ciclo, grazie all'uso di una condizione del tipo "Finchè $J < 10 \rightarrow$ allora ..." nel while magari scrivere while($j < 10$). Quando questa condizione non sarà più soddisfatta nel caso specifico del while o quando al contrario magari in un if che interno al while e che ci potrebbe fare uscire, quando questa condizione sarà vera allora usciremo dal ciclo, interrompendo l'induzione matematica.

Applichiamo questi concetti all'Insertion Sort:

- ➔ **Prima del primo ciclo non abbiamo elementi nell'array A** ➔ spostiamoci al primo ciclo, in questo momento A ha un valore A[0], essendo un solo valore questo potrà essere definito già ordinato
 - Nel secondo ciclo abbiamo due elementi appartenenti ad A[0,1], questi due dopo la logica dell'insertion sort sono ordinati? Sì allora vorrà dire che per tutti i prossimi cicli questo algoritmo funzionerà.
 - In questo modo abbiamo spiegato il primo punto e abbiamo sfruttato il concetto dell'induzione matematica per spiegare che ad ogni ciclo seguendo la logica dell'Insertion Sort alla fine l'invariante di ciclo è sempre VERO.
- ➔ **Conclusione:** quando si verifica una determinata condizione allora terminiamo il codice.

Lista Argomenti

- Cos'è un algoritmo?
- L'importanza di un algoritmo e come indichiamo che un algoritmo è più efficiente di un altro
 - Insertion Sort come funziona?

Random Access Machine

Analizzeremo un classico modello chiamato **Random Access Machine** → RAM, **considereremo l'esistenza di un singolo processore, no thread e nemmeno esecuzioni parallele**. Il modello RAM opera utilizzando un set di operazioni, per essere precisissimi dovremmo definire per le singole operazioni anche il tempo che queste vi impiegano, il tempo sicuramente per eseguire una singola operazione sarà molto veloce ma comunque variabile da operazione in operazione. **Per semplicità considereremo che ogni operazione avrà un tempo **COSTANTE**.**

Il modello RAM contiene istruzioni che si trovano comunemente nei computer reali: istruzioni aritmetiche (addizione, sottrazione, moltiplicazione, divisione, resto, parte intera inferiore o floor, parte intera superiore o ceiling), istruzioni per spostare i dati (load, store, copy) e istruzioni di controllo (salto condizionato e incondizionato, chiamata di subroutine e return). Ciascuna di queste istruzioni richiede una quantità costante di tempo.

Questo modello ideale ha delle caratteristiche

1. **Le istruzioni di un algoritmo sono eseguite in maniera sequenziale**; quindi, non stiamo contemplando la possibilità di avere più processori che eseguono operazioni in parallelo. Non stiamo nemmeno considerando l'uso di thread.
2. Ogni istruzione elementare o ad accesso elementare impiega tempo costante.
3. Il modello RAM non tiene conto della gerarchia di memoria
4. Scrivere del Pseudo Codice e non scriveremo in effettivo codice in un linguaggio di programmazione questo per aumentare ancora di più la portabilità di un algoritmo tra architetture diverse.

Le istruzioni elementari sono le seguenti:

- Operazioni aritmetiche (+, -, x, /, %, ceiling)
- Spostamento di dati (Upload, save, copy)
- Controllo (branching condizionale e incondizionale, chiamate di e return)

Tutte richiedono tempo costante.

I dati possono essere di 3 tipi:

- Interi
- Floating Point
- Caratteri

Ogni stringa di dati è codificata da un numero limitato di bit.

Se devo codificare una scritta di lunghezza n posso assumere che essa sia rappresentata con

$$c \times \log_2 n \quad \exists c \in \mathbb{Q}, \quad c \geq 1$$

Nonostante tutte queste semplificazioni di questo modello potremmo comunque riuscire a capire in maniera efficace le performance di un algoritmo. Infatti, dobbiamo sapere che l'efficienza di un algoritmo si misura in funzione della lunghezza dell'input su cui deve lavorare.

Gli strumenti matematici richiesti possono includere la teoria delle probabilità, la combinatorica, destrezza algebrica e capacità di identificare i termini più significativi in una formula. Poiché il comportamento di un algoritmo può essere diverso per ogni possibile input, occorrono strumenti per sintetizzare tale comportamento in formule semplici e facili da capire

Cos'è il tempo computazionale?

#One-Note-1

Il tempo computazionale di un algoritmo è il numero di istruzioni e accessi elementari che fa l'algoritmo in funzione al numero di input che gli vengono passati. Per spiegare più nel dettaglio cos'è facciamo un esempio con un algoritmo che conosciamo ovvero **l'Insertion sort**

```

1. For i=2 to n
2.   Key = A[j]
3.   j = i - 1
4.   while j > 0 and A[j] > key
5.     A[j + 1] = A[j]
6.     J = j - 1
7.   A[j + 1] = Key

```

COSTO	RIPETIZIONE
Riga C1	n
Riga C2	n - 1
Riga C3	n - 1
Riga C4	$\sum_{i=2}^n ti$
Riga C5	$\sum_{i=2}^n (ti - 1)$
Riga C6	$\sum_{i=2}^n (ti - 1)$
Riga C7	n - 1

Spiegazione dei costi:

- C1 ha costo n perché il for in realtà viene letto nel programma n volte ma eseguirà il codice al suo interno n-1** volte questo perché nella lettura numero n il for viene letto e quando viene letto la condizione viene meno e quindi uscirà dal for. Ma per sapere se la condizione viene meno va letto nuovamente il for quindi di per se il for avrà n iterazioni. Cosa uguale nella riga C4 perché il while viene letto tot volte ma eseguirà il codice al suo interno solo tot-1 volte.

- T_i : tempo che impiega il ciclo while per tutte le esecuzioni
- La C prima della riga indica tempo Costante perché, come detto prima, le istruzioni o accessi elementari impiegano tutti tempo costante in questo modello ideale.

Sommiamo adesso tutti i tempi ottenendo:

$$T(n) = c_1 * n + c_2(n-1) + c_3(n-1) + c_4 \left(\sum_{i=2}^n t_i \right) + c_5 \left(\sum_{i=2}^n t_i - 1 \right) + c_6 \left(\sum_{i=2}^n t_i - 1 \right) + c_7(n-1)$$

Studiamoci adesso i vari casi di questo algoritmo prendendo in esame il caso Peggior e il caso Migliore

Best Case → L'array è già ordinato

$$t_i = 1 \quad \forall_i \in \{2, \dots, n\}$$

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

Nel caso migliore non entriamo nemmeno nel file, quindi ci eviteremo le sommatorie → Semplifichiamo moltiplicando e raccogliendo tutte le n.

$$= (c_1 + c_2 + c_3 + c_4 + c_7) n + (-c_1 - c_2 - c_3 - c_4 - c_7)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7) = \mathbf{a}$$

$$= (-c_1 - c_2 - c_3 - c_4 - c_7) = \mathbf{b}$$

$$= \mathbf{an + b}$$

L'insertion sort nel caso migliore ha una complessità di n, quindi un tempo di computazione lineare.

Lower Case → L'array è inversamente ordinato

$$t_i = 1 \quad \forall_i \in \{2, \dots, n\}$$

Per capire come risolvere questo problema useremo **la formula di Gaus** che ci permetterà di tradurre le sommatorie in un'operazione con incognita n.

$$T(n) = c1 * n + c2(n-1) + c3(n-1) + c4\left(\sum_{i=2}^n i\right) + c5\left(\sum_{i=2}^n i - 1\right) + c6\left(\sum_{i=2}^n i - 1\right) + c7(n-1)$$

Con La formula di Gaus che sarebbe $\frac{n(n-1)}{2}$ le sommatorie diventerebbero

$$\begin{aligned} = T(n) = & c1 * n + c2(n-1) + c3(n-1) + c4\left(\frac{n(n-1)}{2}\right) \\ & + c5\left(\frac{(n-1) * n}{2}\right) + c6\left(\frac{(n-1) * n}{2}\right) + c7(n-1) \end{aligned}$$

Facendo le operazioni c4, c5, c6 avranno una n^2 otterremo così:

$$\begin{aligned} = & \left(\frac{c4 + c5 + c6}{2}\right)n^2 + \left(c1 + c2 + c3 + \frac{c4}{2} - \frac{c5}{2} + \frac{c6}{2}c7\right)n + \\ & (-c2 - c3 - c4 - c7) \end{aligned}$$

Secondo quello che abbiamo fatto prima possiamo dire che $an^2 + bn + c$ quindi possiamo dire che il tempo computazione dell'algoritmo insertion sort nel caso PEGGIORE è quadratico.

Quando parliamo degli algoritmi e dobbiamo dire a priori la loro complessità dobbiamo considerare sempre il caso peggiore, **perché considerando il caso peggiore possiamo dire che peggio di così questo algoritmo non può fare nonostante nel caso migliore l'algoritmo è molto efficiente.** Infatti, come possiamo vedere la differenza tra caso peggiore e caso minore è abissale.

Note:

- Il caso peggiore identifica un caso limite, in cui l'algoritmo effettivamente peggio di così non potrà fare. Il caso peggiore dipende i contesti potrà essere più o meno frequente.
- Il caso medio invece è difficile da identificare nella realtà poiché dovremmo considerare l'esatta probabilità che le operazioni possono verificarsi, noi per semplicità penseremo che la probabilità di ogni operazione sia appunto equiprobabile alle altre.
- Il caso migliore è il caso a cui ambiamo, meglio di così questo algoritmo non potrà fare.

Tasso di crescita

Per tasso di crescita si intende come crescono il numero di operazioni all'aumentare dei valori in input. Per farlo useremo il metodo usato quando abbiamo calcolato $Tn()$ e considereremo il termine con il grado maggiore, proprio quando studiavamo il limite che tende ad infinito di un polinomio quello che faremo sarà isolare e considerare solo il termine con grado maggiore.

$$Tn() = c_2n^2 + c_3n + c_1 \rightarrow \rightarrow \text{Crescita sarà di } \Theta(n^2)$$

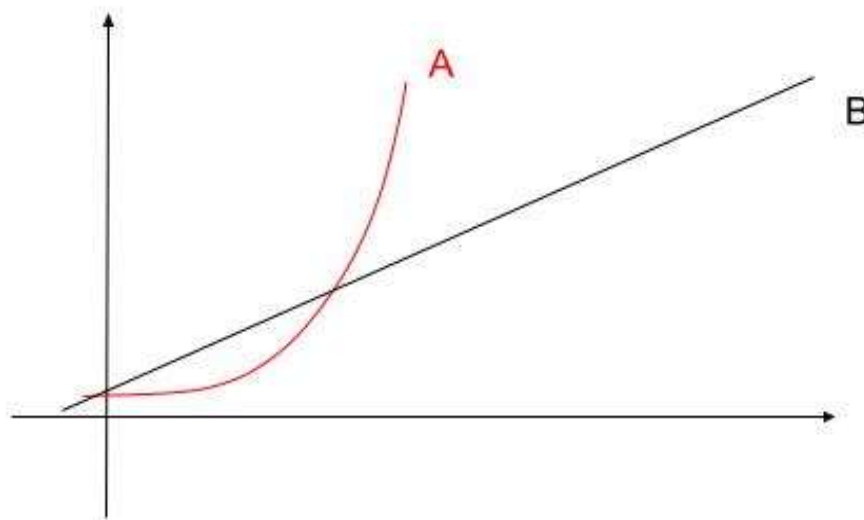
Lower bound $\Omega(n), \Omega(\log n) \dots$

Upper bound $O(n^3), O(n^4) \dots$

Notazione Asintotica

Abbiamo detto all'inizio che un algoritmo può essere più o meno efficiente di un altro ma come faccio a capire questa cosa? Inoltre perché usiamo la notazione asintotica? Usiamo la notazione asintotica per definire il tempo di esecuzione degli algoritmi. Ricorda bene → le notazioni asintotiche sono un insieme di funzioni.

Facciamo un esempio:



Gli algoritmi vanno studiati asintoticamente (in analisi dicevamo definitivamente) non importa se all'inizio per un breve periodo l'algoritmo risulta meno efficiente ma quello che importa è quello che succede da un certo punto in poi, da quel punto in poi l'algoritmo B continuerà ad essere sempre più efficiente dell'algoritmo A e a noi importa studiare questo.

Notazione Big-Oh

Caso Upper Bound "O"

La denotazione big-Oh denota **un Upper Bound asintotico**

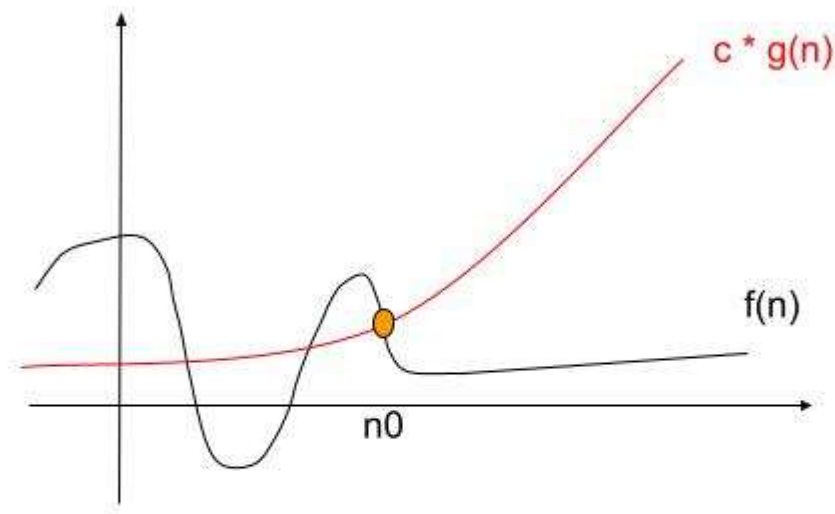
$$f(n) = 7n^3 + 10n^2 - 20n + 6$$

Quello che noi dobbiamo fare è tenere conto nell'esponente con il grado più alto, questa funzione non cresce più velocemente della funzione $k(n) = 8n^3$

$$O(g(n)) = \{f(n) \mid \exists c \in \mathbb{Q} > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad 0 \leq f(n) \leq c \times g(n)\}$$

Quello che stiamo dicendo è che esiste un certo O di una funzione che sarà maggiore o uguale alla funzione $f(n)$, un esempio di funzioni maggiori o uguali alla funzione $f(n)$ sono $O(n^3)$, $O(n^4)$, $O(n^5)$, $O(n^3 \times \log n)$, etc...

Diversamente da analisi quando non mettiamo log con una base ci riferiamo a log in base 2 mentre in analisi quando non mettiamo log ci riferiamo alla base del numero di Nepero ovvero e.



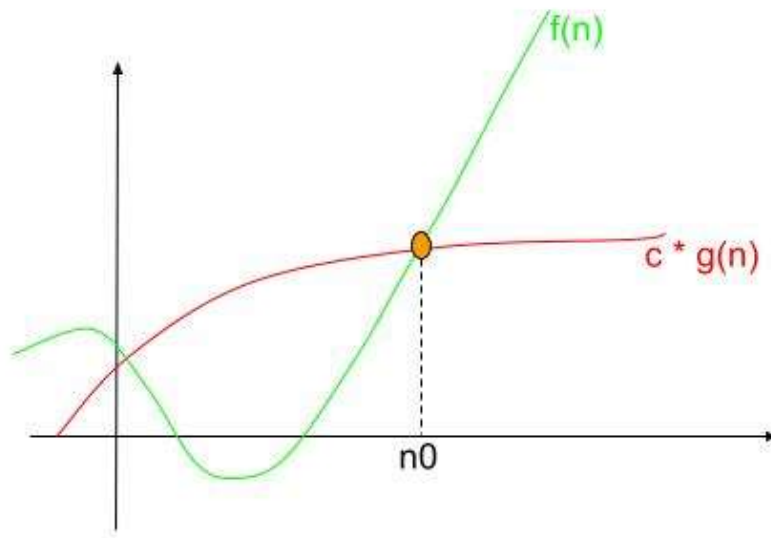
Dopo il punto n_0 asintoticamente $c \cdot g(n)$ crescerà sempre più velocemente rispetto $f(n)$

Con O , stiamo dicendo che esisterà un certo punto n_0 nel quale $c \times g(n)$ sarà sempre maggiore della funzione $f(n)$. Da questo punto n_0 in poi $c \times g(n)$ vincerà sempre su $f(n)$. Sarà una notazione ampiamente utilizzata quando analizzeremo gli algoritmi questo perché ci dà implicitamente un Upper Bound in cui l'algoritmo peggio di così non potrà fare → noi infatti saremmo interessati a conoscere il caso peggiore in cui si comporta l'algoritmo.

Caso Lower Bound " Ω "

$\Omega \rightarrow$ **Denota un Lower Bound asintotico**, il contrario del Upper Bound.

Esisterà un punto n_0 in cui $f(n)$ cresce sempre più velocemente di $c \times g(n)$.



Dopo il punto n_0 asintoticamente $f(n)$ crescerà sempre più velocemente rispetto $c \times g(n)$ — Il contrario del caso precedente con O

$$\Omega(n) = \{f(n) \mid \exists c \in \mathbb{Q} > 0, \quad \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad 0 \leq c \times g(n) \leq f(n)\}$$

Quindi Ω saranno tutte quelle funzioni minori o uguali di $n^3 \rightarrow$ quindi saranno $\Omega(n^3)$, $\Omega(n^2)$, $\Omega(n)$.

Dato che per Upper Bound valgono funzioni maggiori uguali di n^3 (nel nostro caso perché siamo partiti con la funzione iniziale $f(n) = 7n^3 + 10n^2 - 20n + 6$ poteva essere se avessimo avuto come esponente $7n^{10}$ allora avremmo preso in esame n^{10} e che per Lower Bound valgono funzioni minori uguali di esponente a n^3 allora per n^3 varrà sia il O che Ω , allora creiamo $\Theta(n^3)$

Θ : denota un limite stretto

Per capirci $f(n^3)$ non cresce ne più velocemente di $8n^3$ e nemmeno più lentamente di $7n^3$ perché quello che dobbiamo guardare è l'esponente di n proprio come quando in analisi 1 facevamo il limite di una funzione, o $\lim_{n \rightarrow \infty} n^3 = \infty$ o $\lim_{n \rightarrow \infty} 5n^3 = \infty$ sempre più infinito fa, perché non è rilevante il coefficiente davanti ma ha più peso la potenza di n . Stessa cosa qui, $f(n^3)$ è dello stesso ordine di grandezza sia di $8n^3$ sia di $7n^3$, a variare sarà solo la costante davanti all'incognita n con esponente.

$$\Theta(g(n)) \text{ comprende } = \{f(n) \mid \exists c_1, c_2 \in \mathbb{Q} > 0, \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Teorema:

$$\forall f(n), g(n) \text{ NON negativa} \quad f(n) = \Theta(g(n)) \leftrightarrow \{f(n) = O(g(n)), f(n) = \Omega(g(n))\}$$

Quello che abbiamo detto prima, si avrà Θ solo per la stessa funzione uguale sia in O che in Ω .

Notazione Little-oh

o per Upper Bound non asintoticamente stretti

Serve per limiti superiori o inferiori che però non sono stretti. Facciamo un esempio $2n$ e $2n^2$ hanno come $O(n^2)$

- $2n^2 = O(n^2) \wedge 2n^2 = \Theta(n^2) \rightarrow \text{allora} \rightarrow 2n^2 \neq o(n^2)$
- $2n = O(n^2) \wedge 2n \neq \Theta(n^2) \rightarrow \text{allora} \rightarrow 2n = o(n^2) \rightarrow$ Questo è un limite superiore STRETTO perché $O(n^2)$ è dello stesso ordine di grandezza di $2n^2$. Con " o " indichiamo un Upper Bound non asintoticamente stretto quindi, indichiamo funzioni solo maggiori e non maggiori uguali come nell'Upper Bound " O ". **Con " o " indichiamo solo funzioni maggiori a $f(n)$ con invece " O " indichiamo funzioni maggiori o uguali a $f(n)$ con $f(n)$ che in questo caso è $2n$.**

$$o(g(n)) = \{f(n) \mid \forall c \in \mathbb{Q} > 0, \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad 0 \leq f(n) < c \times g(n)\}$$

- **PROPOSIZIONE 1:** $f(n) = o \times g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \rightarrow g(n)$ avrà grado maggiore rispetto a $f(n)$
- **PROPOSIZIONE 2:** $f(n) = O(g(n))$ e $f(n) \neq \Theta(g(n)) \leftrightarrow f(n) = o(g(n)) \rightarrow$ Che è la cosa che abbiamo detto prima ovvero che " o " è **per upper bound non asintoticamente precisi; quindi**, se $f(n) \neq \Theta(g(n))$ vuol dire che $f(n)$ è una funzione di grado minore a $\Theta(g(n))$ ma che comunque ha $O(g(n))$.
 - Esempio $2n = O(n^2)$ ma $2n \neq \Theta(n^2)$ perché per essere Θ deve valere sia O che Ω il che in questo caso esiste ma non sarà n^2 ma solo n , quindi avremo $\Theta(n)$ e non $\Theta(n^2)$. Allora $f(n) = o(g(n))$ il che è vero, quindi dire $2n = o(n^2)$ è vero ed è corretto.

Possiamo quindi dire che O piccolo vale per Upper Bound non asintoticamente stretti e quindi per funzioni solo maggiori a $f(n)$. O grande vale per Upper Bound anche asintoticamente stretti quindi per funzioni sia maggiori che uguali a $f(n)$.

$$2n^2 = O(n^2) \rightarrow \text{Vero}$$

$$2n^2 = o(n^2) \rightarrow \text{Falso Ma sar\`a vero dire } 2n^2 = o(n^3)$$

ω per Lower Bound non asintoticamente stretti

$$\omega(g(n)) = \begin{cases} \forall c \in \mathbb{Q} > 0, \exists n_0 \in \mathbb{N}: \forall n \geq n_0 \\ 0 \leq cg(n) < f(n) \end{cases}$$

Stesso medesimo ragionamento per " ω ". La ω denota Lower Bound non asintoticamente stretti quindi funzioni con grado SOLO minore alla funzione $f(n)$. [Il tutto considerando sempre l'esponente massimo dell'equazione.]

Esempio:

$$2n = \Omega(n), \quad 2n = \Theta(n), \quad 2n \neq \omega(n)$$

$$2n^2 = \Omega(n), \quad 2n^2 \neq \Theta(n), \quad 2n^2 = \omega(n)$$

Consideriamo $f(n)$ come $2n^2$ e consideriamo come $g(n)$ n . Se mettessimo in rapporto le due funzioni e applicassimo il limite per x che tende a infinito il risultato sarebbe sempre infinito.

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(n)} = \infty$$

Proprietà transitiva:

$$f(n) = \Theta(g(n)) \quad \wedge \quad \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \quad \wedge \quad O(h(n)) = O(h(n))$$

Stesso ragionamento con tutti gli altri " ω " e " ω "

[Nota: Non sempre posso usare la notazione asintotica per confrontare due funzioni
esempio: n e $n^{1+\sin(n)}$ NON sono comparabili]

$f(n) = O(g(n))$	equivale a	$a \leq b$
$f(n) = \Omega(g(n))$	equivale a	$a \geq b$
$f(n) = \Theta(g(n))$	equivale a	$a = b$
$f(n) = o(g(n))$	equivale a	$a < b$
$f(n) = \omega(g(n))$	equivale a	$a > b$

NOTA IMPORTANTE: Una funzione oscillante non può essere confrontata con una funzione lineare o esponenziale, questo perché al variare di N che questo sia pari o dispari la funzione oscilla tra -1 e 1 o tra altri valori. Una funzione oscillante è sempre del tipo $(-1)^n$ oppure $n^{(1+\sin(n))}$ etc...

#One-Note-2

Lista Argomenti

- Notazioni Asintotiche:
 - Big O
 - Big Omega
 - Theta
 - Little O
 - Little Omega

Algoritmi di ordinamento

Questo capitolo ci baseremo su una tabella che sicuramente verrà ripetuta più volte:

Nome	Caso Pessimo	Caso Migliore	Adattività	In loco
Selection Sort	$O(n^2)$	$O(n^2)$	X	V
Insertion Sort	$O(n^2)$	$O(n)$	V	V
Bubble Sort	$O(n^2)$	$O(n^2)$	X	V

Spieghiamo quello che abbiamo scritto all'interno di questa tabella partendo dal primo algoritmo.

- Il Selection Sort è un algoritmo che non ha variazione di tempo computazionale tra caso migliore e caso peggiore.
- **L'Insertion Sort ha una complessità di $O(n^2)$ nel caso pessimo. Il caso pessimo nell'Insertion sort si verifica quando l'array è inversamente ordinato.** Tra i 3 algoritmi di base è l'unico che risulta essere adattivo, ovvero **sfrutta porzioni di array già ordinato.**
- Bubble Sort si basa sullo scambio di coppie di numeri. **Di base questo algoritmo non ha controlli che possono minimizzare il numero di scambi.** Noi studieremo gli algoritmi di base proprio come nascono è ovvio che modificandoli potremmo avere risultati migliori.

Cosa indica la colonna in loco?

Un algoritmo di ordinamento si dice **in loco** quando non utilizza memoria aggiuntiva, lavorerà solo con la memoria data in input più al massimo qualche variabile di appoggio. In sostanza non verranno sicuramente usati altri vettori di sostegno all'array.

Stabilità

Stabilità è un concetto fondamentale dello studio di algoritmi. Per capirlo dobbiamo fare dei piccoli esempi che ci porteranno passo dopo passo a capire tale concetto

Concetto di Stabilità

13	6	10	3	4	5	11	7
----	---	----	---	---	---	----	---



Vogliamo ordinarlo, da questo insieme di valori vogliamo ricavare un output con questi stessi valori ma ordinati in maniera crescente.

3	4	5	6	7	10	11	13
---	---	---	---	---	----	----	----



Ma siamo stati fortunati perché non abbiamo valori uguali tra di loro ma tutti valori diversi. Se avessimo avuto valori uguali tra di loro come magari più numeri 3 o più numeri 10, le soluzioni per questo ordinamento non sarebbe più stata 1 ma avremmo avuto diverse soluzioni, perché avere più numeri 3 non vuol dire avere la stessa entità. Possiamo avere più valori 3 in celle diverse dell'array ma che però sono entità diverse. Per capire questo concetto immaginiamo che i numeri vanno in ospedale al pronto soccorso. Vogliamo sempre ordinarli per numero crescente ma dobbiamo tenere conto del loro arrivo. Se due numeri 3 si sono rotti il piede io voglio prendere il numero 3 che è arrivato prima. Per capire questo concetto mettiamo delle piccole lettere vicino i numeri, il numero con la lettera A va servito prima del numero con la lettera B. In questo modo creeremo delle priorità

13A	3A	10A	3B	10B	10C	11A	13B
-----	----	-----	----	-----	-----	-----	-----



3B	3A	10C	10A	10B	11A	13A	13B
----	----	-----	-----	-----	-----	-----	-----

Ad esempio questo ordinamento non tiene conto dell'ordine di arrivo, 3A si è rotto il piede come 3B ma 3A è arrivato prima, quindi va inserito prima di 3B. Piccola nota Noi non stiamo ordinando per arrivo, ma stiamo ordinando sia tenendo conto del numero quindi se è 3,10 etc... sia del loro arrivo dando una priorità maggiore se i numeri sono uguali al numero arrivato prima. Se entrambi hanno codice rosso farò entrare il primo codice rosso che è arrivato. Successivamente entreranno i codici arancioni dando priorità al primo codice arancione arrivato e così via.



3A	3B	10A	10B	10C	11A	13A	13B
----	----	-----	-----	-----	-----	-----	-----

Output corretto e stabile, che rispetta le priorità assegnate. Quando un algoritmo è stabile il risultato sarà 1 e solo 1

La stabilità è una caratteristica intrinseca degli algoritmi, ovvero è una cosa che si può avere con delle modifiche oppure non si potrà mai avere. Infatti, non tutti gli algoritmi possono diventare stabili, alcuni come vedremo possono diventarlo con piccolissime modifiche logiche altri proprio per come sono strutturati non possono.

Piccole nozioni sulla stabilità:

1. **La stabilità fornisce un solo risultato come output effettivo in cui gli stessi elementi uguali hanno una priorità tra di loro.**
2. La stabilità tiene conto dell'ordine di arrivo degli oggetti, infatti un concetto fondamentale quando parliamo di ordinamento è che un **algoritmo di ordinamento in realtà fornisce più soluzioni quando esistono elementi uguali con lo stesso valore**. Nella grafica sopra mostrata viene portato un esempio, un array che ha più valori uguali può scambiare questi valori nonostante siano le stesse quantità, ma quei due valori sono entità diverse entità che hanno però lo stesso numero, scambiandole otterremo un altro risultato. **Con la stabilità questo non avviene perché si tiene conto anche dell'ordine di arrivo di questi elementi**, ovviamente non esisterà mai un elemento che ha lo stesso numero e lo stesso arrivo. Possiamo immaginare la lettera accanto al numero come una pseudo chiave che identifica la priorità che ha un elemento su un altro che ha il suo stesso valore.
3. **La stabilità è una qualità degli algoritmi. Solitamente si può ottenere con piccole modifiche logiche, in altri casi a causa della struttura stessa dell'algoritmo non è possibile ottenere una buona stabilità.**

Esempio di Pseudo Codice → Selection Sort

Max(A,M) //funzione che restituisce il massimo tra due elementi.

M=0 //indice per il massimo

For $i \leftarrow 1$ to $M - 1$ do

 If $A[M] \leq A[i]$ //Se $A[M]$ è minore uguale all'elemento puntato da i allora si scambiano

 Then $M \leftarrow i$

Return M

L'algoritmo originale ha il $<$ e non il \leq con questa piccolissima modifica potremmo rendere l'algoritmo stabile... O forse no?

In realtà il selection sort ha una problematica di base ovvero che tende a scambiare anche elementi MOLTO distanti tra di loro, ma se allora scambia elementi molto distanti tra di loro e non elementi vicini perdiamo la priorità e non sappiamo più se

quello scambiato ha una priorità maggiore o minore il ciclo successivo. Quindi il selection sort è un algoritmo che nonostante delle modifiche (ovviamente non possiamo stravolgere il codice) non potrà essere stabile in tutti i casi.

Cosa diversa avviene con l'Insertion Sort che viene mantenuto stabile perché scambia elementi in posizioni adiacenti e quindi vicine, non scambia elementi lontani tra di loro. Stessa cosa avviene per il Bubble Sort che anche se poco efficiente perché con nessun controllo di base fa scambi adiacenti tra di loro su coppie vicine a due a due.

Modifichiamo allora la tabella di prima aggiungendo ora che conosciamo il concetto di Stabilità anche la colonna denominata stabile.

Nome	Caso Pessimo	Caso Migliore	Adattività	In loco	Stabilità
Selection Sort	$O(n^2)$	$O(n^2)$	X	V	X
Insertion Sort	$O(n^2)$	$O(n)$	V	V	V
Bubble Sort	$O(n^2)$	$O(n^2)$	X	V	V

Insertion sort

Tra i 3 algoritmi di ordinamento possiamo definirlo il più efficiente perché ha un caso migliore sicuramente più veloce in termini di tempo di computazione rispetto gli altri due algoritmi inoltre è **adattivo** ovvero **sfrutta l'ordinamento parziale di un array, se per puro caso l'array ha degli elementi iniziali già ordinati l'algoritmo diminuirà le sue iterazioni.** Vediamo il suo codice iterativo composta da cicli e successivamente lo trasformiamo in un algoritmo ricorsivo.

Iterativo

For int $i \leftarrow$ to $n - 1$ do

 Intert(A,i) //chiamata a funzione insert

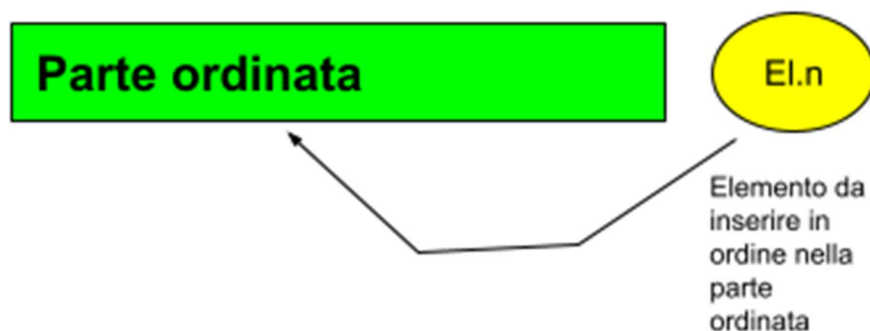
Insert(A, i)

 While($A[j - 1] > A[j]$) do

 Scambia(A, j-1, j)

$j \leftarrow j - 1$

Adesso trasformiamo la procedura da iterativa a ricorsiva. Come prima cosa partiamo dal caso base. **La logica dell'insertion sort è dividere l'array in 2 parti**, una parte già ordinata e una parte da ordinare, **ogni volta per ogni iterazione la parte ordinata aumenta sempre di più grazie al fatto che prendiamo il primo elemento dopo la parte ordinata e lo mettiamo in ordine all'interno della sezione ordinata**. Per trasformare questa cosa in ricorsiva, dobbiamo dare per scontato che già l'array è ordinato e non partiremo dal primo elemento ma dall'ultimo. Ma come partiamo dal dare per scontato che l'array è ordinato? Si proprio così a noi non interessa per ora è ordinato anche se non lo è, se scriviamo e pensiamo bene il codice farà tutto il PC. Quindi partiamo dalla logica che l'array è ordinato successivamente prendiamo un elemento e lo mettiamo in ordine nella parte ordinata questo in maniera ricorsiva finché non arriverò al caso base in cui ho un solo elemento.



Insertion sort(A,m)

 If $n \leq 1$ then return

 Insertion sort (A, n-1)

 Insert (A, n-1)

Algoritmi Ricorsivi → Quick Sort e Merge Sort

Alla tabella di prima aggiungiamo questi 2 nuovi algoritmi di ordinamento ricorsivi. In modo da studiare più in profondità come funziona la ricorsione.

Nome	Caso Pessimo	Caso Migliore	Adattività	In loco	Stabilità
Selection Sort	$O(n^2)$	$O(n^2)$	X	V	X
Insertion Sort	$O(n^2)$	$O(n)$	V	V	V
Bubble Sort	$O(n^2)$	$O(n^2)$	X	V	V
Quick Sort	$O(n^2)$	$O(n \log n)$	X	V	X
Merge Sort	$O(n \log n)$	$O(n \log n)$	X	X	V

Spieghiamoli in parallelo partendo dalle loro logiche di base.

Il quick sort è un algoritmo estremamente casuale dato che la sua efficienza è data dalla corretta scelta del Pivot. Se il Pivot divide perfettamente a metà l'array e quindi si ha un ottimo bilanciamento il quick sort si trova nel caso migliore, se invece il Pivot preso casualmente si trova alle estremità dell'array ci ritroviamo nel caso peggiore.

Il merge sort è in maniera teorica il miglior algoritmo di ordinamento dato che per un caso peggiore il massimo che possiamo riuscire a fare è proprio avere una complessità $O(n \log n)$ meglio di così non possiamo fare. Entrambi come già detto sono algoritmi ricorsivi e quindi pensati in maniera diversa dai 3 algoritmi spiegati precedentemente.

Algoritmi ricorsivi

Un algoritmo ricorsivo si applica ad un problema che può essere scomposto in sottoproblemi che però hanno la stessa natura. La risoluzione di un problema in maniera ricorsiva avviene quasi per magia dato che fin dall'inizio si dà per certo che la soluzione è avvenuta. **Scrivere codice in maniera ricorsiva è sicuramente più elegante tanto da rendere la soluzione capibile in poco tempo,** il vantaggio della ricorsione è puramente concettuale; infatti, non è un modo diverso di risolvere problemi ma **un modo diverso di**

pensare al problema stesso. Ritornando al caso dei due algoritmi di ordinamento esposti prima entrambi dividono il problema in 2 sottoproblemi ...

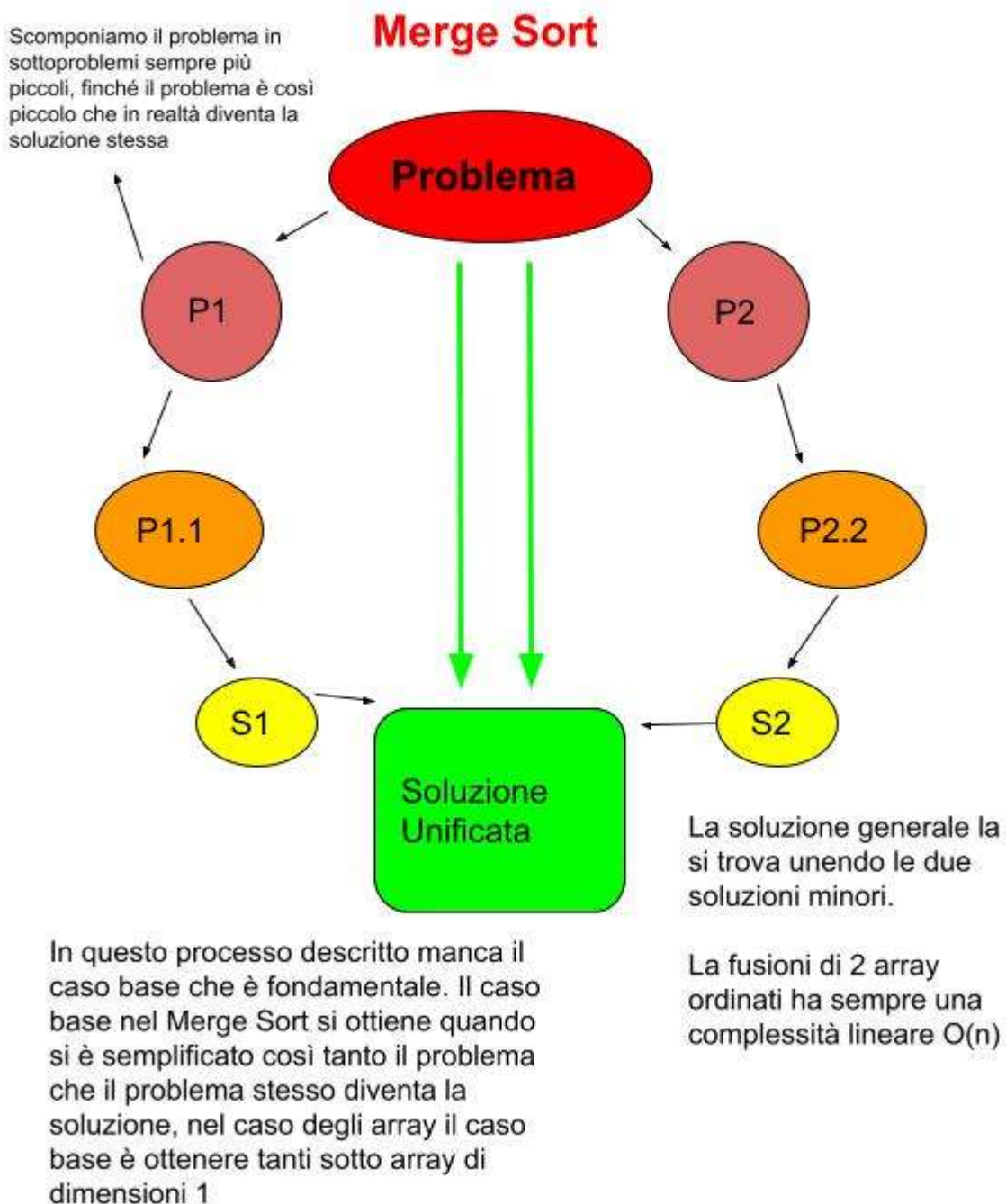
Spiegazione alternativa:

Gli algoritmi ricorsivi sono algoritmi molto utili per risolvere alcuni problemi, sono degli algoritmi che richiamano se stessi risolvendo un determinato problema che ha una certa caratteristica.

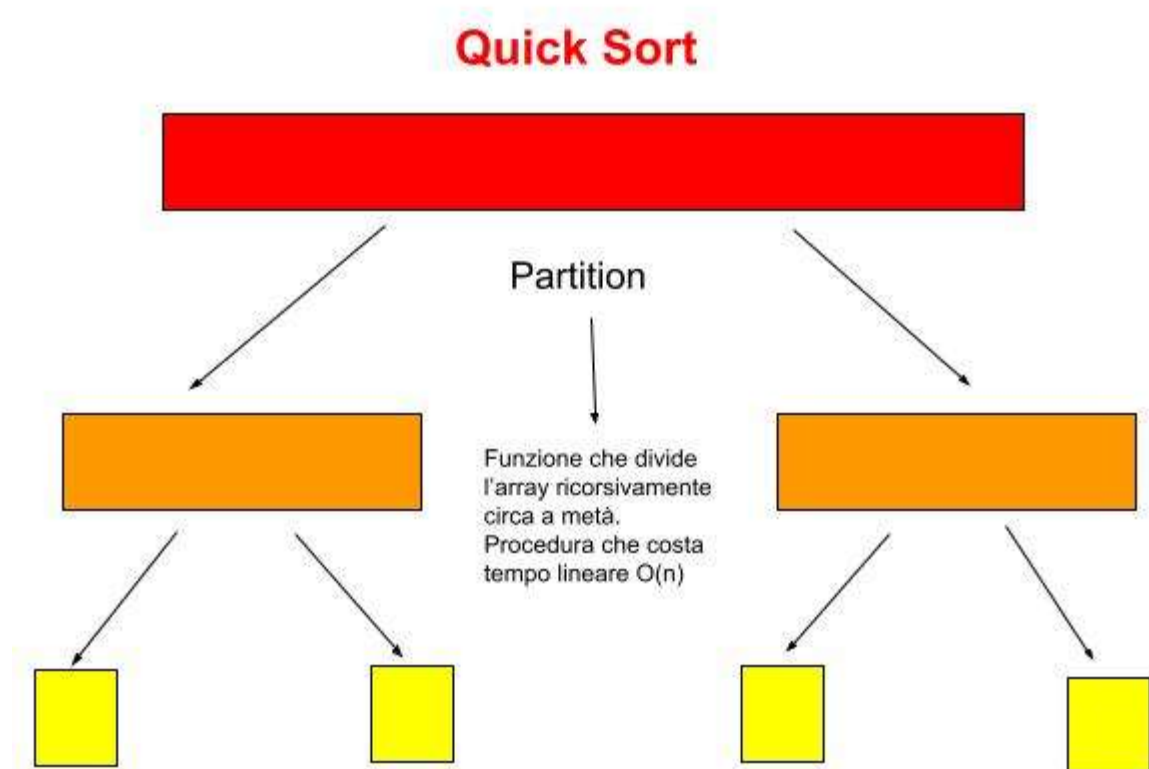
Generalmente questi algoritmi ricorsivi adottano un approccio **Dividi et Impara** → ma che vuol dire e in cosa consiste questo approccio?

L'approccio Dividi et Impara si divide in 3 fasi:

1. **Divide:** divisione del problema in sottoproblemi → questi sottoproblemi sono più piccoli e più semplici da risolvere rispetto la risoluzione del problema più grande. Questo ci permette di trovare soluzioni più semplici a problemi più complessi.
2. **Impera:** i sottoproblemi vengono risolti in maniera ricorsiva → ma che vuol dire ricorsione? Abbiamo detto che quello che farà la funzione sarà richiamare se stessa è quindi fondamentale che in una funzione ricorsiva vi sia sempre un caso base, **un caso noto che ha una risoluzione nota ed esplicita**. Le altre soluzioni si appoggeranno sulla soluzione nota.
3. **Combina:** le soluzioni dei sottoproblemi vengono combinate per generare la soluzione al problema finale.



Il Quick Sort è un algoritmo che non fornisce output stabili perché appunto ha una logica molto casuale. Inoltre, una differenza sostanziale che ha con il merge sort è che preso un Pivot, riordina a sinistra i valori minori e a destra i valori maggiori al pivot, successivamente le altre due metà vengono ridivise e riordinate in maniera ricorsiva.



Per sapere nel dettaglio come funziona il quick sort guarda il suo capitolo su programmazione 2.

Le differenze sostanziali tra Quick Sort e Merge sort sono che:

1. **Quick Sort lavora in loco**, quindi non utilizza altri array di supporto. Si limita a dividere e riordinare parzialmente
2. **Il merge sort non lavora in loco**, infatti ha bisogno di un array di appoggio per poter conservare le informazioni momentaneamente. **In maniera teorica il merge sort è il migliore algoritmo di ordinamento trovato grazie al fatto che il suo caso migliore e peggiore si eguagliano in complessità e dato che per i casi peggiori il meglio che siamo riusciti a trovare è stato un algoritmo di complessità $O(n \log n)$** , ma, **nonostante ciò, nel pratico il merge sort risulta essere più lento del quick sort a causa dell'utilizzo di questo array di supporto.**
3. **Il Merge Sort dà più focus alla fusione degli elementi mentre il Quick Sort riordina prima gli elementi**, mentre la fusione avviene in maniera gratuita dato che alla fine come risultato avremo 2 metà di array già ordinate secondo il primo Pivot preso.
4. **Il Quick sort non è un algoritmo stabile perché molto casuale sia nella scelta del pivot sia nella sua implementazione.** La divisione a metà secondo un Pivot potrà invalidare la priorità che si ha tra elementi.

Ricorsione

La ricorsione parte dai problemi, analizza il problema e grazie al problema stesso trova la soluzione. Un esempio super classico di ricorsione è il fattoriale con

$$f(n) = n! = 1 \times 2 \times 3 \times 4 \times \dots \times n$$

Definizione per casi nonché la più corretta:

$$\begin{cases} 1 & \text{se } n = 1 \\ n \times f(n-1) & \text{se } n > 1 \end{cases}$$

Questa definizione per casi spiega già il concetto di ricorsione e come risolvere questo problema ricorsivamente, **n=1 è il caso base, mentre la seconda riga si costruisce in funzione del caso base, dato che n-1 alla fine arriverà ad essere = 1. Noi supponiamo di avere già il problema del problema più piccolo in quel caso n-1.** A noi non interessa fare i conti farà tutto la macchina noi ci limitiamo a dare le regole delle cose da fare.

Soluzione Ricorsiva in Pseudo codice:

If n=1 then return 1

Return n x fattoriale (n-1)

Soluzione iterativa in Pseudo Codice:

Fattoriale(n)

P ← 1

For i ← 2 to n do

 P ← P x i

Return P

Eseguendo una funzione ricorsiva possiamo dar vita al così detto **ALBERO DI RICORSIONE** composto da tutte le chiamate ricorsive fatte durante l'esecuzione del programma.

Facciamo un esempio affrontando il problema della moltiplicazione:

$$\text{MUL}(X, Y) = X \times Y \quad \text{con } X, Y > 1$$

Immaginiamo che siamo al PC e si rompe il tasto della moltiplicazione, quindi non lo possiamo usare, siamo costretti a calcolare la moltiplicazione con una serie di somme. Facciamo un esempio: $3 \times 5 = 3+3+3+3+3$ oppure a $5 \times 5 \times 5$.

Calciamo queste serie di somme ricorsivamente, scriveremo:

$$MUL(X, Y) = \begin{cases} X & \text{se } Y = 1 \\ X + MUL(X, Y - 1) & \text{se } Y > 1 \end{cases}$$

- Spiegando il tutto, **se Y = 1 allora il risultato sarà X perché un numero moltiplicato ad 1 sarà sempre il numero stesso.**
- **Se Y è maggiore di 1 sommiamo il numero tante volte finché Y-1 non è = 1.** Alla fine, Y arriverà al caso base, e avrà fatto tutte le somme di X + se stesso Y volte.

Ma rendiamolo più efficiente, invece che decrementare Y perché non lo dividiamo circa a metà, in questo modo avremo più chiamate ricorsive ma che avvengono in parallelo.

È proprio in questi casi che abbiamo bisogno di vedere e creare un albero di ricorsione in modo da vedere ciò che succede.

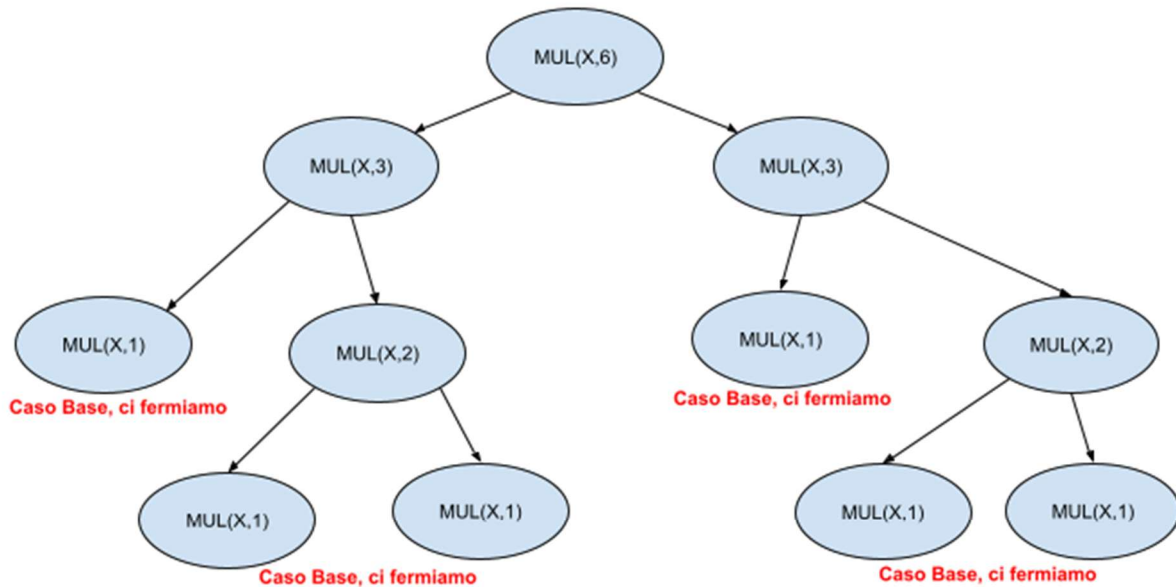
$$MUL(X, Y) = \begin{cases} X & \text{se } Y \leq 1 \\ MUL\left(X, \left\lceil \frac{Y}{2} \right\rceil\right) + MUL\left(X, \left\lfloor \frac{Y}{2} \right\rfloor\right) & \text{se } Y > 1 \end{cases}$$

Consideriamo una moltiplicazione come 3x6 e va fatta semplicemente usando delle semplici somme.

1. $MUL(3, 6/2) + MUL(3, 6/2) \rightarrow MUL(3, 3) + MUL(3, 3)$
2. $MUL(3, 3/2) + MUL(3, 3/2) \rightarrow MUL(3, 1,5) + MUL(3, 1,5)$
3. $MUL(3, 3/4) + MUL(3, 3/4) \rightarrow MUL(3, 0.75) + MUL(3, 0.75)$: Ci fermiamo ritornando ricorsivamente il valore 3 ottenendo quindi $3+3+3+3+3+3 = 3 \times 6 = 18$

1. Moltiplicazione di $2 \times 4 = 2+2+2+2=8$
2. $MUL(2, 2) + MUL(2, 2) = 2+2$
3. $MUL(2, 1) + MUL(2, 1) = 2+2+MUL(2, 1)+MUL(2, 1) = 2+2+2+2=8 \rightarrow$ Il primo numero è X ovvero quello che andremo a sommare il secondo numero invece sono tutte le ripetizione che ci restano da fare, se il numero di ripetizione è ≤ 1 allora ci fermiamo.

ALBERO DI RICORSIONE - Somma



In totale sommiamo X tante volte quanto il caso base, il caso base è MUL(X,1), che appunto saranno 6.

Queste chiamate ricorsive avvengono in parallelo a rispetto delle chiamate singole.

Serie di Fibonacci

Vogliamo calcolare la serie di Fibonacci con un algoritmo ricorsivo, **la serie di Fibonacci somma l'elemento in posizione N-1 con l'elemento in posizione N-2**, quindi sarebbe $1+1+2+3+5+8+13+21+\dots$ etc... [la serie di Fibonacci ricorsivamente con un pseudo albero di ricorsione lo abbiamo affrontato anche su Programmazione 2, nella sezione ricorsione.]

Algoritmo ricorsivo:

$$F(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ F(n-2) + F(n-1) & \text{se } n > 2 \end{cases}$$

Algoritmo iterativo:

FIB(n)

IF $n \leq 2$ THEN RETURN 1

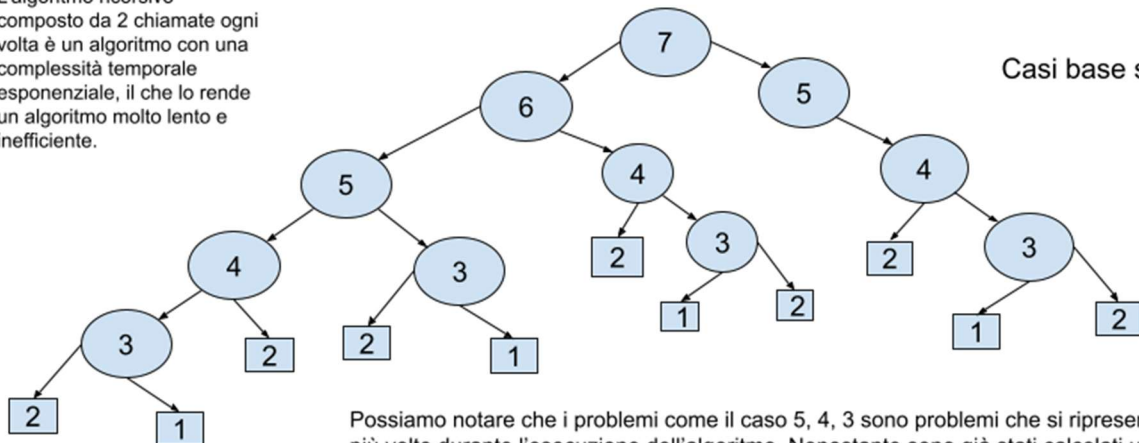
RETURN FIB(n-1) + FIB(n-2)

L'algoritmo ricorsivo rispetto all'algoritmo iterativo ha una complessità temporale esponenziale ma il problema non è dovuto alla ricorsione in sé ma a come è strutturato l'algoritmo.

ALBERO DI RICORSIONE - FIBONACCI

L'algoritmo ricorsivo composto da 2 chiamate ogni volta è un algoritmo con una complessità temporale esponenziale, il che lo rende un algoritmo molto lento e inefficiente.

Casi base sono 1 e 2



Possiamo notare che i problemi come il caso 5, 4, 3 sono problemi che si ripresentano più volte durante l'esecuzione dell'algoritmo. Nonostante sono già stati calcolati una volta, verranno calcolati nuovamente tutte le volte che appariranno.

Per risolvere questo problema memorizziamo il risultato delle operazioni già svolte, e considereremo tutti i casi già svolti proprio come se fossero tanti casi base dato che li conosciamo perché già calcolati una volta.

Cerchiamo di rendere più efficiente questo algoritmo ricorsivo memorizzando ogni volta i vari risultati che riusciamo a crearci. **In questo modo il compilatore non dovrà eseguire il problema 3 ad esempio 4 volte, ma semplicemente una volta sola.** Per fare questo dobbiamo trattare ogni sottoproblema come un caso base, dando per scontato che abbiamo già tale risultato.

Riscriviamo allora il PSEUDO codice per trovare un algoritmo ricorsivo molto più efficiente, tenendo in considerazione i vari risultati.

FIB(n)

If $n \leq 2$ THEN RETURN 1

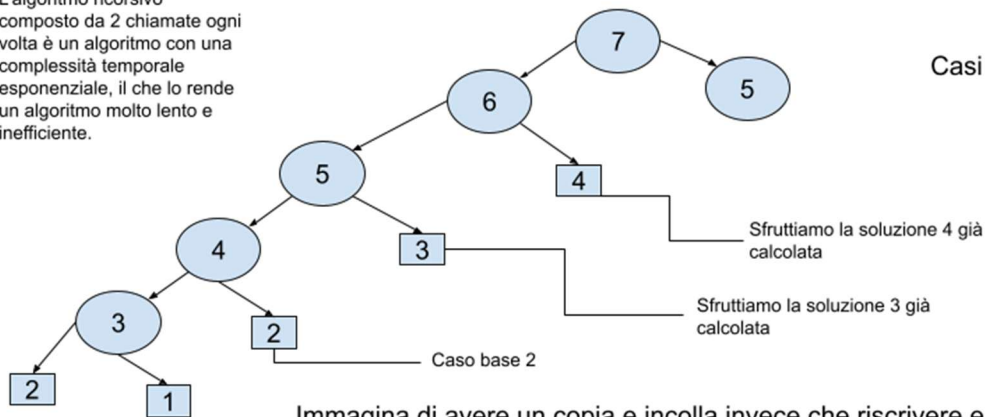
IF F[n-1] = NIL THEN F[n-1] = FIB[n-1]

//potremo fare un secondo if per n-2 ma non lo faremo perché non c'è bisogno perché per calcolare 6 ho bisogno di 5 e di 4 quindi non ha senso fare un altro if per n-2.

RETURN F[n-1] + F[n-2]

ALBERO DI RICORSIONE - FIBONACCI

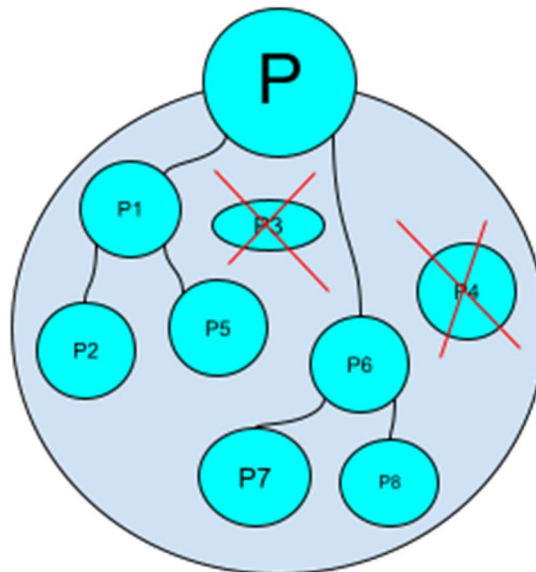
L'algoritmo ricorsivo composto da 2 chiamate ogni volta è un algoritmo con una complessità temporale esponenziale, il che lo rende un algoritmo molto lento e inefficiente.



Immagina di avere un copia e incolla invece che riscrivere e calcolare le soluzioni. La complessità ora non è più esponenziale ma lineare, una complessità simile a quella iterativa.

Ecco un concetto fondamentale degli algoritmi ricorsivi

Problema Grande



Nella soluzione iterativa vengono esplorati tutti i sottoproblemi. Nella ricorsione al contrario vengono guardate solo alcune soluzioni delle varie operazioni.

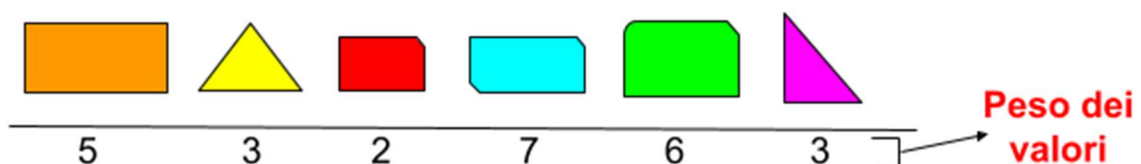
Problema dello zaino

Caso 1

Immagina di essere un ladro, che va a rubare con un solo zaino (che ladro scarso/nabbo). Questo zaino non lo possiamo riempire troppo perché si rompe (quindi non solo scarso pure povero). **Adesso entra in questa gioielleria strana, in cui tutti gli oggetti hanno lo stesso valore, il nostro scopo da ladri è prendere più oggetti possibili.**

Quindi dato che lo zaino se riempito troppo si rompe, diciamo che può sopportare un peso massimo di 10 kili, e noi dobbiamo riuscire a prendere più oggetti possibili.

Iniziamo a prendere oggetti con il peso minore, per farlo ordiniamo tutti gli oggetti a nostra disposizione, studiamoci un algoritmo che ci permetta di portare più oggetti possibili ricordando la definizione iniziale che ogni oggetto ha lo stesso valore ma peso diverso.



Ordiniamo i pesi in maniera crescente in modo da prendere prima tutti quelli con il peso più basso. Questo perché sappiamo che tutti gli oggetti hanno gli stessi valori, ma cambiano solo di peso.



$Z=(k, i) \rightarrow$ Studiamo un algoritmo ricorsivo partendo sempre dal caso base, o dai casi base.

$$Z(k, n) = \begin{cases} \emptyset & i > n \\ \emptyset & p[i] > k \\ 1 + Z(k - p[i], i + 1) & \text{otherwise} \end{cases}$$

- $\emptyset \quad i > n \rightarrow$ non posso prendere più nessun elemento perché già sono stati presi tutti e quindi l'indice ha superato il numero di elementi prendibili. In pratica tutti gli elementi sono entrati nello zaino.
- $\emptyset \quad p[i] > k \rightarrow$ quel peso di quell'oggetto è maggiore alla capienza dello zaino.
- $1 + Z(k - p[i], i + 1) \rightarrow$ Si divide in 2 parti, prendo l'oggetto e lo metto nello zaino allora la capienza dello zaino stesso diminuirà perché c'è meno spazio ecco a cosa serve $k - p[i]$ però ora dobbiamo controllare gli altri elementi e allora andiamo avanti con l'indice puntando al prossimo elemento.

Seguendo questa logica dato che tutti gli elementi hanno lo stesso valore e cambia solo il peso, quello che faremo è prendere l'oggetto con il peso sempre più piccolo tra quelli rimasti.

Caso 2

Adesso ogni oggetto ha un valore proporzionale al suo peso. Pesa 2kg e allora varrà 2euro.

Durante la lezione la logica di un ragazzo è **stato prendere l'oggetto con il peso maggiore** ma ovviamente minore alla capienza dello zaino, **successivamente prendere l'oggetto leggermente minore e se ci entra metterlo nello zaino**. Inizialmente potrebbe essere una buona idea ma effettivamente non lo è.

Facciamo sempre finta che lo **zaino sopporta un massimo di 10kg e ho 4 oggetti: 9, 5, 5, 2**. **Con questa logica io dovrei prendere l'oggetto che pesa e vale di più quindi l'oggetto 9**, ma non è la soluzione migliore perché in questo modo lascio vuoto 1 kg che non uso nello zaino, **La soluzione migliore in questo caso è scegliere i due 5, ma per farlo io devo conoscere tutte le varie combinazioni tra gli oggetti**.

Allora invece di trovare una soluzione totale pensiamo sempre ricorsivamente, la vera domanda è questo oggetto lo prendo oppure no? Ci limitiamo a decidere per un problema più piccolo.

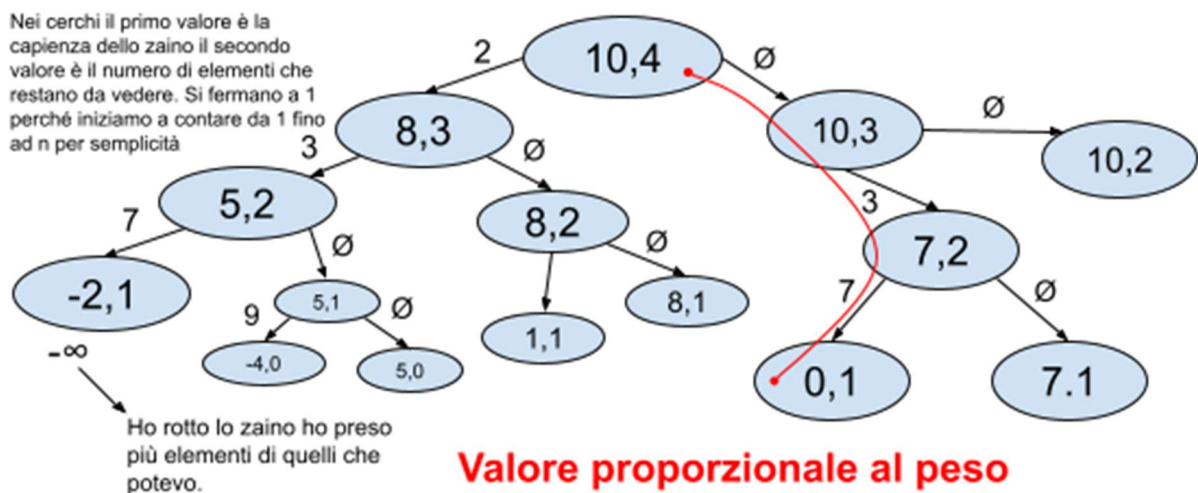
$Z = (k, n)$ l'unica domanda a cui dobbiamo rispondere è l'oggetto $P[n]$ lo prendo oppure no?

- **$P[n] \rightarrow$ Si: allora $Z = (k - P[n], n - 1)$** . // decrementiamo la capienza dello zaino e andiamo avanti.
- **$P[n] \rightarrow$ Si: allora $Z = (k, n - 1)$** . // Andiamo avanti ma non decrementiamo la capienza dello zaino.

$$Z(k, n) = \begin{cases} 0 & \text{se } n = 0 \\ -\infty & k < 0 \\ \max(P[n] + Z(k - P[n], n - 1), Z(k, n - 1)) & \end{cases}$$

- $0 \quad \text{se } n = 0 \rightarrow$ se $n = 0$ allora 0 primo caso base
- $-\infty \quad k < 0 \rightarrow$ se k minore di 0 vuol dire che abbiamo caricato lo zaino più del dovuto sforzando il peso massimo trasportabile che è k , allora diamo il valore $-\infty$ che sarebbe un valore non accettato, un errore.
- $\max(P[n] + Z(k - P[n], n - 1), Z(k, n - 1)) \rightarrow$ **Quello che faremo e trovare il massimo valore peso raggiunto restando sempre minori di k . Le strade possono essere 2 o scegliamo un oggetto oppure no, se scegliamo un oggetto sommiamo $P[n]$ il valore di quell'oggetto, decrementiamo la capienza dello zaino e decrementiamo il numero di elementi da analizzare**, cioè n che sarebbe il nostro indice. Se non scegliamo il valore non decrementiamo la capienza dello zaino ma decrementiamo comunque l'indice n per poter passare al prossimo elemento. Ovviamente nel caso in cui NON scegliamo l'oggetto non lo sommeremo a $P[n]$.

Albero di ricorsione di questo caso 2, del problema dello zaino.



In questo caso appena descritto abbiamo sommato il peso degli oggetti ma se invece volessimo sommare il valore degli oggetti, il procedimento e la logica sarebbero perfettamente uguali ci sarà solo l'aggiunta di un nuovo caso base.

$$\begin{cases} 0 & n = 0 \\ 0 & k = 0, \text{ terzo caso base} \\ -\infty & k < 0 \\ \text{MAX}(V[n] + Z(k - p[n], n - 1), Z(k, n - 1)) \end{cases}$$

Nota: ho inglobato caso 2 e 3 perché su word il massimo sono 3 righe dentro la parentesi graffa, quindi ci adattiamo. Comunque, studiamo questa regola:

- $0 \quad n = 0 \rightarrow$ Facile, se $n = 0$ vuol dire che non ci sono elementi da prendere.
- $0 \quad k = 0 \rightarrow$ Se $k=0$ vuol dire che lo zaino è completamente occupato e quindi non abbiamo spazio libero.
- $-\infty \quad k < 0 \rightarrow$ Se sforiamo il peso dello zaino vuol dire che lo abbiamo rotto, e quindi il risultato delle scelte non è accettabile, poniamo tale risultato come $-\infty$

Questi sono i 3 casi base, passiamo al caso NON di base, ma costruito su di essi.

- **$\text{MAX}(V[n] + Z(k - p[n], n - 1), Z(k, n - 1)) \rightarrow$ Sommiamo non più i pesi ma i valori degli oggetti che mettiamo dentro lo zaino, scegliendo sempre il massimo tra tutte le scelte che possiamo fare.** Analogamente al caso del peso, se scegliamo di prendere l'oggetto decrementiamo la capacità dello zaino e decrementiamo il puntatore andando al prossimo elemento, alla fine sommiamo questo valore preso e messo nello zaino. **Se invece non scegliamo di portare l'oggetto, non decrementiamo la capacità dello zaino ma decrementiamo l'indice per puntare al prossimo elemento.**

Equazione di Ricorrenza

#One-Note-3

Le equazioni di ricorrenza sono lo strumento più importante per studiare la complessità degli algoritmi ricorsivi.

"Una funzione ricorsiva è una formula che descrive i valori di una funzione in corrispondenza di un argomento x in termini del valore della funzione su argomenti di misura in genere più piccole." → Spiegazione fatta dalla professoressa Caterina Viola, coo-professoressa del professore Simone Faro di Algoritmi nel corso di informatica.

Diamo una nostra spiegazione più dettagliata.

Quando siamo dinanzi ad un algoritmo ricorsivo, quest'ultimo farà una serie di chiamate a sé stessa, il suo tempo di esecuzione potrà essere descritto tramite un **EQUAZIONE DI RICORRENZA**. **L'equazione di ricorrenza calcola il tempo di esecuzione totale di un problema in funzione del tempo di esecuzione degli input più piccoli.** Adesso che abbiamo fatto queste assunzioni entriamo più nello specifico, supponiamo che **$T(n)$** , sia il tempo di esecuzione di un algoritmo. Quando studieremo il tempo di esecuzione di un algoritmo dobbiamo dividere i casi base dai casi NON base.

- **I casi base sono quei casi che hanno una risoluzione lineare**, e sono il problema più piccolo che siamo riusciti a creare. Questa come già detto è la soluzione di base sui cui verranno creati tutti gli altri problemi più complessi.
- **Ipotizziamo di aver suddiviso il problema più grande in "a" sottoproblemi, ogni sottoproblema ha un tempo di risoluzione di $T(n/b)$.** Allora per calcolare il tempo dei sottoproblemi "a", faremo **$a * T(n/b)$** . Adesso dobbiamo sommare a questo tempo **$a * T(n/b)$** il **tempo per la divisione del problema in sottoproblemi, questo tempo lo chiameremo $D(n)$** ed **un tempo usato per combinare tutte le mini-soluzioni tra di loro che sarà $C(n)$** . Adesso scriviamo tutte queste informazioni sotto forma di regole effettive.

$$T(n) = \begin{cases} O(1) & \text{se } n \leq n_0 \\ a * T\left(\frac{n}{b}\right) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

- $se\ n \leq n_0 \rightarrow$ è la condizione per cui il caso base è verificato
- *divide* $\rightarrow D(n)$ tempo per la divisione dei vari sottoproblemi
- *conquer* \rightarrow tempo per a sottoproblemi considerando che ogni sottoproblema ha tempo $T\left(\frac{n}{b}\right)$
- *combine* \rightarrow il tempo per combinare/unire tutti i sottoproblemi.

Per analizzare la complessità di un algoritmo possiamo usare vari metodi che ci permetteranno di creare e risolvere un'equazione di ricorrenza, associata all'algoritmo ricorsivo.

Esistono 3 metodi principali, che vedremo durante questo corso:

- Metodo di sostituzione
- Metodo dell'albero di ricorsione
- Metodo Master

Adesso con le nostre conoscenze analizziamo il Merge Sort.

Analisi del Merge Sort

Analisi del Merge Sort con il metodo dell'albero di Ricorsione.

Metodo dell'alberello pazzo sgravato, anche chiamato **metodo dell'albero di ricorsione**

Noi vogliamo analizzare il caso peggiore del Merge Sort. Il merge sort divide l'array in 2 sotto array, ripetendo questa divisione ricorsivamente anche per tutti gli array generati. Alla fine di tutto questo processo avremo un caso base che sarà appunto l'array composto da un solo elemento. Per il caso base in cui $n = 1$ avremo tempo costante quindi $O(1)$. Adesso facciamo specificiamo i vari momenti del Merge Sort.

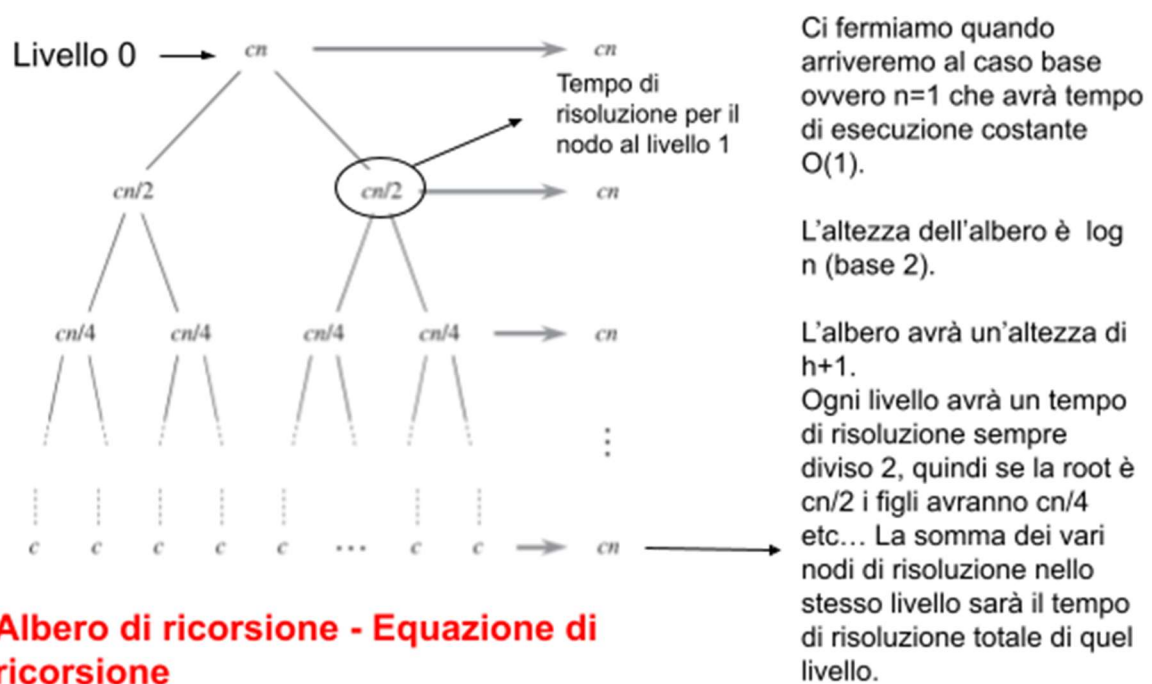
- **Divide:** durante questo passo calcoliamo solo il centro dell'array. Questa operazione sarà sempre 1 indipendente dal numero di elementi, quindi avrà tempo costante. $O(1)$. Calcola la metà dell'array $(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil)$: $\rightarrow D(n) = \Theta(1)$
- **Impera:** dopo aver diviso l'array a metà avremo due sotto array di dimensione $n/2$. Lavorare sui 2 array contribuirà con un tempo di $2T(n/2)$. Con $a=2$ problemi di dimensione $n/2$.
- **Combina:** La combinazione dei vari risultati dato che deve passare per ogni cella dell'array ci starà tempo lineare quindi $O(n)$.

$$T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad \forall n \geq n_0$$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \text{ Caso base} \\ \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n \geq 2 \end{cases}$$

$\Theta(1)$ è un dato anche trascurabile. Possiamo non scriverlo

Per risolvere questa equazione applicheremo l'albero di ricorsione.



Spieghiamo tutto senza creare confusione. **Questo è l'albero di ricorsione che abbiamo creato simulando il funzionamento del Merge Sort.** L'albero ha inizio da un Heap, da una radice come la vogliamo chiamare la chiamiamo che sarebbe l'array nella sua interezza, ad ogni passo questo array verrà diviso a metà ottenendo prima 2 sotto array, ricorsivamente divideremo anche questi due sotto array a metà ottenendo 4 sotto array, ricorsivamente divideremo questi 4 sotto array a metà ottenendo 8 sotto array etc... fino a quando ogni sotto array avrà un solo elemento. **Per accedere ad ogni elemento dell'array nel caso iniziale in cui questo è intero, avremo tempo lineare quindi $T(n)$,** nell'immagine sopra è indicato da cn . Dividendo questo array a metà avrò due array con circa la metà elementi dell'array iniziale quindi avrò un tempo di $T(n/2)$, nell'immagine sarebbe $cn/2$. **Andando avanti di un altro livello ogni array avrà una dimensione di ancora metà del precedente; quindi, avremo per ogni sotto array al secondo livello $T(n/4)$** (i livelli si iniziano a contare da 0). **Questo processo reiterato ricorsivamente fino a quando non otterremo un tempo di accesso ad ogni sotto array che sarà di tempo costante ovvero $\theta(1)$.**

Adesso proviamo a sommare per ogni livello il costo dei propri array. Nel livello 0 sappiamo già essere cn . Nel primo livello facciamo $(cn/2 + (cn/2) = cn$. Nel secondo livello facciamo $(cn/4) + (cn/4) + (cn/4) + (cn/4) = cn$. (Sarebbe come dire $4 * c(n/4) = cn$).

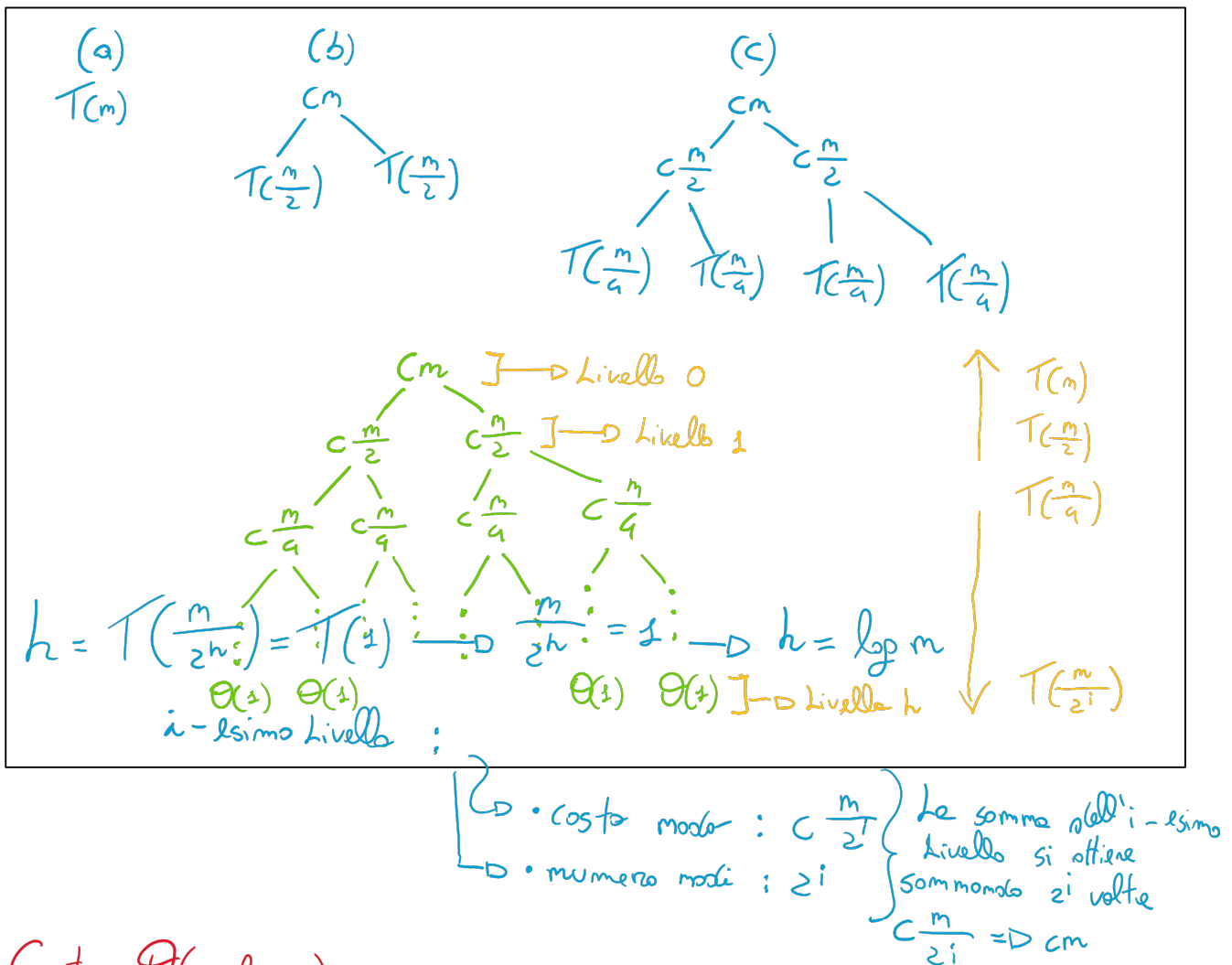
Ogni livello ha un costo complessivo di cn . Trodiciamo queste somme in moltiplicazioni e invece di mettere $\frac{cn}{\text{'un numero'}}$, metteremo $\frac{cn}{2^i}$.

$$\text{Allora avremo che } 2^i * c \left(\frac{n}{2^i} \right) = cn$$

Qual è il costo di tutto questo procedimento che sembra super complicato ma non lo è?

Considerando che ci sono $n+1$ livelli dato che n sono i livelli con foglie e l'ultimo è il livello delle foglie avremo che $cn (\log n - 1) \rightarrow cn \log n - cn$. **Possiamo allora dire che la complessità di questo algoritmo è in realtà il risultato dell'equazione di ricorrenza è $\Theta(n \log n)$ sia nel caso PESSIMO sia nel caso MIGLIORE.**

Definizione della professoressa Viola:



Costo $\Theta(m \log m)$

- $\log m \rightarrow$ costo della Divisione che ha a che fare con l'altezza dell'albero.
- $m \rightarrow$ costo per combinare tutti i sottoproblemi tra di loro.

- In un albero di ricorsione ogni nodo rappresenta una parte del costo del sottoproblema. Si sommano i costi dei singoli nodi ad ogni livello ed il risultato sarà sempre cn .

- Ogni sottoalbero corrisponde ad un sottoproblema
- Le foglie rappresentano le istanze del caso base e per risolvere un'equazione di ricorrenza con questo metodo dell'albero di ricorsione si sommano i costi di ogni livello.

Analisi del Merge Sort con il metodo di sostituzione

Il metodo di sostituzione per funzionare ha bisogno di 2 passi fondamentali:

1. Supporre la soluzione
2. Induzione matematica → che ha lo scopo di trovare le costanti e verificare che la soluzione che abbiamo dedotto prima funziona.

Il metodo di sostituzione è indubbiamente un metodo molto potente per risolvere equazioni di ricorsione ma non è proprio possibile applicarlo sempre; infatti, necessitiamo che la soluzione possa essere dedotta in maniera un po' più semplice.

Facciamo un esempio:

$$T(n) = 2T([n/2]) + \theta(n)$$

Riconosci questa equazione di ricorrenza è proprio quella del merge sort...

Supponiamo la soluzione pensando che sia **$O(n \log n)$** $\forall n \geq n_0$.

Avremo un caso base che ha un tempo di risoluzione $n < n_0$, adesso dobbiamo verificare che la soluzione ipotizzata sia corretta e allora applichiamo l'induzione.

Ipotesi Induttiva: l'induzione ci permette di capire se la nostra logica funziona per il passaggio successivo. Se funziona per n e funziona per $n/2$ nel nostro caso allora funzionerà per tutti gli altri casi. Quindi consideriamo il prossimo passaggio che dato che stiamo dividendo sempre per 2 sarà la prima divisione per 2. Tutti gli n diventeranno $n/2$.

$T(n) \leq cn \log n \rightarrow T(n/2) \leq c(n/2) \log (n/2) \rightarrow T(n/2)$ non ci piace moltiplichiamo tutto per 2.

$$T(n) \leq 2(c[n/2] * \log ([n/2])) + n$$

$$Tn \leq cn \log (n/2) + n$$

//equazione associata e risolviamo $cn \log (n/2) + n$ ottenendo

$$= cn \log n - cn \log 2 + n$$

$$= cn \log n - cn + n$$

// risultato dell'equazione sarà

$$T(n) \leq cn \log n$$

C è un valore > 0

Adesso troviamo i casi base sostituendo ad n il valore dei casi base che abbiamo supposto essere 0.

- Se prendo $n = 0$ avrò $c \cdot 0 \cdot \log 0$, $\log 0$ non esiste quindi questo non potrà essere un valore utile.
- Se prendo $n = 1$ avrò $c \cdot 1 \cdot \log 1 = 0$. Allora n sarà valido per valori ≥ 2

Allora diremo che $n_0 \geq 2$ la formula ha senso di esistere e può essere applicata correttamente. Mentre il nostro caso base è $n_0 = 1$, non consideriamo 0 perché quel caso nemmeno è da prendere in considerazione dato che non è utile per nulla.

Viola:

Esempio (a): $T(m) = 2T(\lfloor \frac{m}{2} \rfloor) + \theta(m)$

guess $\rightarrow T(m) = O(m \log m) \rightarrow T(m) \leq cm \log m$

(1) Caso base: $m < m_0$ $T(m) = \theta(1)$ [Caso base sempre tempo costante]

(2) Ipotesi induttiva: assumiamo che
 $T(\lfloor \frac{m}{2} \rfloor) \leq c \lfloor \frac{m}{2} \rfloor \log(\lfloor \frac{m}{2} \rfloor)$
Con $m \geq m_0$, $\frac{m}{2} \geq m_0$

Sostituiamo

$$\begin{aligned} T(m) &= 2T(\lfloor \frac{m}{2} \rfloor) + \alpha m \leq \\ &2(c \lfloor \frac{m}{2} \rfloor \log \lfloor \frac{m}{2} \rfloor) + \alpha m \leq \\ &2c \frac{m}{2} \log \frac{m}{2} + \alpha m = cm (\log m - \log 2) + \alpha m = \\ &= cm \log m - cm + \alpha m \stackrel{?}{\leq} cm \log m \end{aligned}$$

\nwarrow
per $c \geq \alpha$

Caso Base $\rightarrow m_0 = 2$

(base dell'induzione): $T(m) \leq cm \log m$

$$\left. \begin{array}{l} m \geq m_0 \\ \frac{m}{2} \leq m_0 \end{array} \right\} \rightarrow m_0 \leq m < 2m_0$$

$$m_0 = 2 \rightarrow 2 < m < 4 \rightarrow m = 2, m = 3$$

$$c = \max \{T(2), T(3)\}$$

$$\bullet T(2) \leq c \leq c \cdot 2 \log 2 \quad \checkmark$$

$$\bullet T(3) \leq c \leq c \cdot 3 \log 3 \quad \checkmark$$

TRICK OF THE TRADE

Es: (c)

$$T(m) = 2T\left(\frac{m}{2}\right) + \Theta(1)$$

• guess $\rightarrow T(m) = O(m) \rightarrow T(m) \leq cm - d$

• Sostituzione

$$T(m) \leq 2T\left(\frac{m}{2}\right) + d \leq 2c \frac{m}{2} + d = cm + d \leq cm$$

$$T(m) \leq 2T\left(\frac{m}{2}\right) + d \leq 2\left(c \frac{m}{2} - d\right) + d =$$

$$cm - 2d + d =$$

$$cm - d \leq cm$$

Es (d): $T(m) = 2T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + \Theta(m)$

guess: $T(m) = O(m) \rightarrow T(m) \leq cm$

Sostituiamo $\rightarrow T(m) \leq 2T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + \Theta(m) \leq 2O\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + \Theta(m)$

$$= O(m) + \Theta(m) = O(m)$$

Sost $\rightarrow T(m) \leq 2T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + \Theta(m) \leq 2c \left\lfloor \frac{m}{2} \right\rfloor + \Theta(m) \leq cm + \Theta(m)$
 $= O(m)$

La coda con priorità

Questo che spiegheremo adesso è veramente un bellissimo argomento che è stato spiegato in modo magistrale dal professore Simone Faro. Questo argomento leggermente più lungo del precedente ci farà capire moltissime cose sull'uso di alberi etc...

La struttura dati che vedremo viene dall'era del BOOM dell'informatica ovvero anni 80/90. **La coda con priorità è una struttura dati che va immaginata in maniera astratta come un albero** ma che **in realtà in maniera fisica è molto più simile ad un array con determinate caratteristiche**, ma su questo concetto ci ritorneremo più tardi.

Per capire la coda con priorità immaginiamola in maniera astratta come un albero. Questo albero o HEAP ha delle caratteristiche particolari e molto interessanti.

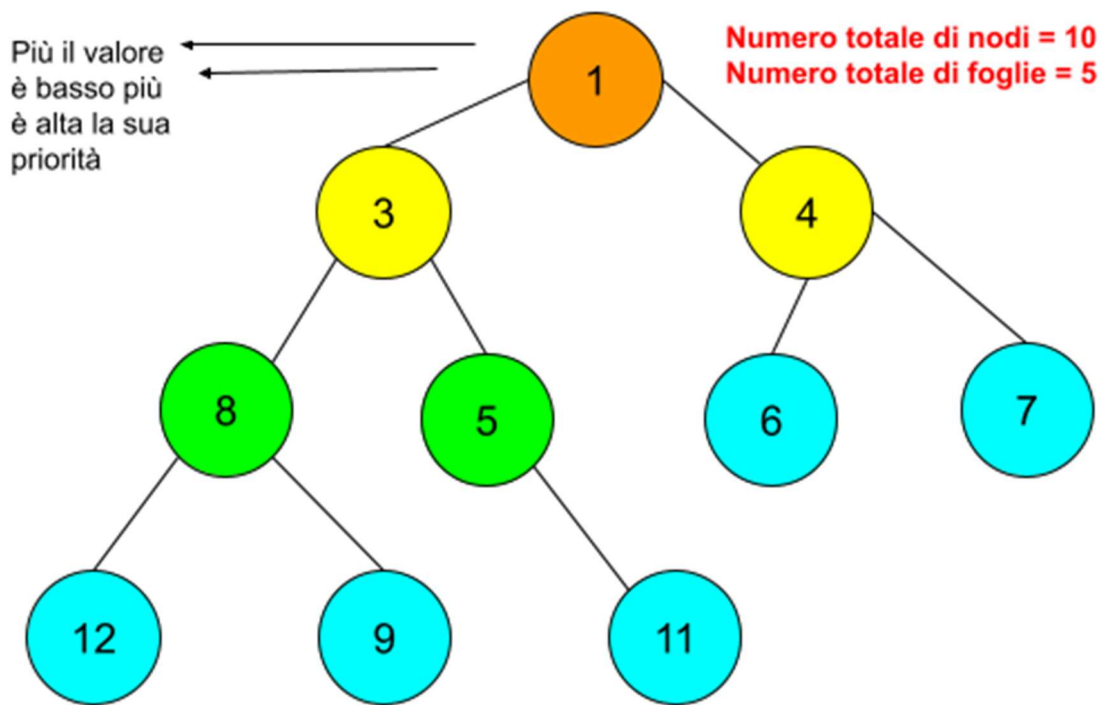
L'HEAP che caratterizza la coda con priorità è un albero binario, posizionale completo. Ma cosa vogliono dire questi nomi:

- **Un albero binario è un albero che al massimo ha due figli per ogni nodo.** Un nodo padre potrà quindi avere massimo 2 figli.
- **Un albero posizionale è un albero che dà valore alla posizione dei figli.** In un albero **non posizionale non importa quale figlio viene disegnato prima perché un padre con un figlio avrà comunque un solo figlio** indipendentemente che lo disegniamo a destra o a sinistra. **Un albero posizionale invece no, un genitore con un figlio a sinistra è diverso da un genitore con un figlio a destra proprio perché viene dato valore alla posizione con cui vengono disegnati i figli.**
- **Un albero completo è un albero che possiamo definire simmetrico alla root,** quindi alla radice. **È un albero con tutti i livelli completi di nodi fatta ad eccezione dell'ultimo livello composto da foglie.**

Unendo tutte le 3 caratteristiche possiamo dedurre che un albero binario, posizionale completo **è un albero il quale ogni nodo può avere al massimo 2 figli, è un albero completo fatta ad eccezione dell'ultimo livello che è composto dalla foglie** (l'unico strato che potrà anche non essere completo) **e tale albero è anche posizionale ovvero viene dato valore alla posizione in cui vengono disegnati i nodi**, motivo per cui ogni nodo verrà scritto da sinistra verso destra, le foglie quindi che saranno la cosa più visibile di questa logica saranno da disegnate da sinistra verso destra, possiamo allora dedurre che sarà IMPOSSIBILE avere un nodo genitore solo con un figlio destro, al massimo avrà avere solo un figlio sinistro e MAI SOLO destro.

Adesso dopo aver compreso le caratteristiche dell'Heap che rappresenta in maniera astratta la nostra coda con priorità, possiamo avere due tipi di Heap.

- **Min-Heap:** **più il valore è basso più la priorità è alta.** Un esempio sono le nostre borse di studio dell'università, più l'isee è basso più è alta la priorità nel prendere la borsa di studio.
- **Max-Heap:** **più il valore è alto più la priorità è alta.** Un esempio potrebbe essere le agevolazione per anziani, più sei anziano più è alta la tua priorità di accedere all'agevolazione. Sveglia mio bis-bis-bis-nonno e continuiamo.



Questo è un esempio di **Min-Heap**.
 12, 9, 11, 6, 7 sono foglie. Le foglie in un albero binario
 corrispondono SEMPRE alla metà del numero di tutti i nodi

Cambiando l'ordine di alcuni nodi senza mutare la logica del Min-Heap otterremo risultati diversi. Ad esempio, invertendo di posizione il 6 con il 7 otterremo un risultato diverso dal precedente, nonostante sia il tutto ancora un Min-Heap. **Si dice quindi che è un ordinamento parziale, in cui possono esistere più soluzioni e configurazioni valide.** [Cosa che non avviene nell'albero di ricerca, in cui si avrà sempre e solo una singola configurazione per gli elementi dell'albero]