Costo computazionale

- Tempo di esecuzione
- Necessità di memoria

Il tasso di crescita del tempo che impieghiamo nell'algoritmo si divide in tre notazioni

- Worst case O()
    - abbiamo f(n) = 3n e il suo O(f(n)) = O(n) perchè abbiamo una c >= 3 tale che ci ritroviamo con un valore maggiore da n0 in poi "n >= n0"
      n <= O(n) o n <= c*n per un n >= n0 e con una c >= 3

- Best case Ω()
    - abbiamo f(n) = 3n e il suo Ω(f(n)) = Ω(n) perchè abbiamo una c <= 3 tale che ci ritroviamo con un valore inferiore a f(n) da n0 in poi "n >= n0 "
      n >= Ω(n) o n >= c*n per un n >= n0 e con una c <= 3

- Average case Θ()
    - il caso medio ci permette di dimostrare che senza cambiare quello che c'è all'interno di Θ(un "generico") possiamo aggiunge anche Ω() e O() quindi sappiamo che esiste una c' tale che f(n) <= c' * n ed esiste una c'' tale che f (n) >= c'' * n e questo vale per un certo punto n0 abbastanza grande.

Proprietà e dimostrazione dell' algebra della notazione asintotica

1. **∀ k > 0 se f(n) → O(g(n)) allora anche k*f(n) → O(g(n)).**
   Dim
    - sappiamo che esistono delle c tali che f(n) <= c*g(n) ∀ n > n0
      allora k*f(n) <= k*c*g(n) == k*f(n) <= c'*g(n) k*f(n) <= k*c*g(n) e c' = c*k

2. **∀ f(n),d(n) > 0 se f(n) → g(n) e d(n) → h(n) allora**
   f(n) + d(n) → (O(g(n) + h(n)) = O(max(g(n), h(n)))
   Dim
    - sappiamo che abbiamo quattro costanti c', c'', n0' e n0'' tali che
      f(n) <= c'*g(n) ∀ n > n0' e d(n) <= c''*h(n) ∀ n > n0'' ed allora sappiamo che
      f(n) + d(n) <= c'*g(n) + c''*h(n) <= max(c', c'') * (g(n) + h(n)) ∀ n > max(n', n'')
      cioè f(n) + d(n) <= c * (g(n) + h(n)) ∀ n > n0 == f(n) + d(n) → O(g(n) + h(n))
      e per le proprietà della notazione asintotica sappiamo che dobbiamo prendere solo il massimo tra la somma di funzioni quindi
      f(n) + d(n) → O(max (g(n), h(n)))

3. **∀ f(n),d(n) > 0 se f(n) → g(n) e d(n) → h(n) allora**
   f(n) * d(n) → O(g(n)*h(n))
   Dim
    - sappiamo che abbiamo quattro costanti c', c'', n0' e n0'' tali che
      f(n) <= c'*g(n) ∀ n > n0' e d(n) <= c''*h(n) ∀ n > n0'' ed allora sappiamo che
      f(n) * d(n) <= (c' * c'')*(g(n) * h(n)) ∀ n >= max(n0', no'') allora
      f(n) * d(n) → O(g(n) * h(n)) con la c = c' * c'' e n = max(n0', no'')

Verifica della notazione asintotica

Attraverso i limiti possiamo capire se la nostra notazione è corretta o sbagliata e questa si può verificare per i diversi casi che può verificare un limite → + ∞ di f(n) → O/Ω/Θ (g(n))

1. lim n → + ∞ di f(n)/ g(n) = n ∈ ℝ allora f(n) → Θ (g(n)) ovviamente anche O & Ω
2. lim n → + ∞ di f(n)/ g(n) = 0 allora g(n) → + ∞ più velocemente di f(n) f(n) → O(g(n))
3. lim n → + ∞ di f(n)/ g(n) = 0 allora f(n) → + ∞ più velocemente di g(n) f(n) → Ω(g(n))

Sommatoria:

1. i=0 → n di ∑ i = n *((n+1)/2) = Θ(n^2)
2. i=x → n di ∑ c^i = (c^(n+1) - c^x) / (c - 1) = Θ(c^n)

Calcolo costo Computazionale

1. I for rispettano il range o il numero di elementi in un array/qualsiasi altra cosa.
2. I while hanno diverse modalità e condizioni come while n > 1(o qualsiasi numero) :
    a. n // x con x un qualsiasi numero O(log(x) n)
    b. n - x con x un qualsiasi numero O(n)
    c. i = 1 i*i >= n con i +=1 allora abbiamo O(n^(½))
3. I for/while con for/while annidati, dobbiamo calcolare le informazioni e poi moltiplicare il tutto
4. In caso di un caso migliore e peggiore, prendiamo innanzitutto quello maggiore e quello che si ripete al crescere di n sennò poniamo c come il max.

Ricorsione

La ricorsione è ispirata dalle funzioni ricorsive, e serve per risolvere un problema in un modo naturale come ad esempio trovare il fattoriale di n cioè n * (n-1)!, quest'ultimo avrà come caso baso n == 0 return 1.

Pro della ricorsione:

- la soluzione è più naturale comparata con la soluzione iterativa.

Contro della ricorsione:

- Utilizza molta memoria perché si deve ricordare i casi precedenti .
- Non è scalabile all’infinito a causa del grande utilizzo della memoria.

Calcolo costo computazionale per la ricorsione

1. **Metodo Iterativo:**
    - Sostituiamo all'interno della f(n) fino a quando non arriviamo a T(1)
    - T(n) = T(n-1) + Θ(1) con T(1) = Θ(1)
    - T(n) = [T(n-2) + Θ(1)] + Θ(1) ... T(n) = [T(n-3) + Θ(1)] +Θ(1) +Θ(1) ……
    - da qui possiamo dire che dobbiamo trovare un k tale che n-k = 1 e con un paio di semplificazioni arriviamo che k = n-1 e questo è il limite della sommatoria ed infine ad ogni T(n) abbiamo un +Θ(1) quindi la nostra sommatoria sarà i=0 → n-1 ∑+Θ(1) che sarebbe Θ(n) - Θ(1)

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

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)\right] + \Theta(n)$$

$$T(n) = 2\left[2\left[2T\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right)\right] + \Theta\left(\frac{n}{2}\right)\right] + \Theta(n)$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \frac{n}{2^i} \cdot 2^i \qquad k = \log_2 n$$

$$T(n) = n \cdot \Theta(1) + \sum_{i=0}^{\log n - 1} n$$

$$T(n) = \Theta(n) + n + n + \dots + n$$

$$T(n) = \Theta(\log_2 n \cdot n) \equiv (\log_2 n^n)$$

2. **Metodo di Sostituzione:**
    - È consigliata al fine di dimostrare la veridicità di un costo computazionale.
    - T(n) = T(n-1) + c T(1) = d
    - Il nostro Principale obiettivo è quello di verificare il Ω e O, al fine di poter dire che T(n) <= k*n (O) e T(n) <= h*n (Ω) e in tutto questo avremmo un T(1) = d tale che per (O) d <= k e per (Ω) d >= h.
    - Andiamo a vederlo per O(n) della T(n) sopra, sappiamo che la k >= d, e che T(n) = k*(n-1) + c e che T(n) <= k*n, quindi k*n - k + c <= kn == k >= c. quindi sappiamo che T(n) = O(n) se e solo se [k >= c] e [k >= d].
    - Questo vale anche per Ω(n) ma con k*n <= T(n).

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

$T(n) = O(n \cdot \log n) \equiv T(n) \leq k \cdot (n \cdot \log n)$

$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \qquad T(1) = d$

① Caso Base $n = 1$

$T(1) \leq k \cdot 1 \cdot \overset{=0}{\log_2 1} \equiv d \leq 0$ FALSO

$d > 0$ per definizione

$O(n \cdot \log n) = k \cdot n \log n + hn$

$d \leq h$ ok!

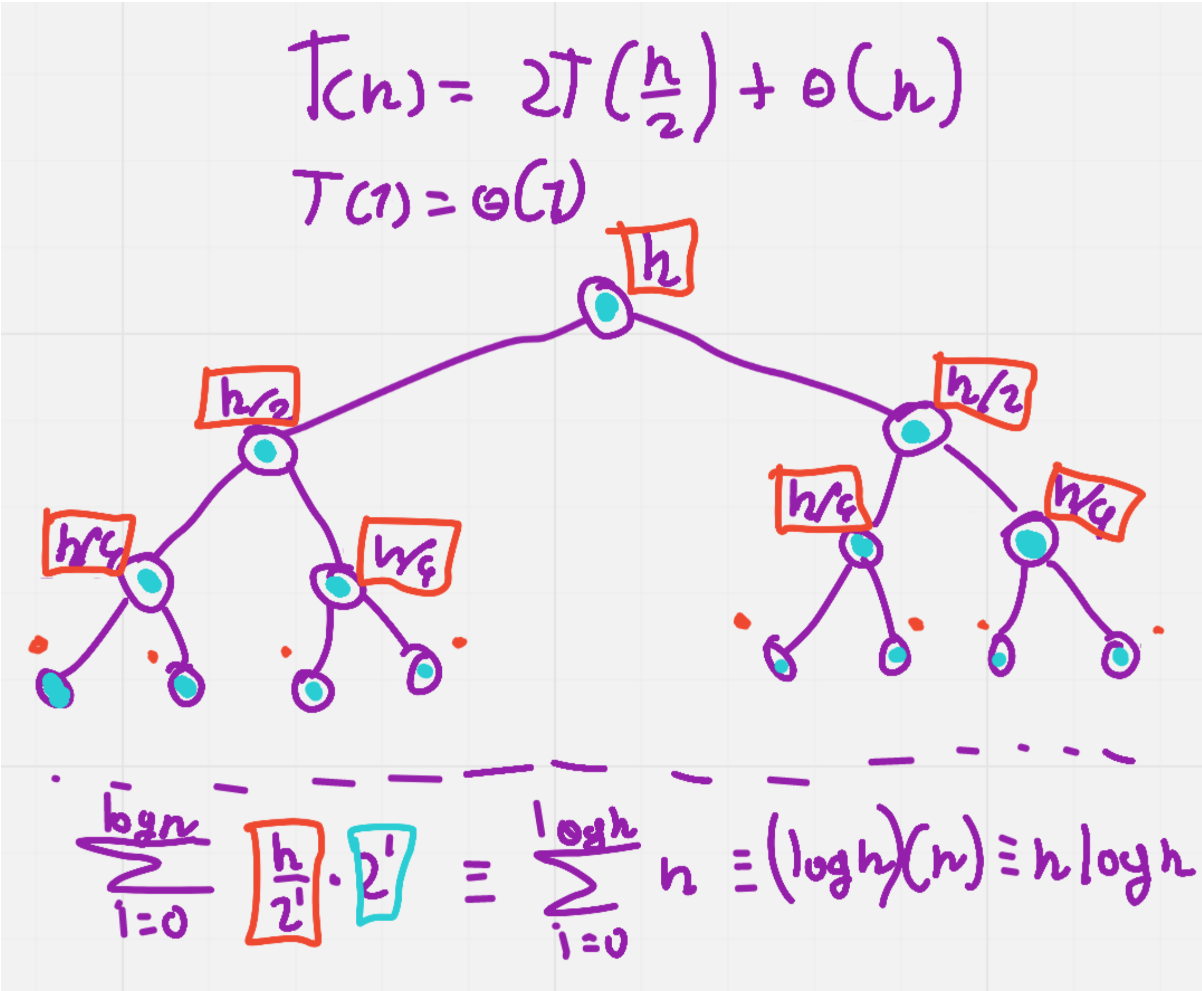
② Passo induttivo ASSUMIAMO CHE $\forall m < n \; T(m) = O(n \log n)$

$T(n) \leq 2\left(k\frac{n}{2}\log\frac{n}{2} + h\frac{n}{2}\right) + cn \equiv \underbrace{kn(\log n - \log_2 2) + hn + cn}_{\text{confronto}}$

$kn\log n - kn + hn + cn \leq kn\log n + hn \equiv cn \leq kn$

$c \leq k$ ok!

3. **Metodo dell'Albero**
   - È molto simile al Metodo Iterativo, soltanto che qui cerchiamo di dare anche una rappresentazione grafica al fine di capire come si sviluppa la ricorsione sotto forma d'albero.
   - $T(n) = T(n-1) + \Theta(1)$
   - disegniamo il costo per $T(n) = \Theta(1)$ e lo colleghiamo tramite un nodo a $T(n-1)$ e gli diamo $T(n-1) = \Theta(1)$ e così via alla fine il avremmo un abero con dei valori ai quali in alcuni casi gli potremmo assegnare delle variabili, e dobbiamo scoprire quanto ci mette $T(n) \rightarrow T(1)$ in questo caso n-1 Time quindi avremo $i=0 \rightarrow n-1 \sum+\Theta(1)$ che sarebbe $\Theta(n) - \Theta(1)$
   - Ora vediamo $T(n) = 2T(n/2) + \Theta(n)^2$ con $T(1) = \Theta(1)$
   - In cima abbiamo $\Theta(n)^2$, al 2° livello= $2*(n/2)^2$, al 3° livello= $4*(n/4)^2$, da qui vediamo una ripetizione di $2^i*(n/2^i)^2 == n^2 / 2^i$, e per vedere dove dobbiamo fermarci sappiamo che $n/2^k = 1$ quindi $n = 2^k$ 1 quindi $k = \log n$ e abbiamo $i=0 \rightarrow \log n \sum n^2/2^i$ che sarebbe
     (siccome non dipende da i)$n^2 * \sum 1/2^i$ che sarebbe $\Theta(n^2)$ * un num finito.



4. **Metodo Principale:**
   - È molto semplice ma abbiamo delle condizioni da rispettare come:
     - T(n) = a*T(n/b) + f(n) e T(1) = Θ(1).
   - S = n^(logb (a)).
   - Se f(n) = O(S) per qualche costante ε tale che S = n^(logb (a- ε))
     allora T(n) = n^(logb (a))
   - Se f(n) = Θ(S) allora T(n) = n^(logb (a)) * log n
   - Se f(n) = Ω(S) per qualche costante ε tale che S = n^(logb (a) + ε)
     e se a*f(n/b) <= c * f(n) per qualche c costante < 1 allora T(n) = Θ(f(n))

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

$$a = 2 \qquad b = 2 \qquad f(n) = n$$

$$n^{\log_2 2} = n^1 = f(n)$$

2 CASO

$$f(n) = \Theta(n^{\log_2 2})$$

$$\Theta(n)!$$

$$T(n) = \Theta(n \log n)$$

Linked List (liste puntate)

Le linked list sono formate da due valori principali, uno rappresenta il valore del puntatore e un altro il valore nella memoria del prossimo puntatore, in questo modo riusciamo ad avere una complessità spaziale costante a differenza degli array.

- **Caratteristiche delle liked List**
  Le linked list sono abbastanza complesse da utilizzare visto che al fine di ritornare all'inizio dobbiamo utilizzare la ricorsione o salvarci da qualche parte il puntatore iniziale, infine trova, elimina e inserisce elementi in O(n), apparte gli inserimenti alla fine della linked list che avviene in Θ(n)

- **Double Linked List (Liste doppiamente puntate)**
  A differenza delle normali Linked List, quest'ultime oltre a puntare al prossimo elemento, puntano anche al precedente, ovviamente il primo elemento ha come precedente None, quest'ultima cosa è molto utile nel caso dell'eliminazione di elementi/puntatori, ad esempio abbiamo None <--> 1 <--> 2 <--> 3 <--> 4 → None se abbiamo il puntatore dell'intero 2, basterà dire che il next di 1 sarà 2.next e dire che il precedente di 3 sarà il precedente del 2, tutto questo in modo costante.

Stack (pila)

Lo stack è rappresentato come una lista puntata, ma lui si ricorda sempre dell'ultimo puntatore, semplicemente riportando sempre l'ultimo puntatore(quello che ha come next None), lo stack è utile perché riusciamo a estrarre l'ultimo elemento in tempo costante, e l'inserimento viene fatto in tempo costante, perché possiamo inserire gli elemento sono in ultima posizione.

Nel complesso lo stack è poco utilizzato nei problemi informatici, ma dove è veramente utile fa la differenza, lascerò qua sotto un codice in python che ci permette di visitare un albero in modo iterativo grazie allo stack.

```
[
def visitTree(root):
    stack = []  #utilizziamo la lista di python come stack utilizzando append() e pop()
    counterOfElements = 0 #numero di elementi nello stack
    print(root.val)
    while (counterOfElements != 0 or root != None):
        if root == None or root.left == None:
            root = stack.pop()
            counterOfElements -= 1
        else:
            root = root.left
            if root.right != None:
                counterOfElements += 1
                stack.append(root.right)
]
```

HashSet

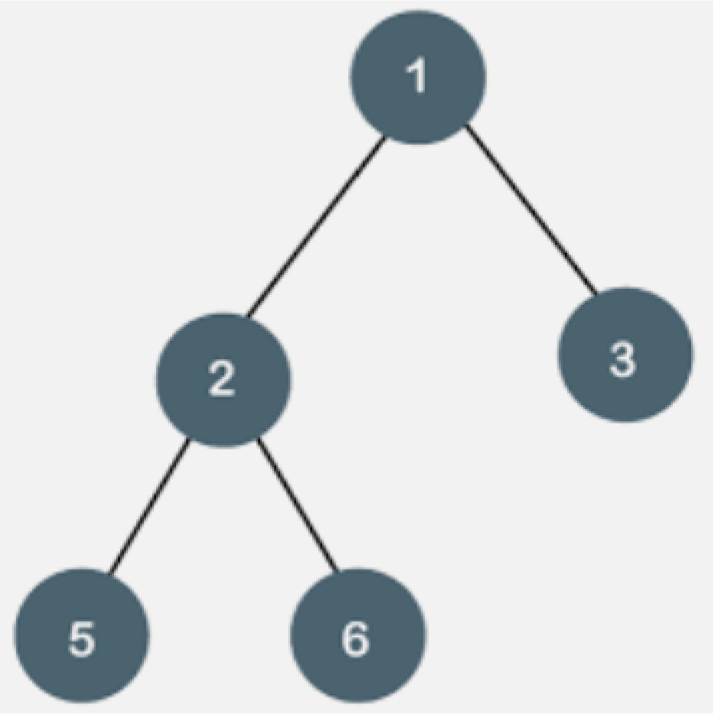
Gli HashSet sono delle strutture dati che fanno uso degli array e formule matematiche, la costruzione di un HashSet può avvenire in diversi modi, io descriverò, uno visto in privato, quindi la prima cosa è costruire un array lunga 4 e ogni volta che quest'ultima ha occupati i ¾ della sua capienza massima, ricreiamo una nuova array lunga il doppio, la cosa più negativa è che dobbiamo reinserire tutti gli elementi di nuovo, questo fa sì che il costo computazionale sia pari a $O(n)$, ma questo vuol dire allo stesso tempo che riusciamo a estrarre ed eliminare elementi in $\Theta(1)$, quindi ci sono vantaggi e svantaggi nell'HashSet, una possibile soluzione sarebbe di creare direttamente un array lunga n al fine di non ricalcolare ogni volta l'array lungo 2^h, bene ora vediamo come vengono inseriti questi elementi, in

pratica è stata creata questa funzione f(x) che prende come input un elemento e lo trasforma in un numero, da qui facciamo il % di python che fa f(input) % sizeArray = k, questo k rappresenta la posizione dell'array dove verrà inserito l'input, ovviamente la f(x) è la più randomica possibile sui moduli, ma allo stesso tempo sappiamo che riusciamo a ottenere da f(input) = I, h(I) = input, e questa cosa è la caratteristica principale dell'hashSet, ovviamente nel caso peggiore impiegheremo O(n) nel trovare un elemento.
Un modo che peggiora diminuisce il caso peggiore sarebbe considerare l'array della posizione k come un albero binario di ricerca con l'implementazione di alberi black and red, al fine di avere un albero binario di ricerca equilibrato, e al fine di inserire e ricercare elementi in un max di O(log2 n).

Alberi binari

Gli alberi binari sono di fatto degli elementi che hanno un valore e possono puntare a due diversi elementi anche chiamati left, right, per esempio la radice dell'albero avrà una parte left e una parte right (non escludiamo possano essere uguale a None), e a loro volta se diversi da None, avranno una parte left e right, e così via.

Ci sono diverse possibili visite per gli alberi, lascerò qui sono il codice in python



inorder Visit:
[5, 2, 6, 1, 3]

preorder Visit:
[1, 2, 5, 6, 3]

postorder Visit:
[5, 6, 2, 3, 1]

[
def **inorderVisit**(root):

```
    if root.left == None and root.right == None:
        print(root.val)
        pass

    if root.left != None:
        inorderVisit(root.left)

    print(root.val)
```

```
        if root.right != None:
                inorderVisit(root.right)

        pass


def preorderVisit(root):
        if root.left == None and root.right == None:
                print(root.val)
                pass

        print(root.val)
        if root.left != None:
                preorderVisit(root.left)
        if root.right != None:
                preorderVisit(root.right)

        pass


def postorderVisit(root):
        if root.left == None and root.right == None:
                print(root.val)
                pass

        if root.left != None:
                postorderVisit(root.left)
        if root.right != None:
                postorderVisit(root.right)

        print(root.val)
        pass
]
```

Ricerca di un valore (V) in una lista casuale

1. Tempo minimo Θ(n).
2. Ci calcoliamo la lunghezza della lista.
3. Scorriamo la lista per indici e se troviamo V ci segniamo l'indice e breakiamo il for.
4. ritorniamo quello che ci chiede l'esercizio (indice).

Ricerca di un valore (V) in una lista ordinata " Binary search "

1. n = len(Lista)
1. Considerando che è una lista ordinata possiamo dividere la lista log n volte

2. Quindi andremo a vedere l’elemento in lista[n//2] e se uguale a V ritorniamo n//2, altrimenti se V è maggiore allora controlleremo [n//2 + n//4] e ripeteremo le stesse operazioni, invece se minore controlleremo [n//2 - n//4] e ripeteremo le stesse operazioni.
3. Il tempo sarà uguale a $\Theta(n)$ se ci calcoliamo la len(Lista) altrimenti $\Theta(\log n)$.