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)
    - $T(n) = T(n-1) + \Theta(1)$ con $T(1) = \Theta(1)$
    - $T(n) = [\, T(n-2) + \Theta(1)\,] + \Theta(1)$ ... $T(n) = [\, T(n-3) + \Theta(1)\,] + \Theta(1) + \Theta(1)$ ……
    - 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)

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).

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) = Θ(1) e lo colleghiamo tramite un nodo a T(n-1) e gli diamo T(n-1) = Θ(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) → T(1) in questo caso n-1 Time quindi avremo i=0 → n-1 ∑+Θ(1) che sarebbe Θ(n) - Θ(1)
    - Ora vediamo T(n) = 2T(n/2) + Θ(n)^2 con T(1) = Θ(1)
    - In cima abbiamo Θ(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 → log n ∑ n^2/2^i che sarebbe (siccome non dipende da i)n^2 * ∑ 1/2^i che sarebbe Θ(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 $f(n/b) <= c * f(n)$ per qualche c costante < 1 allora $T(n) = \Theta(f(n))$
Ricordo che $n^{(1+\varepsilon)} > n*\log n$ per $n \to +\infty$

Ricerca di un valore (V) in una lista casuale

1. Tempo minimo $\Theta(n)$.
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 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 $\Theta(\log n)$.