Corso di laurea in Informatica
Introduzione agli Algoritmi

Il costo di un algoritmo

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Costo computazionale

Esaminiamo ora come calcolare effettivamente il costo computazionale di un algoritmo, utilizzando il **criterio della misura di costo uniforme**.

- E' ragionevole supporre che il costo computazionale, inteso come la funzione che rappresenta il tempo di esecuzione di un algoritmo, sia **una funzione monotona non decrescente** rispetto alla dimensione dell'input.

- Prima di procedere col calcolo del costo, è quindi necessario definire quale sia la dimensione dell'input.

- Trovare questo parametro è, di solito, piuttosto semplice:
  - in un algoritmo di ordinamento corrisponde al numero di elementi da ordinare,
  - in un algoritmo di ricerca corrisponde al numero di elementi in cui effettuare la ricerca,
  - in un algoritmo che opera su una matrice corrisponde al numero di righe e colonne, ecc..

- La notazione asintotica viene sfruttata pesantemente per il calcolo del costo computazionale degli algoritmi, quindi – in base alla definizione stessa – tale costo computazionale potrà essere ritenuto **valido *solo* asintoticamente**.

- In effetti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un certo comportamento, mentre per dimensioni maggiori un altro.

Pseudocodice

Per poter valutare il tempo computazionale di un algoritmo, esso deve essere **formulato in un modo che sia chiaro, sintetico e non ambiguo.**

Si adotta il cosiddetto ***pseudocodice***, che è una sorta di linguaggio di programmazione "informale":

- si usano, come nei linguaggi di programmazione, i costrutti di controllo (***for, if then else, while***, ecc.);
- si può usare il linguaggio naturale per specificare alcune operazioni;
- si omettono dettagli (ad es. la gestione degli errori), per esprimere solo l'essenza della soluzione.

Non esiste una notazione universalmente accettata per lo pseudocodice.

In questo corso useremo spesso i costrutti del Python ma, a volte, potremo usare altre convenzioni intuitive, ad esempio il simbolo ≠ per verificare che il contenuto di 2 variabili sia differente

Regole generali per valutare la complessità computazionale di un algoritmo:

- **Istruzioni elementari:** Operazioni come assegnazioni ($a = b$), letture del valore di una variabile, scritture (ad esempio, la stampa di una variabile), operazioni aritmetiche e logiche su un numero costante di operandi (ad esempio, $a + b$, $a > b$, ecc.) hanno un costo costante $O(1)$.

- **Istruzioni condizionali:**

```
if condizione:
    Blocco1 di istruzioni
else:
    Blocco2 di istruzioni
```

ha un costo dato da:

- il costo di verifica della condizione (di solito costante $O(1)$);
- più il massimo tra il costo delle istruzioni nel $Blocco1$ e il costo delle istruzioni nel $Blocco2$, poiché uno solo dei due blocchi viene eseguito.

Regole generali per valutare la complessità computazionale di un algoritmo:

- **Istruzioni iterative (cicli):** Le istruzioni iterative (ad esempio *for*, *while*, *repeat*) hanno un costo pari alla somma dei costi di tutte le iterazioni. Ogni iterazione ha il costo delle istruzioni eseguite all'interno del ciclo in quella iterazione.

- **Costo totale dell'algoritmo:** Il costo complessivo dell'algoritmo è pari alla somma dei costi di tutte le sue istruzioni, calcolati in base alle regole sopra descritte.

Dipendenza del costo dall'input

Un algoritmo può avere tempi di esecuzione diversi a seconda dell'input fornito. Per una valutazione completa del suo comportamento computazionale, è utile analizzare tre casi distinti:

- **Caso migliore:** rappresenta la situazione più favorevole, in cui l'algoritmo raggiunge il tempo di esecuzione minimo possibile.
- **Caso peggiore:** descrive la situazione più onerosa, cioè l'input che causa il massimo tempo di esecuzione.
- **Caso medio:** considera il tempo di esecuzione atteso su una distribuzione probabilistica degli input, fornendo un'indicazione del comportamento tipico dell'algoritmo.

Importanza del caso peggiore. Il caso peggiore è spesso il più studiato, perché:

- fornisce un limite superiore sul tempo di esecuzione, garantendo che l'algoritmo non supererà mai tale limite per nessun input;
- è fondamentale per applicazioni critiche, dove si devono evitare sorprese in termini di prestazioni, anche nelle condizioni meno favorevoli.

Il caso medio. Il caso medio, sebbene più complesso da calcolare, offre una visione realistica del comportamento di un algoritmo nella pratica. Tuttavia, per determinarlo è necessario:

- definire una distribuzione probabilistica sugli input (ad esempio, assumendo che tutti gli input siano equiprobabili);
- calcolare il tempo atteso di esecuzione su tale distribuzione.

Il caso medio è utile soprattutto in contesti pratici, dove interessa sapere come l'algoritmo si comporta in condizioni tipiche, piuttosto che in scenari estremi.

Considerazioni pratiche. In molti casi, l'analisi si concentra sul caso peggiore, poiché:

- è spesso più semplice da calcolare rispetto al caso medio;
- garantisce previsioni conservative utili per la progettazione e l'ottimizzazione degli algoritmi.

Tuttavia, quando il caso medio può essere calcolato, fornisce informazioni più utili per valutare le prestazioni reali dell'algoritmo. È importante anche considerare il caso migliore, che può evidenziare situazioni particolarmente efficienti (ad esempio, quando l'input è già parzialmente ordinato in un algoritmo di ordinamento).

Notazione asintotica per il calcolo del costo. La notazione asintotica (O, Ω, Θ) è uno strumento essenziale per descrivere il comportamento degli algoritmi al crescere della dimensione dell'input, ignorando dettagli come costanti moltiplicative e termini di ordine inferiore:

- **Notazione O:** indica un limite superiore sul costo. Nel caso peggiore, assicura che il tempo di esecuzione non superi una certa funzione.
- **Notazione Ω:** fornisce un limite inferiore, garantendo che il costo non sia inferiore a una certa funzione.
- **Notazione Θ:** quando un algoritmo ha un comportamento asintotico esattamente descritto da una funzione, la notazione Θ fornisce una caratterizzazione precisa.

Questo approccio fornisce una metodologia chiara e sistematica per calcolare il costo computazionale di un algoritmo, consentendo di confrontare l'efficienza di diverse soluzioni per lo stesso problema.

Esempi elementari dove con $T(n)$ si indica la complessità della funzione su input di dimensione n.

1. Funzione per il calcolo del massimo all'interno di una lista con n valori:

```
def TrovaMax(lista):
    massimo=lista[0].                    Θ(1)
    for i in range(1,len(lista)):          n − 1 iterazioni
        if lista[i]> massimo:               Θ(1)
            massimo=lista[i]                Θ(1)
    print(massimo).                      Θ(1)
```

Per il costo computazionale si ha :

$$T(n) = \Theta(1) + n[\Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

2. Funzione per il calcolo della somma dei primi n interi:

```
def somma(n):                                          
    '''calcola la somma dei primi n interi positivi'''
    somma = 0                                          Θ(1)
    for i in range(1,n+1):                             n iterazioni
        somma += 1                                         Θ(1)
    print(somma)                                       Θ(1)
```

Per il costo computazionale si ha:

$$T(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) = \Theta(n)$$

Si può fare di meglio:

```
def somma(n):
    '''calcola la somma dei primi n interi positivi'''
    somma = n*(n+1)//2                    Θ(1)
    print(somma)                          Θ(1)
```

Per il costo computazionale si ha:

$$T(n) = \Theta(1) + \Theta(1) = \Theta(1).$$

Operazioni a complessità non costante in Python

In Python, alcune operazioni non hanno un costo unitario, cioè la loro complessità dipende dalla struttura sottostante dei dati. Un esempio significativo è l'operatore $\in$, che viene utilizzato per verificare se un elemento appartiene a una sequenza, come una lista, un dizionario o un set. La complessità di questa operazione dipende dalla struttura della sequenza e dalle operazioni che essa richiede per essere attraversata.

Operatore $\in$ con liste, insiemi e dizionari

Quando usiamo l'operatore $\in$ con una `lista`, Python deve eseguire una *ricerca sequenziale*, scorrendo l'intera struttura dati per verificare se l'elemento cercato è presente. Pertanto, la complessità di questa operazione è lineare, ovvero:

$$O(n)$$

dove n è la lunghezza della lista.
Al contrario, quando usiamo l'operatore $\in$ su un *set* o un *dizionario*, la ricerca avviene in tempo costante in media grazie all'uso di una struttura di dati chiamata *tabella di hash* che vedremo nel seguito del corso. In questo caso, la complessità media dell'operazione è:

$$O(1)$$

la complessità può aumentare, arrivando a $O(n)$ nel caso peggiore.

In conclusione l'operatore $\in$ ha una complessità che dipende dalla struttura dei dati:

- Per `liste`, la complessità è $O(n)$, poiché la ricerca è sequenziale.
- Per *set* e *dizionari*, la complessità è $O(1)$ in media, grazie all'uso dell'hashing ma $O(n)$ nel caso pessimo.

Questa differenza evidenzia l'importanza della scelta della struttura dati in relazione alle operazioni previste, per ottenere il miglior compromesso tra facilità d'uso e prestazioni.

Operazione di inserimento e cancellazione in liste, insiemi e dizionari

Le operazioni di inserimento o eliminazione in una lista, specialmente quando vengono eseguite all'inizio o nel mezzo della lista, richiedono lo **spostamento degli altri elementi** per mantenere la coerenza della struttura. In particolare:

- **Inserire un elemento all'inizio o nel mezzo della lista**: La complessità di queste operazioni è $O(n)$, poiché gli altri elementi devono essere spostati per fare spazio all'inserimento.
- **Eliminare un elemento dall'inizio o nel mezzo della lista**: Anche l'eliminazione richiede lo spostamento degli elementi rimanenti, quindi la complessità è anch'essa $O(n)$.

Le operazioni di inserimento e cancellazione in *set* e *dizionari* hanno una complessità ben diversa rispetto alle stesse operazioni su liste. Grazie all'implementazione di queste strutture tramite tabelle hash queste operazioni hanno un costo $O(1)$ al caso medio ma al caso pessimo possono richiedere tempo $O(n)$.

Operazione di slicing su liste o stringhe

Il costo dell'istruzione Python $A[a : b]$ dipende dalla dimensione della slicing generata, ossia dal numero di elementi inclusi nell'intervallo $[a, b)$. Ogni elemento nell'intervallo deve essere copiato dalla lista originale alla nuova lista, pertanto il cosot dell'operazione è proporzionale alla lunghezza del risultato: $O(b - a)$. Il discorso è analogo anche nel caso di slicing di stringa.

Operazioni di modifica di stringhe

In Python, le stringhe sono *immutabili*, il che significa che il loro contenuto non può essere modificato dopo la creazione. Se desideri "modificare" una stringa, devi generare una nuova stringa basata sulla modifica desiderata. Ad esempio, l'operazione $stringa = stringa +' a'$ che aggiunge un carattere alla stringa, ha un costo proporzionale alla lunghezza della stringa. Questo perché viene creata una copia della stringa originale con l'aggiunta del simbolo $'a'$.
Se è necessario modificare frequentemente una stringa s, è preferibile utilizzare delle **liste di caratteri**. Questo approccio consiste nel convertire la stringa in lista con l'operazione $lista = list(s)$), effettuare le modifiche desiderate sulla lista e poi riconvertirla in stringa (con l'operatore "".$join(lista)$".

Conclusioni generali Quelli appena visti sono solo alcuni esempi. In conclusione le operazioni non di costo costante dipendono fortemente dalla struttura dei dati e dalla posizione dell'elemento su cui si opera. Le operazioni che richiedono spostamenti di elementi (come l'inserimento in una lista non alla fine o la cancellazione) tendono a essere più costose rispetto a quelle che operano in modo diretto su strutture più efficienti (come *set* o *dizionari*).
Questa differenza evidenzia l'importanza della scelta della struttura dati in relazione alle operazioni previste, per ottenere il miglior compromesso tra facilità d'uso e prestazioni.

Di seguito 10 programmi e per ognuno va valutata la complessità computazionale

1. **Esempio 1**

```
def es1(n):
    t = 0
    n = abs(n)
    if n > 100: return 1
    for i in range(n):
        t += 3
    return t
```

1. **Esempio 1**

```python
def es1(n):
    t = 0
    n = abs(n)
    if n > 100: return 1
    for i in range(n):
        t += 3
    return t
```

La funzione es1 esegue le seguenti operazioni:

(a) Calcolo del valore assoluto di n tramite $abs(n)$, che ha complessità $O(1)$.

(b) Verifica della condizione $if\, n > 100$, che ha complessità $O(1)$. Se la condizione è vera, la funzione restituisce immediatamente 1, anch'essa in tempo $O(1)$.

(c) Se $n \leq 100$, viene eseguito un ciclofor che itera esattamente n volte. Ogni iterazione richiede $O(1)$ per incrementare la variabile t di 3.

Tuttavia, il ciclo viene eseguito solo se $n \leq 100$. In questo caso, il numero massimo di iterazioni è una costante ($n \leq 100$), e il tempo totale del ciclo risulta anch'esso costante.

Complessità Asintotica: $O(1)$.

2. Esempio 2

```
def es2(n):
    while n >= 0:
        if n%2: return 1
        n -= 2
    return 0
```

2. **Esempio 2**

```
def es2(n):
    while n >= 0:
        if n%2: return 1
        n -= 2
    return 0
```

La funzione $es2$ esegue le seguenti operazioni:

(a) Controlla la condizione $n >= 0$ a ogni iterazione del ciclo $while$. Questa operazione ha complessità $O(1)$.

(b) Verifica se n è dispari tramite $n\%2$, che ha complessità $O(1)$. Se n è dispari, la funzione termina immediatamente restituendo 1, in tempo $O(1)$.

(c) Se n è pari, la funzione riduce n di 2 a ogni iterazione $n-=2$). Questo processo continua finché $n < 0$.

- **Caso migliore:** Quando n è dispari, la condizione $n\%2$è verificata alla prima iterazione, e la funzione restituisce 1. In questo caso, il ciclo viene eseguito solo una volta, e la complessità è $O(1)$.
- **Caso peggiore:** Quando n è pari, il ciclo $while$ viene eseguito $\lfloor n/2 \rfloor + 1$ volte (poiché n viene ridotto di 2 a ogni iterazione). Di conseguenza, la complessità del caso peggiore è $\Theta(n)$.

In conclusione, la complessità asintotica del programma è:

- **Caso peggiore:** $\Theta(n)$, che si verifica quando n è pari.
- **Caso migliore:** $O(1)$, che si verifica quando n è dispari.

3. Esempio 3

```
def es3(n):
    n = abs(n)
    x = r = 0
    while x*x < n:
        x += 1
        r += 3*x
    return r
```

3. **Esempio 3**

```
def es3(n):
    n = abs(n)
    x = r = 0
    while x*x < n:
        x += 1
        r += 3*x
    return r
```

La funzione $es3$ esegue le seguenti operazioni:

- L'istruzione $n = abs(n)$ è un'operazione iniziale con complessità $O(1)$.
- Le assegnazioni iniziali di $x = 0$ e $r = 0$ hanno complessità $O(1)$.
- Il ciclo $while$ esegue le iterazioni finché $x^2 < n$. Ad ogni iterazione:
  - La variabile x viene incrementata di 1.
  - La variabile r viene aggiornata con un valore proporzionale a x, ovvero $r+ = 3x$.
  - Ogni iterazione comporta un numero costante di operazioni, quindi ha complessità $O(1)$.

Il ciclo termina quando $x^2 \geq n$. Questo implica che il numero totale di iterazioni del ciclo è proporzionale alla radice quadrata di n, cioè $O(\sqrt{n})$. Durante ogni iterazione viene eseguito un numero costante di operazioni ($O(1)$).

In conclusione

- Le operazioni iniziali e finali del programma hanno complessità $O(1)$, ma non influenzano significativamente il tempo complessivo di esecuzione.
- La complessità complessiva del programma è dominata dal ciclo $while$, che ha complessità $O(\sqrt{n})$.

Complessità Asintotica: $O(\sqrt{n})$

4. Esempio 4

```
def es4(n):
    x = r = 0
    while n > 1:
        r += 2
        n = n//3
    return r
```

□

4. **Esempio 4**

```
def es4(n):
    x = r = 0
    while n > 1:
        r += 2
        n = n//3
    return r
```

La funzione es4 esegue le seguenti operazioni:

- Le inizializzazioni $x = 0$ e $r = 0$ hanno complessità $O(1)$.
- Il ciclo *while* continua ad iterare finché $n > 1$. Durante ogni iterazione:
  - La variabile r viene incrementata di 2 ($O(1)$).
  - Il valore di n viene ridotto a $\lfloor n/3 \rfloor$ ($O(1)$).

Ad ogni iterazione, il valore di n viene ridotto di un fattore costante ($n = \lfloor n/3 \rfloor$). All'iterazione i, il valore di n è approssimativamente $\frac{n}{3^i}$. Il ciclo termina quando $\frac{n}{3^i} < 1$, ovvero quando $i > \log_3 n$.

Pertanto, il numero totale di iterazioni del ciclo è $O(\log_3 n)$.

Ogni iterazione del ciclo richiede un numero costante di operazioni ($O(1)$), e il ciclo viene eseguito $O(\log_3 n)$ volte. Di conseguenza, la complessità complessiva del programma è:

$$O(\log_3 n)$$

Poiché i logaritmi in basi diverse differiscono solo per una costante moltiplicativa, la complessità può essere espressa anche come $O(\log n)$.

La complessità del programma è:

Complessità Asintotica: $O(\log n)$

5. Esempio 5

```
def es5(n):
    t = x = 1
    for i in range(n):
        t = 3*t
    while t > x:
        x += 2
        t -= 2
    return x
```

—

5. **Esempio 5**

```python
def es5(n):
    t = x = 1
    for i in range(n):
        t = 3*t
    while t > x:
        x += 2
        t -= 2
    return x
```

Il programma $es5$ contiene due cicli principali: un ciclo for e un ciclo $while$. Analizziamo la loro complessità separatamente.

Il ciclo for esegue n iterazioni. Durante ogni iterazione, il valore di t viene triplicato ($t = 3 \cdot t$), un'operazione di complessità $O(1)$. Pertanto, la complessità complessiva del ciclo è:

$$O(n)$$

Al termine del ciclo, il valore di t sarà pari a 3^n.

Dopo il ciclo for, il valore iniziale di t è 3^n, mentre x è inizializzato a 1.

Il ciclo $while$ continua ad iterare finché $t > x$. Durante ogni iterazione:

- Il valore di x viene incrementato di 2 ($x = x + 2$).
- Il valore di t viene decrementato di 2 ($t = t - 2$).

Ad ogni iterazione, la differenza $t - x$ si riduce di 4. Inizialmente, questa differenza è pari a:

$$t - x = 3^n - 1$$

Pertanto, il numero totale di iterazioni del ciclo $while$ è:

$$\frac{3^n - 1}{4} = O(3^n)$$

La complessità del ciclo $while$ è quindi $O(3^n)$.

La complessità complessiva del programma è determinata dal ciclo con la complessità maggiore. Poiché $O(3^n)$ cresce molto più velocemente di $O(n)$, la complessità dominante è quella del ciclo $while$.

Complessità Asintotica: $O(3^n)$

6. Esempio 6

```
def es6(n):
    p = 2
    while n >= p:
        p = p*p
    return p
```

6. **Esempio 6**

```
def es6(n):
    p = 2
    while n >= p:
        p = p*p
    return p
```

Il programma $es6$ contiene un ciclo $while$ che parte con $p = 2$ e continua finché $p \leq n$. Ad ogni iterazione, p viene aggiornato elevandolo al quadrato ($p = p \cdot p$).

Evoluzione di p:
All'inizio, $p = 2$. Dopo ogni iterazione, il valore di p diventa:

$$p = 2^{2^i},$$

dove i è il numero di iterazioni eseguite fino a quel momento. Il ciclo termina quando $p > n$, ossia quando:

$$2^{2^i} > n.$$

Applicando il logaritmo in base 2 su entrambi i membri:

$$2^i > \log_2 n.$$

Applichiamo il logaritmo una seconda volta:

$$i > \log_2(\log_2 n).$$

Da qui, il numero totale di iterazioni è approssimativamente:

$$i = O(\log \log n).$$

Ogni iterazione del ciclo while esegue un numero costante di operazioni, quindi la complessità di ogni iterazione è $O(1)$. Poiché il ciclo viene eseguito $O(\log \log n)$ volte, si ha:

Complessità Asintotica: $O(\log \log n)$.

7. Esempio 7

```
def es7(n):
    u,t,s = 1
    while u <= n:
        for j in range(t):
            s += 1
        u += 1
        t += 1
    return s
```

7. **Esempio 7**

```
def es7(n):
    u,t,s = 1
    while u <= n:
        for j in range(t):
            s += 1
        u += 1
        t += 1
    return s
```

Il programma $es7$ contiene un ciclo $while$ che si ripete finché $u \leq n$.
Ad ogni iterazione del ciclo:

- u viene incrementato di 1;
- t viene incrementato di 1;
- Viene eseguito un ciclo for che si itera esattamente t volte.

Numero di Iterazioni Totali del Ciclo for:
All'iterazione i del ciclo $while$, il valore di t è pari a i. Pertanto, il ciclo for si ripete i volte. Il numero totale di iterazioni del ciclo for in tutte le iterazioni del ciclo $while$ è dato dalla somma dei numeri da 1 a n:

$$1+2+3+\ldots+n=\frac{n(n+1)}{2}.$$

La somma dei numeri da 1 a n è di ordine $O(n^2)$. Poiché ogni iterazione del ciclo for richiede un numero costante di operazioni ($O(1)$), il contributo complessivo del ciclo for è $O(n^2)$.

Il ciclo $while$ viene eseguito n volte, ma il costo aggiuntivo delle operazioni fuori dal ciclo for è costante ($O(1)$) e non influisce sull'ordine di crescita complessivo.

Complessità Asintotica: $O(n^2)$.

8. Esempio 8

```
def es8(n):
    n = abs(n)
    s = t = n
    p = 0
    while s >= 1:
        s = s//4
        p += 1
    while n-p > 0:
        n -= p
        t += 5
    return t
```

Esempio 8

```
def es8(n):
    n = abs(n)
    s = t = n
    p = 0
    while s >= 1:
        s = s//4
        p += 1
    while n-p > 0:
        n -= p
        t += 5
    return t
```

La complessità del programma è determinata principalmente dalla somma delle complessità dei due cicli *while*. Analizziamoli separatamente.

Primo Ciclo *while* :
Il primo ciclo continua finché $s \geq 1$, e ad ogni iterazion s viene diviso per 4. Questo ciclo termina quando $s < 1$, i che avviene dopo $\log_4 n$ iterazioni. In termini di complessit asintotica, il numero di iterazioni è:

$$O(\log n),$$

poiché $\log_4 n$ differisce da $\log n$ solo per una costante molti plicativa.

Secondo Ciclo *while*:
Il secondo ciclo continua finché $n - p > 0$. Ad ogni iterazione:

- n viene decrementato di p, dove $p = \lfloor \log_4 n \rfloor$;
- t viene incrementato di una costante $(+5)$.

Il numero di iterazioni di questo ciclo è approssimativamente pari a:

$$\frac{n}{p} \approx \frac{n}{\log_4 n}.$$

Poiché $\log_4 n = O(\log n)$, il numero di iterazioni è $O\left(\frac{n}{\log n}\right)$.

La complessità complessiva del programma è determinata dal ciclo dominante. Poiché il primo ciclo ha complessità $O(\log n)$ e il secondo ciclo ha complessità $O\left(\frac{n}{\log n}\right)$, il secondo ciclo domina:

Complessità Asintotica: $O\left(\frac{n}{\log n}\right)$.

9. Esempio 9

```
def es9(n):
    n = abs(n)
    s,p = n, 2
    i,r = 1
    while s >= 1:
        s = s//5
        p += 2
    p = p*p
    while i*i*i < n:
        for j in range(p):
            r += 1
        i += 1
    return r
```

9. **Esempio 9**

```
def es9(n):
    n = abs(n)
    s,p = n, 2
    i,r = 1
    while s >= 1:
        s = s//5
        p += 2
    p = p*p
    while i*i*i < n:
        for j in range(p):
            r += 1
        i += 1
    return r
```

La complessità del programma *es9* è data essenzialmente dalla somma delle complessità dei due cicli *while*. Analizziamoli separatamente.

Primo Ciclo *while*: Il primo ciclo parte con $p = 2$ e $s = n$ continua finché $s \geq 1$. Ad ogni iterazione:

- s viene diviso per 5;
- p viene incrementato di 2.

Il numero di iterazioni è proporzionale a quante volte s (inizialmente uguale a n) può essere diviso per 5 prima di diventare minore di 1. Pertanto, il numero di iterazioni è $O(\log_5 n)$, che equivale a $O(\log n)$, poiché la base del logaritmo è una costante.

Secondo Ciclo *while*: Alla fine del primo ciclo, p, che è stato incrementato di 2 per ogni iterazione, vale $p = O(\log n)$. Dopo l'istruzione $p = p * p$, p diventa $O(\log^2 n)$.
Nel secondo ciclo *while*:

- La condizione $i*i*i < n$ implica che il ciclo termina quando $i \geq n^{1/3}$. Pertanto, il numero di iterazioni è $O(n^{1/3})$.
- In ogni iterazione del ciclo, c'è un ciclo *for* che esegue p iterazioni. Poiché $p = O(\log^2 n)$, ogni iterazione del ciclo *while* ha un costo $O(\log^2 n)$.

Il costo totale del secondo ciclo *while* è quindi:

$$O\left(n^{1/3} \cdot \log^2 n\right).$$

Il primo ciclo *while* ha complessità $O(\log n)$, mentre il secondo ciclo *while* ha complessità $O(n^{1/3} \cdot \log^2 n)$. Poiché il secondo ciclo domina:

Complessità Asintotica: $O\left(n^{1/3} \cdot \log^2 n\right)$.

10. Esempio 10

```
def es10(n):
    tot = n
    if n <= 100: return 5
    j = 512
    while j >= 2:
        k = 1
        while k*k <= n:
            k = 2*k
        tot += 5
        j = j//2
    return tot
```

10. **Esempio 10**

```
def es10(n):
    tot = n
    if n <= 100: return 5
    j = 512
    while j >= 2:
        k = 1
        while k*k <= n:
            k = 2*k
        tot += 5
        j = j//2
    return tot
```

Per analizzare la complessità del programma $es10$, esaminiamo il comportamento di ciascun ciclo $while$.

Il ciclo esterno ha la condizione $j \geq 2$ e inizia con $j = 512$. Ad ogni iterazione, j viene dimezzato ($j = j//2$). Pertanto, il ciclo esterno si ripete fino a quando j diventa minore di 2. Il numero di iterazioni di questo ciclo è $O(\log j)$. Poiché j parte da 512, il ciclo esterno esegue $O(\log 512) = O(9)$ iterazioni, che è una costante, quindi $O(1)$.

Nel ciclo interno, k parte da 1 e viene raddoppiato ad ogni iterazione ($k = 2 \cdot k$). Il ciclo continua finché $k \cdot k \leq n$. Quindi, il ciclo termina quando k diventa maggiore di $\sqrt{n}$. Poiché k raddoppia in ogni iterazione, dopo i iterazioni k vale 2^i e il numero di iterazioni del ciclo è $O(\log_2 \sqrt{n}) = O(\log n)$.
In conclusione:

Complessità Asintotica: $O(1) \cdot O(\log n) = O(\log n)$.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

1. Determina la complessità dei seguenti codici e spiega il tuo ragionamento:

(a)
```
def f(n):
    i = 1
    while i < n:
        print(i)
        i *= 2
```

(b)
```
def f(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print(i, j, k)
```

(c)
```
def f(n):
    i = 1
    while i * i <= n:
        x = i
        while x > 0:
            x //= 2
        i += 1
    return i*x
```

(d)

```
m, tot = 1, 0
for _ in range(n):
    m *= 3
for j in range(1,m+1):
    t = j
    while t > 0:
        if t % 2 == 1:
            tot += 1
        t //= 2
return tot
```

(e)

```
tot = 0
for i in range(n):
    s, t = 0, i
    while t>0:
        t /=2
        s+=1
    j= n
    while j>0:
        j -= s
        if j % 7 :
            tot += 1
return tot
```

2. per ciascuna delle seguenti funzioni in Python:
   - Calcolane la complessità.
   - Determina cosa fa
   - proponi un algoritmo alternativo che abbia complessità inferiore

(a)
```python
def es(lista):
    R = []
    for i in range(len(lista)):
        s=set()
        for j in range(i+1)
            s.add(lista[j])
        R.append(len(s))
    return R
```

(b)
```python
def es(lista):
    tot = 0
    for i in range(len(lista)):
        for j in range(i, len(lista)):
            if lista[i] != lista[j]:  tot += 1
    return tot
```

(c)

```
def es(lista , x):
    for i in range(len(lista) - 1):
        for j in range(i + 1, len(lista)):
            if lista[i] + lista[j] == x:
                return True
    return False
```

(d)

```
def es(lista):
    m=1
    n= len(lista)
    for i in range(n):
        j=i+1
        while j<n  and A[j-1]<A[j]:
            j+= 1
        m = max(m, j-i)
    return m
```