Costo computazionale

▼ ***INDICE***

3 Costo computazionale

Fino ad ora abbiamo visto il costo computazionale di funzioni ipotetiche. Ora andiamo a vedere come calcolare effettivamente il costo computazionale di un algoritmo usando il criterio della misura di costo uniforme.

> ⚠️ ***ATTENZIONE:***
>
> Ovviamente il costo computazionale inteso come funzione che va a rappresentare il tempo di esecuzione di un algoritmo è una funzione monotona non decrescente, poiché aumentando l'input ovviamente il tempo di esecuzione aumenta.

Visto che il tempo di esecuzione è prettamente basato sulla quantità di dati in input bisogna saper riconoscere all'interno del codice il parametro corrispondente ad esso:

- In un algoritmo di ordinamento sarà il numero di dati da ordinare;
- In un algoritmo che lavora sulla matrice sarà il numero di righe e colonne(quindi, 2 parametri);
- In un algoritmo che opera su alberi sarà il numero di nodi;
- Ecc…

Quindi prima di calcolare il costo è necessario stabilire quale sia la variabile(o variabili) di riferimento.

La notazione asintotica viene sfruttata per il calcolo del costo computazionale degli algoritmi, quindi tale costo potrà essere ritenuto valido solo asintoticamente.

Per poter valutare il tempo computazionale useremo lo pseudocodice, ossia un linguaggio di programmazione "informale".

Nel nostro caso useremo spesso i costrutti di Python essendo molto intuitivi, ma potremo usare simboli non utilizzati nel codice, come per esempio $\neq$ per verificare se il contenuto di 2 variabili sia diverso.

3.1 Costo delle istruzioni

Principalmente siamo in grado di individuare 3 categorie di istruzioni:

- Istruzioni elementari: sono tutte istruzioni che non dipendono dalla dimensione dell'input quindi con un tempo di esecuzione costante(per esempio: lettura e scrittura di una variabile, operazioni aritmetiche, stampa ecc..).

  Avendo un tempo di esecuzione costante il costo equivale a $\Theta(1)$.

  Ad esempio, queste istruzioni hanno tutte costo $\Theta(1)$:

📎 ***ESEMPIO DI ISTRUZIONI ELEMENTARI:***

```
var = 7       #Θ(1)
var += 6*5    #Θ(1)+Θ(1)+Θ(1) = Θ(1)
print(var)    #Θ(1)
```

- Blocchi if/else: hanno un costo pari alla somma tra il costo della verifica della condizione e il max tra i costi del blocco if e else:

📎 ***ESEMPIO DI BLOCCHI IF/ELSE:***

```
if (a>b):
  a += b
  print("Il valore di a+b è ",a)
else:
  print("Il valore di a è ",a)
```

In questo caso sarà:

$$T(n) = CostoVerifica + max(CostoIf, CostoElse) = \Theta(1) + max(\Theta(1), \Theta(1)) = 2\Theta(1) = \Theta(1)$$

- Istruzioni iterative(ossia i cicli): hanno un costo pari alla somma dei costi di ciascuna delle iterazioni, quindi se tutte le iterazioni hanno costo uguale, il costo del blocco iterativo sarà il prodotto del costo di una singola iterazione per il numero di iterazioni:

📎 ***ESEMPIO DI ISTRUZIONI ITERATIVE:***

```
def Calcola_Somma_1(n):
  somma = 0             #Θ(1)
  for i range(1,n+1):   #n iterazioni + Θ(1)
    somma += 1           #Θ(1)
  return somma          #Θ(1)
```

In questo il risultato sarà:

$$T(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) + \Theta(1) = 3\Theta(1) + \Theta(n) = \Theta(n)$$

Informalmente, quindi, ricaviamo il costo dell'algoritmo nel suo complesso tramite la somma dei costi delle istruzioni che lo compongono.

Tuttavia ovviamente un algoritmo potrebbe avere costi diversi in base all'input, poiché con un input vantaggioso avremo un caso migliore mentre con uno svantaggioso ne avremo uno peggiore, dunque per avere un'idea sul costo dell'algoritmo, cercheremo il suo comportamento nel caso peggiore, cercando di calcolare il costo in termini di notazione asintotica Θ.

Laddove questo non è possibile essa dovrà essere approssimata per difetto(Ω) o per eccesso(O).

Per capire meglio, proviamo a prendere l'ultimo esempio fatto. Possiamo vedere che il problema può essere svolto in modo molto più efficiente:

📎 ***ESEMPIO MIGLIORIA CODICE:***

```
def Calcola_Somma_2(n):
  somma = n*(n+1)/2   #Θ(1)
  return somma        #Θ(1)
```

In questo caso sarà semplicemente:

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

Come possiamo vedere il costo sarà il risultato sarà $\Theta(1)$ che è decisamente meglio rispetto a $\Theta(n)$.

3.2 Esempi valutazione di un algoritmo

Vediamo un primo esempio molto semplice, dove proviamo ad analizzare l'algoritmo per il calcolo del massimo in un vettore disordinato contenente n valori:

📎 ***ESEMPIO CALCOLO DEL MASSIMO IN UN ARRAY:***

```
def Trova_Max(A):
  n = len(A)             #θ(1)
  max = A[0]             #θ(1)
  for i in range(1,n):   #(n-1) iterazioni + θ(1)
    if A[i] > max:        #θ(1)
      max = A[i]           #θ(1)
  return max             #θ(1)
```

Proviamo a calcolare il costo computazione a blocchi:

1. Per prima cosa prendiamo i viola:

$$T_1 = \Theta(1) + \Theta(1) = \Theta(1)$$

2. Successivamente calcoliamo i verdi:

$$T_2 = \Theta(1) + \Theta(1) = \Theta(1)$$

3. Infine calcoliamo il costo complessivo dell'algoritmo, che sarà:

$$T(n) = T_1 + (n-1) \cdot T_2 + \Theta(1) + \Theta(1) = 3\Theta(1) + \Theta(n-1) = \Theta(n)$$

📖 ***NOTA:***

Nella terza fase possiamo vedere che verso l'ultimo passaggio trasformiamo $\Theta(n-1)$ in $\Theta(n)$ questo perché prendiamo il massimo tra i due costi che in questo caso è n per l'appunto.

Ora andiamo a vedere un esempio un po' più complicato rispetto al precedente, dove vogliamo valutare un polinomio espresso nella seguente forma:

$$\sum_{k=0}^{n} a_i \cdot x^i$$

📎 ***ESEMPIO VALUTAZIONE DI UN POLINOMIO IN UN PUNTO:***

```
def Calcoa_Polinomio_1(A, c):
  somma = A[0]                 #θ(1)
  for i in range(len(A)):      #n iterazioni + θ(1)
    potenza = 1                 #θ(1)
    for j in range(i):          #i iterazioni + θ(1)
      potenza = c*potenza        #θ(1)
    somma +=  A[i]*potenza      #θ(1)
  return somma                 #θ(1)
```

Proviamo come prima cosa a calcolare il blocco di colore blu:

$$T_1 = \Theta(1) + i(\Theta(1) + \Theta(1)) + \Theta(1) + \Theta(1) = 3\Theta(1) + i(2\Theta(1)) = \Theta(1) + \Theta(i) = \Theta(i)$$

Ora che abbiamo calcolato questo andiamo a calcolare i costo complessivo dell'algoritmo:

$$T(n) = \Theta(1) + \sum_{i=1}^{n} \cdot T_1 = \Theta(1) + \Theta\left(\sum_{i=1}^{n} i\right) = \Theta(1) + \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Dall'ultimo esempio ci viene come risultato $\Theta(n^2)$, per questo l'algoritmo sarà lentissimo, per questo dobbiamo provare ad alleggerirlo, in questo modo:

```
def Calcola_Polinomio(A,c):
  somma = A[0]                #θ(1)
  potenza = 1                 #θ(1)
  for i in range(1,len(A)):   #n iterazioni + θ(1)
    potenza = c*potenza        #θ(1)
```

```
    somma += A[i]*potenza      #Θ(1)
  return somma                #Θ(1)
```

Di conseguenza il costo complessivo dell'algoritmo sarà:

$$T(n) = \Theta(1) + n\cdot(\Theta(1) + \Theta(1) + \Theta(1)) + \Theta(1) = 4\Theta(1) + n(3\Theta(1)) = \Theta(1) + \Theta(n) = \Theta(n)$$

Così avremo un costo computazionale di $\Theta(n)$ invece che di $\Theta(n^2)$.

3.3 Tempi di esecuzione

Compreso come valutare il costo di un logaritmo, possiamo effettivamente capire quanto sono grandi i tempi di esecuzione di un algoritmo in base al suo costo computazionale.

Ipotizziamo di avere un calcolatore in grado di fare un operazione elementare in un nanosecondo(ossia 10^9 operazioni al secondo) e supponiamo che la dimensione de idati in un input sia $n = 10^6$.

- Tempi di un algoritmo con costo $\mathrm{O}(n)$:

$$T = \frac{10^6}{10^9\ op/s} = 10^{-3} = 1\ millisecondo$$

- Tempi di un algoritmo con un costo $\mathrm{O}(n \cdot log(n))$:

$$T = \frac{10^6 \cdot log(10^6)}{10^9\ op/s} = \frac{3 \cdot log(10)}{500} \approx 20\ millisecondi$$

- Tempi di un algoritmo con costo $\mathrm{O}(n^2)$:

$$T = \frac{(10^6)^2}{10^9\ op/s} = 1000\ secondi \approx 17\ minuti$$

Da questi esempi possiamo notare che la differenza tra $\mathrm{O}(n)$ ed $\mathrm{O}(n^2)$ è abissale, ma cosa succede se il costo computazionale cresce esponenzialmente, cioè ad esempio è in $\mathrm{O}(2^n)$?

Ovviamente il tempo di esecuzione avrà cifre enormi, infatti già un input di piccole dimensioni come $n = 100$ ha come tempo di esecuzione molto grande:

$$T = \frac{2^{100}}{10^9\ op/s} = 10^3 = 1,26 \cdot 10^21 secondi \approx 3 \cdot 10^3\ anni$$

Dunque possiamo concludere che un algoritmo con un costo esponenziale sia inutilizzabile, infatti anche con le tecnologie che abbiamo ora non troviamo una soluzione a questo problema.

Esercizi

SELECTION SORT:

```
def Selection_Sort(A):
  for i in range(len(A)-1):
    m=i                                   #Θ(1)
    for j in range(i+1,len(A)):
      if A[j] < A[m]:                      #Θ(1)
        m = j                               #Θ(1)
    A[m],A[i]=A[i],A[m]                   #Θ(1)
```

Come prima cosa essendo un ciclo annidato il primo ciclo varrà:

$$\sum_{i=1}^{n} + \Theta(1)$$

Mentre il secondo ciclo varrà $(n - i)$.

Quindi:

$$T(n) = \sum_{i=1}^{n} (\Theta(1) + (n-i) \cdot \Theta(1)) + \Theta(1) = \Theta(n) + \Theta(1) + \Theta(n^2) = \Theta(n^2)$$

BUBBLE SORT:

```
def Bubble_Sort(A)
  for i in range(len(A)-1):
    for j in range(len[A]-i-1):
      if (A[j] > A[j+1]):            #Θ(1)
        A[j],A[j+1]=A[j+1],A[j]       #Θ(1)
```

Come prima cosa essendo un ciclo annidato il primo ciclo varrà:

$$\sum_{i=1}^{n} +\Theta(1)$$

Mentre il secondo ciclo varrà $(n-i)$.

Quindi:

$$T(n) = \sum_{i=1}^{n}(\Theta(1) + (n-i)\cdot\Theta(1)) = \Theta(n) + \Theta(1) + \Theta(n^2) = \Theta(n^2)$$

INSERTION SORT:

```
def Insertion_Sort(A):
  for j in range (1,len(A)):
    x = A[j]                        #Θ(1)
    i = j - 1                       #Θ(1)
    while (i>=0)and(A[i]>x):
      A[i+1] = A[i]                  #Θ(1)
      i=i-1                          #Θ(1)
    A[i+1]=x                        #Θ(1)
```

Come prima cosa essendo un ciclo annidato il primo ciclo varrà:

$$\sum_{j=0}^{n-1} +\Theta(1)$$

Mentre il secondo ciclo potrà fare al più $(j-1)$ volte perché potrebbe fare meno cicli nel caso $A[i] < x$, quindi questo algoritmo abbiamo caso peggiore e migliore diversi:

- Caso peggiore:

$$T(n) = \sum_{j=0}^{n-1}(\Theta(1) + (j-1)\cdot\Theta(1)) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

- Caso migliore:

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

1. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es(1):
  if n<0: n=-n              #θ(1)
  while n:
    if n%2: return 1         #θ(1)
    n -= 2                   #θ(1)
  return 0                  #θ(1)
```

Come prima cosa partiamo dallo studio del ciclo while.

Per far questo dobbiamo controllare cosa accade nelle istruzioni al suo interno. Notiamo subito che se n è dispari eseguirà l'istruzione $return$ e il ciclo terminerà immediatamente. Se succede questo il ciclo while avrà il valore di una semplice condizione(ossia $\Theta(1)$) non avendo terminato neanche un singolo ciclo:

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = 4\Theta(1) = \Theta(1)$$

Possiamo affermare con certezza che questo sia il caso migliore, ma dobbiamo studiare anche il caso peggiore ossia il caso in cui n è un numero pari.

Vediamo il comportamento del ciclo nel caso in cui il numero è pari:

n.iterazione	1	2	3	...	k
Valore di n	n-2	n-4	n-6	...	n-2k

Quindi la condizione necessaria a far terminare il ciclo è $n - 2k = 0$, che con dei semplici calcoli muterà in:

$$n - 2k = 0 \rightarrow n = 2k \rightarrow k = \frac{n}{2}$$

quindi:

$$T(n) = \Theta(1) + \frac{n}{2}(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(\frac{n}{2}) = \Theta(1) \cdot \frac{1}{2}\Theta(n) = \Theta(n)$$

Possiamo dire che il caso peggiore quindi è $\Theta(n)$.

2. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es2(n):
  n = abs(n)       #θ(1)
  x = r = 0        #θ(1)
  while x*x<n:
    x+=1            #θ(1)
    r*=3*x          #θ(1)
  return r         #θ(1)
```

Andiamo a studiare il comportamento del ciclo while:

n.iterazione	1	2	3	...	k
Valore di x	1	2	3	...	k
x*x	1^2	2^2	3^2	...	k^2

Quindi la condizione necessaria a far terminare il ciclo è $k^2 = n$, che con dei semplici calcoli muterà in:

$$k^2 = n \rightarrow k = \sqrt{n}$$

Quindi il costo sarà:

$$T(n) = \Theta(1) + \Theta(1) + \sqrt{(}n)(\Theta(1)) + \Theta(1) = 3\Theta(1) + \Theta(\sqrt{n}) = \Theta(\sqrt{n})$$

In questo caso non avendo condizioni con altri return questo risultato vale sia per il caso peggiore che migliore.

3. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es6(n):
  n = abs(n)         #O(1)
  x = r = 0          #O(1)
  while n>1:
    r += 2            #O(1)
    n = n//3          #O(1)
  return r           #O(1)
```

Anche in questo dovremo andare a osservare il comportamento del ciclo while:

n.iterazione	1	2	3	...	k
n	$\frac{n}{3}$	$\frac{n}{3^2}$	$\frac{n}{3^3}$	...	$\frac{n}{3^k}$

Quindi la condizione necessaria a far terminare il ciclo è $\frac{n}{3^k} = 1$, che con dei semplici calcoli muterà in:

$$\frac{n}{3^k} = 1 \rightarrow n = 3^k \rightarrow k = log_3(n)$$

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) : \Theta(1) + log_3(n)(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(log_3(n)) = \Theta(log_3(n))$$

4. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es4(n):
  n = abs(n)          #O(1)
  x=t=1               #O(1)
  for in range(n):
    t=3*t              #O(1)
  while t>=x:
    x+=2               #O(1)
    t-=2               #O(1)
  return x            #O(1)
```

Andiamo come prima cosa ad analizzare il ciclo for.

Vediamo subito che il ciclo viene ripetuto $n\ volte$ ma ciò che importa a noi è come muta t all'interno del ciclo:

n.iterazione	1	2	3	...	k
t	3^1	3^2	3^6	...	3^n

Quindi possiamo trarre da questa tabella che $t = 3^n$.

Fatto questo andiamo a osservare il comportamento del secondo ciclo, ossia il while:

n.iterazione	1	2	3	...	k
Valore di x	3	5	7	...	1+2k
Valore di t	$3^n - 3$	$3^n - 5$	$3^n - 7$	...	$3^n - (1 + 2k)$

Quindi la condizione necessaria a far terminare il ciclo è $3^n - (1 + 2k) = 2k$, che con dei semplici calcoli muterà in:

$$3^n - (1 + 2k) = 2k \rightarrow 3^n - 1 = 4k \rightarrow k = \frac{3^n - 1}{4}$$

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) = \Theta(1) + n \cdot (\Theta(1)) + \frac{3^n - 1}{4} \cdot \Theta(1) + \Theta(1) = \Theta(n) + \Theta(3^n) = \Theta(3^n)$$

5. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es5(n):
  n = abs(n)       #Θ(1)
  p = 2            #Θ(1)
  while n >=p:
    p = p*p         #Θ(1)
  return p         #Θ(1)
```

Andiamo a studiare il comportamento del ciclo while:

n.iterazione	1	2	3	...	k
p	2^1	2^2	2^6	...	2^{2^k}

Quindi la condizione necessaria a far terminare il ciclo è $2^{2^k} = n$, che con dei semplici calcoli muterà in:

$$2^{2^k} = n \to 2^k = log_2 n \to k = log_2(log_2(n))$$

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) = \Theta(1) + log_2(log_2(n))(\Theta(1)) + \Theta(1) = \Theta(1) + \Theta(log_2(log_2(n))) = \Theta(log_2(log_2(n)))$$

6. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es6(n):
  n = abs(n)              #Θ(1)
  i,j,t,s = 1             #Θ(1)
  while i*i<=n:
    for j in range(t):
      s+=1                  #Θ(1)
    i = i+1                #Θ(1)
    t+=1                   #Θ(1)
  return s                #Θ(1)
```

Andiamo a studiare il comportamento del ciclo while:

n.iterazione	1	2	3	...	k
t	3	5	7	...	(k+1)
i	2	3	4	...	$(k+1)$
i^2	2^2	2^3	2^4	...	$(k+1)^2$

Quindi la condizione necessaria a far terminare il ciclo è $(k+1)^2 = n$, che con dei semplici calcoli muterà in:

$$(k+1)^2 = n \to k = \sqrt{n} - 1$$

Adesso passiamo all'analisi del ciclo for, dove possiamo notare che non è proprio costante perché ad ogni ciclo del while t aumenta di 1 quindi per calcolare il suo numero di iterazioni faremo:

$$\sum_{t=1}^{\sqrt{n}-1} t = \frac{\sqrt{n} \cdot (\sqrt{n} - 1)}{2} = \frac{n - \sqrt{n}}{2}$$

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) = \Theta(1) + \frac{n - \sqrt{n}}{2}(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(n) = \Theta(n)$$

7. Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

```
def es7(n):
  n = abs(n)         #θ(1)
  t,s = n            #θ(1)
  p = 0              #θ(1)
  while s>=1:
    s = s\\4          #θ(1)
    p += 1            #θ(1)
  while n-s>0:
    n -= s           #θ(1)
    t += 5           #θ(1)
  return t          #θ(1)
```

Come prima cosa andiamo a studiare il comportamento del primo ciclo while:

n.iterazione	1	2	3	...	k
$\mathrm{s = n}$	$\frac{n}{4}$	$\frac{n}{4^2}$	$\frac{n}{4^3}$	...	$\frac{n}{4^k}$
$\mathrm{p} = 0$	1	2	3	...	k

Quindi la condizione necessaria a far terminare il ciclo è $\frac{n}{4^k} = 1$, che con dei semplici calcoli muterà in:

$$\frac{n}{4^k} = 1 \rightarrow k = \frac{1}{2} \cdot log(n)$$

Quindi il valore del primo ciclo sarà $log(n)$.

Andiamo ora a studiare il secondo ciclo while:

n.iterazione	1	2	3	...	k
n-s	$n-1$	$n-2$	$n-3$	...	$n-k$

Scriviamo -1 perché s per uscire dal primo ciclo doveva essere uguale ad 1. Ora andiamo a calcolare la condizione necessaria per terminare questo ciclo while, ossia $n-k=0$

$$n - k = 0 \rightarrow k = n$$

Quindi il valore del secondo ciclo sarà n.

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) = \Theta(1) + log(n)(\Theta(1)) + n(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(log(n)) + \Theta(n) = \Theta(n)$$

8.Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

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

Come prima cosa andiamo a studiare il comportamento del primo ciclo while:

n.iterazione	1	2	3	...	k
s = n	$\frac{n}{4}$	$\frac{n}{4^2}$	$\frac{n}{4^3}$	...	$\frac{n}{4^k}$
p = 0	1	2	3	...	k

Quindi la condizione necessaria a far terminare il ciclo è $\frac{n}{4^k} = 1$, che con dei semplici calcoli muterà in:

$$\frac{n}{4^k} = 1 \to k = \frac{1}{2} \cdot log(n)$$

Quindi il valore del primo ciclo sarà $log(n)$.

Andiamo ora a studiare il secondo ciclo while:

n.iterazione	1	2	3	...	k
n-log(n)	$n - log(n)$	$n - 2log(n)$	$n - 3log(n)$	...	$n - (k+1)log(n)$

Come p usiamo $log(n)$ perché nel ciclo primo p andava a contare i cicli del while.

Quindi la condizione necessaria a far terminare il ciclo è $n - (k+1)log(n) = 0$, che con dei semplici calcoli muterà in:

$$n - (k+1)log(n) = 0 \to k = \frac{n}{log(n)}$$

Quindi il valore del secondo ciclo sarà $\frac{n}{log(n)}$.

Avendo questo andiamo a calcolarci il costo dell’algoritmo:

$$T(n) = \Theta(1) + log(n)(\Theta(1)) + \frac{n}{log(n)}(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(log(n)) + \Theta(\frac{n}{log(n)}) = \Theta(\frac{n}{log(n)})$$

9.Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

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

Come prima cosa andiamo a studiare il comportamento del primo ciclo while:

n.iterazione	1	2	3	...	k
s = n	$\frac{n}{5}$	$\frac{n}{5^2}$	$\frac{n}{5^3}$	...	$\frac{n}{5^k}$
p = 2	4	6	8	...	$2+2k$

Quindi la condizione necessaria a far terminare il ciclo è $\frac{n}{4^k} = 1$, che con dei semplici calcoli muterà in:

$$\frac{n}{5^k} = 1 \rightarrow k = log(n)$$

Quindi il valore del primo ciclo sarà $log(n)$.

Prima di studiare il secondo ciclo andiamo a studiare il comportamento di p. Poiché il primo ciclo viene ripetuto $log(n)$ volte anche p verrà ripetuto $log(n)$ volte quindi:

$$p = 2 + 2(log(n))$$
$$p = (2 + 2(log(n)))^2$$

Quindi p varrà $log(n)$.

Ora andiamo a studiare il comportamento del secondo ciclo:

n.iterazione	1	2	3	...	k
i	1	2	3	...	k
i*i*i	1^3	2^3	3^3	...	$(k+1)^3$

Quindi la condizione necessaria a far terminare il ciclo è $(k+1)^3 = n$, che con dei semplici calcoli muterà in:

$$(k+1)^3 = n \rightarrow k = \sqrt[3]{n} - 1$$

Quindi il valore del secondo ciclo sarà $\sqrt[3]{n}$.

Il terzo ciclo invece varrà $log(n)$ visto che ciclerà p volte.

Avendo questo andiamo a calcolarci il costo dell'algoritmo:

$$T(n) = \Theta(1) + log(n)(\Theta(1)) + (\sqrt[3]{n} - 1)((log(n) \cdot \Theta(1)) + \Theta(1)) + \Theta(1) = 3\Theta(1) + \Theta(log(n)) + \Theta(\sqrt[3]{n} \cdot log(n)) + \Theta(\sqrt[3]{n}) = \Theta(\sqrt[3]{}$$