Complessità Temporale di Algoritmi Iterativi

Algoritmi 1 – Lezione 2

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

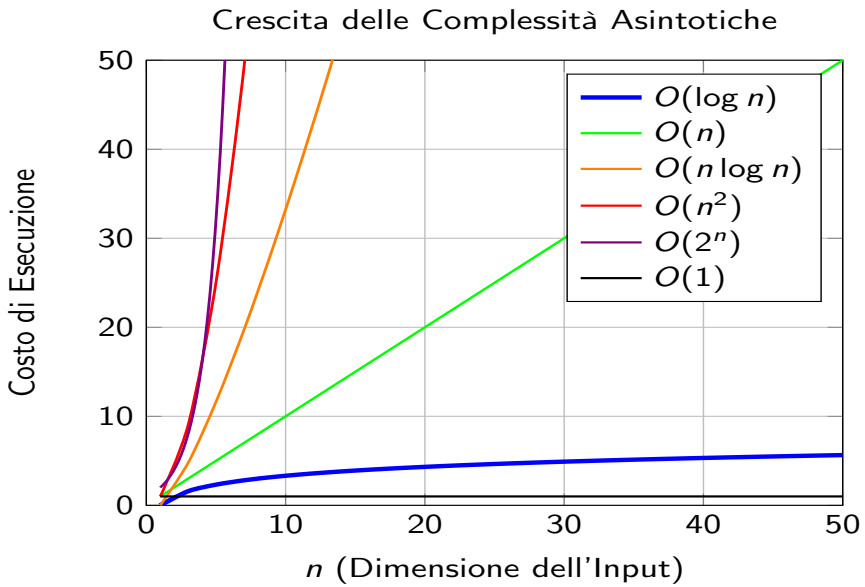
Calcolo del costo computazionale degli algoritmi

- Il **costo computazionale** di un algoritmo rappresenta il suo **tempo di esecuzione** in funzione della **dimensione dell'input**.
- Questo parametro, indicato comunemente con n, è solitamente semplice da individuare:
  - in un **algoritmo di ordinamento** è il numero di dati da ordinare,
  - in un **algoritmo di ricerca** è il numero di elementi da esaminare.
  - ...
- Esistono algoritmi che, per dimensioni dell'input relativamente piccole, presentano un **comportamento specifico**, mentre per dimensioni più grandi il loro comportamento può cambiare. In genere, ciò che ci interessa maggiormente è come l'algoritmo si comporta per **valori elevati** della dimensione dell'input.
- Per questo motivo, la **notazione asintotica** è ampiamente utilizzata nel calcolo del costo computazionale degli algoritmi. Essa fornisce una **stima del comportamento** dell'algoritmo quando l'input è grande, ed è pertanto valida solo in un **contesto asintotico**.

Notazione Asintotica

- **Notazione** O: indica un *limite superiore* sul costo; garantisce che il tempo di esecuzione non supererà mai una certa funzione (caso peggiore).
- **Notazione** Ω: fornisce un *limite inferiore*, assicurando che il costo non sia mai inferiore a una certa funzione.
- **Notazione** Θ: descrive il comportamento preciso di un algoritmo quando il suo costo è limitato sia superiormente sia inferiormente dalla stessa funzione.

Visualizzazione della crescita delle complessità



Pseudocodice

- **Linguaggio informale** per descrivere algoritmi evidenziandone la logica senza la sintassi rigida di un linguaggio di programmazione. Ecco le sue **caratteristiche principali**:
  - Possibilità di usare **linguaggio naturale** per maggiore chiarezza.
  - **Omissione di dettagli implementativi** specifici, per concentrarsi sulla essenza dell'algoritmo.
  - Utilizza **costrutti fondamentali** come *for*, *if then else*, *while*.
  - In questo corso si adotteranno prevalentemente **costrutti ispirati a Python** per la loro intuitività.

Nota: Lo pseudocodice non è pensato per essere eseguibile, ma leggibile.

Regole generali per valutare la complessità computazionale

Nel **modello a costo unitario** adottato in questo corso, si assume che ogni **istruzione elementare abbia costo costante**. Questo consente di stimare la complessità di un algoritmo in modo semplice e intuitivo.

Regole generali:

- **Istruzioni elementari:**
  Operazioni come assegnazioni ($a = b$), lettura/scrittura di variabili, operazioni aritmetiche e logiche su un numero costante di operandi ($a + b$, $a > b$, ecc.) hanno costo $O(1)$.

- **Istruzioni condizionali:**
  ```
  if condizione:
      # Blocco 1
  else:
      # Blocco 2
  ```
  ha un costo pari a:
  - il **costo di verifica della condizione** (tipicamente $O(1)$),
  - più il **massimo tra i costi** del Blocco 1 e del Blocco 2 (poiché non si eseguono entrambi).

- **Istruzioni iterative (cicli):**
  Il costo di un ciclo `for` o `while` è la **somma dei costi di tutte le iterazioni**. Ogni iterazione contribuisce con il costo delle istruzioni al suo interno.

- **Costo totale dell'algoritmo:**
  È la **somma dei costi di tutte le istruzioni**. La complessità asintotica è **dominata dalla parte più costosa**.

Analisi dei Casi: Migliore, Peggiore e Medio

Il tempo di esecuzione di un algoritmo può variare in base all'**input**. Per una valutazione completa, si analizzano tre scenari distinti:

- **Caso migliore:** rappresenta la situazione più favorevole, in cui l'algoritmo raggiunge il **tempo minimo** di esecuzione.
- **Caso peggiore:** descrive la situazione più onerosa, ovvero l'input che causa il **tempo massimo** di esecuzione.
- **Caso medio:** considera il **tempo atteso** su una distribuzione probabilistica degli input, offrendo una stima del **comportamento tipico** dell'algoritmo.
- Esempio: ricerca in una lista ordinata
  - **Caso migliore:** elemento in prima posizione, $O(1)$.
  - **Caso peggiore:** elemento non presente, $O(n)$.
  - **Caso medio:** elemento a metà, $O(n)$.

Considerazioni pratiche:

- L'analisi si concentra spesso sul **caso peggiore**, poiché:
  - è più semplice da stimare rispetto al caso medio;
  - garantisce valutazioni conservative utili in fase di progettazione.
- Quando disponibile, l'**analisi del caso medio** fornisce informazioni più realistiche sulle prestazioni.

Operazioni Python a complessità non costante I

In Python, alcune operazioni hanno una **complessità variabile**, cioè il loro costo dipende dalla struttura dati utilizzata. Un caso tipico è l'**operatore** `in` (appartenenza), la cui efficienza cambia a seconda che venga usato su una `lista`, un `insieme` o un `dizionario`.
Un esempio significativo è l'operatore *in*:

Struttura Dati	**Complessità**
Liste	$O(n)$ (ricerca sequenziale)
Set e Dizionari	$O(1)$ in media $O(n)$ nel caso peggiore

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.

Operazioni Python a complessità non costante II

Un'altra categoria importante è quella delle **operazioni di inserimento e cancellazione**, che variano significativamente in base alla struttura dati usata.

Struttura Dati	**Complessità Inserimento / Cancellazione**
Lista	$O(n)$ (se all'inizio o nel mezzo)
Set / Dizionario	$O(1)$ media
	$O(n)$ nel caso peggiore

Operazioni Python a complessità non costante III

Altre operazioni Python comuni con complessità non costante includono:

Concatenazione di liste:

- L'istruzione `C = A + B`, dove A ha lunghezza n e B lunghezza m, ha costo $O(n+m)$.

Slicing su liste o stringhe:

- `A[a:b]` ha costo $O(b-a)$, poiché gli elementi vengono copiati in una nuova lista.

Modifica di stringhe:

- Le stringhe sono **immutabili**: ogni modifica genera una nuova stringa.
- L'operazione `s = s + 'a'` ha costo $O(n)$, con $n = \text{len}(s)$.

Suggerimento: se è necessario modificare frequentemente una stringa, conviene convertirla in **lista di caratteri** con $lista = list(s)$, modificarla e poi riconvertirla in stringa con `"".join(lista)`.

Esempio 1

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

Esempio 1

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

Analizziamo i casi possibili:

1. Se $n > 100$, l'algoritmo restituisce il valore 1 e termina immediatamente. Costo: $O(1)$.
2. Se $n \leq 100$, l'algoritmo esegue al più 100 iterazioni, ciascuna di costo $O(1)$. Costo complessivo: $O(1)$.

Complessità asintotica: $O(1)$

Esempio 2

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

Esempio 2

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

Analizziamo i casi limite:

1. **Caso ottimo:** si verifica quando n è dispari. L'algoritmo termina immediatamente restituendo 1. Costo: $O(1)$.
2. **Caso pessimo:** si verifica quando n è pari. Il ciclo `while` itera per circa $n/2$ volte prima di restituire 0. Costo: $O(n)$.

Esempio 3

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

Esempio 3

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

Analiziamo il numero di iterazioni del ciclo *while*:

- Dopo i iterazioni si ha $x = i$
- il ciclo termina quando $i^2 \geq n$ ovvero per $x \approx \sqrt{n}$

Complessità asintotica: $O(\sqrt{n})$.

Esempio 4

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

Esempio 4

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

Analizziamo il numero di iterazioni del ciclo `while`:

- Dopo i iterazioni, il valore di n sarà al più $\frac{n}{3^i}$
- Il ciclo termina quando $\frac{n}{3^i} < 1$, ovvero per $i \approx \log_3 n$

Complessità asintotica: $O(\log n)$

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
```

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
```

- Il ciclo `for` richiede tempo $O(n)$.
- Alla i-esima iterazione del ciclo `for` si ha $t = 3^i$. Al termine, quindi, $t = 3^n$.
- All'inizio del ciclo `while`, si ha $t = 3^n$ e $x = 1$, quindi la differenza tra i due è circa 3^n.
- A ogni iterazione del ciclo `while`, la differenza tra t e x si riduce di 4. Infatti:
  - Il valore di x viene incrementato di 2 ($x = x + 2$)
  - Il valore di t viene decrementato di 2 ($t = t - 2$)
- Il ciclo `while` termina dopo i iterazioni, dove $3^n = 4i$, ovvero $i = \frac{3^n}{4} = O(3^n)$.

Complessità asintotica: $O(n) + O(3^n) = O(3^n)$

Esempio 6

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

Esempio 6

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

Analizziamo il comportamento del ciclo `while`:

- All'inizio del ciclo, si ha $p = 2$, e a ogni iterazione il valore di p viene elevato al quadrato.
- Alla i-esima iterazione, si ha $p = 2^{2^i}$.
- Il ciclo termina quando $2^{2^i} \geq n$, ovvero quando $i \approx \log_2 \log_2 n$.

Complessità asintotica: $O(\log \log n)$

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
```

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
```

Ad ogni iterazione del ciclo `while` accadono due cose:

- Il ciclo `for` viene eseguito t volte.
- Le variabili u e t vengono incrementate di 1.

Contributo complessivo del ciclo `for`:
Il numero totale di iterazioni è:

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

Ogni iterazione ha costo costante $O(1)$, quindi il contributo complessivo del ciclo `for` è $O(n^2)$.

Contributo delle altre istruzioni del ciclo `while`:
Le istruzioni `u += 1` e `t += 1` vengono eseguite una volta per ciascuna delle n iterazioni del ciclo `while`, e hanno costo costante. Contributo complessivo: $O(n)$.

Complessità asintotica: $O(n^2) + O(n) = O(n^2)$

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
```

Analizziamo separatamente il contributo dei due cicli `while`:

- **Primo ciclo `while`:**
  Ad ogni iterazione, il valore di s viene diviso per 4. Il ciclo termina quando $s < 1$, ovvero dopo circa $\log_4 n$ iterazioni.
  Al termine, $p = \lfloor \log_4 n \rfloor$, e il costo del ciclo è: $O(\log n)$
- **Secondo ciclo `while`:**
  - Ad ogni iterazione, n viene decrementato di $p = \lfloor \log_4 n \rfloor$.
  - Dopo i iterazioni, si ha $n = n - i \cdot p$.
  - Il ciclo termina quando $n - i \cdot p \leq 0$, ovvero per $i \approx \dfrac{n}{\log_4 n}$.
  - Il numero di iterazioni è quindi:

$$O\left(\frac{n}{\log n}\right)$$

Complessità asintotica: $O(\log n) + O\left(\frac{n}{\log n}\right) = O\left(\frac{n}{\log n}\right)$

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
```

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
```

Analizziamo separatamente il contributo dei due cicli `while`:

- **Primo ciclo `while`:**
  La variabile s, inizialmente uguale a n, viene divisa per 5 a ogni iterazione. Il ciclo termina quando $s < 1$, quindi il numero di iterazioni è:

  $$O(\log_5 n) = O(\log n)$$

  Ad ogni iterazione, p viene incrementato di 2, quindi al termine si ha $p = O(\log n)$.
- **Calcolo di p:**
  Dopo il primo ciclo, p viene elevato al quadrato: $p = O(\log^2 n)$.
- **Secondo ciclo `while`:**
  - Il ciclo termina quando $i^3 \geq n$, quindi si esegue per: $O(n^{1/3})$
  - Ad ogni iterazione, il ciclo `for` esegue $p = O(\log^2 n)$ iterazioni.
  - Costo del secondo ciclo: $O(n^{1/3} \cdot \log^2 n)$

Complessità asintotica: $O(\log n) + O(n^{1/3} \cdot \log^2 n) = O(n^{1/3} \cdot \log^2 n)$

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
```

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
```

Analizziamo separatamente i due cicli `while` annidati:

- **Ciclo esterno** `while j >= 2`**:**
  - La variabile `j` parte da 512 e viene dimezzata a ogni iterazione (`j = j // 2`).
  - Il ciclo termina quando `j < 2`, ovvero dopo circa $\log_2 512 = 9$ iterazioni.
  - Il numero di iterazioni è costante, quindi il costo complessivo del ciclo esterno è $O(1)$.
- **Ciclo interno** `while k * k <= n`**:**
  - La variabile `k` parte da 1 e viene raddoppiata a ogni iterazione (`k = 2 * k`).
  - Il ciclo termina quando $k^2 > n$, cioè quando $k > \sqrt{n}$.
  - Dopo i iterazioni, si ha $k = 2^i$, quindi:

$$2^i > \sqrt{n} \quad \Rightarrow \quad i > \log_2 \sqrt{n} = \frac{1}{2} \log_2 n$$

  - Il numero di iterazioni è dunque $O(\log n)$.

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

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

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

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

2
```
def f(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print(i, j, k)
```

3
```
def f(n):
    i = 1
    while i * i <= n:
        x = i
        while x > 0:
            x //= 2
        i += 1
    return i*x
```

Esercizi

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

1

```
def f(n):
    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
```

2

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

Esercizi

- Calcolane la complessità.
- Determina cosa fa
- Proponi un algoritmo alternativo che abbia complessità inferiore

1
```
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
```

2
```
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
```

Esercizi

- Calcolane la complessità.
- Determina cosa fa
- Proponi un algoritmo alternativo che abbia complessità inferiore

1

```
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
```

2

```
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
```