Dati e algoritmi

0. Indice

[Lezione 01]

1. Nozioni

1.1. Problema computazionale

Un problema computazionale è costituito da

- un insieme I di *istanze* (i *possibili input*)
- un insieme S di *soluzioni* (i *possibili output*)
- una relazione Π che a ogni istanza $i \in I$ associa *una o più* soluzioni $s \in S$

> **Osservazione**
>
> Π è un sottoinsieme del prodotto cartesiano $I \times S$

1.1.1. Esempi

Somma di Interi ($\mathbb{Z}$)

- $\mathcal{I} = \{(x, y) \ : \ x, y \in Z\}$;
- $\mathcal{S} = Z$;
- $\Pi = \{((x, y), s) \ : \ (x, y) \in \mathcal{I}, s \in \mathcal{S}, s = x + y\}$.
  Ad es: $((1, 9), 10) \in \Pi$; $((23, 6), 29) \in \Pi$ $((13, 45), 31) \notin \Pi$

Ordinamento di array di interi

- $\mathcal{I} = \{A \ : \ A = \text{array di interi}\}$;
- $\mathcal{S} = \{B \ : \ B = \text{array ordinati di interi}\}$;
- $\Pi = \{(A, B) \ : \ A \in \mathcal{I}, B \in \mathcal{S}, B \text{ contiene gli stessi interi di } A\}$.
  Ad es.
  $(< 43, 16, 75, 2 >, < 2, 16, 43, 75 >) \in \Pi$
  $(< 7, 1, 7, 3, 3, 5 >, < 1, 3, 3, 5, 7, 7 >)$
  $(< 13, 4, 25, 17 >, < 11, 27, 33, 68 >)$

Ordinamento di array di interi (ver.2)

- $\mathcal{I} = \{A \ : \ A = \text{array di interi}\}$;
- $\mathcal{S} = \{P \ : \ P = \text{permutazioni}\}$;
- $\Pi = \{(A, P) \ : \ A \in \mathcal{I}, P \in \mathcal{S}, P \text{ ordina gli interi di } A\}$.
  Ad es.
  $(< 43, 16, 75, 2 >, < 4, 2, 1, 3 >) \in \Pi$
  $(< 7, 1, 7, 3, 3, 5 >, < 2, 4, 5, 6, 1, 3 >) \in \Pi$
  $(< 7, 1, 7, 3, 3, 5 >, < 2, 5, 4, 6, 1, 3 >) \in \Pi$
  $(< 13, 4, 25, 17 >, < 1, 2, 4, 3 >)$

Osservazione

- Istanze diverse possono avere la stessa soluzione (come la somma)
- Un'istanza può avere diverse soluzioni (come l'ordinamento ver. 2)

1.1.2. Esercizi

Esercizio

Specificare come problema computazionale Π la verifica se due insiemi finiti di oggetti da un universo U sono disgiunti oppure no.

$I \equiv \{(A, B) : A, B \subseteq U, \ A, B \text{ finiti}\}$
$S \equiv \{true, \ false\}$
$\Pi \equiv \{((A, B), s) \text{ se } A \cap B = \emptyset \text{ allora } s = true, \text{ se } A \cap B \neq \emptyset \text{ allora } s = false\} \quad s \in S, (A, B) \in I$

Esercizio

Specificare come problema computazionale Π la ricerca dell'inizio e della lunghezza del più lungo segmento di 1 consecutivi in una stringa binaria.

$I \equiv \{A : A \text{ è una stringa binaria}\}$
$S \equiv \{(i,l) : i,l \in \mathbb{N}_0\}$
$\Pi \equiv \{(A,(i,l)) : i \text{ la casella di inizio del segmento di 1 consecutivi più numeroso}, l \text{ la lunghezza del segmento di 1 consecutivi più lungo}\}$

1.2. Algoritmo e modello di calcolo

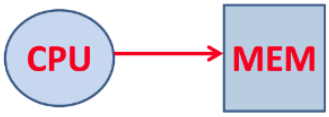
1.2.1. Algoritmo

Un *algoritmo* procedura computazionale ben definita che trasforma un dato *input* in un *output* eseguendo una sequenza finita di *operazioni elementari*.
□

L'algoritmo fa riferimento a un *modello di calcolo*, ovvero un'astrazione di computer che definisce l'insieme di operazioni elementari.

Le operazioni elementari sono: assegnamento, operazioni logiche, operazioni aritmetiche, indicizzazione di array, return di un valore da parte di un metodo, ecc.

1.2.2. Modello di calcolo RAM (Random Access Machine)



In questo modello input, output, dati intermedi (e il programma) si trovano in memoria.

Un algoritmo A risolve un *problema computazionale* $\Pi \subseteq I \times S$ *se*:

1. A calcola una funzione da I a S e quindi,
   - riceve come input istanze $i \in I$
   - produce come output soluzioni $s \in S$
2. Dato $i \in I$, A produce in output $s \in S$ tale che $(i,s) \in \Pi$.
   Se Π associa più soluzioni a una istanza i, per tale istanza A ne calcola una (quale, dipende da come è stato progettato).

[Lezione 02]

1.3. Pseudocodice

Per semplicità e facilità di analisi, descriviamo un algoritmo utilizzando uno pseudocodice strutturato come segue:

> **Algoritmo** *nome*(*parametri*)
> **Input**: *breve descrizione dell'istanza di input*
> **Output**: *breve descrizione della soluzione restituita in output*
> *Descrizione chiara dell'algoritmo tramite costrutti di linguaggi di programmazione e, se utile ai fini della chiarezza, anche tramite linguaggio naturale, dalla quale sia facilmente determinabile la sequenza di operazioni elementari eseguita per ogni dato input.*

"Algoritmo nome (parametri)" è la firma (signature) dell'algoritmo; i parametri sono l'input.
Si può usare il linguaggio naturale per cose semplici, che sarebbero lunghe da scrivere con i costrutti del linguaggio di programmazione.

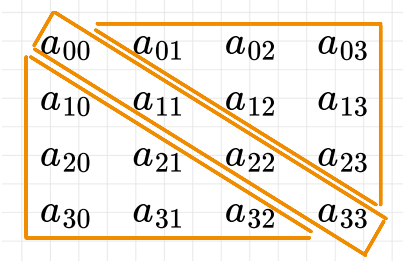
Usare *sempre* questa struttura per lo pseudocodice!

Per maggiori dettagli fare riferimento alla dispensa sulla Specifica di Algoritmi in Pseudocodice, disponibile sul Moodle del corso.

1.3.1. Esempio - Trasposta di una matrice $n \times n$ di interi

Problema computazionale: $\Pi \subseteq I \times S$
$I \equiv \{A:$ matrice $n \times n$ di interi$\}$
$S \equiv \{B$ matrice $n \times n$ di interi$\}$
$\Pi \equiv \{(A,B) : A \in I, B \in S, B = A^t\}$

Algoritmo: idea per $n = 4$



Scambio ciascuna entry a_{ij} della matrice triangolare superiore ($a_{ij} : i = 0, \ldots, n-2, j > i$) con a_{ji} (l'omologa del triangolo inferiore)

Algoritmo transpose(A)
input matrice A $n \times n$ di interi a_{ij}, $0 \leq i,\ j < n$

output matrice A^t

```
for i <- 0 to n-2 do{
    for j <- i+1 to n-1 do{
        scambia a_ij con a_ji
    }
}
return A
```

1.4. Taglia di un'istanza

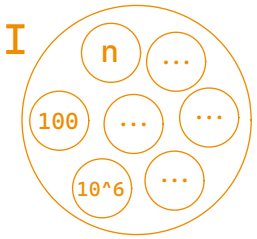
L'analisi di un algoritmo (ad es., la determinazione della sua complessità) viene solitamente fatta *partizionando le istanze in gruppi in base alla loro taglia (size)*, in modo che le istanze di un gruppo siano tra loro *confrontabili*.

> La *taglia di un'istanza* è espressa da uno o più valori che ne caratterizzano la grandezza.

1.4.1. Esempi

- Se ho n elementi da ordinare con un MergeSort, la taglia è n. In questo caso fa riferimento alla dimensione fisica dell'istanza.

La taglia serve per dividere in gruppi omogenei da analizzare separatamente.



- Ordinamento di un array:
  Taglia n = numero di elementi dell'array
- Trasposta di una matrice:
  Prendendo una generica matrice $n \times m$ ho più modi di definire la taglia, in base al tipo di analisi che si vuole condurre:
  - Taglia $x = n * m$
  - Taglia (n, m), con n e m rispettivamente il numero di righe e numero di colonne
  - (Se $n = m$) Taglia $n =$ numero di righe/colonne

1.5. Struttura dati

Le strutture dati sono usate dagli algoritmi per organizzare e accedere in modo sistematico ai dati di input e ai dati generati durante l'esecuzione.

1.5.1. Struttura dati - definizione

Una *struttura dati* è una collezione di oggetti corredata di *metodi* di accesso e/o modifica.

Vi sono due *livelli di astrazione*:

1. *Livello logico*: specifica l'organizzazione logica degli oggetti della collezione, e la relazione input-output di ciascun metodo (a questo livello si parla di *Abstract Data Type* o ADT)
2. *Livello fisico*: specifica il layout fisico dei dati e la realizzazione dei metodi tramite algoritmi.

1.5.1.1. Esempio

In Java a livello logico ho (gerarchia di) *interfacce*, mentre a livello fisico ho (gerarchia di) *classi*.
□

1.5.2. Esercizio

Siano A e B due array di n e m interi, rispettivamente. Scrivere un algoritmo in pseudocodice per calcolare il seguente valore:

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \frac{a_n}{A[i] \cdot B[j]}$$

1.5.3. Esercizio

Sia A un array di n interi distinti, ordinato in senso crescente, e x un valore intero. Scrivere due algoritmi in pseudocodice che implementano la ricerca binaria per trovare x in A. Entrambi gli algoritmi restituiscono l'indice i tale che $A[i] = x$, se x appare in A, e -1 altrimenti. Il primo algoritmo deve essere iterativo e usare un solo ciclo, e il secondo algoritmo deve essere ricorsivo.

2. Analisi degli algoritmi

L'analisi di un algoritmo A mira a studiarne l'*efficienza* e l'*efficacia*. In particolare, essa può valutare la ***complessità*** di

tempo e *spazio*, e la ***correttezza*** di *terminazione* (se termina o rimane in un loop) e della *soluzione del problema computazionale*.

Noi ci concentreremo sulla complessità di tempo e sulla correttezza della soluzione del problema computazionale.

2.1. Complessità in tempo

L'*obiettivo* è *stimare il tempo di esecuzione ("running time") di un algoritmo* al fine di valutarne l'efficienza e poterlo confrontare con altri algoritmi per lo stesso problema.

2.1.1. Requisiti per la complessità in tempo

La complessità in tempo deve:

1. Riguardare *tutti gli input*;
2. Permettere di *confrontare algoritmi* (senza necessariamente determinare il tempo di esecuzione esatto);
3. Essere *derivabile dallo pseudocodice*.

2.1.1.1. Esempio

Stimare sperimentalmente il tempo di esecuzione di un algoritmo (ad esempio in Java con `System.currentTimeMillis()`) è utile, ma non soddisfa i requisiti. **Perché?**

- Non si possono considerare tutti gli input (se infiniti);
- Richiede di implementare l'algoritmo con un programma e l'impatto dell'implementazione può influenzare il confronto tra algoritmi;
- La stima dipende dall'ambiente hardware/software.

2.1.1.2. Approccio che soddisfa i requisiti

- Analisi al *caso pessimo* (*worst-case*) in funzione della taglia dell'istanza (requisiti 1 e 2)
- Conteggio delle operazioni elementari nel modello RAM (requisiti 2 e 3)
- Analisi asintotica (per semplificare il conteggio)

> **Osservazione**
>
> Esiste anche l'analisi al caso medio (*average case*) e l'analisi probabilistica.

2.1.2. Complessità (in tempo) al caso pessimo - definizione

Sia A un algoritmo che risolve $\Pi \subseteq I \times S$.

> La complessità (in tempo) al caso pessimo di A è una funzione $t_A(n)$ definita come *il massimo numero di operazioni elementari che A esegue per risolvere un'istanza di taglia n*.

In altre parole, se chiamiamo $t_{A,i}$ il numero di operazioni eseguite da A per l'istanza i, abbiamo che

$$t_A(n) = max\{t_{A,i} : \text{ istanza } i \in I \text{ di taglia } n\}$$

La definizione rispetta i tre requisiti definiti sopra.

2.1.3. Difficoltà nel calcolo di $t_A(n)$

Determinare $t_A(n)$ per ogni n è arduo, se non impossibile, perché è *difficile identificare l'istanza peggiore di taglia n* e perché è *difficile contare il massimo numero di operazioni richieste per risolvere tale istanza* peggiore (il conteggio richiederebbe anche una specifica dettagliata del set di operazioni elementari del modello RAM).
Ma *non è necessario determinare esattamente $t_A(n)$*, infatti il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso, in più le diverse operazioni elementari del modello RAM possono avere impatto diverso sui tempi di esecuzione a seconda delle architetture.
Ci accontentiamo dei *limiti superiori* e i *limiti inferiori* a $t_A(n)$.

2.1.4. Esempio (`arrayMax`)

Algoritmo `arrayMax(A)`
Input array $A[0 \div n-1]$ di $n \geq 1$ interi
Output: max intero in A

```
currMax<-A[0]
for i<-1 to n-1 do{
    if(currMax < A[i]) then currMax<-A[i];
}
return currMax
```

Taglia dell'istanza: n (ragionevole)

2.1.4.1. Stima di $t_{\text{arrayMax}}(n)$

È facile vedere che per una qualsiasi istanza di taglia n:

- al di fuori del ciclo for `arrayMax` esegue un numero costante (rispetto a n) di operazioni;
- in ciascuna delle $n-1$ iterazioni del ciclo for si esegue un numero costante di operazioni.
  Esistono allora quattro costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n valga $t_{\text{arrayMax}}(n) \leq c_1 n + c_2$ (cioè il *limite superiore*) e $t_{\text{arrayMax}}(n) \geq c_3 n + c_4$ (cioè il *limite inferiore*). Non è necessario stimare le quattro costanti per le stesse ragioni per cui non è necessario stimare esattamente la complessità.

2.2. Analisi asintotica

Si ricorre quindi all'analisi asintotica, ignorando i *fattori moltiplicativi costanti* (rispetto alla taglia dell'istanza) e i *termini additivi non dominanti*.

2.2.1. Esempio (`arrayMax`)

Riprendendo l'esempio, nel caso di `arrayMax` possiamo affermare che $t_{\text{arrayMax}}(n)$ è *al più* proporzionale a n (limite superiore) ed è anche *almeno* proporzionale a n (limite inferiore), ne consegue quindi che $t_{\text{arrayMax}}(n)$ è *proporzionale* a n (limite stretto).
□

> Per esprimere affermazione come quelle fatte sopra si usano gli **ordini di grandezza**: $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$.

[lezione 03]

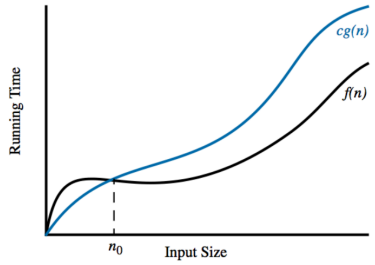
2.3. Ordini di grandezza

Siano , funzioni da $\mathbb{N}$ a $\mathbb{R}^+ \cup \{0\}$.

2.3.1. O-grande

> $f(n) \in O(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, *costanti rispetto a n*, tali che
>
> $$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè se $f(n)$ è al più proporzionale a $g(n)$, ovvero se $f(n)$ non cresce asintoticamente più di $c \cdot g(n)$.



2.3.1.1. Esempi

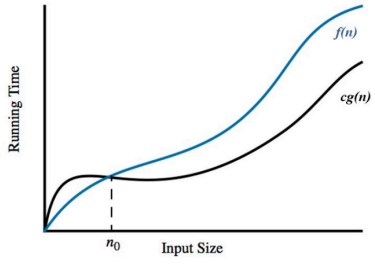
$f(n)$	$O(\cdot)$	c	n_0
$3n+4$ per $n \geq 1$	$O(n)$	4	4
$n+2n^2$ per $n \geq 1$	$O(n^2)$	3	1
2^{100} per $n \geq 1$	$O(1)$	2^{100}	1
$c_1 n + c_2$ per $n \geq 1$, $c_1, c_2 > 0$	$O(n)$	$c_1 + c_2$	1

Si può dire che $3n+4 \in O(n^5)$, $c=7$, $n_0=1$, ma non sarebbe molto utile.

2.3.2. Omega-grande

> $f(n) \in \Omega(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, *costanti rispetto a n*, tali che
>
> $$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè $f(n)$ è almeno proporzionale a $g(n)$.



2.3.2.1. Esempi

$f(n)$	$\Omega(\cdot)$	c	n_0
$3n+4$ per $n \geq 1$	$\Omega(n)$	1	4
$n+2n^2$ per $n \geq 1$	$\Omega(n^2)$	1	1
2^{100} per $n \geq 1$	$\Omega(1)$	2^{100}	1
$c_1 n + c_2$ per $n \geq 1$, $c_1, c_2 > 0$	$\Omega(n)$	1	1

2.3.3. Theta

> $f(n) \in \Theta(g(n))$ se
>
> $$f(n) \in O(g(n)) \quad \text{e} \quad f(n) \in \Omega(g(n))$$

Cioè $f(n)$ è (*esattamente*) proporzionale a $g(n)$.

2.3.3.1. Esempi

- $f(n) = 3n + 4 \in \Theta(n)$;
- $f(n) = n + 2n^2 \in \Theta(n^2)$;
- $f(n) = 2^{100} \in \Theta(1)$;
- $t_{\text{arrayMax}}(n) \in \Theta(n)$.

2.3.4. o-piccolo

> $f(n) \in o(g(n))$ se
>
> $$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Cioè $f(n)$ è *asintoticamente* più piccola (cresce meno) di $g(n)$.

2.3.4.1. Esempi

- $f(n) = 100n$ per $n \geq 1 \implies f(n) \in o(n^2)$;
- $f(n) = \frac{3n}{\log_2 n}$ per $n \geq 1 \implies f(n) \in o(n)$.

2.3.5. Proprietà degli ordini di grandezza

1. $max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ per ogni $f(n), g(n) : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$;
2. $\sum_{i=0}^{k} a_i n^i \in \Theta(n^k)$, se $a_k > 0$, k, a_i costanti, e $k \geq 0$.
   Ad esempio: $(n+1)^5 \in \Theta(n^4)$.
   Cioè tengo il termine con l'esponente maggiore;
3. $\log_b n \in \Theta(\log_a n)$, se $a, b > 1$ sono costanti;

   > **Osservazione**
   > La proprietà deriva dalla relazione $\log_b n = (\log_a n)(\log_b a)$ e, grazie a essa, *la base dei logaritmi*, se costante, *si omette negli ordini di grandezza*, a meno che il logaritmo non sia all'esponente.

4. $n^k \in o(a^n)$, se $k > 0$, $a > 1$ sono costanti;
5. $(\log_b n)^k \in o(n^h)$ se b, k, h sono costanti, con $b > 1$ e $h, k > 0$.

2.3.6. Strumenti matematici

2.3.6.1. Parte bassa

> $\forall x \in \mathbb{R}$, si definisce $\lfloor x \rfloor$ come il più grande intero tale che sia $\leq x$.

2.3.6.1.1. Esempi

- $\lfloor \frac{3}{2} \rfloor = 1$;
- $\lfloor 3 \rfloor = 3$.

2.3.6.2. Parte alta

> $\forall x \in \mathbb{R}$, si definisce $\lceil x \rceil$ come il più piccolo intero tale che sia $\geq x$.

2.3.6.2.1. Esempi

- $\lceil \frac{3}{2} \rceil = 2$;
- $\lceil 3 \rceil = 3$.

2.3.6.3. Modulo

> $\forall x, y \in Z$, con $y \neq 0$, si definisce $x \bmod y$ come il resto della divisione intera x/y (l'operatore "%" in Java).

2.3.6.3.1. Esempi

- $29 \bmod 7 = 4$;
- $80 \bmod 4 = 0$.

2.3.6.4. Sommatorie notevoli

$\forall n \in \mathbb{Z}$ e $a \in \mathbb{R}$, con $n \geq 0$ e $a > 0$ vale:

- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$;
- $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$ per $a > 1$;
- $\sum_{i=1}^{n} a^i = \frac{a^{n+1}-1}{a-1} - 1 \in \Theta(a^n)$ per $a > 1$.

2.4. Analisi di complessità in pratica

Dato un algoritmo A e detta $t_A(n)$ la sua complessità al caso pessimo, si cercano limiti asintotici superiori e/o inferiori a $t_A(n)$.

2.4.1. Limite superiore (upper bound) - definizione

$$t_A(n) \in O(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande" e per ciascuna istanza di taglia n l'algoritmo esegue $\leq c \cdot f(n)$ *operazioni*, con c costante (e che non serve determinare).

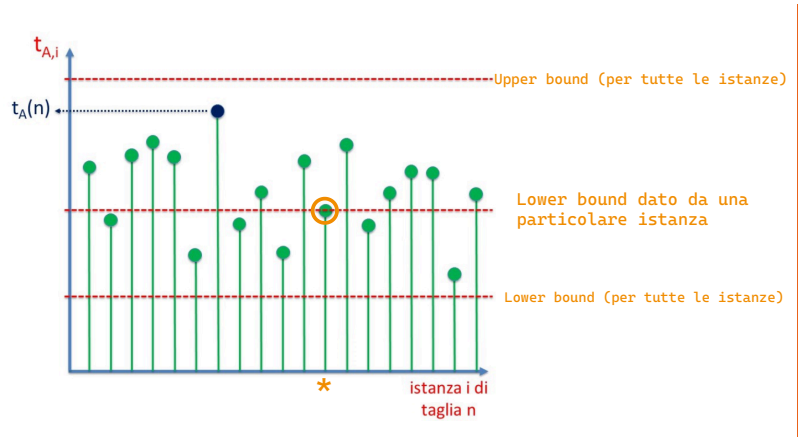
2.4.2. Limite inferiore (lower bound) - definizione

$$t_A(n) \in \Omega(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande", *esiste* un'istanza di taglia n per la quale l'algoritmo esegue $\geq c \cdot f(n)$ *operazioni*, con c costante (e che non serve determinare).
In alcuni casi è comodo argomentare che per *ciascuna* istanza di taglia n l'algoritmo esegue $\geq c \cdot f(n)$ operazioni.

> ⚠ **Attenzione**
>
> Sia che si provi $t_A(n) \in O(f(n))$ o che si provi $t_A(n) \in \Omega(f(n))$
>
> - $f(n)$ deve essere più vicino possibile alla complessità vera (*tight bound*)
> - $f(n)$ deve essere più semplice possibile, quindi senza costanti e termini additivi di ordine inferiore, solo con i *termini essenziali*!

2.4.3. Limiti superiori e inferiori



2.4.4. Terminologia per complessità

- logaritmica: $\Theta(\log n)$, base 2 o costante > 1
- lineare: $\Theta(n)$
- quadratica: $\Theta(n^2)$
- cubica: $\Theta(n^3)$
- polinomiale $\Theta(n^c)$, $c > 0$ costante
- esponenziale: $\Omega(a^n)$, $a > 1$ costante
- polilogaritmica: $\Theta((\log n)^c)$, $c > 0$ costante

2.4.5. Esempio (prefix averages)

Si consideri il seguente problema computazionale.
Dato un array di n interi $X[0 \dots n-1]$ calcolare un array $A[0 \dots n-1]$ dove $A[i] = \left(\sum_{j=0}^{i} X[j]\right) \frac{1}{i+1}$, per $0 \leq i < n$.

Vedremo adesso due algoritmi di cui uno banale e inefficiente, poi uno più furbo ed efficiente. Per entrambi gli algoritmi vale la seguente specifica di input-output:
Input: $X[0 \dots n-1]$ array di n interi
Output: $A[0 \dots n-1] : A[i] = \left(\sum_{j=0}^{i} X[j]\right) \frac{1}{i+1}$ per $0 \leq i < n$.

2.4.5.1. Algoritmo inefficiente

Algoritmo `prefixAverages1`

```
for i <- 0 to n-1 di{
    a <- 0;
    for j <- 0 to i do {
        a <- a+X[j];
    }
    A[i] <- a/(i+1);
}
return A
```

- Fuori dal for esterno: $\Theta(1)$ operazioni
- Per ciascuna iterazione i del for esterno $(i = 0, \dots, n-1)$:
  - $\Theta(i+1)$ operazioni nel for interno
  - $\Theta(1)$ altre operazioni

La complessità è quindi:
$t_{pA1}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} i + 1\right) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{(n-1)n}{2}\right) = \Theta(n^2)$.

2.4.5.2. Algoritmo efficiente

Algoritmo `prefixAverages2`

```
s <- 0;
for i <- 0 to n-1 do{
    s <- s + X[i];
    A[i] <- s/(i+1);
}
return A
```

- Fuori dal for: $\Theta(1)$ operazioni
- Iterazioni i del for $(i = 0, \ldots, n-1)$: $\Theta(1)$ operazioni

La complessità è quindi: $t_{pA2}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} 1\right) = \Theta(n)$

Possiamo affermare che $t_{pA2}(n) \in o(t_{pA1}(n))$, cioè è migliore.