Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Tempo di esecuzione di un algoritmo

Dato un algoritmo, vogliamo avere una stima di quanto *tempo* ci mette ad essere completato su una data macchina.

Esempio 1: ipotizziamo di avere un algoritmo in grado di effettuare la somma dei primi n numeri naturali, con n dato in input, e di voler sapere quanto tempo impiega per essere eseguito su una data macchina.

ipotizziamo che le istruzioni

- aritmetiche
- logiche
- scrittura/lettura di memoria

siano considerate elementari ed impieghino tutte lo stesso tempo (e.g. 1 ms) per essere eseguite su una data architettura.

Date queste premesse, supponiamo di voler sapere quanto tempo impiega il seguente codice per essere eseguito.

In [ ]:
```
// algoritmo 1.1
int main()
{

    int n;
    printf("inserire numero: ");
    scanf("%d", &n);

    int s = 0;
    for(int i = 1; i<=n; i++)
        s = s + i;

    printf("la somma dei numeri da 1 a %d è %d\n", n, s);

    return 0;
}
```

Supponendo di inserire `5` come valore di `n`, il seguente algoritmo effettuerà:

- 2 inizializzazioni `int s = 0; int i = 1;`
- 6 confronti `i<=5`

- 5 incrementi `i++`, che possono essere visti come 5 assegnazioni + 5 operazioni aritmetiche
- 5 somme `s + i`
- 5 assegnazioni `s = ...`

totale: $2 + 6 + 2 \cdot 5 + 5 + 5 = 28$ operazioni da 1 ms, per un totale di 28 ms (Nota: solo per semplicità, abbiamo trascurato il tempo per le operazioni di lettura dalla memoria).

Generalizzando ad un n qualsiasi:

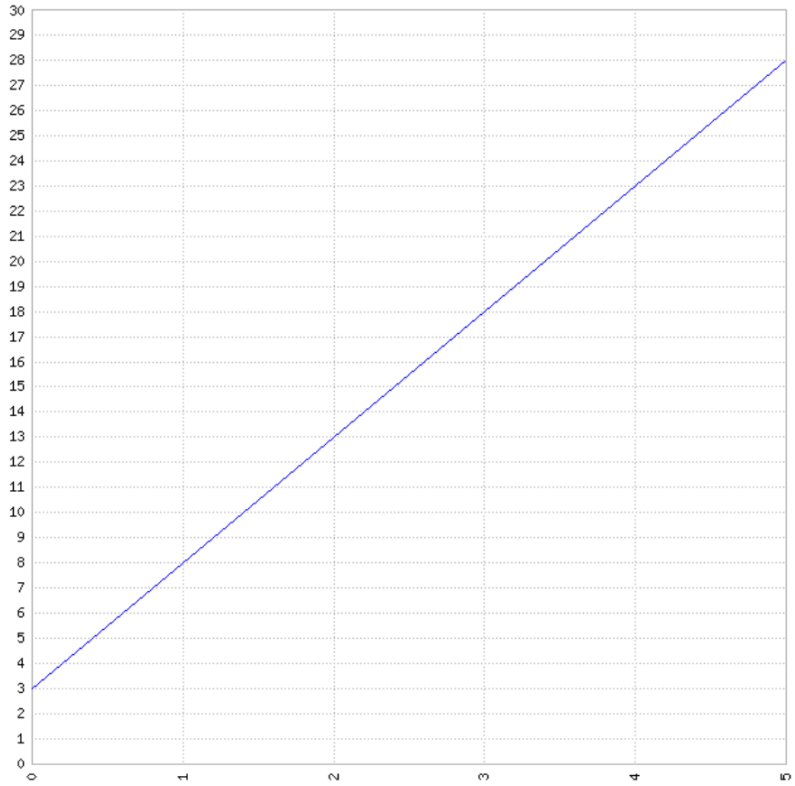
$2 + n + 1 + 2n + n + n = 5n + 3$ ms.

Notiamo che:

- il tempo t_{alg} di esecuzione di questo algoritmo dipende da quanto è grande la variabile n data in input.
- Tale tempo può essere "spezzato" in due parti:
  1. una parte di tempo t_{iter}, costituita dal tempo impiegato per una singola iterazione del ciclo, moltiplicata per il numero di volte in cui si ripete il ciclo;
  2. una parte di tempo t_{cost}, indipendente dal numero di volte in cui si ripete il ciclo.
- Dato che il numero di volte che viene ripetuto t_{iter} corrisponde ad n, il tempo complessivo dell'algoritmo si può quindi esprimere come $t_{alg} = t_{iter} \cdot n + t_{cost}$
- al crescere di n, il termine $t_{iter} \cdot n = 5n$ diventa preponderante sul termine $t_{cost} = 3$, che quindi può essere considerato trascurabile
- indipendentemente dal fatto che l'architettura di riferimento impieghi $1ms$ o $1\mu s$ o 5 s o 10 ns per ogni istruzione elementare, l'espressione per il calcolo del tempo sarà sempre la stessa (quindi posso evitare di esprimere l'unità di tempo)
- il tempo complessivo $t_{alg} = t_{iter}n + t_{cost}$ posso vederla come $t_{alg} = a \cdot n + b$ con $a, b \in \mathbb{R}$, $\Rightarrow t_{alg}$ è una funzione $lineare$ sulla variabile di input n, quindi si dice che tale algoritmo, per essere completato, impiega un tempo **lineare** rispetto all'input.

Possiamo quindi fare un grafico per vedere come varia il tempo al variare del valore di n

(ossia avente sulle y: tempo t_{alg} e sulle x: n)



Possiamo fare di meglio? In altri termini, esiste un algoritmo che mi permette di risolvere lo stesso problema ma in tempo minore?

In questo caso specifico, sì.

Ricordandoci la formula di Gauss per il calcolo della somma dei primi n numeri naturali:

$$\frac{n(n+1)}{2}$$

possiamo implementare il seguente algoritmo:

In [ ]:
```
// algoritmo 1.2
int main()
{

    int n;
    printf("inserire numero: ");
    scanf("%d", &n);

    int s = n*(n+1)/2;

    printf("la somma dei numeri da 1 a %d è %d\n", n, s);

    return 0;
}
```

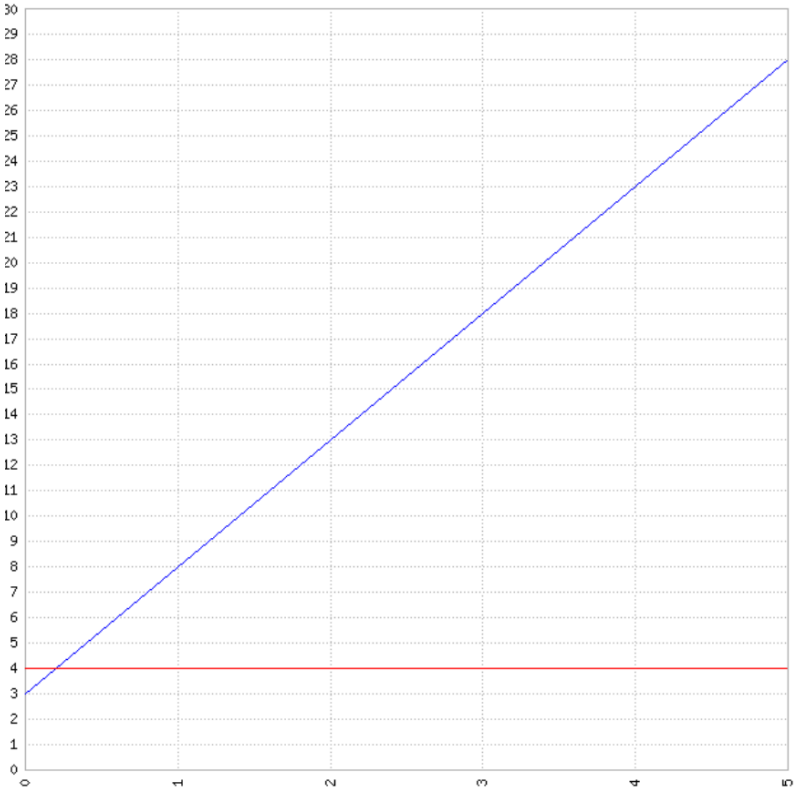

in questo caso, il tempo di esecuzione di questo algoritmo sarà dato da:

- 1 assegnazione
- 1 somma
- 1 moltiplicazione
- 1 divisione

totale: 4 operazioni elementari

In questo caso il tempo necessario all'algoritmo non dipende dalla dimensione della variabile n di input, ma sarà sempre pari a 4 **indipendentemente** dal valore assegnato ad n.

Si dirà quindi che tale algoritmo impiega un tempo $costante$ rispetto all'input (ossia il tempo neessario ad essere completato non dipende dall'input).



Osserviamo il seguente grafico, avente come ascisse il valore n e come ordinate il tempo t di esecuzione:

La linea blu rappresenta il tempo dell'algoritmo 1.1 in funzione di n, ossia $5n + 3$, la linea rossa invece il tempo dell'algoritmo 1.2, ossia 4 a prescindere da n.

Ovviamente, il secondo algoritmo è preferibile al primo in termini di tempo, in quanto indipendente dall'input.

Esempio 2: ipotizziamo di voler sapere quanto tempo impiega ad essere completato il seguente algoritmo che restituisce la somma di tutti gli elementi contenuti in un vettore di lunghezza `n`

In [ ]:
```
int somma_vett(int v[], int n)
{
    int s = 0;
    for(int i = 0; i<n; i++)
        s = s + v[i];

    return s;
}
```

Si hanno:

- 2 inizializzazioni `s=0` e `i=0`
- $n+1$ confronti `i<n`
- n incrementi `i++` (ognuno dei quali vale 2)
- n addizioni `s +...;`
- n assegnazioni `s=...`

totale: $2+n+1+2n+n+n = 5n+3$.

Anche in questo caso il tempo può essere espresso come $t_{iter}n + t_{cost}$

$\Rightarrow$ l'algoritmo impiega un tempo *lineare* rispetto alla dimensione n dell'array per essere completato.

Esempio 3: ipotizziamo di voler realizzare un algoritmo che, dato un array `v` di lunghezza `n` ed un valore `k`, restituisca l'indice in cui si trova `k`.

In [ ]:
```
// algoritmo 3.1
int cerca_vett(int v[], int n, int k)
{
    int idx_k = -1;
    for(int i = 0; i < n; i++)
        if(k == v[i])
            idx_k = i;
    return idx_k;
}
```

Anche in questo caso il tempo di esecuzione dell'algoritmo sarà composto da una parte costante t_{cost} che è pari al tempo per effettuare l'inizializzazione `idx_k=-1`, e da una parte variabile t_{loop} che dipende dal numero di iterazioni di un ciclo, quindi $t_{alg} = t_{cost} + t_{loop}$.

Calcoliamo momentaneamente il tempo impiegato da *una singola* iterazione (quindi t_{iter}):

- 1 confronti `i<n`
- 1 confronto `k == v[i]`
- 1 assegnazione (solo se il confronto `k == v[i]` da esito positivo)
- 1 incremento `i++` (che vale 2)

tale iterazione sarà ripetuta n volte, a cui bisogna aggiungere il tempo dell'assegnazione `idx_k = -1`.

Ogni singola iterazione avrà quindi tempo 4 oppure 5 in base al fatto che il confronto `k == v[i]` dia esito positivo o meno.

Semplifichiamo considerando due casi:

1. il valore `k` non è presente nell'array (ossia 0 volte): in questo caso ogni iterazione avrà durata $t^0_{iter} = 4$ => il tempo complessivo del ciclo sarà $t_{loop} = t^0_{iter} n = 4n$
2. il valore `k` è presente in ogni posizione dell'array (ossia n volte): in questo caso ogni iterazione avrà durata $t^n_{iter} = 5 \Rightarrow$ il tempo complessivo del ciclo sarà $t_{loop} = t^n_{iter} n = 5n$

è facile vedere che in tutti gli altri casi t^i_{iter}, $0 < i < n$, la durata complessiva del ciclo sarà $4n < t_{loop} < 5n$.

Quindi in generale l'algoritmo avrà tempo di esecuzione $t_{alg} = t_{cost} + t_{loop} = 1 + w \cdot t_{loop}$, con $4 \leq w \leq 5$.

Si vede quindi che, indipendentemente dal valore effettivo di w, il tempo necessario all'algoritmo per convergere al risultato finale è **lineare** rispetto alla dimensione dell'array.

Si può fare di meglio?

In [ ]:
```
// algoritmo 3.2
int cerca_vett(int v[], int n, int k)
{
    int idx_k = -1;
    int  i = 0;
    while(i < n && idx_k==-1)
    {
        if( k == v[i])
        {
            idx_k = i;
        }
        i++;
    }
    return idx_k;
}
```

Tralasciamo per ora il tempo t_{cost} impiegato dalle istruzioni fuori dal ciclo.

Calcoliamo momentaneamente il tempo impiegato da *una singola* iterazione (quindi t_{iter}):

- 2 confronti `i<n` , `idx_k==-1`
- 1 and logico `&&`
- 1 confronto `k == v[i]`
- 1 assegnazione (solo se il confronto `k == v[i]` da esito positivo)
- 1 incremento `i++` (che vale 2)

totale: una singola iterazione impiega un tempo pari a $t_{iter} = 6$ unità per essere completato, o 7 se `k == v[i]`, ma in quel caso il ciclo terminerebbe subito dopo.

Semplifichiamo considerando soltanto $t_{iter} = 6$, tanto come abbiamo visto non cambia molto.

Vediamo adesso quante volte il blocco con tempo t_{iter} viene eseguito:

- se il valore `k` si trova nella prima posizione di `v`, il loop sarà eseguito 1 sola volta, in quanto il corpo del ciclo sarà eseguito una sola volta
- se il valore `k` si trova nella seconda posizione di `v`, il loop sarà eseguito 2 volte $\Rightarrow 2 \cdot t_{iter}$

$\vdots$

- se il valore `k` si trova nell'ultima posizione di `v` (o non esiste in `v`), il loop impiegherà $n \cdot t_{iter}$ tempo per essere completato

Facciamo una distinzione su cosa succede nei diversi casi, ossia:

- *caso migliore*: il valore da cercare è in prima posizione $\Rightarrow$ trascurando t_{cost}, il tempo necessario per completare l'algoritmo è pari a $t_{alg} = t_{iter}$ (quindi indipendente dal numero dei dati)
- *caso peggiore*: il valore da cercare non è presente in `v` oppure è in ultima posizione $\Rightarrow$ trascurando t_{cost}, il tempo necessario per completare l'algoritmo è pari a $t_{alg} = n \cdot t_{iter}$
- in tutti gli altri casi: trascurando t_{cost}, il tempo necessario sarà compreso tra t_{iter} ed $n \cdot t_{iter}$. Solitamente il tempo di questi casi viene espresso come tempo necessario nel *caso medio*, che tiene conto di come l'algoritmo si comporta *in media* al variare degli input [non trattato in questo corso].

Ricapitolando, questo algoritmo impiega un tempo:

- *nel caso migliore, indipendente dalla dimensione dell'input (ossia dal numero di dati in input), quindi* ***costante***
- *nel caso peggiore, lineare** sulla dimensione dell'input*

Ciò è comunque meglio dell'avere un algorimo che impiega un tempo lineare sulla dimensione dell'input *in ogni caso*.

Ipotizziamo di avere un array con un milione di elementi. Se abbiamo la fortuna che l'elemento `k` si trovi in una posizione $< n$ (magari la prima!!), allora il tempo da attendere sarà $t_{alg} < 1000000 \cdot t_{iter}$, mentre nel caso del primo algoritmo sarà sempre $t_{alg} = 1000000 \cdot t_{iter}$, indipendentemente da dove si trovi il valore di `k`.

Esempio 4: dato un vettore `v` di lunghezza `n`, vogliamo calcolare quanto tempo impiega per essere completata la seguente funzione che restituisce l'indice di posizione del suo valore minimo

In [ ]:
```
float idx_min_array(float v[], int n)
{
    int idx_min = 0;
    float min   = v[0];
    for (int i=1; i < n; i++)
        if (v[i] < min)
        {
            min     = v[i];
            idx_min = i;
        }
    return idx_min;
}
```


Indichiamo con t_{alg} il tempo necessario al completamento dell'algoritmo riportato, i.e. il tempo necessario per la la ricerca dell'indice del minimo in un array al variare della lunghezza n.

In generale, la ricerca del minimo su un array di dimensione n impiegherà un tempo complesivo pari alla somma de:

- il tempo per eseguire 3 assegnazioni iniziali: `idx_min = 0`, `min = v[0]`, `i=0` (parte di tempo costante t_{cost})
- il tempo t_{iter} per eseguire il corpo del ciclo, costituito da 2 confronti (`i<n` e `v[i] < min`) e 2 eventuali assegnazioni (solo se `v[i] < min`).
- il ciclo `for` va da 1 ad n escluso, quindi viene eseguito $n - 1$ volte.

Il tempo impiegato da questo algoritmo sarà quindi, nel caso peggiore,
$t_{alg} = t_{iter} \cdot (n - 1) + t_{cost} = 4(n - 1) + 3$.

Quindi il tempo necessario per la ricerca del minimo è $lineare$ sulla dimensione dell'input -1.

Esempio 5: ipotizziamo di voler vedere quanto tempo impiega un algoritmo che, dato un array `v` di lunghezza `n`, restituisca un vettore `freq` contenente, in ogni posizione `i`-esima, la frequenza di ogni valore `v[i]` in `v` (ossia quante volte ogni `v[i]` è contenuto in `v`).

In [ ]:
```
void freq(int v[], int n, int freq[])
{
    // ciclo esterno
    for(int i = 0; i<n; i++)
    {
        freq[i] = 0;
        // ciclo interno
        for(int j = 0; j<n; j++)
        {
            if(v[j] == v[i])
                freq[i]++;
        }
    }
}
```

Analizziamo il tempo di esecuzione del ciclo interno:

Trascurando le inizializzazioni che darebbero luogo a t_{cost}, il corpo del ciclo più interno è composto composto da:

- 2 confronti: `v[j] == v[i]` e `j<n`
- 2 incrementi: `freq[i]++` e `j++` (ognuno dei quali vale 2)

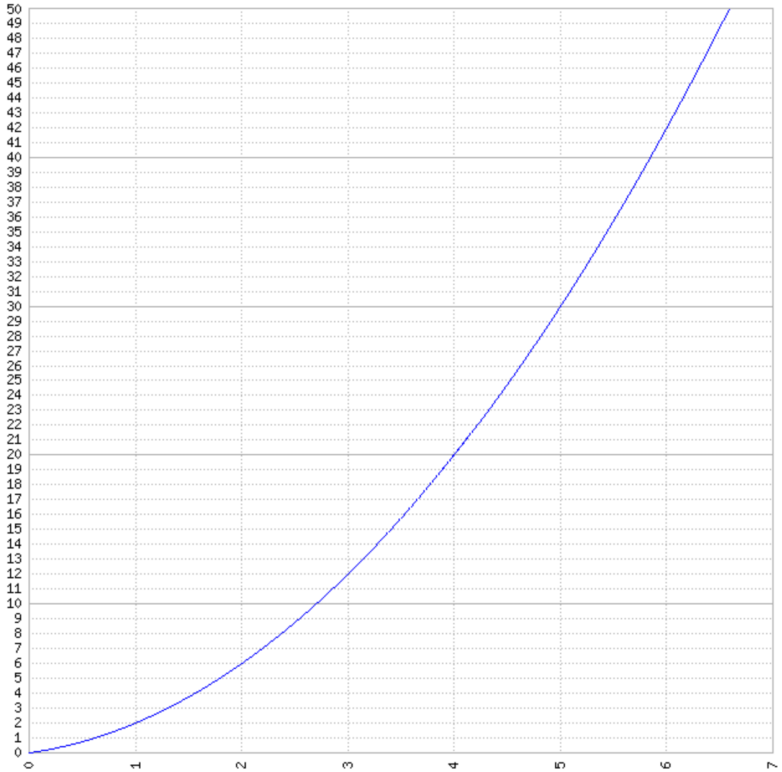
totale: $t_{iter} = 6$ operazioni elementari.

- Tali 6 operazioni vengono ripetute n volte $\Rightarrow 6n$, o anche $t_{iter} \cdot n$ (per semplificare, trascuriamo il tempo del primo confronto `j<n` di ingresso nel ciclo).
- Tali $t_{iter} \cdot n$ operazioni vengono ripetute n volte nel ciclo esterno, più un'assegnazione `freq[i] = 0`

$\Rightarrow t_{alg} = n \cdot (1 + t_{iter} n) = t_{iter} n^2 + n$

Per questo algoritmo, il tempo di esecuzione necessario essere completato varia in maniera $quadratica$ rispetto alla dimensione dell'input.

$\Rightarrow$ Tempo abbastanza alto anche per vettori non troppo grandi.



Esempio, per $n = 10$ il tempo necessario è dato da $t_{iter}n^2 + n = t_{iter} \cdot 100 + 10$, per $n = 1000$ il tempo necessario diventa $t_{iter} \cdot 1000000 + 1000$!

Considerazioni

- Si nota che la parte davvero preponderante (soprattutto all'aumentare di n) del tempo necessario per un algoritmo è generalmente **funzione dell'input** n, motivo per cui il tempo utilizzato t_{cost} per la parte rimanente (inizializzazioni, ecc.) può essere trascurato.
- Esistono ovviamente anche algoritmi con tempi di esecuzione molto più alti, ad esempio *cubico* sulla dimensione dell'input, (n^3) o, addirittura, *esponenziale* sulla dimensione degli input (2^n). Un algoritmo con tempi alti come l'esponenziale sono generalmente considerato inattuabili nella pratica, in quanto anche per piccoli valori di n, risultano impiegare troppo tempo per avere risultati in tempi utili. Esempio: per $n = 30$, $2^{30} = 1073741824$!!!!
- Con il termine **complessità di tempo** intendiamo quanto tempo un dato algoritmo impiega per essere completato *al variare degli input*.
- Con il termine **complessità di spazio** intendiamo quanto spazio in memoria un dato algoritmo necessita per essere completato al variare degli input [non trattata in questo corso].
- Ogni algoritmo ha le propria complessità. In teoria, dato un problema, si cerca l'algoritmo risolutivo che abbia la complessità di tempo e/o di spazio più bassa/e possibile.

02/04/23, 12:09 LABPROG_L4_tempoalgoritmi