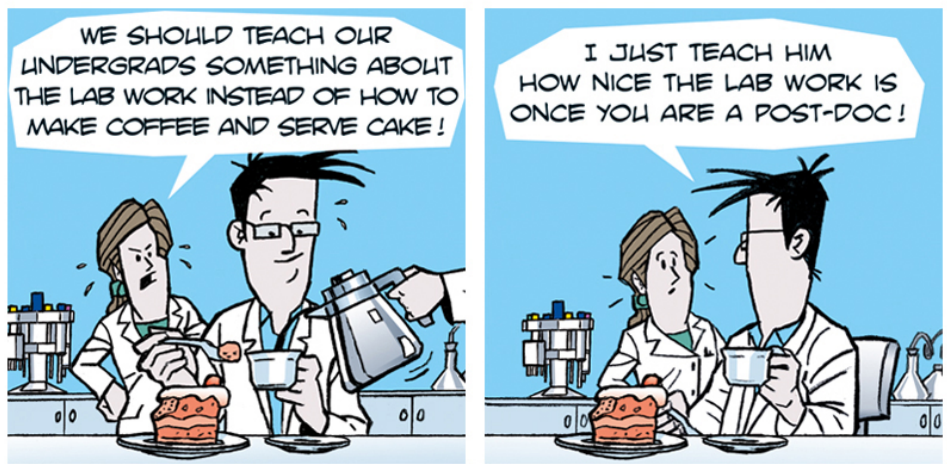
Laboratorio di Algoritmi e Strutture Dati

Primo esercizio, prima parte: *InsertionSort*(punti: 0)



Introduzione al laboratorio

Nel corso di Algoritmi vediamo circa un'ottantina di diversi algoritmi, tecniche, e soluzioni.

La mera implementazione di queste, pur presentando certi problemi non banali di adattamento pseudo-codice -> codice, non si può considerare sufficiente all'apprendimento delle idee.

Gli esercizi di laboratorio che sono proposti sono invece pensati per porre una sfida non del tutto triviale che serve per scontrarsi con problemi di implementazione di un livello adeguato per una laurea di primo livello. Questi esercizi vanno affrontati con **rigore metodologico**, **attenzione ai dettagli**, e curiosità.

Introduzione al laboratorio

La differenza tra pseudo-codice e codice vero non è solo sintattica. Lo pseudo-codice esiste per fare emergere le idee; il codice per fare in maniera che la macchina capisca in maniera non ambigua ciò che deve fare.

L'origine, la semantica, e le interazioni tra diverse parti in un linguaggio di medio/basso livello come il *C* sono di per sè problemi complicatissimi. Nel corso di Programmazione si riesce a vedere unicamente la punta di questo iceberg. In questo corso di laboratorio vedremo alcuni problemi non banali di programmazione che però non hanno a che fare con il linguaggio in sè.

Diventare **programmatore** è qualcosa che può essere identificato con diventare **informatico** fino ad un certo livello; oltre questo, le due strade si dividono molto chiaramente.

Introduzione al laboratorio

Gli esercizi vanno affrontati tenendo conto dei seguenti aspetti:

1. **Correttezza del codice.** A fine lavoro non ci possono essere errori di sintassi, né **warnings** (oppure si è compreso il significato e l'origine di ognuno di essi), il codice esegue il lavoro previsto correttamente in **tutti i casi**, anche quelli estremi.
2. **Pulizia del codice.** Le variabili, le funzioni, e le strutture dati hanno nomi sensati, coerenti, e puliti; il codice è correttamente identato, ed è, almeno in parte, ottimizzato riducendo ove possibile il numero degli **statements**, oltre ad essere **correttamente e abbondantemente commentato**.
3. **Efficienza.** Nei limiti del possibile, si utilizza la migliore soluzione possibile tra quelle che si vedono nel corso.

Laboratorio: complessità asintotica e sperimentale

Durante il corso abbiamo introdotto la nozione di tempo di esecuzione, indicato con $T(n)$, dove n è la dimensione dell'input.

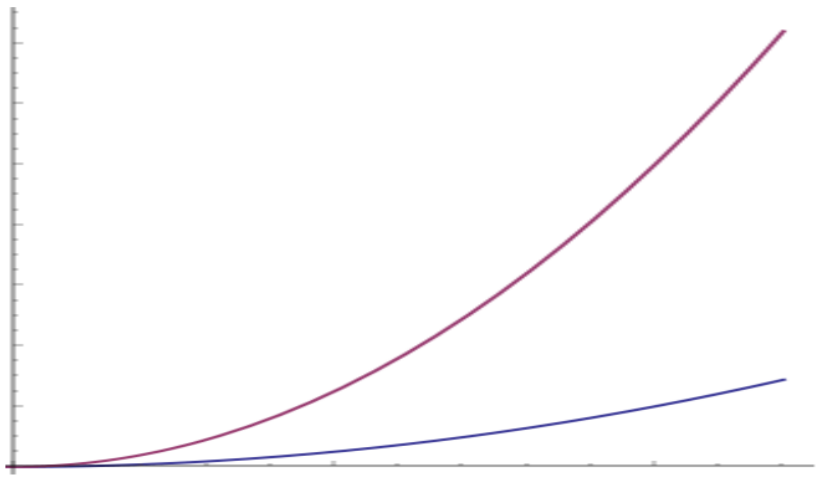
In casi semplici (esempio: ordinare un array), n è semplicemente il numero di oggetti. In casi più complessi può misurare l'input in maniera non banale (per esempio nei grafi).

Il tempo di esecuzione calcolato nella teoria è **asintotico**, cioè di tendenza. Sappiamo però che la forma asintotica **nasconde** la complessità analitica corretta, e quindi anche quella sperimentale.

Laboratorio: complessità asintotica e sperimentale

Prendiamo, ad esempio, *InsertionSort*, e consideriamo la sua complessità nel caso peggiore: $\Theta(n^2)$.

Studiare la complessità sperimentale, significa differenziare tra un'implementazione efficiente (esempio: $T(n) = 2 \cdot n^2$), ed una non molto efficiente (esempio: $T(n) = 10 \cdot n^2$). Entrambe queste implementazioni sono tali che $T(n) = \Theta(n^2)$: la notazione asintotica ci nasconde l'efficienza sperimentale.



Laboratorio: complessità asintotica e sperimentale

Inoltre, la complessitá sperimentale dovrebbe essere sempre calcolata sul **caso medio**. Il caso medio si ottiene **randomizzando** l'input, e per fare in maniera che un esperimento sia statisticamente valido, bisogna ripeterlo più volte, e calcolare la media aritmetica (o geometrica) dei risultati.

Supponiamo ad esempio di voler sperimentare *InsertionSort*. Immaginiamo di volerlo fare su un array di lunghezza fissata; poichè il tempo dipende da un'input generato casualmente, può variare anche senza variare la lunghezza.

Uno di questi esperimenti, dunque, consiste nel generare un array casuale, ordinarlo, prendere il tempo di questo ordinamento, e ripetere.

Laboratorio: complessità asintotica e sperimentale

Il modo più ovvio e naturale di misurare il tempo di esecuzione di una procedura è utilizzare il **clock** interno della macchina, che misura, in forma indipendente, il numero di cicli che passano. Per essere esatti, chiamare una funzione che restituisce questo valore significa ottenere il numero di cicli che sono passati da un momento predeterminato nel passato (tipicamente, il primo gennaio 1970) al momento della chiamata. Questo numero intero è misurato immediatamente prima dell'esecuzione, e immediatamente dopo. Calcolando la differenza tra questi due, si ottiene il numero di cicli di clock che sono passati durante l'esecuzione.

Ad esempio, in *C*, la libreria di sistema `time.h` mette a disposizione la funzione *clock*() e il tipo di dato `clock_t` (che è un intero).

Laboratorio: complessità asintotica e sperimentale

```
proc SingleExperiment (length, max_instances)
{ t_tot = 0
  for (instance = 1 to max_instances)
  { A = GenerateRandomArray(length)
    t_start = clock()
    InsertionSort(A)
    t_end = clock()
    t_elapsed = t_end − t_start
    t_tot = t_tot + t_elapsed
  return t_tot/max_instances
}
```

	tentativo					
lungh.	1	2	3	4	5	media
10	7	9	6	5	9	7.2

Laboratorio: complessità asintotica e sperimentale

In questo modo, ci assicuriamo che input **troppo favorevoli** o **troppo sfavorevoli** non influenzino il nostro esperimento. Ma la complessità di un algoritmo e quindi di una implementazione si misura per input di lunghezza crescente.

Quindi, il passo successivo, è ripetere l'esperimento per array più lunghi.

Laboratorio: complessità asintotica e sperimentale

```
proc Experiment (min_length, max_length)
{ max_instances = 5
  step = 10
  for (length = min_length to max_length step step)
  { time = SingleExperiment(length, max_instancs)
    print(time)
```

	tentativo					
lungh.	1	2	3	4	5	media
10	7	9	6	5	9	7.2
20	27	29	41	21	25	28.6
30	60	55	62	70	58	61
40	110	123	109	111	118	114.2
50	179	170	185	167	173	174.8

Laboratorio: dimostrare la correttezza

Come possiamo essere ragionevolmente sicuri che ciò che abbiamo fatto funziona? Parliamo di correttezza formale del codice, correttezza formale dell'algoritmo, e tecniche di testing, e cerchiamo di contestualizzare tutto ciò all'interno del corso.

Laboratorio: dimostrare la correttezza

Correttezza formale del codice. Seguendo la teoria di Hoare, è possibile dimostrare formalmente che un insieme di istruzioni in codice è corretto. Questa teoria si insegna in corsi di programmazione avanzata, ma sfortunatamente è considerata poco applicabile, e comunque ristretta a programmi di dimensioni piccole (**programmi giocattolo**).

Noi non la vediamo, e probabilmente la maggioranza degli informatici moderni non la vede in tutta la sua carriera.

Laboratorio: dimostrare la correttezza

Correttezza formale dell'algoritmo. Nella teoria di Algoritmi noi ci occupiamo di mostrare formalmente che un algoritmo presentato è corretto. Questo è necessario per la corretta comprensione, e utile per il ragionamento astratto. Purtoppo non si applica direttamente al codice utilizzato in programmazione e non ci aiuta a raggiungere il nostro obbiettivo.

Laboratorio: dimostrare la correttezza

Tecniche di testing. Queste vengono imparate nei corsi di ingengeria del software. Qui, vediamo degli accenni molto brevi. L'idea è che un programma puó essere testato in maniera sistematica, e tipicamente questo significa: **testare casi normali ed estremi**, **testare in maniera randomizzata**, e **testare attraverso funzioni antagoniste**.

Una funzione antagonista puó essere una soluzione semplice ed efficace. Partendo dall'idea che risolvere un problema è piú complesso che controllare che una soluzione sia corretta, possiamo costruire delle funzioni che effettuano questo controllo. Nell'esempio di un algoritmo di ordinamento questo significherebbe testare se un array è giá ordinato.

In casi piú complessi bisognerá studiare caso per caso la fattibilitá di una tale funzione.

Laboratorio: esercizio richiesto

In questa lezione di laboratorio vogliamo dunque realizzare un esperimento implementando *InsertionSort* e misurando il tempo di esecuzione sperimentale per input di lunghezza crescente, con multiple ripetizioni per la stessa lunghezza.

Possiamo riassumere un insieme di punti che devono essere rispettati per considerare l'esercizio **corretto**:

- Il programma compila correttamente?
- Ci sono errori/warnings?
- Il programma funziona?
- Restituisce una risposta corretta in ogni esecuzione?
- Le costanti scelte sono ragionevoli?
- Quali tecniche di correttezza sono state usate, e come?
- I risultati dell'esperimento sono stati riassunti in maniera grafica?
- L'aspetto della curva è quello atteso? Se no, perchè?

Laboratorio: una nota sulla generazione di numeri casuali

La generazione di numeri casuali è un problema noto in informatica.

Poichè le macchine sono tutte deterministiche, non esiste un modo per generare un numero realmente **casuale**. Quello che si fa è utilizzare un generatore di numeri **pseudo-casuali**.

Quali sono le caratterische attese di questi numeri? Essi devono **sembrare** casuali, e quindi sotto certe condizioni ci si aspetta che siano **equamente** distribuiti, **con la stessa probabilità di essere estratti** in ogni momento. Come si ottiene questo da una macchina deterministica?

Laboratorio: una nota sulla generazione di numeri casuali

Tipicamente, le più semplici funzioni causali sono quelle cosiddette **lineari**:

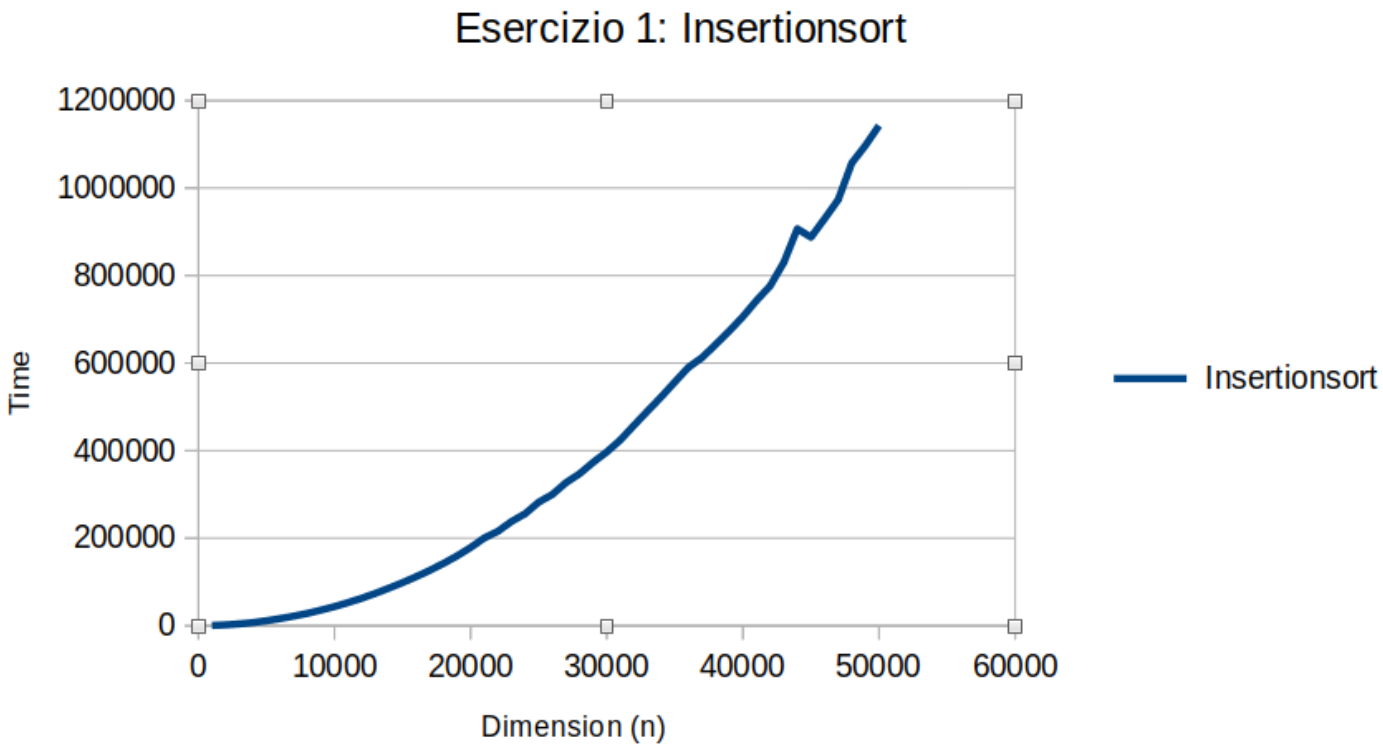
$$x_{n+1} = (ax_n + c) \ mod \ m$$

dove a, c, m sono costanti predefinite. Il valore x_0 è quello che, in un linguaggio come il *C*, è chiamato **seed** (seme).

Sebbene, per un determinato seme, chiedere al generatore di estrarre numeri in sequenza dia l'impressione che questi siano casuali, partire dallo stesso seme genererà la stessa sequenza.

Questo per noi è un vantaggio: pur generando input casuali, possiamo ripetere un esperimento. Quindi, il nostro protocollo prevede di selezionare un seme iniziale, e **non cambiarlo mai durante l'esperimento**.

Laboratorio: possibile risultato per questo esercizio



I parametri utilizzati per questo esperimento sono: $min\_length = 1000$, $max\_length = 50000$, $step = 1000$, $max\_instances = 50$, e $seed = 13$. Si possono utilizzare anche altri, a seconda del caso, una volta capito il problema; perché si è scelto di usare altri parametri, o questi, per gli esperimenti?

Conclusione

In questo primo esercizio abbiamo usato un algoritmo molto noto (*InsertionSort*) inserito in un contesto di esperimento scientifico, per verificare che le previsioni in termini di complessitá asintotica corrispondono alla realtá sperimentale. Questo schema sará la base di tutti i nostri esercizi.