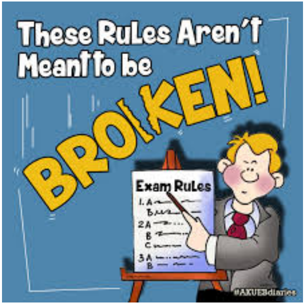
Algoritmi e strutture dati

Introduzione, organizzazione, e regole d'esame



Menú di questa lezione

In questa lezione vedremo la struttura del corso, le regole d'esame, le regole di valutazione, e gli esercizi di laboratorio. Inoltre daremo le prime definizioni e introdurremo il concetto di intuizione algoritmica.

Il docente

Io sono:

- Nome: **prof. Guido Sciavicco**.
- Sede: via Macchiavelli, terzo piano.
- Indirizzo email: *guido.sciavicco@unife.it*.

Voi potete:

- Passare a trovarmi in ogni momento del giorno, senza avvertire, per chiedere qualunque cosa relativa al corso e al suo svolgimento.
- Scrivermi prima, se volete avere la certezza di trovarmi in ufficio.
- Contattarmi via Skype se necessario (utente: *guido7504*).
- Scrivermi i vostri dubbi se non richiedono lunghe risposte o integrazioni (in quel caso è meglio venire).

Informazioni utili

Le lezioni saranno in modalità frontale con uso della lavagna elettronica per poter rilasciare gli appunti. Prima lezione: 29 settembre 2022. Orari delle lezioni:

- Lunedi 10.30 - 13.30, aula B1 del Palazzo Manfredini, complesso di matematica.
- Giovedi 8.30 - 11.30, aula B1 del Palazzo Manfredini, complesso di matematica (tranne 29 settembre, che si terrá in aula F8 del Chiostro di Santa Maria delle Grazie).
- Venerdi 14 - 16, laboratorio F8-F9 del Chiostro di Santa Maria della Grazie, laboratorio di informatica (prima lezione 7 ottobre 2022).

Sono sospese le seguenti lezioni:

- 31 ottobre (lunedì).
- Dal 7 al 11 novembre (lunedì, giovedì e venerdì).
- 8 e 9 dicembre (giovedì e venerdì).

Ultima lezione: lunedì 19 dicembre 2022.

Informazioni utili

Il corso dell'anno 2017/2018, cosí come quello dell'anno 2020/2021 sono interamente disponibili sul canale YouTube del docente:

https://www.youtube.com/channel/UCg7I1qsab9i3fEYijlsbNiQ

Libri di testo di riferimento:

- Introduction to Algorithms - Cormen, Leiserson, Rivest, Stain.
- The Algorithm Design Manual - Skiena, 2nd Edition.

Materiale didattico necessario per il corso:

- Queste slides!
- Lista e codice di tutti gli algoritmi trattati.
- Raccolta di esercizi (contiene anche tutti gli esercizi visti a lezione).

Informazioni utili

La valutazione è organizzata su tre assi:

1. Esame scritto.
2. Esame di laboratorio.
3. Esame orale.

Informazioni utili: esame scritto

L'esame scritto è composto da 13 domande a risposta multipla (**di cui 3 sempre prese dagli esercizi proposti**), con un massimo di 2 punti per ogni risposta esatta, cosí distribuiti:

- Domanda secca:
  - Punti 0: risposta sbagliata.
  - Punti 2 (2.5*): risposta esatta.
- Domanda semi-aperta:
  - Punti 0: risposta sbagliata (indipendentemente dalla spiegazione).
  - Punti 1: risposta esatta con spiegazione assente oppure sbagliata.
  - Punti 2 (2.5*): risposta esatta con spiegazione esatta.

Voto massimo: 26/30**.

*: **per gli studenti di matematica che svolgono l'esame da 6 crediti, senza laboratorio.**

: **per gli studenti di matematica, voto massimo 32

Informazioni utili: esame scritto

Esempio di domanda chiusa: **qual è la complessità di *InsertionSort* nel caso ottimo?** Possibili risposte:

A) $O(n)$.
B) $\Theta(n^2)$.
C) $\Theta(n \cdot log(n))$.
D) Nessuna delle risposte precedenti.

In alcuni casi, ci possono essere molte risposte possibili, ma solo una è esatta (o più precisa delle altre).

Informazioni utili: esame scritto

Esempio di domanda semi-aperta: **qual è la complessità di *InsertionSort* nel caso ottimo, e perchè?** Possibili risposte:

A) $O(n)$.

B) $\Theta(n^2)$.

C) $\Theta(n \cdot log(n))$.

D) Nessuna delle risposte precedenti.

Dimostrazione. Il ciclo piú interno di *InsertionSort* serve a trovare il posto giusto nel quale inserire *key*. Se l'array è ordinato, non viene mai eseguito. Dunque la sua complessitá dipende unicamente dal ciclo piú esterno, che è lungo n, ed è pertanto $O(n)$.

Nelle risposte aperte, viene dato uno spazio di scrittura per la spiegazione che deve essere chiara e coincisa. Il retro del foglio dell'esame puó essere usato per i ragionamenti che portano alla risposta, che peró deve essere, appunto, sintetica.

Informazioni utili: esercizi di laboratorio e valutazione

Vengono proposti 2 esercizi di implementazione, che vengono presentati durante 8 lezioni di laboratorio (durante la prima ora). Questi esercizi sono incrementali: ad ogni lezione vengono aggiunte delle caratteristiche che contrubuiscono alla valutazione.

Informazioni utili: esercizi di laboratorio e valutazione

Primo esercizio: algoritmi di ordinamento, confronto basato su efficienza sperimentale tra diversi algoritmi e diverse implementazioni. Struttura:

1. *InsertionSort*, struttura di un esperimento, tracciamento del grafico. Punti: 0. **Attenzione: questo esercizio è obbligatorio per essere ammessi all'esame finale (eccetto che per gli studenti di matematica che svolgono 6 crediti).**
2. Aggiunta di *MergeSort*, implementazione standard, e della sua versione ibrida con *InsertionSort*, implementazione standard. all'esperimento. Punti 2.
3. Aggiunta di *QuickSort*, implementazione standard, non randomizzato, con scelta del pivot *mediana di tre* (non visto in classe, spiegato in laboratorio). Punti 3.
4. Aggiunta di *QuickSort*, implementazione single tail call e ibrida con *InsertionSort* (non visto in classe, spiegato in laboratorio). Punti 4.

Informazioni utili: esercizi di laboratorio e valutazione

Secondo esercizio: strutture per insiemi disgiunti, confronto tra diverse implementazioni. Struttura:

1. Strutture per insiemi disgiunti basati su liste senza unione pesata e senza esperimento di confronto. Punti 0.
2. Aggiunta della versione con unione pesata ed esperimento di confronto tra le due versioni. Punti 2.
3. Aggiunta della versione basata su alberi k-ari, con union-by-rank e compressione del percorso, con esperimento di confronto tra le diverse versioni. Punti 4.

Informazioni utili: esercizi di laboratorio e valutazione

Gli esercizi sono contestualizzati con una metodologia precisa di sperimentazione, che viene spiegata in una lezione dedicata (la prima). Gli esercizi sono pensati per migliorare la comprensione di certi concetti di teoria, e per mettere lo studente di fronte a problemi di complessitá media in preparazione anche al mondo lavorativo. Sono svolti in linguaggio C per dare continuitá al corso di Programmazione e per obbligare lo studente a comprendere i concetti piú basilari.

Informazioni utili: esame orale

Per quanto riguarda la parte orale dell'esame, abbiamo quanto segue. Si è ammessi all'orale con una somma totale (scritto+laboratorio) di minimo 15/30. L'orale è obbligatorio nei seguenti casi:

- Totale minore di 18 (ma almeno 15).
- Totale maggiore stretto di 26.
- Esplicita richiesta da parte del docente.

In tutti gli altri casi, assumendo di aver raggiunto almeno 18, il voto puó essere registrato arrotondato all'intero inferiore senza orale. In alternativa l'orale è facoltativo, e permette di incrementare il voto finale in una certa misura, senza mai prevedere la possibilitá di una diminuzione.

Informazioni utili

Tutorato didattico. Gli studenti possono prendere appuntamento per questioni relative al laboratorio e per la consegna dello stesso.

- Nome: **dott. Ionel Eduard Stan**.
- Sede: via Macchiavelli, primo piano, stanza dottorandi.
- Indirizzo email: *ioneleduard.stan@unife.it*.

Programma concettuale

Lo scopo di questo corso è riassumere circa 70 anni di storia di algoritmi e strutture dati in 80 ore. Inevitabilmente i concetti che andremo a vedere sono legati tra loro in maniera non lineare, con multiple connessioni su multiple dimensioni, pertanto costruire un ordine totale di argomenti è fondamentalmente impossible. Cercheremo di procedere nel modo piú lineare possible, ma non possiamo aspettarci di studiare in maniera **compartimentata**. E' necessario, come pre-requisito, aver superato con successo un corso di Programmazione (in qualunque linguaggio, preferibilmente imperativo), avendo ben chiare le nozioni di base di diagrammi di flusso e di programma.

Programma concettuale

Array ←
- Algoritmi elementari per l'ordinamento
- Algoritmi non elementari per l'ordinamento
- Algoritmi per l'ordinamento non basato su confronti

Tabelle Hash

Liste

Pile

Code

Code di prioritá

- Inserimento
- Ricerca
- Cancellazione
- (Incremento/Decremento)

Alberi

Alberi binari di ricerca

Alberi red-black

Alberi B

Strutt. per ins. disgiunti

Grafi ←
- Visite
- Percorsi minimi
- Alberi di copertura minimi
- Componenti fortemente connesse
- Ricerca di cicli

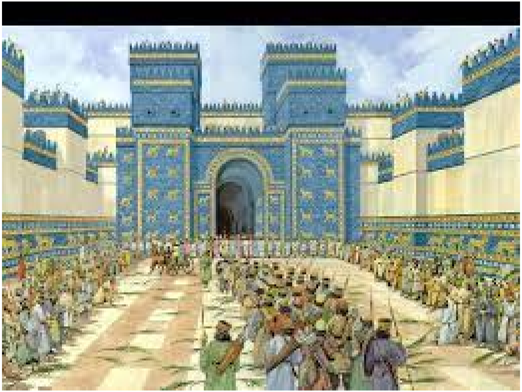
Buon divertimento!

Algoritmi e strutture dati

Intuizione algoritmica

Algoritmi e problemi

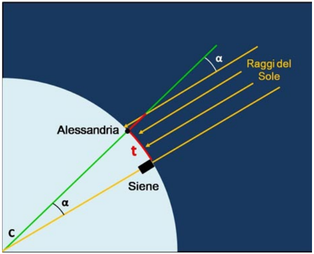
Le prime tracce di **algoritmica** provengono dai Babilonesi, nel IV secolo Avanti Cristo. Si dice che, contemporaneamente all'introduzione dei sistemi numerici e l'evoluzione dell'abaco, dell'algebra, e l'introduzione del concetto di variabile, i Babilonesi scoprirono l'importanza di avere un **metodo** per tenere traccia delle riserve di grano e della popolazione del bestiame.



Algoritmi e problemi

Euclide, Archimede, e soprattutto Eratostene (tutti vissuti tra il IV ed il III secolo Avanti Cristo) ci forniscono i primissimi esempi di algoritmi che sono rimasti fino a noi. Euclide, ad esempio, forní un metodo, ancor oggi utilizzato, per trovare il massimo comun divisore di due interi positivi, ed Archimede trovó una prima approssimazione sistematica di π.

Particolarmente interessante è il lavoro di Eratostene, vissuto tra il 276 ed il 194 Avanti Cristo. Oltre al famoso **crivello** (un metodo per elencare tutti i numeri primi), il suo contributo piú famoso è certamente una stima della circonferenza terrestre ottenuta con una tecnica riproducibile, appunto, un algoritmo. Capiamo dunque che è questa una delle caratteristiche piú importanti: un algoritmo deve essere **riproducibile**.



Algoritmi e problemi

Ma la parola **algoritmo** per come la conosciamo noi deriva dal nome di un matematico persiano del nono secolo (quindi 12 secoli dopo), Abu Abd Allah Muhammad ibn Musa **al-Khwarizmi**, cioé Mohammad, padre di Adbdulla, figlio di Mosè il **Khawariziano**, cioè colui che viene dalla città di **Kwarizm** (oggi Khiva, Uzbekistan, repubblica dell' ex-unione sovietica).

Algoritmi e problemi

Quella regione meridionale del moderno Uzbekistan ha prodotto, nel tempo, una curiosamente alta percentuale di matematici nel periodo seguito dal declino della civiltà greca e prima dell'avvento dell'epoca cosiddetta moderna. Tra questi esponenti troviamo, al-Biruni e, appunto, al-Khwarizmi. Al Biruni, di un centinaio di anni posteriore a al-Khwarizm, diede alcune importanti contributi alla matematica, alla geografia, e la cartografia, tra cui anche la **carta egocentrica**, oggi ancora usata nella pianificazione del volo aereo.

Al-Khwarizm invece operó durante il regno del califfo al-Mamun (tra il 813 e il 833 Dopo Cristo) in una biblioteca chiamata Casa del Sapere. Egli curò la traduzione delle grandi opere matematiche greche e indiane in arabo, da cui trasse diversi insegnamenti, tra cui alcuni metodi per la risoluzione di equazioni.

Algoritmo
non è una parola di orgine greca, nonostante il suffisso -**ritmo** che potrebbe ricordare il greco -**rhythmos**; la teoria più accreditata, è invece che provenga dalla latinizzazione del nome, appunto, Al-Khwarizm.
Allo stesso modo, probabilmente anche **algebra** è la latinizzazione del suo libro più famoso: **al-Jabr**. Il suo lavoro comprende l'introduzione del sistema di posizionamento decimale al mondo occidentale, assieme al primo metodo di soluzione di equazioni lineari e quadratiche.

Algoritmi e problemi

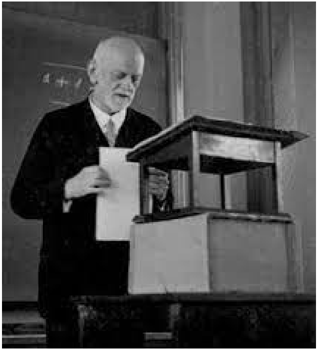
Al-Khwarizmi ha popolarizzato l'uso del sistema decimale (forse anche il simbolo dello zero). Grazie anche agli sforzi successivi del matematico Leonardo da Pisa (noto come Fibonacci), l'uso di questo sistema per la manipolazione di numeri ha progressivamente sostituito l'abaco. Fino all'epoca moderna, la parola **algoritmo** indicava quindi un **metodo per la manipolazione simbolica dei numeri**.

Algoritmi e problemi

Un **algoritmo**, per noi, è qualsiasi procedura computazionale che da un **input** produce un **output**. É necessario un algoritmo per **comunicare** la soluzione a un problema in una maniera formale, verificabile, riproducibile, ed implementabile. Non ci possono essere passi **magici** nella ricetta, e non ci possono essere passi **imprecisi** (esempio: **un pizzico di origano**). Inoltre, la relazione tra problema ed algoritmo non è uno a uno! Ci possono essere problemi risolvibili da tanti algoritmi diversi, e problemi per i quali non esiste un algoritmo che lo risolve.

Algoritmi e problemi

In questa definizione, giá propria dell'informatica moderna, si inseriscono altri contributi molto piú recenti. David Hilbert, famosissimo matematico a cavallo ormai tra il 1800 e 1900, pubblicó proprio nell'anno 1900 una lista di 20 problemi, al tempo irrisolti, due dei quali si sono rivelati pietre miliari dell'informatica moderna: la soluzione algoritmica delle equazioni diofantee (cioè con coefficienti interi, problema 10) e l'assiomatizzazione corretta e completa dell'aritmetica (problema 2).



Algoritmi e problemi

Noi diciamo che un **problema** è ciò che dobbiamo risolvere, e chiameremo **problema risolvibile** un problema per il quale esiste un algoritmo (sotto certe condizioni) che lo risolve. Un problema puó essere perfettamente definito anche in assenza di un algoritmo che lo risolve (i problemi 2 e 10 di Hilbert sono proprio due esempi di questo!). Una **istanza** è un particolare input ad un problema e una **soluzione** è l'output che corrisponde ad un particolare input. Un esempio di problema è **ordinare un insieme di numeri interi**; un possibile algoritmo che lo risolve è *InsertionSort*; una istanza è $\langle 23, 1, 45, -10, 7\rangle$, e la soluzione a questa istanza è $\langle -10, 1, 7, 23, 45\rangle$.

Algoritmi e problemi

È con Turing, nell'epoca della seconda guerra mondiale, che assistiamo alla nascita della teoria della Calcolabilitá e della Complessitá. In maniera indiretta Turing mostró che esistono problemi non computabili, e che il suo modello di computazione permetteva di parlare di complessitá computazionale in maniera generale ed oggettiva.

Algoritmi e problemi

Con Turing entriamo nell'epoca moderna dell'informatica, e ci addentriamo nello studio degli algoritmi di problemi che sono risolvibili. Diciamo che la **computabilitá** è una caratteristica del **problema** e non dell'algoritmo che lo risolve, mentre la **complessità** è invece una caratteristica di entrambi il problema e un algoritmo che lo risolve. Un problema puó essere **incomputabile** (indecidibile, irrisolvibile). La caratteristica di questi ultimi di non essere risolvibili in maniera algoritmica dipende dalla natura del problema e non dalla nostra capacitá di pensare ad un algoritmo e neppure dalle capacitá degli attuali sistemi computazionali. Focalizzandoci sui problemi che invece possono essere risolti, ci dobbiamo chiedere come comunicare questa soluzione in maniera formale, e dunque ricordare il concetto di **linguaggio** e poi di **programma**.

Scrivere algoritmi

In **linguistica** si distinguono tre aspetti di un linguaggio: la **sintassi**, cioè come strutturo una frase; la **semantica**, cioè cosa significa una frase; e la **pragmatica**, cioè lo studio di qual è il miglior modo di esprimere un concetto, fissata la semantica che si vuol dare e la sintassi che si vuol utilizzare. Per studiare algoritmica dobbiamo fissare un linguaggio, dunque la sua sintassi e la sua semantica, ed essere pragmatici nell'utilizzarlo. Ma gli algoritmi **non** sono programmi. I programmi **descrivono** gli algoritmi, nascondendo, di fatto, l'intuizione che soggiace all'algoritmo stesso. C, C++, Java, Phyton, Prolog, MatLab, R, Javascript, ...: le loro idiosincrasie non rivestono alcuna importanza nella progettazione di un algoritmo, e, di fatto, ne costituiscono un limite. **Tutti i linguaggi di programmazione sufficientemente espressivi sono ugualmente espressivi**. Questo significa che possiamo scegliere un linguaggio o un altro e non perdere nulla in termini di capacitá di risolvere un problema. Pertanto scegliamo il **nostro** linguaggio (uno **pseudo codice**) con le seguenti caratteristiche: deve essere intuitivo e assomigliare il piú possibile ai linguaggi di programmazione classici (imperativi, tipo C, C++, o Java).

Scrivere algoritmi

Lo pseudo codice si caratterizza per essere molto semplice ed intuitivo:

```
proc MyFunction (MyArg1, MyArg2, ...)
  for (i = Something to Something)
  {...
  while (Condition)
  {...
  if (Condition)
    then
  {...
  MyOtherFunction(MyArg3, MyArg4, ...)
  return Something
```

Scrivere algoritmi

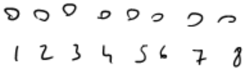
Facciamo le seguenti ipotesi. Le istruzioni semplici si eseguono in tempo costante (una unitá di tempo): operazioni algebriche sui numeri interi e non, assegnamenti, controlli di condizioni logiche, movimenti semplici in memoria. Tutto il resto non è costante, in particolare i cicli e le chiamate ricorsive o a funzioni da noi precedemente definite o assunte. I numeri sono rappresentati in base binaria, per cui il numero n occupa $c \cdot log(n)$ bit di memoria per qualche costante c - la **parola** di memoria è di lunghezza costante. La **dimensione dell'input** si misura, in generale, come il **numero di bit** che l'input occupa. Il **tempo di computazione** è il numero di passi semplici che si impiegano espresso in termini della dimensione dell'input, e tiene conto di **costanti** che nascondono i dettagli implementativi. Infine, **gli array cominciano dalla posizione 1** e non 0, se non esplicitamente detto il contrario.

Caratteristiche degli algoritmi

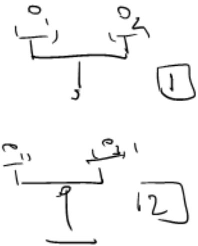
L'esistenza di un algoritmo per un problema ci dá un tetto alla complessità del problema stesso: nel peggior caso, il problema è non piú difficile dell'algoritmo che lo risolve - puó, d'altra parte, essere piú semplice, e quindi esistere un altro algoritmo che lo risolve in maniera piú efficiente. In questo corso, in generale, ci occupiamo di studiare algoritmi (quindi, problemi risolti) e le loro complessità. La complessità si misura in termini di tempo e di spazio (di memoria). Per esempio, un algoritmo di ordinamento potrebbe avere complessità in tempo $n \cdot log(n)$: ció significa che se una particolare istanza è lunga n, ci si mette (circa!) $n \cdot log(n)$ passi **elementari** per risolverlo.

Caratteristiche degli algoritmi

Per comprendere correttamente queste affermazioni, dovremo essere piú precisi (presto lo faremo). Concentriamoci però sui dati che possiamo comprendere adesso: quanto è importante saper studiare la complessità di un algoritmo? Immaginiamo di voler scrivere un algoritmo per risolvere il seguente problema: date n monete indistinguibili, tutte di peso identico tranne una (falsa) che pesa di piú, ed una bilancia a bracci, trovare la moneta falsa. In questo esercizio consideriamo una singola pesata come una operazione elementare.



In una prima soluzione (chiamiamola soluzione **naïve**, o soluzione ingenua), usiamo il seguente algoritmo, dove $Coin_1, \ldots, Coin_n$ sono le monete, $Weigth(\mathcal{C}_i, \mathcal{C}_j)$ è una operazione elementare a costo unitario che restituisce uno di tre valori: uno, se il gruppo di monete $\mathcal{C}_i$ pesa meno del gruppo $\mathcal{C}_j$, due se il gruppo di monete $\mathcal{C}_i$ pesa più del gruppo $\mathcal{C}_j$, e zero altrimenti:




```
proc IsFalse (Coin_1, ..., Coin_n)
  for (j = 2 to n)
    Res = Weight({Coin_1}, {Coin_j})
    if (Res ≠ 0)
      then
        if (Res = 1)
          then return Coin_j
          else return Coin_1
```

Caratteristiche degli algoritmi

Su 8 monete = 7 pesate
su n = (n−1)

Immaginiamo di eseguire il nostro algoritmo su una macchina che riesce ad eseguire 100000 ‘pesate’ al secondo. Il numero totale di pesate del nostro algoritmo è uguale numero di monete meno uno (cioè $n - 1$). Immaginiamo adesso di eseguire il nostro algoritmo per trovare una moneta falsa in un insieme di 100 milioni di monete. Il tempo totale necessario è:

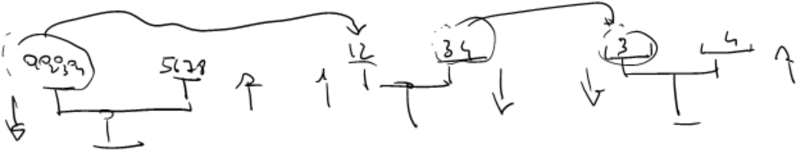
$$\frac{10^8}{10^5} = 10^3 \text{ secondi} > 16 \text{ minuti}$$

Nella seconda soluzione usiamo una tecnica piú intelligente.

proc IsFalse-2 $(Coin_1, \ldots, Coin_n)$

- $i = 1$
- $j = n$
- **while** $(i < j)$
  - $k = \frac{i+j}{2}$
  - $Res = Weights(\{Coin_i, \ldots, Coin_k\}, \{Coin_{k+1}, \ldots, Coin_j\})$
  - **if** $(Res = 1)$
    - **then** $i = k + 1$
    - **else** $j = k$
- **return** $Coin_i$

divisione intera = $\lfloor \ \rfloor$



Caratteristiche degli algoritmi

Supponiamo, di nuovo, che il costo della pesata sia unitario. Questa volta però, eseguiamo l'algoritmo su una macchina peggiore di quella precedente, che riesce ad eseguire solo 10000 'pesate' al secondo. Il numero totale di pesate necessarie all'esecuzione di questa tecnica è una per la prima separazione (che ci lascia con $\frac{n}{2}$ monete), una per la seconda separazione, e così via fino a trovarci con una sola moneta. Questa successione è scrivibile in un modo molto elegante:

$$\underbrace{1 + 1 + 1 \ldots + 1}_{log(n)},$$

perchè ci vogliono precisamente $log(n)$ (in base 2) divisioni a metà per passare dalla quantità n alla quantità 1 (se n è una potenza di 2; quante sono se non lo è?). Il tempo totale necessario per trovare la moneta falsa tra 100 milioni è:

$$\frac{log(10^8)}{10^4} = \frac{26{,}57\ldots}{10^4} = 0{,}002657\ldots \text{ secondi.}$$

Conclusione

Durante questo corso impareremo una serie di fatti algoritmici. Ad esempio scopriremo che *MergeSort* è un algoritmo **divide et impera** che costa $O(n \cdot log(n))$, e che **il costo di eliminare un elemento da un albero binario di ricerca è lineare nel caso pessimo**, e molte altre affermazioni simili. Ma lo scopo del corso non è imparare una serie di nozioni; è, invece, stimolare lo sviluppo di una **intuizione** algoritmica. Questa è la ragione per la quale siete autorizzati a portare con voi degli appunti durante l'esame. La maniera di costruire questa intuizione è certamente quella di porre (una lunga serie di) problemi e imparare, da soli e assieme, come questi si risolvono, per essere preparati, domani, ad affrontare un problema nuovo. Questo processo è, in ultima analisi, **divertente**; spero che vi divertirete tanto quanto me nel farlo.