Informatica per Statistica
Riassunto della lezione del 05/10/2012

Igor Melatti

Introduzione agli algoritmi

- La parola *algoritmo* deriva dal nome di un matematico persiano del IX secolo, *Al-Khowarizmi*
- Non esiste una definizione ben precisa del concetto *generale* di algoritmo, ma è possibile caratterizzarlo in modo formale
- In questo corso, se ne darà un'idea intuitiva, e si procederà per esempi
- Si comincia da un *problema computazionale*
  - in soldoni, qualcosa che si vuole risolvere tramite un (programma per) computer
- Un problema computazionale specifica, tipicamente usando un linguaggio matematico, la relazione desiderata tra *input* (informazione in ingresso) ed *output* (informazione in uscita)
  - praticamente è la *specifica* di una funzione matematica
- Un algoritmo è lo strumento per *risolvere* un problema computazionale
  - in un problema computazionale si dice *che cosa* si vuole ottenere
  - con un algoritmo si dice *come* lo si ottiene
  - non sempre è possibile risolvere un problema computazionale con un algoritmo (vedere più avanti)
- Per far sì che il "come" sia *computazionalmente* realizzabile, ovvero che lo si possa realizzare con un (programma per) computer, un algoritmo deve avere determinate caratteristiche, riassumibili informalmente come segue:
  - un algoritmo è una *sequenza finita di passi computazionali "ben definiti"*, che trasformano l'input nell'output nella maniera desiderata

- cioè in quella specificata dal problema
- il "ben definiti" di cui sopra verrà illustrato tramite esempi

- Problema della conversione tra numeri senza segno dalla base 10 alla base 2

  input: numero $n \in \mathbb{N}$

  output: sequenza di bits $\langle b_1, \ldots, b_k \rangle$, tale che il valore di $b_k \ldots b_1$ sia n

  - il valore di una sequenza di bit $\langle b_k, \ldots, b_1 \rangle$ intesa come intero senza segno in notazione posizionale è $\sum_{i=1}^{k} b_i 2^{i-1} = b_k 2^{k-1} + b_{k-1} 2^{k-2} + \ldots + b_2 2^1 + b_1 2^0$
  - quindi si vuole che $\sum_{i=1}^{k} b_i 2^{i-1} = n$

- Algoritmo per la conversione (già visto, ma riscritto con la notazione che verrà usata nel resto del corso): Figura 1

  - **Esercizio**: provare ad eseguire l'algoritmo *Convert* sulla macchina di Von Neumann avendo come input 62

```
1  Convert(n)
2    i ← 1
3    h ← n
4    do
5      A[i] ← h mod 2 /* nasconde alcune difficoltà */
6      i ← i + 1
7      h ← h div 2
8    while h ≠ 0
9    return A
```

Figure 1: Algoritmo *Convert*

- Problema dell'ordinamento

  input: sequenza di n numeri $\langle a_1, \ldots, a_n \rangle$

  output: sequenza di n numeri $\langle a'_1, \ldots, a'_n \rangle$, che sia un riordinamento della sequenza in input e t.c. $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

  - volendo essere pienamente formali: per "riordinamento" (o *permutazione*) di $\langle a_1, \ldots, a_n \rangle$ si intende una sequenza $\langle a'_1, \ldots, a'_n \rangle$ t.c. esiste una funzione biiettiva $f : \{1, \ldots, n\} \leftrightarrow \{1, \ldots, n\}$ t.c. $\forall i \in \{1, \ldots, n\}$. $a'_i = a_{f(i)}$

- Esistono molti algoritmi per risolvere il problema dell'ordinamento

- Ad esempio, algoritmo insertion sort (letteralmente, ordinamento per inserzione) in Figura 2

  - **Esercizio**: provare ad eseguire l'algoritmo *Insertion-Sort* sulla macchina di Von Neumann avendo come input la sequenza $A = \langle 5, 4, 3, 1, 2 \rangle$

```
1 Insertion-Sort(A)
2   for j ← 2 to length(A) do
3     key ← A[j]
4     /* Inserimento di A[j] nella sequenza ordinata A[1...j − 1] */
5     i ← j − 1
6     while i > 0 and A[i] > key do
7       A[i + 1] ← A[i]
8       i ← i − 1
9     A[i + 1] ← key
```

Figure 2: Algoritmo *Insertion Sort*

- Da questi esempi si possono notare alcune cose

  1. gli algoritmi vengono descritti in *pseudocodice*
     - per distinguerlo dal *codice* (sottinteso: *sorgente*), che è il modo con cui si indica un programma scritto in C o un qualunque altro linguaggio di programmazione
     - mentre il codice, che va "capito" ed eseguito da un computer, va scritto secondo regole ben precise, e stando attenti a tutti i dettagli, con lo pseudocodice si possono astrarre alcune cose
       * ad esempio, non è grave (pur restando un errore, una volta decisa una notazione...) se si scrive `length[`A`]` invece di `length(`A`)`
       * si possono usare simboli come $\neq$ o $\leq$ (non sono caratteri ASCII, i linguaggi di programmazione non vanno oltre quello)
     - inoltre, può non occuparsi di dettagli come l'allocazione di memoria
       * come il passo 5 di Figura 1
     - l'importante per lo pseudocodice non è che lo capisca una macchina, ma che lo capisca un essere umano che poi lo possa facilmente trascrivere in un codice vero e proprio
     - caratteristiche dello pseudocodice adottato in questo corso:

    * i blocchi di istruzioni all'interno di strutture di controllo di flusso come `for`, `while` etc sono individuati solo dalle indentazioni
        · cioè dagli spazi di rientro
        · ad esempio nella Figura 2 le righe dalla 3 alla 9 hanno tutte 2 spazi all'inizio in quanto sono da intendersi dentro il `for` di riga 2
        · analogamente, le righe dalla 7 alla 8 hanno tutte altri 2 spazi all'inizio (quindi in totale 4) in quanto sono da intendersi non solo dentro il `for` di riga 2, ma anche dentro il `while` di riga 6
        · nel C, gli analoghi costrutti andranno specificati aprendo e chiudendo parentesi graffe
        · ma esistono altri linguaggi di programmazione più semplici, ad esempio il Python, che vanno ugualmente ad indentazione
    * i controllori di flusso hanno lo stesso significato che nel C
    * gli algoritmi trattano spesso di sequenze, ovverosia di insiemi ordinati; tali sequenze verranno indicate da una variabile (tipicamente maiuscola), e per indicare un certo elemento i di una sequenza A si userà l'espressione A`[`i`]`
    * il primo elemento di una sequenza A è A`[1]`, l'ultimo è A`[length(`A`)]`
        · come si vedrà, nel C, invece, si va da A`[0]` a A`[`n `- 1]`, con n lunghezza dell'array (non esiste una funzione predefinita `length`)
- in un algoritmo, sarà sufficiente dare una funzione (in alcuni rari casi con 2 o 3 funzioni cosiddette *ausiliarie*) che risolvono il problema
    * negli esempi dati sopra, sono state definite le funzioni *Convert* in Fig. 1 e *Insertion-Sort* in Fig. 2
- in C, occorrerà scrivere una sequenza di funzioni (qualcuna potrebbe essere già stata scritta, e basta solo riusarla), di cui una obbligatoriamente chiamata `main`, che sia la prima ad essere eseguita dal computer
    * inoltre, in C sarà tipicamente necessario leggere in qualche modo l'input (da tastiera, da file, da internet...) e scrivere in qualche modo l'output (su schermo, su file, su internet...)
    * tipicamente, per leggere l'input si usa un comando (o meglio, come si vedrà, una funzione) chiamata `scanf`; invece per scrivere l'output si usa la `printf`
    * questo potrebbe portare a pensare che le `scanf` del C siano sempre l'input delle funzioni di un programma C, così come le `printf` siano sempre l'output delle stesse funzioni

    - *non* è così, dovrà essere chiaro nel prosieguo di questo corso
    - al più, `scanf` e `printf` (quando presenti) potranno essere considerate l'input e l'output del programma inteso nella sua intierezza
  2. i problemi sono descritti dando prima l'input, poi l'output insieme con la proprietà che l'output stesso deve rispettare con riferimento all'input
  3. la tipica "filiera" è la seguente
     (a) c'è un problema da risolvere in modo computazionale (esempio: occorre ordinare dei numeri)
     (b) lo si specifica come problema computazionale, ovvero dando la relazione tra input ed output (esempio: la formalizzazione vista sopra)
     (c) si scrive un algoritmo che risolva il problema (esempio: Insertion Sort)
     (d) ci si convince che l'algoritmo risolva effettivamente il problema
     (e) si *implementa* l'algoritmo, scrivendo un programma (o una funzione) in un linguaggio di programmazione

- È importante distinguere tra *problema* e *istanza di un problema*
  - un problema è la formulazione generale come sopra
  - un'istanza di un problema è un caso particolare di un suo input
    - ad esempio, $\langle 34, 45\rangle$ e $\langle 100, 0, 42\rangle$ sono due istanze del problema dell'ordinamento
    - ad esempio, 1023 e 20345 sono due istanze del problema della conversione
  - formalmente, un problema può anche essere definito dall'insieme delle sue istanze con le relative soluzioni
  - i problemi "interessanti" hanno un numero infinito di istanze (almeno matematicamente)

- Un algoritmo si dice *corretto* rispetto al problema che deve risolvere, o equivalentemente si dice che *risolve* il problema, se e solo se fornisce in output la risposta corretta *per ogni* istanza del problema
  - per dimostrare che un algoritmo è corretto occorre tipicamente una vera e propria dimostrazione, come se si trattasse di un teorema (ed in effetti, lo è)
  - per esempio, se ne potrebbe fare uno per dimostrare che l'*Insertion Sort* o *Convert* risolvono correttamente i rispettivi problemi
  - in questo corso ci si accontenterà di dare l'idea del perché un algoritmo è corretto

- tipicamente è più facile far vedere che un algoritmo è scorretto
- si consideri ad esempio l'algoritmo *Stupid Sort* in Figura 3
  - **Esercizio**: provare ad eseguire l'algoritmo *Stupid Sort* sulla macchina di Von Neumann avendo come input la sequenza $A = \langle 5, 4, 3, 1, 2 \rangle$

```
1 Stupid-Sort(A)
2   if (length(A) ≥ 2) then
3     swap A[length(A) − 1] and A[length(A)]
```

Figure 3: Algoritmo *Stupid Sort*

- ci sono (infinite) istanze su cui *Stupid Sort* funziona bene
- per esempio, sulle istanze $\langle 34, 45, 35 \rangle$, o $\langle 2 \rangle$, o $\langle 12, 27, 30, 44, 42 \rangle$...
- ma basta trovare anche un solo controesempio: in questo caso ce ne sono infiniti, per esempio $\langle 12, 27, 30, 42, 44 \rangle$, oppure $\langle 27, 12, 30, 42, 44 \rangle$...
- quindi, *Stupid Sort* non è corretto, o equivalentemente non risolve il problema dell'ordinamento
- si consideri ora l'algoritmo *Insertion Sort Bis* in Figura 4
  - **Esercizio**: provare ad eseguire l'algoritmo *Insertion-Sort-Bis* sulla macchina di Von Neumann avendo prima come input la sequenza $A = \langle 5, 4, 3, 1, 2 \rangle$, poi la sequenza $A = \langle 1, 2, 3 \rangle$

```
1 Insertion-Sort-Bis(A)
2   for j ← 2 to length(A) do
3     key ← A[j]
4     i ← j − 1
5     while i > 0 and A[i] > key do
6       A[i + 1] ← A[i]
7     A[i + 1] ← key
```

Figure 4: Algoritmo *Insertion Sort Bis*

- se l'input è già ordinato, sembra corretto
- ma se deve eseguire anche una sola volta il corpo del ciclo `while`, non ne esce più
- ovvero ci sono istanze sulle quali *non termina*
- dato che le specifiche del problema richiedono che l'algoritmo fornisca sempre un output, questo vuol dire che anche *Insertion Sort Bis* non è corretto

  - questo nonostante il fatto che, quando termina, produca sempre il risultato corretto

- La correttezza non è l'unico parametro di bontà di un algoritmo: ce n'è un altro, che è la sua *complessità*
  - *complessità temporale*: se l'input è "lungo" (ossia, ha dimensione) n, quanto tempo, in funzione di n, occorre aspettare per vederlo completato?
  - *complessità temporale* (più realistico): se l'input è "lungo" (ossia, ha dimensione) n, e ho due algoritmi diversi che risolvono uno stesso problema, c'è un qualche valore di n a partire dal quale uno dei due algoritmi è sempre migliore dell'altro in termini di tempo di esecuzione?
  - *complessità spaziale*: se l'input è "lungo" (ossia, ha dimensione) n, quanta RAM aggiuntiva (a parte l'input stesso), in funzione di n, occorre usare per eseguire l'lgoritmo?
  - *complessità spaziale* (più realistico): se l'input è "lungo" (ossia, ha dimensione) n, e ho due algoritmi diversi che risolvono uno stesso problema, c'è un qualche valore di n a partire dal quale uno dei due algoritmi è sempre migliore dell'altro in termini di utilizzo di RAM ulteriore?
  - in questo corso si tratterà quasi esclusivamente di complessità temporale, ed in maniera intuitiva
  - quando si parla di complessità, ci si può riferire al caso peggiore, medio o migliore
  - ciò è dovuto al fatto che un algoritmo può andare benissimo su alcune istanze e malissimo su altre (ne saranno forniti esempi)
  - pertanto, la complessità del caso peggiore fa riferimento all'istanza che fa andare l'algoritmo al peggio delle sue possibilità; quella del caso medio considera le prestazioni su un'istanza "media" (quella che si dovrebbe presentare nella media dei casi); quella del caso migliore fa riferimento all'istanza che fa andare l'algoritmo al meglio delle sue possibilità
  - in questo corso, si farà sempre riferimento al caso peggiore, che è quello che *garantisce* le prestazioni di un algoritmo
    * di più non occorre aspettare...
  - da notare anche che si fa sempre riferimento ad una dimensione dell'input: per ogni problema occorre definire tale dimensione in modo appropriato
    * nel problema dell'ordinamento, la dimensione dell'input coincide con il numero di elementi nell'array
    * nel caso della conversione, è il numero di bit necessarie a rappresentare il numero in input

– esempio di complessità: si consideri nuovamente l'*Insertion Sort* (Figura 2) e il *Total Sort* (Figura 5)
  * **Esercizio**: provare ad eseguire l'algoritmo *Total Sort* sulla macchina di Von Neumann avendo come input la sequenza $A = \langle 3, 1, 2 \rangle$

```
1 Total-Sort(A)
2   for B in permutations(A)
3     ok ← true
4     for i ← 1 to length(B) - 1 do
5       if B[i+1] < B[i]
6         ok ← false
7    if ok
8      copy B into A
9      return
```

Figure 5: Algoritmo *Total Sort*

  * da notare anche che il *Total Sort* astrae parecchio dai "dettagli"
  * non dice come si enumerano le permutazioni, assumendo che chi debba implementare l'algoritmo lo sappia fare
  * qui è ok perché serve solo a far vedere cosa succede con gli algoritmi inefficienti
  * nello pseudocodice si può, con un linguaggio di programmazione "vero" no...
– è possibile mostrare che l'*Insertion Sort* ha complessità nel caso pessimo di circa n^2, mentre il *Total Sort* ha complessità sempre nel caso pessimo di circa $n(n!)$
– qualche conto: supponendo che l'*Insertion Sort* impieghi esattamente n^2 microsecondi per un'istanza di dimensione n, e il *Total Sort* impieghi esattamente $n(n!)$ microsecondi sulla stessa istanza, allora si ha che nello stesso lasso di tempo (circa 7 mesi e mezzo) l'*Insertion Sort* ordina 4.428 milioni di numeri, mentre il *Total Sort* ne ordina 15...
– in generale e con un po' di semplificazioni, se un algoritmo ha una complessità *polinomiale* allora va bene, se ha complessità *esponenziale* allora va male
  * il fattoriale ha comportamento asintotico superiore all'esponenziale e^n ma inferiore a n^n (formula di Stirling: $n! \approx \sqrt{2n\pi}\frac{n^n}{e^n}$)
– per quanto riguarda la complessità spaziale, l'*Insertion Sort* usa solo una RAM aggiuntiva costante, mentre il *Total Sort* necessita di altri n numeri in RAM (per l'array B)

- Problematiche a latere che verranno accennate solo qui:
  - esistono problemi *non calcolabili*
  - vuol dire che non è possibile scrivere un algoritmo che li risolva
  - alcuni perché non è chiara la relazione input-output (ad es: dire se un quadro dato in input è bello oppure no)
  - altri perché, pur essendo descritti in modo matematicamente corretto, contengono potenziali paradossi al loro interno; ad esempio:

    input: un programma scritto in C (ma vanno bene anche altri linguaggi, ad es. Java)

    output: "si" se e solo se il programma si arresta su ogni suo possibile input, "no" altrimenti
  - noi ci occuperemo solo di problemi calcolabili

Introduzione alla programmazione in C

- Primo semplice programma in C: Figura 6

```
1 #include <stdio.h>
2
3 int main()
4 {
5  printf("Ciao mondo!!!!\n");
6  return 0;
7 }
```

Figure 6: Primo semplice programma in C

  - va scritto e salvato in un file di solo testo; supponiamo che il nome del file sia `helloworld.c` (il suffisso `.c` è obbligatorio)
    - su Windows, non è il caso di usare Word, basta il Blocco Note, ma molto meglio usare direttamente DevCpp
    - su Linux, vanno benissimo Gedit, Nedit, Emacs...
  - va *compilato*, creando così un nuovo file eseguibile (`helloworld.exe` per Windows, `a.out` per Linux)
    - su Windows, basta cliccare sull'apposita voce di DevCpp
    - su Linux, occorre aprire un terminale e scrivere (da dentro la directory dove si trova `helloworld.c`) la seguente riga di comando: `gcc helloworld.c`
  - va eseguito

    - su Windows, conviene aprire un Prompt di DOS, navigare fino alla directory che contiene `helloworld.c` (e dove ora sarà stato creato `helloworld.exe` e scrivere `helloworld`
    - su Linux, sullo stesso terminale di sopra, occorre scrivere la seguente riga di comando: `./a.out`
  - in caso di errori, la compilazione listerà tali errori, e il nuovo file eseguibile non verrà creato
    - se eventualmente ne esiste uno vecchio, risultante di una precedente compilazione, tale file non verrà cancellato

- **Esercizio**: provare a fare piccole modifiche (ad es. far scrivere il proprio nome) al programma, ricompilarlo e rieseguirlo, nonché ad introdurre errori (ad es. dimenticandosi qualche parentesi graffa)