Informatica per Statistica
Riassunto della lezione del 09/10/2013

Igor Melatti

Il sistema operativo

- Tutto l'hardware del mondo è inutile senza software
  - nella Fig. 1 si vede che tra l'utente (che usa il computer) e l'hardware ci sono 2 strati
  - quello più vicino all'hardware è il *sistema operativo* (Windows, Linux, MacOS-X, ...)
  - quello più vicino all'utente contiene i *programmi applicativi* (semplicemente *applicazioni* o *programmi*) che l'utente usa direttamente
    - programmi per scrivere (Word)
    - fogli di calcolo (Excel)
    - programmi multimediali (Windows Media Player)
    - questi sono per Windows, sia chiaro che esistono i corrispettivi programmi per Linux e MacOS-X
    - altro (videogiochi, programmazione, ...)
  - per far sì che un software possa essere usato, occorre *installarlo*
    - semplificando, vuol dire copiarlo su disco fisso in modo che possa essere propriamente eseguito quando serve
    - tipica sorgente per l'installazione è un file preso da un qualche supporto di input (rete, CD, DVD, chiave USB...)
    - vale sia per l'intero sistema operativo che per i singoli applicativi
    - a seconda della complessità del programma stesso, può essere un'operazione facile o difficile
    - un sistema operativo è tra le cose più difficili da installare
  - usare un software vuol dire *eseguirlo* (o anche avviarlo, o aprirlo, talvolta *lanciarlo*)
    - semplificando, vuol dire prenderlo dal disco fisso e copiarlo (almeno in parte) in RAM
    - dalla RAM può essere *eseguito* (almeno secondo il modello di Von Neumann)

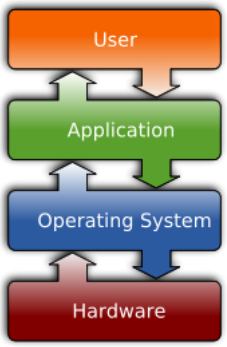

Figure 1: Architettura di un sistema operativo

    - se va tutto bene, un software si installa *una sola volta*, e lo si può solitamente eseguire *molteplici volte*
  - il sistema operativo non viene mai modificato direttamente dall'utente "normale"
    - al massimo, si eseguono gli *aggiornamenti* predisposti da chi ha programmato il sistema operativo stesso
    - tali aggiornamenti, ovviamente, modificano il sistema operativo in una qualche parte
  - è invece abbastanza frequente che un utente crei nuovi programmi applicativi, per sé o anche per altri
    - per farlo, occorre conoscere un qualche *linguaggio di programmazione*, ed avere degli opportuni programmi applicativi con la seguente caratteristica: si tratta di programmi (che si chiamano *compilatori*) che creano altri programmi
    - in alcuni casi, anziché di compilatori, si parla di *interpreti*, non trattati in questo corso
    - un po' a parte è il discorso sugli applicativi creati per gestire basi di dati con DBMS; se ne tratterà a fine corso

- tipicamente, si acquista il sistema operativo insieme col computer (sistema operativo *preinstallato*)
  - nel caso dei Mac di Apple, praticamente ci può essere solo una delle versioni di MacOS-X
  - nel caso degli IBM-compatibili, nel 99% dei casi (fino a 2 anni fa quasi il 100%) c'è Windows
  - al giorno d'oggi ci sono anche computer con Linux come sistema operativo

        * costano una 50ina di euro in meno, perché Linux è gratuito
    - si possono acquistare IBM-compatibili anche senza sistema operativo
    - in questo caso, una volta che lo si accende, lui fa qualche controllo e poi si blocca con una schermata nera lamentandosi di non avere un sistema operativo

- di solito, l'utente normale si limita ad *aggiornare* il sistema operativo
    - di solito per correggere qualche problema di sicurezza riscontrato di recente
    - anche cambiare il sistema operativo alla versione successiva (*upgrade*, ad esempio da Windows XP a Windows Vista) è sostanzialmente un aggiornamento
    - qualche volta capita di dover riavviare il computer, ma è comunque una cosa semplice da fare

- tuttavia, è possibile anche *disinstallare* il sistema operativo (pre)installato e installarne un altro
    - operazione non semplicissima, ma neanche impossibile
    - con i Mac, lo fanno solo i superhacker
    - sugli IBM-compatibili ci si può sbizzarrire a volontà
    - la cosa tipica che fa chi si intende un po' di informatica è fare un sistema *dual-boot*
        * quando lo si accende, viene chiesto se far partire Windows o Linux
        * con il termine "boot" si intende quello che fa il computer non appena lo si accende
        * in pratica è la fase in cui viene "caricato" il sistema operativo
        * ovvero, la fase il sistema operativo fa alcune cose iniziali che gli permetteranno poi di funzionare
        * può durare da pochi secondi e un paio di minuti
    - aggiornare un sistema operativo (passare da una versione precedente ad una più recente) è facile
    - cambiare totalmente un sistema operativo (da Windows a Linux o viceversa) è più complicato
        * se non si sta attenti, si perdono tutti i dati

- l'accensione di un computer: cosa succede?
    1. *bootstrap* (o solo *boot*): una parte di software indipendente dal sistema operativo fa alcuni controlli sul funzionamento di alcune parti dell'hardware

   - ad esempio, vede quanta RAM c'è e controlla che funzionino tastiera e monitor
   - questa parte di software si trova su una memoria RAM che però non è volatile, e neanche modificabile
   - per questo motivo viene chiamata ROM (Read Only Memory)
   - nei sistemi IBM-compatibili, questa procedura software si chiama BIOS (Basic Input Output System)
2. l'ultima parte del boot fa partire il sistema operativo, da disco, da CD o da floppy
   - è possibile scegliere la *sequenza di boot* all'inizio
   - infatti, alcune parti del BIOS sono modificabili
   - in pratica, occorre premere un qualche tasto (ad esempio F9) nei primi 2-3 secondi di accensione del computer
3. dopodiché le istruzioni da eseguire vengono tratte dal disco, da CD o da floppy a seconda della scelta precedente
   - il sistema operativo vero e proprio parte praticamente sempre da disco
   - negli altri casi:
     * si è su un sistema piccolo o senza hard disk, e quindi il dischetto o il CD contengono il sistema operativo
     * oppure si vuole installare un nuovo sistema operativo che si trova su dischetto (difficile) o su CD
     * in questo caso, si verrà avviati ad una serie di passi che costituiscono il programma d'installazione
     * oppure è successo qualcosa di grave e il dischetto o il CD contengono programmi di recovery
4. seguendo tali istruzioni (supponendo che non si tratti di installazione o di recovery), il sistema operativo comincia pian piano a posizionare le sue parti vitali in RAM
5. quando è tutto pronto, si arriva alla schermata di login
   - occorre immettere username e password
   - sui PC (soprattutto Windows), questa parte viene spesso saltata
   - in pratica, si accede come utente generico (*user* o *guest*) senza password
   - sui PC non personali (quelli dei laboratori), chi amministra il sistema (dei tecnici pagati per questo) deve aver rilasciato all'utente un *account*, ovvero uno username e una password
   - di solito, viene chiesto di modificare subito la password al primo accesso
   - a meno che non si sia amici dell'amministratore, solitamente non è possibile cambiare lo username

  - c'è sempre uno user onnipotente (solitamente chiamato *superuser*, *root* o *administrator*)
  - è l'unico che può modificare il sistema operativo, installando applicazioni o facendo aggiornamenti

6. poi viene caricata la GUI (Graphical User Interface) e si può cominciare ad usare il computer
   - alcuni vecchi sistemi operativi non avevano la GUI, oppure occorreva farla partire come un'applicazione in un secondo momento
   - in questi casi, c'era una schermata nera con un *prompt* che attendeva comandi da tastiera
   - oggi la situazione è invertita: si parte con la GUI ed eventualmente si fa partire la schermata nera come un'applicazione
7. quando si finisce si fa *logout* (ci si *disconnette*) ed eventualmente si spegne il computer
   - fare logout senza spegnere equivale a permettere ad altre persone con un altro account di accedere
   - lo spegnimento va fatto da software, chiedendo al sistema operativo di spegnersi
   - infatti, per potersi poi riavviare correttamente la prossima volta, occorre che salvi alcune informazioni prima di spegnersi
   - per questo motivo, se si preme normalmente sul pulsante di accensione, il computer resta accesso
   - se si vuole proprio spegnere senza permettere al sistema operativo i suoi salvataggi usuali (cosa ad esempio necessaria se il sistema operativo stesso si è bloccato) occorre tener premuto il tasto di spegnimento per qualche secondo

- per usare il sistema operativo ci sono sostanzialmente 2 modi
  - da interfaccia grafica (quindi con la GUI)
    - si va avanti a clic, doppi clic e trascinamenti (drag and drop) col mouse
    - la tastiera viene usata solo quando è indispensabile
      - per esempio, per scrivere una lettera con Word, prima si apre Word e poi si scrive la lettera
      - per andare su un sito, prima si apre il Browser (es. Internet Explorer o Firefox) e poi si scrive l'indirizzo
    - ci sono svariati modi per eseguire un programma
      - fare doppio clic su un'icona sul Desktop
      - fare clic in una barra di avviamento veloce
      - selezionare il programma dai menu

    * all'interno di un programma, si può usare il mouse o la tastiera in tanti modi, dipendentemente dal programma stesso
        · ad esempio, sul browser si usa prevalentemente il mouse per cliccare sui link, su Word si usa prevalentemente la tastiera per scrivere
  - da interfaccia a linea di comando (CLI, Command Line Interface)
    * è più da geek
    * l'utente "normale" non lo fa mai
    * in maniera pura e "totale" (ovvero usando un sistema operativo che, una volta avviato, mostra solamente lo schermo nero dei comandi), non lo si fa più
    * quello che talvolta qualche utente (non proprio "normale") fa è di eseguire un programma applicativo che apre uno schermetto nero, dove si possono dare comandi direttamente al sistema operativo
    * in Windows, questa applicazione si chiama `Prompt dei comandi` o `DOS prompt`
    * sotto Linux la cosa è più naturale, e ci sono svariate applicazioni: `konsole` e `gnome-terminal` sono le più comuni
    * sotto Mac, la situazione è quella di un Linux "ingentilito"
    * i tipici comandi che si danno riguardano il file-system, o servono per lanciare altri programmi
        · è un po' più potente della GUI, perché si possono lanciare i programmi con i *parametri*
        · non che con la GUI non lo si possa fare, ma talvolta non è altrettanto agevole
    * con la GUI, si fanno le stesse cose ma per quanto riguarda il filesystem occorre prima lanciare qualche altra applicazione (es. Explorer)

- tutti i moderni sistemi operativi (almeno quelli che interessano a noi) sono *multitasking*

  1. lo possono usare contemporaneamente più utenti
     - sia in modo lasco, come ad esempio nella seguente situazione: un utente fa login, lancia e lascia un programma in esecuzione e fa logout, e poi fa login un altro utente che lancia altri programmi (mentre il programma del primo è ancora in esecuzione)
     - sia in modo massivo, cioè il computer è connesso in rete e più utenti contemporaneamente ci accedono con programmi appositi
     - era quello che succedeva nei vecchi mainframe con i *terminali*

2. uno stesso utente può far eseguire più applicazioni contemporaneamente
3. in entrambi i casi, le varie applicazioni si alternano, sotto il comando del sistema operativo, nell'uso delle risorse della macchina
   - prima fra tutte la CPU...
   - la parte del sistema operativo che si occupa di ciò è chiamata scheduler (che si può tradurre con "organizzatore")
   - se due utenti, o uno stesso utente, lanciano due programmi che richiedono molte risorse di computazione (CPU, RAM o altro), allora si potrà notare un decremento delle prestazioni del computer

- file system
  - tutti i vari dispositivi di memorizzazione di massa (dischi fissi e non, chiavi USB, CD e DVD) sono organizzati con un file system
  - vuol dire che le informazioni sono memorizzate in archivi (*files*), ad ognuno dei quali occorre dare un nome
  - è compito dell'utente dare un nome a ciascun file in maniera da ricordare poi che informazioni ci sono dentro
    * fanno eccezione a questa regola i files di sistema
    * cioè quelli che servono al sistema operativo
    * lo stesso sistema operativo è inizialmente un file, prima del boot
    * i files di sistema non possono essere toccati, a meno di non essere superuser
  - dato che i files sono spesso molti, non è comodo dover dare un nome diverso a ciascuno di essi
  - e comunque sarebbe difficile cercare un files tra migliaia di altri simili
  - quindi i file systems offrono di organizzare il tutto in cartelle (*directories*)
    * sono dei files anche le directories, ma sono files speciali
    * servono per contenere altri files
  - le directory sono organizzate ad albero
    * c'è una directory iniziale che contiene tutte le altre (`/` in Linux, `C:\` in Windows)
    * ogni directory può contenere sia files che altre directories
    * all'interno di ogni directory, non ci possono essere 2 files con lo stesso nome
  - operazioni tipiche (per files si intende sia files regolari che directory)
    * creare un file vuoto (con un nuovo nome)

    - cancellare un file
        - nel caso di una directory si cancella tutto quello che c'è dentro
        - tipicamente, il sistema operativo chiede conferma
    - copiare un file in una diversa directory o nella stessa ma con un altro nome
    - spostare un file in una diversa directory o nella stessa ma con un altro nome
  - tipica forma del nome di un file: `qualcosa.estensione`
    - l'estensione è tipicamente di 3 lettere, ma non necessariamente
    - serve per capire con quale applicazione aprire il file relativo

Introduzione agli algoritmi

- La parola *algoritmo* deriva dal nome di un matematico persiano del IX secolo, *Al-Khwarizmi*
- 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, ovvero tale che $\sum_{i=1}^{k} b_i 2^{i-1} = n$

  - da notare che la sequenza di bits in output è da leggersi al contrario, ovvero il primo elemento della sequenza è il bit meno significativo (che dovrebbe quindi stare a destra e non a sinistra)
  - **Esercizio**: riscrivere la specifica del problema in modo tale che la sequenza in output sia nel verso giusto (il bit meno significativo deve quindi essere b_k e non b_1); attenzione a scrivere la sommatoria...

- Algoritmo per la conversione (già visto, ma riscritto con la notazione che verrà usata nel resto del corso): Figura 2

  - **Esercizio**: provare ad eseguire l'algoritmo *Convert* sulla macchina di Von Neumann avendo come input 62
  - **Esercizio**: provare a riscrivere l'algoritmo *Convert* in modo tale che l'output sia nel verso giusto, ed eseguirlo nuovamente con input 62

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

Figure 2: 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 3
  - **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 3: 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 Python 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)

- 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 3 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
    · riguardo ai blocchi di istruzioni, Python si comporta in maniera esattamente uguale
    · non è obbligatorio che l'indentazione sia esattamente 2 spazi; può essere anche, ad esempio, 4 spazi o 1 TAB (singolo carattere ASCII che identifica un numero fisso di spazi consecutivi, solitamente 8)
    · l'importante è che, una volta decisa un'indentazione all'interno di un blocco, si usi sempre quella
    · ad esempio, non si può indentare la riga 7 di 1 spazio e la riga 8 di 3 spazi
    · questo vale sia per lo pseudocodice che, soprattutto, per Python
  * gli algoritmi trattano spesso di *sequenze* (chiamate anche *array*), 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`]`
    · matematicamente, le sequenze verranno scritte come $A = \langle a_1, \ldots, a_n \rangle$
    · da notare che, mentre ad esempio $\{a_1, \ldots, a_n\} = \{a_n, \ldots, a_1\}$ (gli insiemi non hanno ordine), si ha invece $\langle a_1, \ldots, a_n \rangle \neq \langle a_n, \ldots, a_1 \rangle$ (le sequenze hanno ordine)
  * il primo elemento di una sequenza A è A`[1]`, l'ultimo è A`[length(`A`)]`
    · come si vedrà, in Python, invece, si va da A`[0]` a A`[len(`A`) - 1]`
    · in Python, si usa il termine *lista* per indicare una sequenza

- 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. 2 e *Insertion-Sort* in Fig. 3
- in Python, occorrerà scrivere una sequenza di funzioni (qualcuna potrebbe essere già stata scritta, e basta solo riusarla) e di istruzioni da eseguire (e che possono chiamare le funzioni definite in precedenza)
  * inoltre, in Python 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 due comandi (o meglio, come si vedrà, due chiamate a funzione) chiamati `input` (per i numeri) e `raw_input` (per tutto il resto); invece per scrivere l'output si usa la `print`
  * questo potrebbe portare a pensare che le `input` e/o le `raw_input` del Python siano sempre l'input delle funzioni di un programma Python, così come le `print` siano sempre l'output delle stesse funzioni
  * *non* è così, dovrà essere chiaro nel prosieguo di questo corso
  * al più, `input` (e/o `raw_input`) e `print` (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 4
    * **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 4: 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 5

∗ **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 5: 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'algoritmo?
- *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 necessari a rappresentare il numero in input
- esempio di complessità: si consideri nuovamente l'*Insertion Sort* (Figura 3) e il *Total Sort* (Figura 6)
  - **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 6: 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 Python (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 Python

- Primo semplice programma in Python: Figura 7

```
1 print "Ciao mondo!!!!";
```

Figure 7: Primo semplice programma in Python

  - va scritto e salvato in un file di solo testo; supponiamo che il nome del file sia `helloworld.py` (il suffisso `.py` non è obbligatorio, si potrebbe anche non mettere alcun suffisso)
    * su Windows, non è il caso di usare Word, basta il Blocco Note, ma molto meglio usare programmi appositi per linguaggi di programmazione, come NotePad++
    * su Linux, vanno benissimo Gedit, Nedit, Emacs...
  - è poi possibile *eseguire* il file di testo, passandolo all'*interprete Python*
    * su Windows, conviene aprire un Prompt di DOS, navigare fino alla directory che contiene `helloworld.py` e scrivere `python helloworld.py`
    * su Linux, occorre aprire un terminale e scrivere (da dentro la directory dove si trova `helloworld.py`) la seguente riga di comando: `python helloworld.py`
  - in caso di errori, l'esecuzione terminerà mostrando tali errori, senza eseguire il programma

- **Esercizio**: provare a fare piccole modifiche (ad es. far scrivere il proprio nome) al programma e ad eseguirlo, nonché ad introdurre errori (ad es. scrivendo `pr int` anziché `print`)