# Algoritmi (e Complessità)

## Capitolo 5 del testo

Alberto Policriti

IGA
ISTITUTO DI GENOMICA APPLICATA

UNIVERSITAS STUDIORUM UTINENSIS

14 Novembre, 2019

# Algoritmo: definizione

**Kleene**

*E' una risposta finita ad un numero infinito di domande.*

# Algoritmo: definizione

**Kleene**

> *E' una risposta finita ad un numero infinito di domande.*

An algorithm is an ordered set

of unambiguous, executable steps

that defines a terminating process.

# Polya

**Phase 1** Understand the problem.

**Phase 2** Devise a plan for solving the problem.

**Phase 3** Carry out the plan.

**Phase 4** Evaluate the solution for accuracy and for its potential as a tool for solving other problems.

# Polya

**Phase 1** Understand the problem.

**Phase 2** Get an idea of how an algorithmic function might solve the problem.

**Phase 3** Formulate the algorithm and represent it as a program.

**Phase 4** Evaluate the program for accuracy and for its potential as a tool for solving other problems.

# Il problema computazionale

- Un *problema (computazionale)* specifica una *relazione* input-output
- Esempi:
  - un problema di ordinamento
    - Input: una sequenza di numeri naturali
    - Output: la sequenza ordinata
- Un *algoritmo* che risolve il problema specifica una procedura effettiva per ottenere la relazione desiderata

# Efficienza degli algoritmi

- Ogni problema può avere diverse soluzioni algoritmiche
  - o non averne alcuna...
- Qual è l'algoritmo "migliore"?
- Occorre un modo per confrontare l'*efficienza* degli algoritmi
- Bisogna definire che cosa si intende per "efficienza"
- L'efficienza si può misurare sulla base delle *risorse* richieste dall'algoritmo

# Analisi degli algoritmi

- *Analizzare un algoritmo* significa prevedere le risorse che l'algoritmo richiede
- Quali risorse? Principalmente
  - tempo di esecuzione
  - spazio di memoria
- Soprattutto il tempo è un fattore importante
  - lo spazio si può riutilizzare, il tempo no
- Queste risorse dipendono, in generale, dai dati in ingresso

# Modelli di computazione

- Un *modello di computazione* è un'astrazione matematica di ciò che ritieniamo sia una "macchina calcolatrice"
- Definisce la tecnologia con cui sono realizzati gli algoritmi
- Specifica
  - le operazioni primitive e il loro costo
  - le risorse disponibili e il loro costo

# Esempio: ricerca lineare

- Ricerca di un valore $v$ in un array $L$:

```
$i = 0;
while ($i < scalar(@L) and $v != $L[$i]) {
      ++$i;
}
```

- Se $L[0] == $v, la linea 2 è eseguita una volta e la linea 3 mai
- Se $v non occorre in @L, la linea 2 è eseguita $|L| + 1$ volte e la linea 3 $|L|$ volte, dove $|L|$ è la lunghezza dell'array $L$.

# Tempo di esecuzione

- *Tempo di esecuzione* di un algoritmo su un particolare input: è il numero di operazioni primitive, o passi, eseguiti
  - ad esempio, assumiamo che ciascuna linea del codice precedente costituisca un passo...
  - ...e che ogni passo richieda un tempo fissato $T_0$
  - allora, il tempo di esecuzione della ricerca varia da $3\,T_0$ (caso migliore) a $(3 + 2|L|)\,T_0$ (caso peggiore)

# Ricerca lineare: osservazioni

- Il tempo di esecuzione dipende dall'input
  - dalla lunghezza della lista @$L$
  - dalla posizione di \$$v$ in @$L$
- In generale, il tempo di esecuzione cresce con la dimensione dell'input
  - Ma il tempo di *quale* esecuzione?
  - Nella migliore situazione possibile?
  - Nella peggiore situazione possibile?
  - In una situazione intermedia?

# Analisi del caso pessimo

- La stima del tempo di esecuzione nel caso peggiore è particolarmente importante
  - è un limite superiore al tempo di esecuzione
    - "non può andare peggio di così"
- Altre stime interessanti:
  - tempo medio
    - spesso difficile da calcolare
    - richiede assunzioni sulla distribuzione dell'input
  - tempo del caso ottimo
    - limite inferiore alle prestazioni

# Ordine di grandezza

- L'analisi del tempo di esecuzione richiede:
  - un formalismo per la specifica dell'algoritmo
  - l'attribuzione di un costo ad ogni operazione primitiva
- Ulteriore semplificazione:
  - ciò che interessa è l'*andamento*, o *ordine di grandezza*, del tempo di esecuzione
    - l'algoritmo di ricerca, nel caso pessimo, richiede un tempo proporzionale a $|L|$
    - non interessa conoscere le costanti

# Confronto di tempi di esecuzione

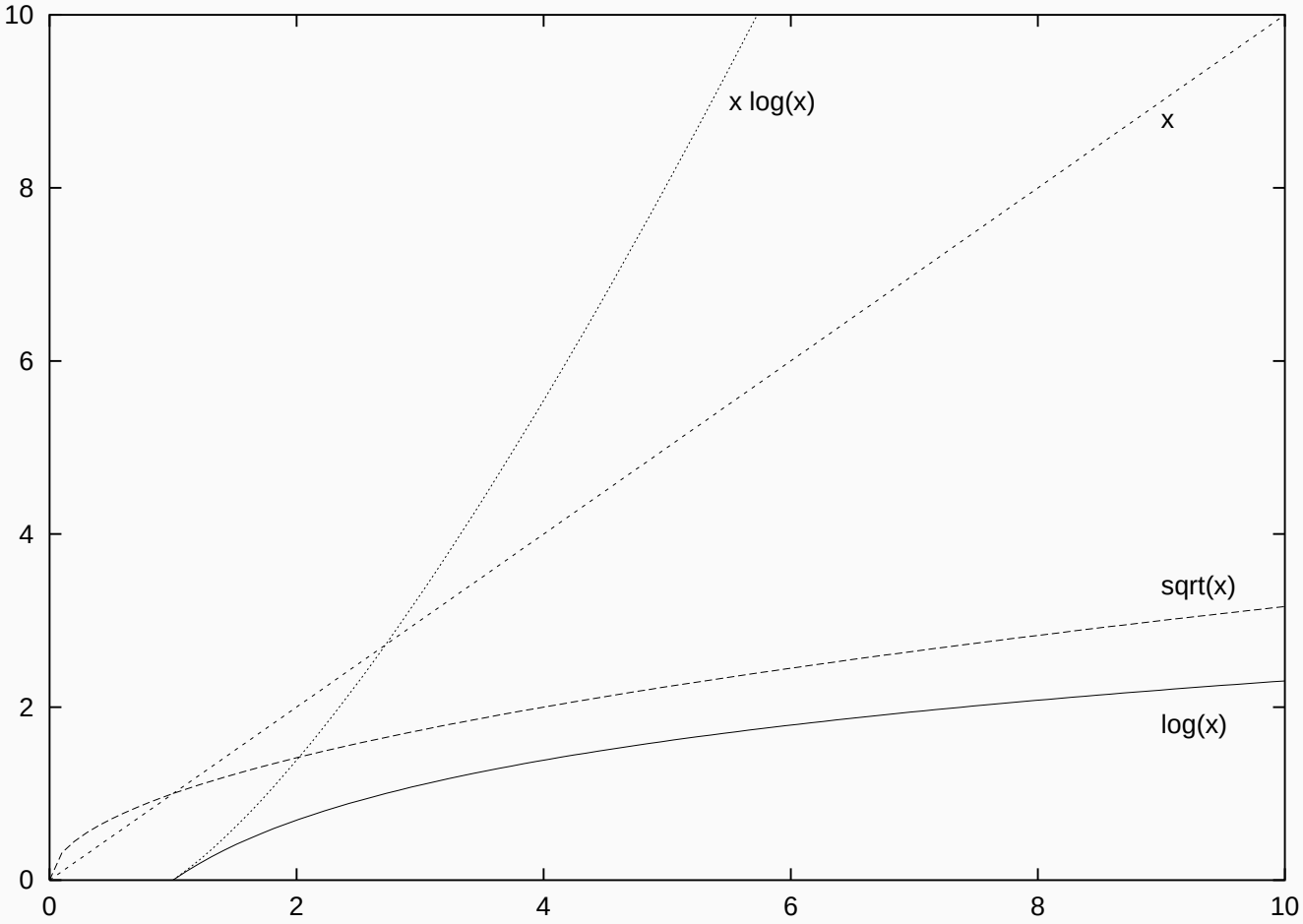
- Un processore *Ghepardo* a 1Ghz esegue un algoritmo di ordinamento che richiede $2n^2$ operazioni per ordinare $n$ numeri
- Un processore *Bradipo* a 1Mhz (anni '80) esegue un algoritmo di ordinamento che richiede $50n \log_2 n$ operazioni su $n$ numeri
- Tempo per ordinare $10^7$ numeri:
  - Ghepardo: $\frac{2 \cdot (10^7)^2}{10^9} = 2 \cdot 10^5 \approx 56 \text{ ore}$
  - Bradipo: $\frac{50 \cdot (10^7) \log_2 10^7}{10^6} \approx 3.2 \text{ ore}$

# Confronto di tempi di esecuzione

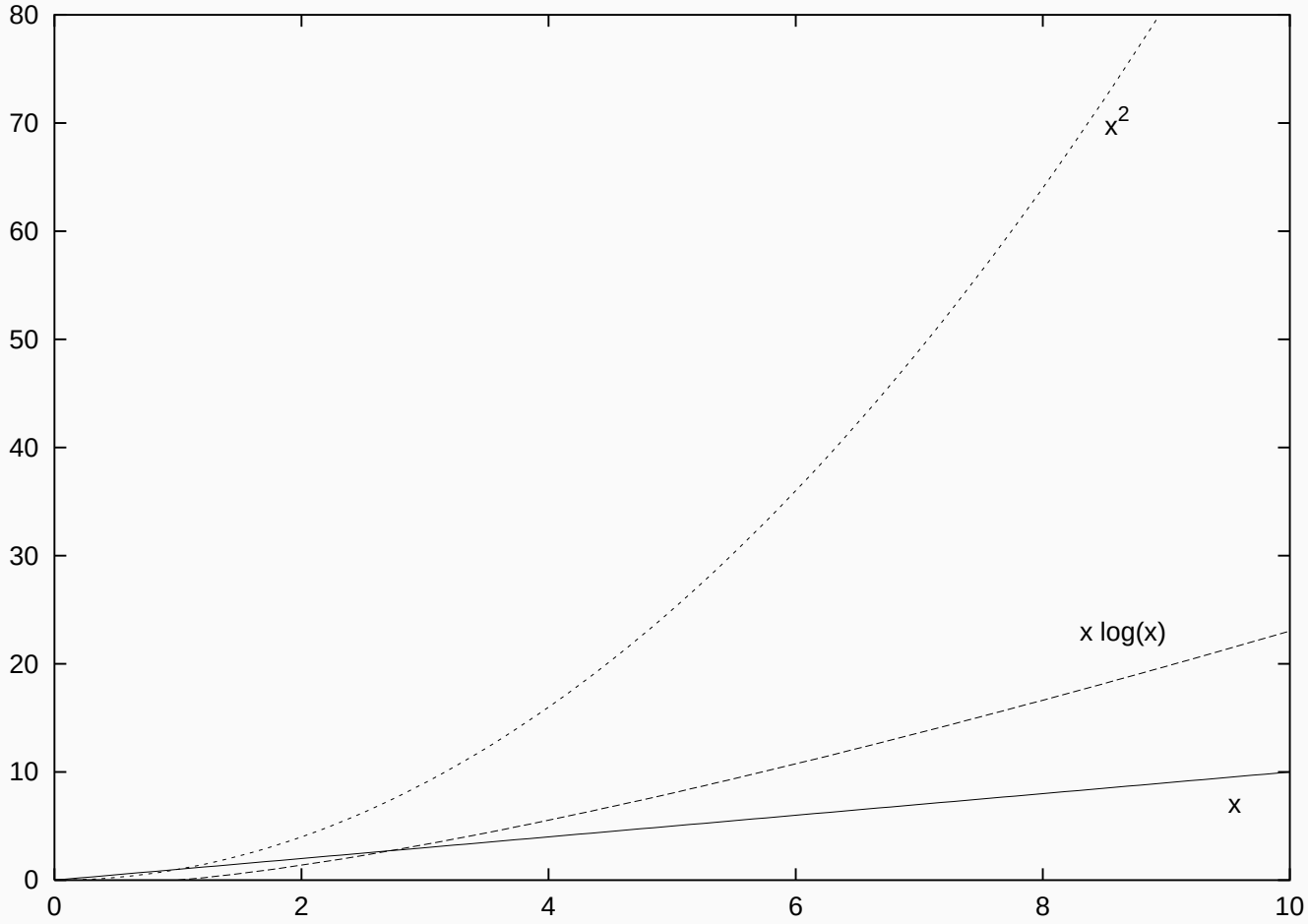
- Sia $\mathcal{A}$ un algoritmo che richiede $f(n) \cdot 10^{-9}$ secondi su un input di dimensione $n$
- Per ogni $f(n)$ e tempo $T$ nella tabella, determinare la massima dimensione $n$ tale che l'esecuzione di $\mathcal{A}$ duri al più $T$.

| | $\log n$ | $\sqrt{n}$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|---|
| 1 s | | | | | | | | |
| 1 min | | | | | | | | |
| 1 ora | | | | | | | | |
| 1 anno | | | | | | | | |

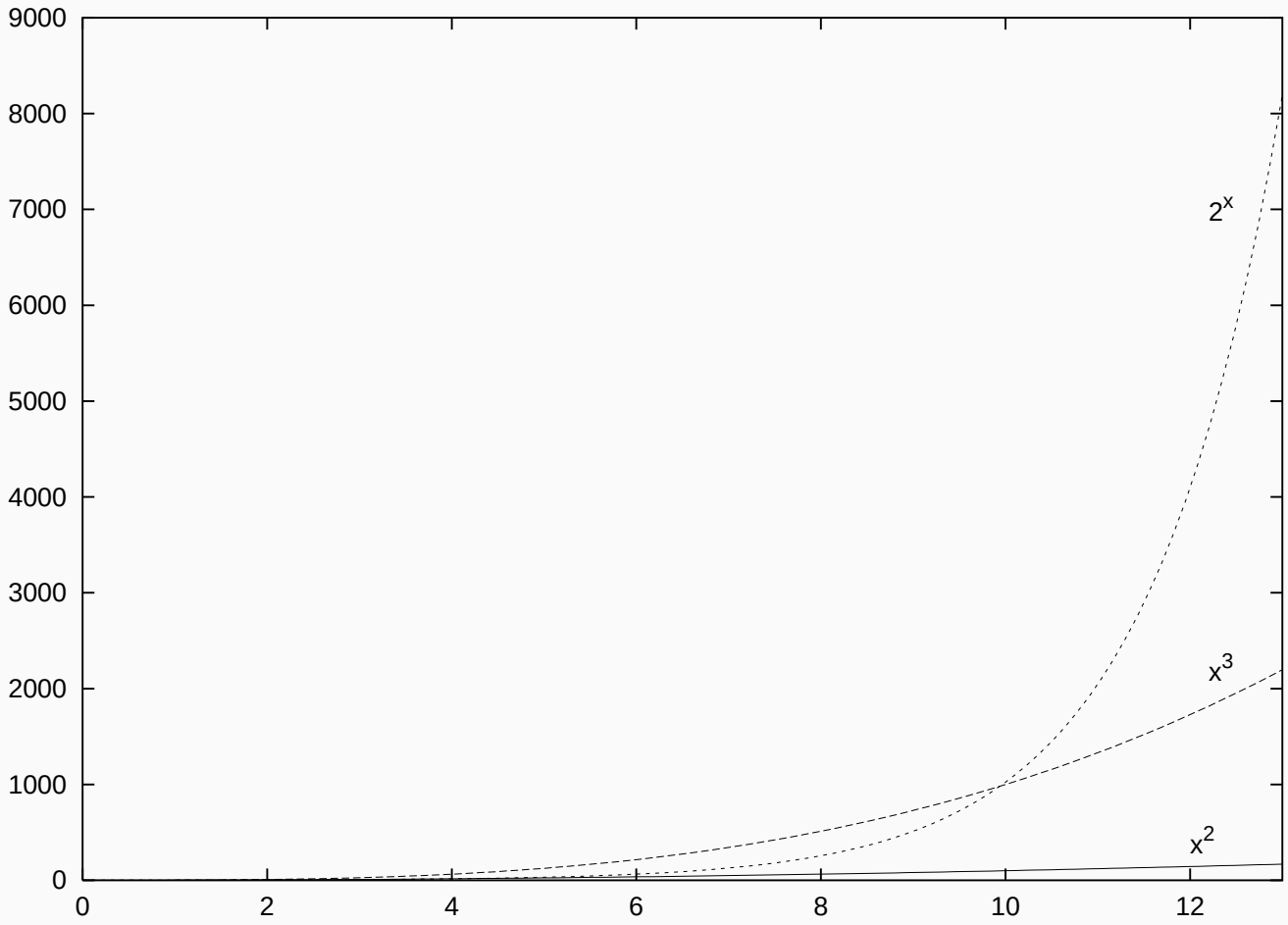
# Grafici di funzioni



# Grafici di funzioni (2)



# Grafici di funzioni (3)



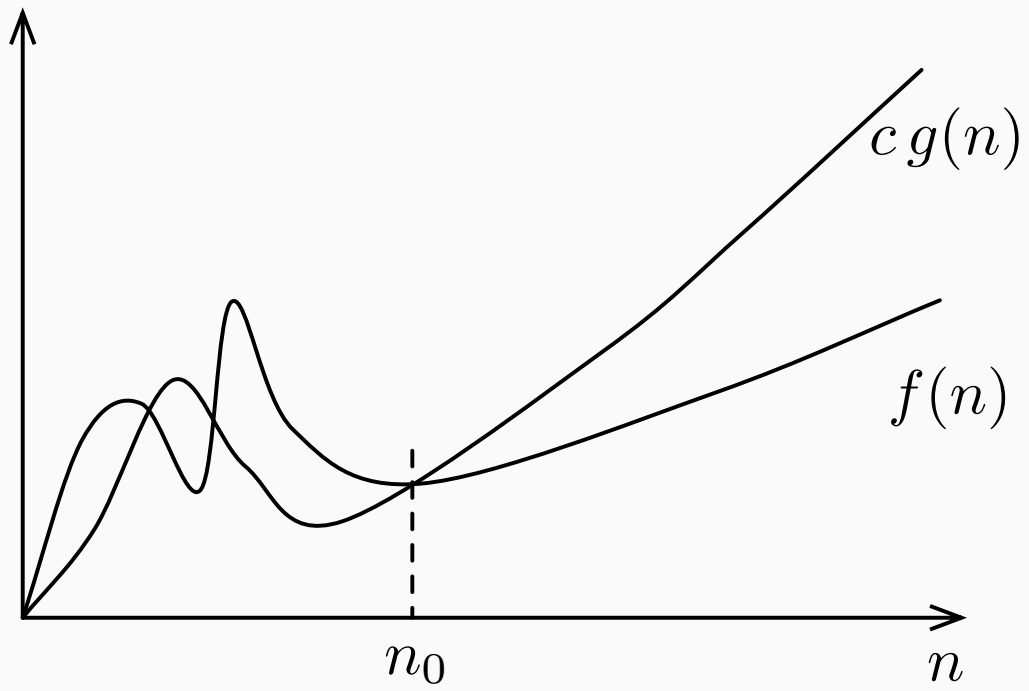
# Notazione asintotica

- *Limite superiore asintotico* per una funzione $g(n)$:

$$\begin{aligned} O(g(n)) \quad = \quad & \{\, f(n) \mid \exists c > 0 \; \exists n_0 > 0 \\ & \forall n \geq n_0 . \, 0 \leq f(n) \leq c \, g(n) \,\} \end{aligned}$$

- La notazione $O$-grande dà un limite superiore a meno di fattori costanti
  - Esempio: l'algoritmo di ricerca lineare prende tempo $O(|L|)$ nel caso peggiore

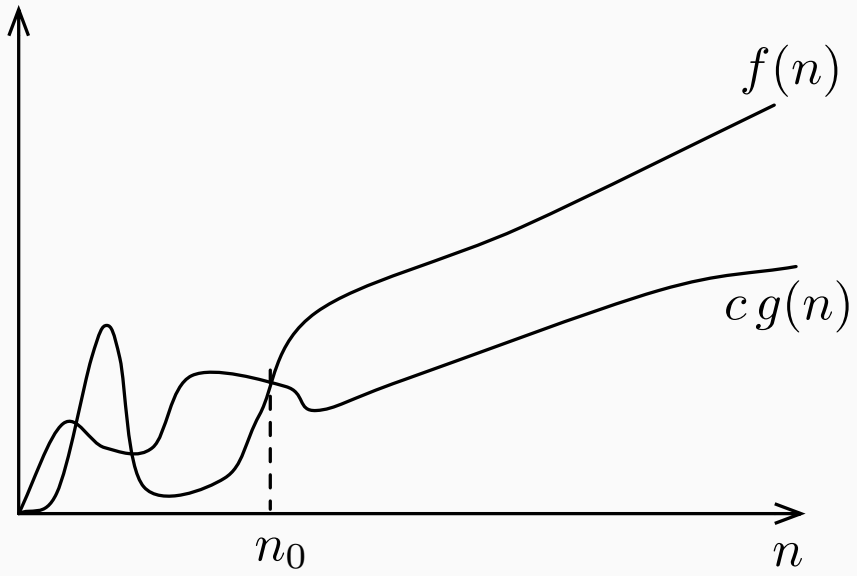
# Notazione asintotica (2)




$$O(g(n)) = \{\, f(n) \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 . 0 \leq f(n) \leq c\,g(n) \,\}$$

# Notazione asintotica (3)

- *Limite inferiore asintotico* per $g(n)$:




$$\Omega(g(n)) = \{\, f(n) \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 . 0 \leq c\,g(n) \leq f(n) \,\}$$

# Esercizi

- Provare che ogni funzione lineare $f(n) = an + b$ è[1] $O(n)$
- Provare che ogni funzione lineare è $O(n^2)$
- Provare che ogni polinomio $\sum_{i=0}^{d} a_i x^i$ è $O(n^d)$
- Provare che $2^{2n}$ non è $O(2^n)$
- Provare che $2^n$ non è $O(n^d)$ per alcun $d$
- Provare che $2^n$ è $O(n^n)$

[1] È convenzione dire che una funzione "è" $O(\cdot)$ intendendo con ciò che "è in" $O(\cdot)$.

# Esercizi (2)

- Scrivere un algoritmo quadratico per la valutazione di un polinomio in un punto
- Scrivere un algoritmo lineare per la valutazione di un polinomio in un punto, sfruttando la *regola di Horner*:

$$\sum_{i=0}^{n-1} a_i x^i = (\cdots(a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0$$

- Scrivere un algoritmo per il problema del pattern matching di stringhe e valutarne la complessità

# Analisi di un programma Perl

- Come determinare la complessità di un programma Perl?
- Istruzioni a tempo costante:
  - assegnamento di un numero

    ```
    $n = 100;
    ```

  - assegnamento di un riferimento a variabile

    ```
    $ref = \@lista;
    ```

  - assegnamento di una stringa di lunghezza uno (carattere)

    ```
    $s = 'z';
    ```

# Analisi di un programma Perl (2)

- Istruzioni a tempo costante (segue):
  - confronto numerico

    ```
    $n > 50, $n != 10, ...
    ```

  - confronto di singoli caratteri

    ```
    'c' eq 'd'
    ```

  - accesso a un elemento di un array o di un hash:

    ```
    $a[10], $h{'chiave'}
    ```

- L'assegnamento di array, array associativi e stringhe e il confronto di stringhe richiedono tempo proporzionale alla loro dimensione

# Analisi di un programma Perl (3)

- Il passaggio di parametri segue le regole dell'assegnamento, così come la restituzione del valore di ritorno di una subroutine
- La complessità delle procedure predefinite è, in generale, più difficile da determinare
  - bisogna consultare la documentazione
- Esercizio:
  - calcolare la complessità degli altri costrutti Perl (**if-then-else**, **do-until**, **for**, **while**, ...)

# Problemi “facili” e “difficili”

- L’analisi di complessità degli algoritmi permette di *classificare i problemi*
  - Dati due problemi $A$ e $B$, supponiamo che:
    - esista un algoritmo per risolvere $A$ in tempo $O(n)$ nel caso pessimo
    - il miglior algoritmo per $B$ sia $O(2^n)$ nel caso pessimo
  - Allora, possiamo concludere che $B$ è un problema “più difficile” di $A$
  - $B$ non ammette soluzioni altrettanto efficienti quanto $A$

# Esempi

- Problemi $O(\log n)$ (complessità logaritmica)
  - ricerca binaria su liste ordinate
- Problemi $O(n)$ (complessità lineare):
  - ricerca lineare
  - pattern matching esatto o approssimato di stringhe
- Problemi $O(n \log n)$
  - ordinamento
- Problemi $O(n^2)$ (complessità quadratica):
  - allineamento di due sequenze

# Problemi trattabili e intrattabili

- La classificazione basata sulla notazione asintotica è una classificazione "fine"
- In generale, è utile classificare i problemi sulla base di una classificazione più grossolana:
  - un problema con complessità polinomiale nel caso pessimo ($O(n^d)$ per qualche $d$) è considerato "trattabile"
  - un problema con complessità almeno esponenziale nel caso pessimo ($O(c^n)$ per qualche $c$) è considerato "intrattabile"
- Ma non è tutto qua...

# Problemi NP

- C'è una classe di problemi, chiamata **NP**, per ciascuno dei quali esiste un algoritmo esponenziale...
- ...ma non si sa se esistano algoritmi polinomiali
- Sia **P** la classe dei problemi risolvibili in tempo polinomiale
- Chiaramente, $\mathbf{P} \subseteq \mathbf{NP}$
- Ma, $\mathbf{NP} \overset{?}{\subseteq} \mathbf{P}$ (Problema aperto)

# Problemi NP in biologia

- Che fare se un problema di interesse biologico è **NP**?
  - Ottenere una soluzione esatta di solito non è possibile
- Si ricorre ad algoritmi che *approssimano* la soluzione
  - È opportuno che l'errore sia quantificabile
- Si ricorre ad algoritmi *probabilistici*
  - La soluzione è esatta con un certo livello di confidenza

# Sequence assembly

- Sequenziamento (*metodo shotgun*):
  - la sequenza di partenza è clonata
  - il DNA è tagliato in posizioni casuali
  - i frammenti (700–800 basi) sono sequenziati
- L'ordinamento dei frammenti è perso
- I frammenti sono sovrapponibili
- Come ricostruire l'ordine corretto dei vari pezzi?

# Sequence assembly (2)

- Dato un insieme $\mathcal{S}$ di stringhe, determinare una stringa $s$ di lunghezza minima tale che ogni $s' \in \mathcal{S}$ sia sottostringa di $s$
- Esempio:

```
CATGCACTCAT
     CACTCATCTGCATTTTAATGA
           CTGCAT
                 TTAATGATAGC
                       ATAGCCAACTACGC
                             AACTACGC
CATGCACTCATCTGCATTTTAATGATAGCCAACTACGC
```

# Sequence assembly (3)

- Il problema precedente è noto come *shortest superstring problem*
- Il problema è **NP**-completo
- Esistono algoritmi polinomiali che forniscono una soluzione approssimata
  - garantiscono che la stringa risultante abbia lunghezza inferiore a tre volte la lunghezza della soluzione esatta

# Allineamento multiplo

```
AQP1.PRO    TLFVFISIGSALGFNYPLERNQTLVQDNVK     30
AQP2.PRO    LLFVFFGLGSALQWA...SS....PPSVLQ     23
AQP3.PRO    LILVMFGCGSVAQVVLSRGTHGGF....LT     26
AQP4.PRO    LIFVLLSVGSTINWG...GSENPLPVDMVL     27
AQP5.PRO    LIFVFFGLGSALKWP...SA....LPTILQ     23
consensus   ***!****!!******    **    **** **
```

```
AQP1.PRO    VSLAFGLSIATL     42
AQP2.PRO    IAVAFGLGIGIL     35
AQP3.PRO    INLAFGFAVTLA     38
AQP4.PRO    ISLCFGLSIATM     39
AQP5.PRO    ISIAFGLAIGTL     35
consensus   ****!!******
```

# Allineamento multiplo (2)

- Date $k$ sequenze, determinare l'allineamento ottimo
- Il criterio di ottimalità si basa su una funzione che associa un punteggio a ciascuna $k$-upla di amminoacidi
- Il punteggio di un allineamento è la somma dei punteggi associati a ciascuna posizione (colonna)
- Il problema è **NP**-completo
- Gli algoritmi usati in pratica approssimano il problema esatto

# Ripiegamento di proteine

- La funzione di una proteina è determinata dalla sua struttura tridimensionale
- Problema:
  - data una sequenza di amminoacidi, determinare la risultante conformazione spaziale (*protein folding problem*)
- Problema ancora aperto
- Problema computazionalmente costoso

# Ripiegamento di proteine (2)

- Un modello semplificato:
  - proteina: stringa $s_1 \cdots s_n$ sull'alfabeto $\{H, P\}$ (idrofilico, idrofobico)
  - due dimensioni
  - posizioni spaziali discrete (griglia)
- Trovare una $f : \{s_1, \ldots, s_n\} \to \mathbb{N} \times \mathbb{N}$ iniettiva tale che
  - $|f(s_i) - f(s_{i+1})| = 1$ (contiguità)
  - sia massimo il numero di "contatti" H-H
- Questa formalizzazione del problema è **NP**-completa

# Ripiegamento di proteine (3)

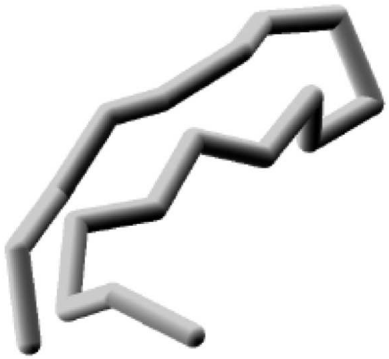


**Fig. 1.** Protein 1PG1 from PDB

**Fig. 2.** Protein 1PG1, HP-model

**Fig. 3.** Protein 1PG1, our model

(da Dal Palú, Dovier, Fogolari, *Protein Folding in CLP(FD) with Empirical Contact Energies)*