UNIVERSITÀ DEGLI STUDI DI FERRARA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di laurea in matematica

UN ALGORITMO PER LA RICOSTRUZIONE DI SEQUENZE DA SEQUENZE PARZIALI IN BIOINFORMATICA

Relatore:

Chiar.mo Prof. Josef Eschgfäller

Laureanda:

Lucia Del Chicca

Anno accademico 2000-2001

INDICE

1. INTRODUZIONE

La bioinformatica è una nuova disciplina che studia le applicazioni delle scienze dell'informazione alle scienze biologiche. Si occupa dell'organizzazione e dell'utilizzo di dati biologici che descrivono sequenze di geni, composizione e struttura di proteine, processi biochimici nelle cellule ed esperimenti biologici di altro tipo. Negli ultimi anni si è arrivati alla sequenziazione completa o quasi completa del genoma di molti organismi, compreso il genoma dell'uomo, ed è così aumentata in misura vertiginosa la quantità di informazione genetica grezza che deve essere catalogata, mantenuta e studiata dai ricercatori non solo della genetica, ma anche della medicina, della farmacologia, della botanica, dell'agricoltura, della zoologia, dell'antropologia, della geobiologia. La bioinformatica nasce quindi dal bisogno di gestire, catalogare ed analizzare grandi quantità di dati biologici che sono molto complessi e voluminosi. Infatti una così immensa quantità di informazioni crea svariati problemi, che vanno dal suo immagazzinamento alla messa a punto di sofisticati sistemi di interrogazione, alla sua analisi.

Lo sviluppo della biologia molecolare, della genetica e della biochimica ha così portato alla produzione di una rilevante quantità di dati, dei quali le biosequenze (acidi nucleici e proteine) costituiscono l'aspetto essenziale e più importante. La caratteristica intrinseca delle biosequenze, ovvero l'essere macromolecole biologiche costituite da tante unità (i nucleotidi e gli aminoacidi) legate l'una dietro l'altra a formare stringhe, ha reso indispensabile l'utilizzo di tecnologie informatiche sia per la loro archiviazione che per la loro analisi. Infatti molte funzioni biologiche corrispondenti alle sequenze degli aminoacidi nelle proteine risultano sconosciute. È possibile, in alcuni casi, confrontarle oppure utilizzare strumenti di analisi per capirne la struttura, identificarne i gruppi funzionali o per operare modifiche (ad esempio nel disegno di farmaci).

Nell'ambito molecolare la bioinformatica comprende numerosi aspetti abbastanza diversi tra loro che vanno dalla creazione di banche dati specializzate, alla produzione di software, all'implementazione di modelli matematici. Ci sono due tipi di banche dati: quelle primarie e quelle specializzate. Sono dette banche dati primarie le banche dati di sequenze di acidi nucleici che contengono i dati grezzi di laboratorio, ossia informazioni molto generiche, cioè quel minimo di informazioni necessarie da associare alla sequenza per identificarla. Le banche dati specializzate invece si sono sviluppate successivamente, sono derivate da quelle primarie e contengono dati associati a quelli delle banche dati primarie. La prima raccolta di dati bio-

logici fu una banca dati di proteine, la NBRF (National Biology Research Foundation), e risale al 1960 ad opera di Margaret Dayhoff, una ricercatrice nel campo dell'evoluzione molecolare che era interessata ad avere la maggior quantità possibile di dati di sequenze di proteine a disposizione per i suoi studi. Nacquero successivamente le banche dati di sequenze di acidi nucleici: nel 1980 la banca dati europea (EMBL Datalibrary), nel 1982 quella americana (Genbank) e infine nel 1986 quella giapponese (DDBJ). Fra queste banche dati si è sviluppata una collaborazione internazionale che ha reso possibile l'individuazione di un certo numero di informazioni da associare a ciascuna sequenza nucleotidica ed ha favorito lo scambio dei dati tanto che, per gran parte, le sequenze e le relative informazioni nelle tre banche dati sono comuni anche se strutturate in modo diverso. Ogni voce della banca dati è caratterizzata da un nome, che in codice indica la specie e la funzione biologica della sequenza che rappresenta, e da un numero che identifica univocamente la sequenza. Altre informazioni relative ad una sequenza per esempio nella banca dati EMBL sono: la lunghezza della sequenza, la data della sua introduzione nella banca dati e le informazioni bibliografiche che riguardano la pubblicazione della sequenza, le informazioni tassonomiche che indicano la specie da cui la sequenza è estratta, le parole chiave che indicano le funzioni biologiche contenute nella sequenza, il *cross referencing* che permette di collegare le banche dati di sequenze di acidi nucleici a quelle di proteine e ad altre banche dati specializzate.

L'aspetto dell'internazionalità è fondamentale per la bioinformatica negli aspetti tecnologici e di servizio. Per quel che riguarda le banche dati di acidi nucleici, Stati Uniti, Europa e Giappone collaborano per la raccolta e l'organizzazione dei dati. In conseguenza all'attenzione rivolta internazionalmente a certi progetti, è sorta l'esigenza di istituire centri internazionali di bioinformatica: in particolare negli Stati Uniti il National Center for Biotechnology Information (NCBI) con sede a Washington e in Europa il European Bioinformatics Institute (EBI) con sede a Hinxton (UK). Tali centri distribuiscono tecnologia e coordinano un servizio incentrato sulle biosequenze a livello internazionale. Queste banche dati non avrebbero avuto la possibilità di esistere e di svilupparsi con questa velocità e fino a questi livelli senza le nuove tecnologie informatiche e telematiche.

Un aspetto fondamentale della bioinformatica è lo studio delle informazioni contenute nelle biosequenze, cioè di tutte le problematiche biologiche e le tecniche informatiche legate alla ricerca e comprensione di messaggi all'interno delle biosequenze. Queste sequenze sono costituite da stringhe

di caratteri: i nucleotidi negli acidi nucleici e gli aminoacidi nelle proteine, la cui successione produce e definisce l'informazione biologica come un testo scritto in una lingua naturale con scrittura alfabetica. Per esempio nel caso degli acidi nucleici l'alfabeto è costituito da quattro elementi: adenina (A), citosina (C), guanina (G) e timina (T) o uracile (U) nel DNA o RNA, mentre per le proteine l'alfabeto ha 20 lettere (i 20 aminoacidi naturali).

Nonostante si conosca da circa quarant'anni il codice genetico che a ogni tripla di nucleotidi fa corrispondere un aminoacido, l'analisi dei testi biologici è ancora agli inizi. Infatti il genoma non contiene solo le ricette per la costruzione delle proteine, ma anche segnali, meccanismi di riconoscimento, ripetitività ancora misteriose.

Quando si determina una nuova sequenza, la prima operazione che di solito si compie per la sua caratterizzazione è la ricerca, all'interno di una banca dati, di sequenze con un grado di similarità statisticamente significativo rispetto alla sequenza in esame. Nelle proteine spesso esiste una stretta, benchè ancora poco compresa, relazione tra struttura e funzione biologica: la funzione biologica di una proteina dipende dalla sua struttura chimica e spaziale, che a sua volta è determinata dalla sua sequenza aminoacidica. La determinazione del grado di similarità tra due o più sequenze fa riferimento alla somiglianza che viene misurata tra sequenze precedentemente allineate, come verrà descritto più avanti in questa introduzione.

La bioinformatica sta diventando sempre più importante per la ricerca medica: un altro dei suoi scopi è mettere a disposizione informazioni che rendono possibile l'ingegneria genetica e lo sviluppo di medicinali che esprimano o sopprimano determinate funzioni nelle cellule di un paziente o di un agente nocivo.

Non conoscevo la bioinformatica fino a quando non ho frequentato un corso in Germania dove ho cominciato a familiarizzare con questa materia. L'inizio è stato un po' difficile sia per i termini tecnici nella lingua straniera che per il fatto che per me la disciplina era nuova, ma dopo aver studiato la teoria, è stato molto interessante provare ad entrare effettivamente in una banca dati e ad analizzarne le informazioni contenute.

La prima parte della tesi è dedicata all'esposizione di un lavoro di Carpi e de Luca, apparso come rapporto di seminario nel 1998 e recentemente in forma ampliata nel *Journal of Theoretical Computer Science*. Questo lavoro contiene forse la prima caratterizzazione matematica degli ostacoli nella ricostruzione di una sequenza da suoi frammenti. Questi ostacoli sono

rappresentati da quei fattori, necessariamente ripetuti, della sequenza (termine preferito dai biologi) o parola (termine usato in informatica teorica) da ricostruire, che sono nondeterminanti, cioè che nella parola completa sono seguiti da due lettere diverse o preceduti da due lettere diverse. Il calcolo combinatorio di questi fattori speciali è presentato in modo abbastanza dettagliato nel lavoro di de Luca.

Anticipiamo brevemente l'idea degli algoritmi. Immaginiamo di avere una sequenza genetica di cui conosciamo solo i pezzi più piccoli che sono il risultato di un trattamento biochimico, ad esempio di un trattamento con endonucleasi di restrizione. Come possiamo ricomporre dai pezzi la sequenza originale? Abbiamo quindi un problema di ricostruzione di una parola, che chiamiamo w, da un insieme dei suoi fattori; indichiamo con $F(w)$ l'insieme dei fattori di una parola. Se ognuno di questi fattori nella parola originale avesse un'unica continuazione verso destra e verso sinistra (se esistesse cioè per ognuno di questi fattori f un'unica lettera a tale che af è ancora contenuto nella parola w originale ed un'unica lettera b tale che fb è fattore di w), allora la ricostruzione dovrebbe essere possibile. La difficoltà nasce da quei fattori che non hanno questa proprietà, cioè da quei fattori f che sono nondeterminanti a sinistra o a destra in w nel senso che esistono lettere a, a' con $a \neq a'$ tali che $af \in F(w)$ e $a'f \in F(w)$ (in questo caso si dice che f è nondeterminante a sinistra), oppure esistono $b, b' \in A$ con $b \neq b'$ tali che $fb \in F(w)$ e $fb' \in F(w)$ (in questo caso si dice che f è nondeterminante a destra).

L'idea di de Luca è adesso di racchiudere questi fattori tra due lettere formando così pacchetti; è anche chiaro che sarà sufficiente conoscere i pacchetti massimali e i fattori non determinanti massimali per poter ricostruire la parola. Siamo riusciti a semplificare l'esposizione nel lavoro originale, da un lato dimostrando che i fattori nondeterminanti a sinistra massimali sono anche nondeterminanti a destra e quindi binondeterminanti massimali e viceversa, dall'altro utilizzando sistematicamente parole chiuse, cioè aggiungendo a sinistra e a destra lettere che non appaiono nella parola originale, per eliminare i casi particolari che altrimenti si presentano alle estremità della parola. Le idee e tecniche combinatorie sono invece già tutte contenute nel lavoro citato.

Abbiamo implementato i due algoritmi più importanti del lavoro di Carpi e de Luca (cioè la ricostruzione di una parola dai suoi pacchetti massimali e la determinazione dei pacchetti massimali in successione ordinata, il ricoprimento canonico) in C++ e in parte in Perl. Abbiamo presentato in un capitolo a parte le versatili possibilità di programmazione orientata agli

oggetti in Perl.

Per algoritmi di carattere più informatico si vedano gli articoli di Crochemore/Vérin e Raffinot, i libri di Crochemore/Hancart/Lecroq o di Gusfield e la tesi di Vérin. Proprio il genoma degli eucarioti contiene molte sequenze ripetute; ciò è ben noto anche ai genetisti; il libro di Gusfield, ricco e difficile, contiene anche molte osservazioni sulle implicazioni biologiche delle ripetizioni nel genoma, riviste anche nell'articolo di Epplen/Riess. La combinatoria delle parole ha molti legami con la teoria dei semigruppi, esposti nel testo recente di de Luca/Varricchio.

Il Perl è oggi proposto molto spesso come linguaggio in ascesa per i problemi di bioinformatica (Tisdall, Gibas/Jambeck – questi libri introducono anche all'uso delle fonti bioinformatiche in Internet). Wall/Christiansen/-Orwant è l'introduzione standard al linguaggio Perl.

Esistono vari metodi statistici per la ricostruzione di sequenze; un modello in cui viene anche esaminato il ruolo dei fattori ripetuti si trova nel lavoro di Arratia/Martin/Reinert/Waterman. Il libro di Waterman ha una forte inclinazione statistica anche quando tratta problemi di carattere combinatorio.

Un altro problema è l'allineamento ottimale di due o più sequenze. L'allineamento tra due sequenze consiste nella determinazione di una relazione tra le lettere della prima sequenza con quelle della seconda in modo da rendere minimo il numero di differenze. L'allineamento cio è stabilisce una corrispondenza tra due sequenze (o parti di esse) in modo da minimizzare il numero di operazioni necessarie per la trasformazione di una nell'altra.

Siano *ACGATAGATATCTGTA* e *CAATTCGAATCAGA* le stringhe da allineare. Le scriviamo l'una sotto l'altra nel modo seguente

AC–GAT–AGATATC–TGTA
–CA–ATTCGA–ATCA–G–A

Abbiamo quindi inserito dei segni – in modo tale che le due stringhe allungate abbiano la stessa lunghezza, cercando di ottenere un massimo numero di posizioni in cui coincidono. Una tale disposizione si chiama allineamento. Possiamo ora definire la somiglianza tra due stringhe allungate (che per brevità chiamiamo anche la somiglianza dell'allineamento): ogni – conta -2; ogni posizione in cui si trovano due lettere uguali vale 1; ogni posizione dove si trovano due lettere differenti conta -1 (– non viene considerato una lettera). Nel nostro esempio abbiamo otto –, una posizione con lettere diverse e dieci posizioni in cui le lettere coincidono. La somiglianza dell'allineamento è quindi $-16 - 1 + 10 = -7$.

Questo problema è stato studiato intensamente con algoritmi più o meno efficienti con generalizzazioni ai casi di allineamenti locali e di allineamenti di più di due sequenze, entrambi molto importanti nella pratica. Abbiamo sperimentato qui il metodo dell'ottimizzazione genetica per l'allineamento di due sequenze. Crediamo che si tratti di una tecnica efficiente, finora poco esplorata (tranne, per quanto sappiamo, nella tesi di Notredame). Il programma, anche per ragioni di velocità, è stato realizzato in C++. La problematica degli allineamenti ottimali è esposta in molti testi di bioinformatica o combinatoria delle parole (Sankoff/Kruskal, Setubal/Meidanis, Pevzner, Waterman, Gusfield, Crochemore/Hancart/Lecroq) o nell'articolo di Altschul. L'ottimizzazione genetica è stata applicata anche al problema di ricostruzione da Parsons/Forrest/Burks.

Una breve esposizione dei meccanismi fondamentali della genetica (conservazione dell'informazione, sua trasmissione e traduzione in funzione biologica) si trova nel libro di Breckow/Greinert. Il trattato di Lodish e.a. è considerato attualmente il testo più completo di biologia molecolare; un'esposizione molto più breve si trova nell'atlante tascabile di Passarge.

Sento il bisogno di ringraziare di cuore il Prof. Eschgfäller, non solo per la competenza e il sostegno che mi ha offerto per la stesura di questo lavoro, ma anche per la grande disponibilità e l'aiuto sempre dimostrati, in particolare per la cura e l'attenzione nella preparazione della lingua tedesca in occasione della mia partenza per il semestre Erasmus, per la collaborazione nella ricerca di un lavoro per la borsa di studio Leonardo e per il suggerimento e l'incoraggiamento a partecipare ad un corso di bioinformatica in Germania.

2. FATTORI DI UNA PAROLA

Situazione 2.1: A sia un insieme finito $\neq \emptyset$, $w \in A^*$ (cfr. Def. 2.2), e $i, j \in \mathbb{N}$.

Definizione 2.2: $A^* := \bigcup_{n=0}^{\infty} A^n$ sia il monoide libero generato da A. Gli elementi di A si chiamano lettere dell'alfabeto A, gli elementi di A^* si chiamano parole. L'elemento neutro di A^* (la parola vuota) è denotata con ε.

Poniamo $A^+ := A^* \setminus \{\varepsilon\}$. Come d'uso nella combinatoria delle parole, per $(a_1, \ldots, a_n) \in A^n$ scriviamo semplicemente $a_1 \cdots a_n$. Identifichiamo A con A^1 e notiamo che $A^0 = \{\varepsilon\}$.

Definizione 2.3: Per $n \geq 0$ e $w \in A^n$ chiamiamo $|w| := n$ la lunghezza di w. In particolare $|\varepsilon| = 0$.

Definizione 2.4: Per $1 \leq i \leq |w| + 1$ e $0 \leq k \leq |w| - i + 1$ usiamo le seguenti abbreviazioni:

(1) $w(i)$ sia l'i-esima lettera di w per $i \leq |w|$.
Quindi $w = w(1)w(2)...w(|w|)$.
(2) $w(i, k) := w(i)...w(i + k - 1)$ per $k \geq 0$,
$w(i, 0) := \varepsilon$ (anche per $i = |w| + 1$).

In particolare $w(i) = w(i, 1)$ per ogni i.

Definizione 2.5: Una parola f si dice *fattore* di w, se esistono $p, q \in A^*$ tali che $w = pfq$.

Denotiamo con $F(w)$ l'insieme dei fattori di w. Gli elementi di $F(w) \setminus \{w\}$ si chiamano fattori *propri* di w.

Per $W \subset A^*$ sia $F(W) := \bigcup_{w \in W} F(w)$.

Per $n \geq 0$ sia $F_n(w)$ l'insieme dei fattori di lunghezza $\leq n$ di w.

Definizione 2.6:

(1) Una parola p si chiama un *prefisso* di w, se esiste $s \in A^*$ tale che $w = ps$.
(2) Una parola s si chiama un *suffisso* di w, se esiste $p \in A^*$ tale che $w = ps$.

Denotiamo con Pref(w) l'insieme dei prefissi di w e con Suff(w) l'insieme dei suffissi.

Definizione 2.7: alf(w) $:= A \cap F(w)$ sia l'insieme delle lettere che compaiono in w.

Definizione 2.8: Per $f \in A^*$ sia

$$\mathrm{Pos}(f, w) := \{\, i \in \mathbb{N} \mid 1 \leq i \leq |w| + 1 \text{ ed } f = w(i, |f|) \,\}.$$

Si osservi che $\mathrm{Pos}(\epsilon, w) = \{1, 2, ..., |w| + 1\}$.
Gli elementi di $\mathrm{Pos}(f, w)$ si chiamano le *posizioni* di f in w.

Definizione 2.9: A contenga anche le lettere $+, -$. Una parola $w \in A^*$ si chiama *chiusa*, se inizia con $+$ e termina con $-$ e non contiene $+$ o $-$ al suo interno.
Questa idea si trova anche in Gusfield e Raffinot.

Definizione 2.10: Per $|w| \geq 2$ sia $\mathrm{int}\ w := w(2, |w| - 2)$ la parola che si ottiene da w togliendo la prima e l'ultima lettera.
Un fattore f di w si chiama *interno*, se $f \in F(\mathrm{int}\ w)$.
f può essere allo stesso tempo un fattore interno $\neq \varepsilon$ e un prefisso e un suffisso di w, come mostra l'esempio $w = aaa$, $f = a$. Ciò non può accadere se w è chiusa. Per $|w| \geq 1$ definiamo anche l'interno destro

$$w + 1 := w(2, |w| - 1)$$

come la parola che si ottiene da w togliendo la prima lettera e l'interno sinistro

$$w - 1 := w(1, |w| - 1)$$

come la parola che si ottiene da w togliendo l'ultima lettera.

3. FATTORI RIPETUTI

Situazione 3.1: A sia un insieme finito che contenga $+$ e $-$. $w \in A^*$ sia una parola chiusa.

Osservazione 3.2: Sia $f \in A^*$, allora f è un fattore di w se e solo se $|\mathrm{Pos}(f,w)| \geq 1$.

Definizione 3.3: $f \in A^*$ si dice *fattore ripetuto* di w, se $|\mathrm{Pos}(f,w)| \geq 2$.
Un fattore si dice *semplice* se non è ripetuto.

Osservazione 3.4: ε è sempre un fattore ripetuto di w. Infatti
$|\ \mathrm{Pos}(\varepsilon, w)\ | = |w| + 1 \geq 3$.

Osservazione 3.5:

(1) Un fattore $\neq \varepsilon$ di w è un prefisso se e solo se contiene $+$.
Ogni prefisso $\neq \varepsilon$ è un fattore semplice.

(2) Un fattore $\neq \varepsilon$ di w è un suffisso se e solo se contiene $-$.
Ogni suffisso $\neq \varepsilon$ è un fattore semplice.

(3) Ogni fattore ripetuto di w è un fattore interno di w. Ciò vale anche per ε che è allo stesso tempo prefisso, suffisso e fattore interno di w.

Osservazione 3.6: Ogni fattore di un fattore ripetuto di w è a sua volta un fattore ripetuto di w.

Definizione 3.7: Sia f un fattore di w ed $a \in A$.
Diciamo che a è un *appoggio sinistro* di f in w, se $af \in F(w)$.
Diciamo che a è un *appoggio destro* di f in w, se $fa \in F(w)$.

Definizione 3.8: Sia f un fattore di w.

(1) f si dice *non determinante a sinistra* (NDS), se f possiede due appoggi sinistri distinti in w. Altrimenti f si dice *determinante a sinistra* (DS).

(2) f si dice *non determinante a destra* (NDD), se f possiede due appoggi destri distinti in w. Altrimenti f si dice *determinante a destra* (DD).

(3) f si dice *non determinante* (ND), se f è NDS o NDD. Altrimenti diciamo che f è *bideterminante*.

(4) f si dice *binondeterminante*(BND), se è NDS e NDD. I fattori BND si chiamano anche *bispeciali* (de Luca) oppure *maximal repeats* (Gusfield).

Osservazione 3.9: Ogni prefisso $\neq \varepsilon$ è bideterminante così come ogni suffisso.

Osservazione 3.10: ε è sempre un fattore BND di w. Infatti ogni lettera di w è sia un appoggio destro che un appoggio sinistro di ε. Però $\mid \text{alf}(w) \mid \geq 2$ perchè $+,- \in \text{alf}\,(w)$.

Osservazione 3.11: Un fattore ND è ripetuto. Non vale l'implicazione inversa come mostra l'esempio $w = +xabyxaby-$.

Definizione 3.12: Per un insieme parzialmente ordinato (p.o.) $(P, \leq)$ denotiamo con Max P l'insieme degli elementi massimali di P.

Osservazione 3.13: $(P, \leq)$ sia un insieme p.o. finito. Allora per ogni $p \in P$ esiste un $m \in \text{Max } P$ con $p \leq m$.

Dimostrazione: Se $p =: p_0$ è massimale, possiamo prendere $m = p$. Altrimenti esiste $p_1 \in P$ con $p_0 < p_1$ (cioè $p_0 \leq p_1$ e $p_0 \neq p_1$). Se p_1 è massimale, possiamo porre $m = p_1$. Altrimenti esiste $p_2 \in P$ con $p_1 < p_2$, e così via. In questo modo otteniamo una successione $p_0 < p_1 < p_2 < ...$ che deve terminare perchè P contiene solo un numero finito di elementi. L'ultimo elemento m della successione è massimale con $p \leq m$.

Lemma 3.14: $(Q, \leq)$ sia un insieme p.o. finito e $P \subset Q$. In P consideriamo la naturale restrizione di $\leq$. Allora:

(1) $P \cap \text{Max } Q \subset \text{Max } P$.
(2) Se Max $Q \subset P$, allora Max $P =$ Max Q.

Dimostrazione.(1) Gli elementi di $P \cap \text{Max } Q$ sono gli elementi di P che sono massimali in Q e quindi a maggior ragione massimali in P.

(2) Dall'ipotesi segue che Max $Q = P \cap \text{Max } Q \subset \text{Max } P$, usando il punto (1). Sia $p \in \text{Max } P$. Per l'oss. 3.13 esiste $m \in \text{Max } Q$ tale che $p \leq m$. Ma per ipotesi $m \in P$, per cui $m = p$. Quindi $p \in \text{Max } P$.

Lemma 3.15: f sia un fattore NDS o NDD massimale di w. Allora f è BND.

Dimostrazione. f sia un fattore NDS massimale. f possiede quindi due appoggi sinistri diversi a_1, a_2 in w, perciò $a_1 \neq a_2$ e $a_1 f, a_2 f \in F(w)$. Per provare che f è BND basta provare che f è NDD. f è un fattore interno, perciò esistono b_1 e $b_2 \in A$ tali che $a_1 f b_1 \in F(w)$ e $a_2 f b_2 \in F(w)$. Supponiamo per assurdo che non esistano due appoggi destri distinti di f in w. Allora $b_1 = b_2 := b$ e quindi $a_1 f b, a_2 f b \in F(w)$ e vediamo che fb è ancora NDS, in contraddizione alla massimalità di f.

Corollario 3.16: Per un fattore f di w sono equivalenti:
(1) f è un fattore NDS massimale di w.
(2) f è un fattore NDD massimale di w.
(3) f è un fattore BND massimale di w.

Dimostrazione. Segue dal lemma 3.15, usando il lemma 3.14 e l'oss. 3.11.

Osservazione 3.17: Sia $w = +ababaa-$.
ab è un fattore ripetuto DD contenuto in un fattore (aba) che è NDD (e BND).

Lemma 3.18: Sia f un fattore ripetuto di w.
(1) Se $+f \in F(w)$, allora f è NDS.
(2) Se $f- \in F(w)$, allora f è NDD.

Teorema 3.19: Sia f un fattore di w. Allora sono equivalenti:
(1) f è un fattore ripetuto massimale di w.
(2) f è un fattore BND massimale di w.

Dimostrazione. Per il lemma 3.14 è sufficiente mostrare che ogni fattore ripetuto massimale è BND. Sia f è un fattore ripetuto massimale (necessariamente interno) di w. Allora esistono i, j con $2 \leq i < j \leq n+1$ tali che $w(i, |f|) = w(j, |f|)$. Siccome $2 \leq i < j$, esistono $a, b \in A$ tali che

$$w(i-1, |f|+1) = af,\ w(j-1, |f|+1) = bf.$$

Se $a = b$, allora af sarebbe un fattore ripetuto di w, in contraddizione alla massimalità di f. Quindi $a \neq b$, perciò f è NDS. Nello stesso modo si mostra che f è NDD e perciò BND.

Corollario 3.20: Un fattore di w è ripetuto se e solo se è un fattore di un fattore BND di w.

Dimostrazione. Ciò segue direttamente dall'oss. 3.13, dal teorema 3.19 e dall'oss. 3.11.

Osservazione 3.21: Sia $\lambda \geq 0$. Se ogni fattore di lunghezza λ di w è DD (rispettivamente DS), anche ogni fattore di lunghezza maggiore è DD (rispettivamente DS). Infatti se f è un fattore NDD (rispettivamente NDS) di w con $|f| \geq \lambda$, allora il suffisso (rispettivamente prefisso) di f di lunghezza λ è anch'esso NDD (rispettivamente NDS).

Osservazione 3.22: Sono equivalenti:
(1) w non possiede fattori ripetuti $\neq \varepsilon$.
(2) w non possiede fattori BND $\neq \varepsilon$.
(3) Tutte le lettere di w sono distinte.

Dimostrazione. Dall'oss. 3.13 e dal teorema 3.19 segue che (1) $\iff$ (2).
È ovvio inoltre che (3) $\iff$ (1).

Definizione 3.23: R_w sia il più piccolo $\lambda \geq 0$ tale che ogni fattore di lunghezza λ di w sia DD e L_w il più piccolo $\mu \geq 0$ tale che ogni fattore di lunghezza μ di w sia DS.

Per l'oss. 3.21 ogni fattore di lunghezza $\geq R_w$ è DD ed ogni fattore di lunghezza $\geq L_w$ è DS.

Definizione 3.24: G_w sia la massima lunghezza di un fattore ripetuto di w. G_w è ben definito perchè con ε esiste sempre un fattore ripetuto di w.

Proposizione 3.25: $L_w = R_w = G_w + 1$.

Dimostrazione. Essendo R_w il più piccolo λ tale che ogni fattore di lunghezza λ di w sia DD, ogni fattore di lunghezza $R_w - 1$ è NDD massimale di w e quindi per il corollario 3.16 NDS e quindi BND. Allo stesso modo si vede che $L_w - 1 = G_w$. Essendo G_w la massima lunghezza di un fattore ripetuto di w, dal teorema 3.19 segue che $R_w - 1 = G_w$.

Osservazione 3.26: Il lavoro di de Luca contiene molti risultati quantitativi sui fattori di una parola.

4. PACCHETTI

Situazione 4.1: A sia un insieme finito che contenga + e −. $w \in A^*$ sia una parola chiusa.

Definizione 4.2: Un fattore f di w si chiama *pacchetto* di w, se $|f| \geq 2$ e int f è BND in w. Denotiamo con P(w) l'insieme dei pacchetti di w.

Osservazione 4.3: Siccome ε è BND, ogni fattore di lunghezza 2 di w è un pacchetto di w.

Osservazione 4.4: Ogni pacchetto di w è contenuto in un pacchetto massimale di w. Ciò segue dall'osservazione 3.13.

Osservazione 4.5: Per $f \in F(w)$ sono equivalenti:

(1) f è un fattore ripetuto di w.
(2) f è un fattore interno di un pacchetto di w.

Dimostrazione: (1) $\Rightarrow$ (2): f sia un fattore ripetuto di w. Per il corollario 3.20 f è fattore di un fattore BND. g è necessariamente un fattore interno di w, per cui esistono $a, b \in A$ tali che $r = agb \in F(w)$. Per definizione r è un pacchetto di w.

(2) $\Rightarrow$ (1): f sia un fattore interno di u, quindi un fattore di int $u =: g$. g è BND e dal corollario 3.20 segue che f è ripetuto .

Proposizione 4.6: Un pacchetto massimale di w è sempre un fattore semplice.

Dimostrazione: Altrimenti, secondo l'osservazione 4.5, sarebbe fattore interno di un altro pacchetto, in contraddizione alla massimalità.

Osservazione 4.7: Sia $w = +a_1a_2...a_n-$. Tutte le lettere di w siano distinte. Allora i pacchetti di w sono esattamente

$$+a_1, a_1a_2, ..., a_{n-1}a_n, a_n-.$$

Sapendo che non ci sono lettere ripetute e conoscendo l'inizio + e la fine − di w, non è difficile ricostruire w dai pacchetti. Il teorema di Carpi/de Luca che dimostreremo in questo capitolo generalizza questa idea e permette di ricostruire una parola dai suoi pacchetti massimali.

Osservazione 4.8: Siano $w_1, w_2 \in A^*$ tali che $F_n(w_1) = F_n(w_2)$ per ogni $n \geq 0$. Allora $w_1 = w_2$.

Teorema 4.9: Sia $w' \in A^*$ un'altra parola chiusa. Assumiamo che

$$\text{Max P}(w) \subset F(w')$$
$$\text{Max P}(w') \subset F(w).$$

Allora $w = w'$.

Dimostrazione: Dimostriamo per induzione che $F_n(w) = F_n(w')$ per ogni $n \geq 0$. Secondo l'oss. 4.6 questo implica che $w = w'$.

$n = 0$: Ovvio.

$n = 1$: Dobbiamo dimostrare che alf$(w) =$ alf (w').
Sia $a \in$ alf(w). Siccome $|w| \geq 2$, esiste un $b \in A$ tale che $ab \in F(w)$ oppure $ba \in F(w)$. Supponiamo che $ab \in F(w)$, allora ab è un pacchetto, quindi esiste un $u \in$ Max P(w) tale che $ab \in F(u)$ e dall'ipotesi Max P$(w) \subset F(w')$ segue $a \in$ alf(w'). Allo stesso modo si dimostra che alf$(w') \subset$ alf(w).

$n \to n+1$: Sia $aub \in F_{n+1}(w)$ con $a, b \in A$ e $u \in F_{n-1}(w)$, per $n \geq 1$.

(1) Se aub è un pacchetto di w, allora esiste un pacchetto massimale v di w tale che $aub \in F(v)$. Ciò implica $aub \in F(w')$. Analogamente si procede per il viceversa.

(2) Assumiamo che aub non sia un pacchetto di w. Allora u non è BND in w, quindi ad esempio u è DS in w. Inoltre $au, ub \in F_n(w) = F_n(w')$. Siccome $Au \subset F_n(w) = F_n(w')$, vediamo che u, e quindi anche ub è DS in w'. Essendo $u(1) \neq +$, ub deve essere un fattore interno di w' e l'unico appoggio sinistro di ub in w' è a. Ciò implica $aub \in F(w')$ e mostra che $F_{n+1}(w) \subset F_{n+1}(w')$ in entrambi i casi. Nello stesso modo si vede che $F_{n+1}(w') \subset F_{n+1}(w)$.

Proposizione 4.10: $w' \in A^*$ sia un'altra parola chiusa. Sia $n := G_w + 2$.
Se $F_n(w) = F_n(w')$, allora $w = w'$.

Dimostrazione: Dal teorema 3.19 segue che $n-2$ è la lunghezza massima di un fattore BND di w, quindi n è la lunghezza massima di un pacchetto di w. L'enunciato segue dal teorema 4.7, se riusciamo a dimostrare che $G_w = G_{w'}$. L'ipotesi implica che ogni fattore BND di w è anche fattore BND di w', per cui $G_w \leq G_{w'}$.
Assumiamo che $G_w < G_{w'}$. Allora esiste un fattore BND f di w' con $|f| \geq G_w + 1 = n - 1$. Il suffisso di lunghezza $n-1$ di f è allora un fattore NDS in w' di lunghezza $n-1$, e perciò anche un fattore NDS di w, perchè $F_n(w) = F_n(w')$. Ma ciò non è possibile perchè $G_w = n - 2$.

Osservazione 4.11: Ogni lettera di w è contenuta in un pacchetto di w.

Dimostrazione: Ogni lettera di w è un fattore di lunghezza 1 ed è contenuta in un fattore di lunghezza 2 (composto da lei e dal suo appoggio destro o sinistro) che, per l'oss. 4.3, è un pacchetto.

Definizione 4.12: $F_P(w) := F(\text{Max } P(w)) = F(P(w))$ sia l'insieme di tutti i fattori di w che sono contenuti in un qualche pacchetto.

Proposizione 4.13: Sia $p \neq w$ un prefisso $\neq \varepsilon$ di w. Sia f il suffisso più lungo di p che sia un fattore BND di w.
p non è BND (oss. 3.5), per cui $f \neq p$, perciò esiste $b \in A$ tale che bf è un suffisso di p. Sia $u := bf$.
Allora u è un suffisso di p con la proprietà che per ogni $a \in A$ vale:

$$pa \in F(w) \iff ua \in F_P(w).$$

Dimostrazione: ε è un suffisso BND di p, perciò f è ben definito. Sia $a \in A$.
(1) Sia $pa \in F(w)$. Allora $ua = bfa$ è un pacchetto, quindi $ua \in F_P(w)$.

(2) Sia ora $ua \in F_P(w)$. Assumiamo per assurdo che $pa \notin F(w)$.
Sia g il suffisso più lungo di p con $ga \in F(w)$. Dato che $g \neq p$, esiste un $c \in A$ tale che cg è un suffisso di p.
Per la massimalità di g vale $cga \notin F(w)$. ga non può essere un prefisso di w (altrimenti $g = p$), quindi esiste un $x \in A$ tale che $xga \in F(w)$. D'altra parte cg è un suffisso di $p \neq w$, quindi esiste un $y \in A$ tale che $cgy \in F(w)$.
Siccome $cga \notin F(w)$, segue che $y \neq a$. g è quindi NDS in w.

Inoltre $xga \in F(w)$ e $cga \notin F(w)$ implicano che anche $x \neq c$, per cui g è NDD e quindi BND in w.
Siccome $ua \in F(w)$, abbiamo $|g| \geq |u| = |f| + 1$, ma questo è assurdo avendo assunto come ipotesi che f fosse il suffisso più lungo di p che sia BND in w.

Osservazione 4.14: Nelle ipotesi e con le notazioni della proposizione 4.13, vediamo che, essendo p un prefisso $\neq \varepsilon$ di w, la lettera $a \in A$ con $pa \in F(w)$ è univocamente determinata, quindi esiste esattamente un $a \in A$ per cui $ua \in F_P(w)$ e per questo a vale $pa \in F(w)$.

Proposizione 4.15: Sia $p \neq w$ un prefisso $\neq \varepsilon$ di w. Sia v il più corto suffisso di p per cui esiste esattamente un $a \in A$ con $va \in F_P(w)$. Allora $pa \in F(w)$.

Dimostrazione: Sia u definito come nella proposizione 4.13. Per l'oss. 4.14 esiste esattamente un $a' \in A$ tale che $ua' \in F_P(w)$. v è quindi ben definito. Siccome v è un suffisso di u, anche $va' \in F_P(w)$, e dall'ipotesi segue che a' deve essere quell'unico $a \in A$ per cui $va \in F_P(w)$.

Osservazione 4.16: Dalla proposizione 4.15 otteniamo un algoritmo per ricostruire w da Max P(w).

(1) $p =$ unico pacchetto massimale di w che contiene $+$.
(2) Se p contiene $-$ allora STOP.
(3) $v =$ suffisso più corto di p per cui esiste esattamente un $a \in A$ con $va \in F_P(w)$. Determiniamo questo a.
(4) $p = pa$.
(5) goto (2).

Osservazione 4.17: Sia $n := G_w + 2$. Se conosciamo $F_n(w)$, possiamo costruire Max P(w) nel modo seguente:

(1) I fattori BND di w sono esattamente gli elementi $f \in F_n(w)$ per cui esistono $a, b, c, d \in A$ con $a \neq b, c \neq d$ e $afc, bfd \in F_n(w)$.
(2) I pacchetti di w sono esattamente gli elementi $u \in F_n(w)$ per cui esistono un fattore BND ed $a, b \in A$ tali che $afb = u$.
(3) Adesso scegliamo i pacchetti massimali.

Insieme all'oss. 4.16 ciò fornisce una nuova dimostrazione della prop. 4.10.

Osservazione 4.18: Ci si può chiedere se la terminazione dell'algoritmo di ricostruzione partendo da un insieme M di fattori di una parola garantisce che si ottenga la parola da cui si è partiti. In genere non è così, come mostrano i seguenti esempi in cui denotiamo con w' la parola costruita dall'algoritmo.

(1) $M = \{+a, a, a-\}$. In questo caso $w' = +a-$, ma si ha anche $M \subset F(+aaaaa-)$.
(2) $M = \{+a, ab, b-\}$. In questo caso $w' = +ab-$, ma si ha anche $M \subset F(+ababab-)$.

Proposizione 4.19: Sia $M \subset A^*$. M soddisfi le seguenti condizioni:

(1) M contiene esattamente un elemento δ che inizia con +.
(2) $\delta \in F(w)$.
(3) Se $v \in F(M), a \in A$ con $va \in F(w)$ ed esiste $b \in A$ con $vb \in F(M)$, allora $va \in F(M)$. L'algoritmo di ricostruzione basato su M termini e conduca alla parola w'. Allora $w = w'$.

Dimostrazione: Siano $w = a_1 a_2 \dots a_n \in A^n$ e $w' = a'_1 a'_2 \dots a'_k \in A^k$.
Per $1 \leq m \leq n$ sia $p_m := a_1 a_2 \dots a_m$ e per $1 \leq m \leq k$ sia $p'_m := a'_1 a'_2 \dots a'_m$.
Per ipotesi esiste $s \geq 1$ tale che $\delta = p_s$. É chiaro che allora anche $p'_s = p_s$.
Dimostriamo per induzione su m per $s \leq m \leq n$ che $p_m = p'_m$. Siccome sia w che w' terminano con -, ciò implica che $w = w'$. Siano $s \leq m < n$ e

$p_m = p'_m$. Allora p_m non può contenere -, perchè altrimenti $m = n$. Quindi esiste un suffisso v di p_m tale che esiste un unico $b \in A$ con $vb \in F(M)$. Questo b è uguale ad a'_{m+1} per costruzione. Da ciò segue che $v \in F(M)$. D'altra parte $p_m a_{m+1} \in F(w)$, quindi $va_{m+1} \in F(w)$, e l'ipotesi (3) implica che $va_{m+1} \in F(M)$. Dall'unicità di b segue che $a_{m+1} = b = a'_{m+1}$, per cui $p_{m+1} = p'_{m+1}$.

Osservazione 4.20: La condizione della proposizione 4.19 matematicamente è quasi banale, è però una condizione relativamente facile da soddisfare in un esperimento biologico quando da una parola fisicamente data, ma sconosciuta come sequenza, si prepara un insieme M di frammenti da cui ricostruire la parola.

5. IL RICOPRIMENTO CANONICO

Situazione 5.1: A sia un insieme non vuoto finito che contiene + e −. $w \in A^*$ sia una parola chiusa.

Definizione 5.2: Un fattore f di w si chiama *quasipacchetto*, se $|f| \geq 2$ e int f è un fattore ripetuto di w.

Ogni pacchetto è quindi un quasipacchetto (per l'oss. 3.11).

Osservazione 5.3: Per $r \in F(w)$ sono equivalenti:

(1) r è un fattore ripetuto di w.
(2) r è un fattore interno di un pacchetto di w.
(3) r è un fattore interno di un quasipacchetto di w.

Dimostrazione: (1) $\Longleftrightarrow$ (2): Oss. 4.5.

(2) $\Longleftrightarrow$ (3): Ovvio.

(3) $\Longleftrightarrow$ (1): Ovvio.

Lemma 5.4: Sia $f = arb \in F(w)$ con $a, b \in A$ ed r un fattore ripetuto di w. f è quindi un quasipacchetto. Sono equivalenti:

(1) f è un quasipacchetto massimale.
(2) ar ed rb sono fattori semplici di w.

Dimostrazione: (1) $\Longrightarrow$ (2): Sia ar ripetuto.

Se f non è un prefisso, allora esiste un $c \in A$ tale che $carb \in F(w)$; ma allora $carb$ è un quasipacchetto più grande di f, in contraddizione alla massimalità di f. Se invece f è un prefisso, allora $a = +$, quindi ar è semplice. Allo stesso modo si dimostra che rb è semplice.

(2) $\Longrightarrow$ (1): Se f non è massimale, esiste un quasipacchetto g della forma $\lambda arb\mu$ con $\lambda, \mu \in A^*$ e $\lambda\mu \neq \varepsilon$. Allora uno dei due fattori ar o rb è fattore interno di g e quindi ripetuto in w per l'oss. 5.3, una contraddizione.

Corollario 5.5: $f = arb$ sia un pacchetto con $a, b \in A$. I fattori ar ed rb siano semplici in w. Allora f è un quasipacchetto massimale e quindi anche un pacchetto massimale.

Teorema 5.6: Per $f \in F(w)$ sono equivalenti:

(1) f è un quasipacchetto massimale.
(2) f è un pacchetto massimale.

Dimostrazione: Siccome ogni pacchetto è un quasipacchetto, per il lemma 3.14 dobbiamo solo dimostrare che ogni quasipacchetto massimale è un pacchetto. f sia quindi un quasipacchetto massimale. Per il lemma 5.4 f è della forma $f = arb$ con $a, b \in A$ ed r un fattore ripetuto di w tale che ar ed rb sono semplici in w. Se $r = \varepsilon$ allora r è BND. Altrimenti r appare almeno due volte in w e non può essere nè un prefisso nè un suffisso. Siccome invece ar ed rb non si

possono ripetere, devono esistere $c, d \in A$ con $c \neq a$ e $d \neq b$ tali che $crd \in F(w)$. Questo implica che r sia BND in entambi i casi e quindi f è un pacchetto.

Corollario 5.7: $f = arb$ sia un pacchetto massimale di w con $a, b \in A$. Allora ar ed rb sono semplici in w.

Dimostrazione: Per il teorema 5.6 f è un quasipacchetto massimale, perciò possiamo applicare il lemma 5.4.

Definizione 5.8: Per $u, v \in A^*$ poniamo

$$u \wedge v := \text{suffisso di lunghezza massimale di } u \text{ che è prefisso di } v = \\ = \text{prefisso di lunghezza massimale di } v \text{ che è suffisso di } u.$$

Adesso esistono, univocamente determinati, $\alpha, \beta \in A^*$ tali che

$$u = \alpha(u \wedge v)$$
$$v = (u \wedge v)\beta.$$

Poniamo allora $u \vee v := \alpha(u \wedge v)\beta$.

Esempio 5.9: Siano $u = abxyz, v = xyzcab$ con $a, b, c, x, y, z \in A$ lettere distinte. Allora

$$u \wedge v = xyz$$
$$u \vee v = abxyzcab$$

mentre

$$v \wedge u = ab$$
$$v \vee u = xyzcabxyz.$$

In particolare vediamo che le operazioni $\wedge$ e $\vee$ non sono commutative, come d'altronde è ovvio.

Osservazione 5.10: Sia $u \in A^*$. Allora $u \wedge u = u \vee u = u$.

Osservazione 5.11: Per $u, v \in A^*$ sono equivalenti:

(1) v è un suffisso di u.
(2) $u \wedge v = v$.
(3) $u \vee v = u$.

Osservazione 5.12: Per $u, v \in A^*$ sono equivalenti:

(1) u è un prefisso di v.
(2) $u \wedge v = u$.
(3) $u \vee v = v$.

Osservazione 5.13: Per $u, v \in A^*$ valgono le seguenti relazioni:

(1) $(u \wedge v) \wedge v = u \wedge v$.
(2) $(u \wedge v) \vee v = v$.
(3) $u \wedge (u \wedge v) = u \wedge v$.
(4) $u \vee (u \wedge v) = u$.

Dimostrazione: $u \wedge v$ è un prefisso di v, quindi (1) e (2) seguono dall'oss. 5.12. $u \wedge v$ è suffisso di u, perciò (3) e (4) seguono dall'oss. 5.11.

Osservazione 5.14: Per $u, v \in A^*$ valgono le seguenti relazioni:

(1) $u \wedge (u \vee v) = u$.
(2) $u \vee (u \vee v) = u \vee v$.
(3) $(u \vee v) \wedge v = v$.
(4) $(u \vee v) \vee v = u \vee v$.

Dimostrazione: u è prefisso di $u \vee v$, perciò (1) e (2) seguono dall' oss. 5.12. v è suffisso di $u \vee v$, quindi (3) e (4) seguono dall'oss. 5.11 .

Osservazione 5.15: Le operazioni $\wedge$ e $\vee$ sono fortemente non associative. Per $a \neq b$ ad esempio si ha

$$(aba \wedge babaa) \wedge abaa = ba \wedge abaa = a$$
$$aba \wedge (babaa \wedge abaa) = aba \wedge abaa = aba.$$

oppure

$$(aba \wedge ab) \wedge a = a \wedge a = a$$
$$aba \wedge (ab \wedge a) = aba \wedge \varepsilon = \varepsilon$$

oppure

$$(aba \wedge abab) \wedge aba = aba \wedge aba = aba$$
$$aba \wedge (abab \wedge aba) = aba \wedge ab = a.$$

In una situazione importante però potremo dimostrare l'associatività (lemma 5.19).

Lemma 5.16: Siano $h \in A^*$ ed $u, v \in F(h)$. t sia un fattore semplice di h tale che $t \in \text{Suff}(u) \cap \text{Pref}(v)$. Allora $t = u \wedge v$ e $u \vee v \in F(h)$.

Dimostrazione: t è sia un prefisso che un suffisso di $u \wedge v$, quindi essendo t semplice, necessariamente $t = u \wedge v$. $t \in \text{Suff}(u) \cap \text{Pref}(v)$ implica che esistono $\lambda, \mu \in A^*$ tali che $u = \lambda t$ e $v = t\mu$. Inoltre necessariamente, sempre essendo t semplice, l'unico prefisso di u che precede t in h è λ e l'unico suffisso di v che lo segue è μ, quindi $u \vee v = \lambda t \mu \in F(h)$.

Corollario 5.17: Siano $h \in A^*$ ed $u, v \in F(h)$. $u \wedge v$ sia un fattore semplice di h. Allora $u \vee v \in F(h)$.

Lemma 5.18: Siano $h \in A^*$ ed $\alpha, \beta, \gamma \in F(h)$ tali che $\alpha \wedge \beta$ e $\beta \wedge \gamma$ siano fattori semplici di h. Allora:

(1) $(\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma) \in F(h)$.
(2) $(\alpha \vee \beta) \wedge \gamma = \beta \wedge \gamma$
$\alpha \wedge (\beta \vee \gamma) = \alpha \wedge \beta$.

Dimostrazione: (1) Siano $\alpha \wedge \beta = t$ e $\beta \wedge \gamma = s, \alpha = \alpha' t, \beta = t\beta' = \beta'' s, \gamma = s\gamma''$ con $\alpha', \beta', \beta'', \gamma'' \in A^*$. Allora $\alpha \vee \beta$ e $\beta \vee \gamma \in F(h)$ per il lemma 5.16. Adesso $\alpha \vee \beta = \alpha' t \beta' = \alpha' \beta'' s$ e, ancora dal lemma 5.16, segue che $(\alpha \vee \beta) \vee \gamma = \alpha' \beta'' s \gamma''$ e $(\alpha \vee \beta) \wedge \gamma = s = \beta \wedge \gamma$, e nello stesso modo $\beta \vee \gamma = \beta'' s \gamma'' = t \beta' \gamma''$, per cui $\alpha \vee (\beta \vee \gamma) = \alpha' t \beta' \gamma'' = \alpha' \beta'' s \gamma'' = (\alpha \vee \beta) \vee \gamma$ e $\alpha \wedge (\beta \vee \gamma) = t = \alpha \wedge \beta$. Il lemma 5.16 implica adesso anche che $(\alpha \vee \beta) \wedge \gamma \in F(h)$.

Lemma 5.19: Siano $h \in A^*$ ed $\alpha_0, \alpha_1, ..., \alpha_n \in F(h)$ (per $n \geq 2$) tali che $\alpha_i \wedge \alpha_{i+1}$ sia semplice per ogni $i = 0, ..., n-1$. Allora l'espressione $\alpha_0 \vee ... \vee \alpha_n$ non dipende da come poniamo le parentesi ed è un fattore di h.

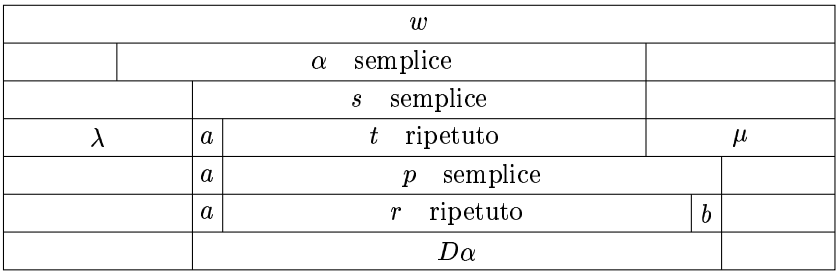
Dimostrazione: Procediamo per induzione.
Per $n = 2$ l'enunciato segue dal lemma 5.18.
Per $n = 3$ consideriamo i seguenti enunciati:
(1) $((\alpha_0 \vee \alpha_1) \vee \alpha_2) \vee \alpha_3 = (\alpha_0 \vee (\alpha_1 \vee \alpha_2)) \vee \alpha_3$.
(2) $((\alpha_0 \vee \alpha_1) \vee \alpha_2) \vee \alpha_3 = (\alpha_0 \vee \alpha_1) \vee (\alpha_2 \vee \alpha_3)$.
(3) $(\alpha_0 \vee \alpha_1) \vee (\alpha_2 \vee \alpha_3) = \alpha_0 \vee (\alpha_1 \vee (\alpha_2 \vee \alpha_3)))$.
(4) $\alpha_0 \vee ((\alpha_1 \vee \alpha_2) \vee \alpha_3) = \alpha_0 \vee (\alpha_1 \vee (\alpha_2 \vee \alpha_3))$.

Proviamo a dimostrarli:
(1) segue direttamente dal lemma 5.18.
(2) è valido se $(\alpha_0 \vee \alpha_1) \wedge \alpha_2$ è semplice. Ma $(\alpha_0 \vee \alpha_1) \wedge \alpha_2 = \alpha_1 \wedge \alpha_2$.
(3) come (2).
(4) come (1).

Inoltre $(\alpha_0 \vee \alpha_1) \wedge \alpha_2 = \alpha_1 \wedge \alpha_2$ e $\alpha_2 \wedge \alpha_3$ sono semplici e ciò implica (sempre per il lemma 5.18) che $((\alpha_0 \vee \alpha_1) \vee \alpha_2) \vee \alpha_3 \in F(h)$. Nello stesso modo, per induzione, si dimostra il caso generale.

Definizione 5.20: α sia un fattore semplice di w. α non sia un suffisso di w. Costruiamo una parola Dα nel modo seguente.

s sia il più corto suffisso di α che è semplice in w. Allora possiamo scrivere $s = at$ con $a \in A$ e t un fattore ripetuto di w. Adesso esistono $\lambda, \mu \in A^*$ con $w = \lambda a t \mu$. λ e μ sono univocamente determinate, perché at è un fattore semplice di w. Per ipotesi α non è un suffisso di w, perciò $t\mu$ è un suffisso $\neq \varepsilon$ di w e quindi semplice.
p sia il più corto prefisso di $t\mu$ che è semplice in w. Poniamo D$\alpha := ap$.
Notiamo che esiste un fattore ripetuto r di w tale che $p = rb$, quindi D$\alpha = arb$.
t è un prefisso di r. Dα si chiama la *continuazione destra* del fattore semplice α in w.



Osservazione 5.21: Nella situazione e con le notazioni della definizione 5.20 si ha :

(1) ar è semplice perchè contiene il fattore semplice at.
(2) $rb = p$ è semplice per costruzione.
(3) r è ripetuto per costruzione.
(4) Quindi Dα è un pacchetto massimale di w (lemma 5.4).
(5) $\alpha \wedge$Dα contiene il fattore semplice at, quindi $\alpha \wedge$D$\alpha = at$ e $\alpha \vee$D$\alpha \in F(h)$.
(6) Dα si estende verso destra di almeno una posizione in più di α (quella occupata dalla lettera b), per cui D$\alpha \neq \alpha$.

Osservazione 5.22: D+ è l'unico pacchetto massimale di w che contiene $+$. Esplicitamente possiamo ripetere la costruzione della def. 5.20 per questo caso particolare: $s = +, t = \varepsilon, w = +\mu$. Anche μ è semplice in w. Allora D+= $+p$.

Proposizione 5.23: α_0 =D+ sia l'unico pacchetto massimale che è un prefisso di w. Formiamo induttivamente

$\alpha_1 :=$ Dα_0, $\alpha_2 :=$ Dα_1, ... , $\alpha_{k+1} :=$ Dα_k,...

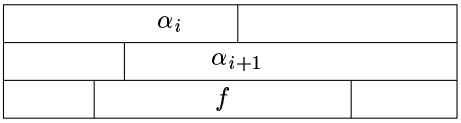
Per il punto (6) dell'osservazione 5.21 esiste un n tale che α_n è un suffisso di w. Allora $w = \alpha_0 \vee \alpha_1 \vee ... \vee \alpha_n$.

Dimostrazione: Per l'oss. 5.21 ed il lemma 5.19 l'espressione $\alpha_0 \vee \alpha_1 \vee ... \vee \alpha_n$ è ben definita. Dal lemma 5.19 segue anche che $f := \alpha_0 \vee \alpha_1 \vee ... \vee \alpha_n$ è un fattore di w. Siccome f è $\neq \epsilon$ e sia un prefisso che un suffisso di w, necessariamente $f = w$.

Definizione 5.24: I pacchetti massimali $\alpha_0, \alpha_1, ..., \alpha_n$ siano definiti come nella proposizione 5.23. La successione $(\alpha_0, \alpha_1, ..., \alpha_n)$ si chiama il *ricoprimento canonico* di w.

Osservazione 5.25: $(\alpha_0, \alpha_1, ..., \alpha_n)$ sia il ricoprimento canonico di w. Allora ogni fattore f di w che non è contenuto in nessuno degli α_j contiene qualche $\alpha_i \wedge \alpha_{i+1}$ come fattore interno.

Dimostrazione: Se f non è fattore di nessuno degli α_j, è possibile solo una situazione come nella seguente figura:



Proposizione 5.26: I pacchetti massimali $\alpha_0, \alpha_1, ..., \alpha_n$ nella proposizione 5.23 sono tutti distinti e costituiscono tutti i pacchetti massimali di w:

$$\text{Max P}(w) = \{\alpha_0, \alpha_1, ..., \alpha_n\}$$

Dimostrazione: Che gli α_i siano tutti distinti segue dal punto (6) dell'oss. 5.21. Sia f un pacchetto massimale di w, distinto da tutti gli α_j e quindi fattore di nessuno di essi. Per l'oss. 5.25 allora f contiene qualche $\alpha_i \wedge \alpha_{i+1}$ come fattore interno. Ma ciò è impossibile, perchè ogni $\alpha_i \wedge \alpha_{i+1}$ è un fattore semplice e non può essere un fattore interno di un pacchetto.

Osservazione 5.27: $(\alpha_0, \alpha_1, ..., \alpha_n)$ sia il ricoprimento canonico di w. Per ogni $i = 0, 1, ..., n-1$ allora $\alpha_i \wedge \alpha_{i+1}$ è il più corto suffisso di α_i semplice in w e il più corto prefisso di α_{i+1} semplice in w.

Dimostrazione: Il primo enunciato segue dalla costruzione (def. 5.20), il secondo per simmetria.

Proposizione 5.28: $(\alpha_0, \alpha_1, ..., \alpha_n)$ sia il ricoprimento canonico di w. Allora i fattori $\alpha_0 \wedge \alpha_1, \alpha_1 \wedge \alpha_2, \alpha_{n-1} \wedge \alpha_n$ sono tutti distinti. L'insieme

$$\{+, \alpha_0 \wedge \alpha_1, \alpha_1 \wedge \alpha_2, ..., \alpha_{n-1} \wedge \alpha_n, -\}$$

è l'insieme dei fattori semplici minimali di w.

Dimostrazione: (1) Gli $\alpha_i \wedge \alpha_{i+1}$ sono tutti semplici e da ciò e dalla costruzione segue che sono tutti distinti.
(2) Dall'oss. 5.27 discende direttamente che, togliendo una lettera a destra o a sinistra di $\alpha_i \wedge \alpha_{i+1}$ si ottiene un fattore ripetuto di w. Perciò $\alpha_i \wedge \alpha_{i+1}$ è un fattore semplice minimale di w.

6. GLI ALGORITMI DI CARPI/DE LUCA IN C++

Questo capitolo contiene i programmi per due algoritmi nei lavori di Carpi/de Luca che sono stati spiegati nei capitoli precedenti di questa tesi. Il primo effettua la ricostruzione di una parola dai suoi pacchetti massimali secondo l'algoritmo dell'oss. 4.16, il secondo trova invece il ricoprimento canonico con i metodi esplicati nella prop. 5.23.

L'algoritmo di ricostruzione

Il file *rico.c* inizia con le dichiarazioni delle funzioni locali.

```
static void errore(int), rico(char**);
static char cont(char*,char**),cont(char*,char*,char**),*ultimo(char*);
static size_t luntotale(char**);
```

La funzione *ricostruzione* che viene chiamata dal file principale *alfa.c* contiene soprattutto le funzioni di lettura dei dati dal file: devono essere eliminati i commenti e predisposto il vettore delle stringhe che rappresenta l'insieme dei pacchetti massimali.

```
void ricostruzione()
// I frammenti devono essere separati da spazi bianchi.
{char a[80],b[80],prefisso[]="Files/frammenti/";
char *X,*Y,**M,*Testo,*W; int k,commento; size_t lun;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) {errore(1); return;}
Testo=new char[lun+5]; W=new char[lun+4];
if (!caricafile(b,Testo,lun)) {errore(1); delete Testo; return;}
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n ') commento=0; else if (*X=='# ') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W+1;*X;X++) if (*X>1) *(Y++)=*X; *Y=0;
for (W[0]=1, X=W;*X;X++) if (*X<=32) *X=1;
for (X=W,k=0;X;X=strpbrk(X,"\1"))
{X+=strspn(X,"\1"); k++;} M=new (char*)[k+1];
for (X=W,k=0;X;X=strpbrk(X,"\1"))
{X+=strspn(X,"\1"); M[k]=X; k++;} M[k]=0;
for (X=W;*X;X++) if (*X==1) *X=0;
rico(M); delete Testo; delete W; delete M;}
```

Il testo viene prima caricato in formato grezzo nell'indirizzo *Testo*, dal quale verra' copiato, dopo l'eliminazione dei commenti, nell'indirizzo *W*.

Partiamo ponendo *X* sul primo byte della stringa; se **X* è il #, attiviamo la modalita' di commento, che viene invece disattivata da un carattere di nuova riga. Quando ci troviamo in modalita' commento, sostituiamo il carattere sotto *X* con 1 e quindi per tutta la lunghezza del commento fino al successivo carattere di nuova riga. Lo scopo è quello di ottenere il testo ripulito dai commenti, quindi ponendo il puntatore all'inizio della stringa e procedendo verso destra riscriviamo solo i caratteri maggiori di 1.

Ripartendo poi da *W* sostituiamo 1 ogni volta che incontriamo un carattere con codice ASCII ≤ 32, quindi anche nel caso che incontriamo un carattere spazio. A questo punto possiamo costruire un vettore *M* di stringhe (cioè un vettore di puntatori a caratteri). Ripartiamo con *X* in *W* ed utilizzando la funzione *strpbrk* del C in un primo momento contiamo solo quanti frammenti ci sono. Ripetiamo l'operazione per assegnare i puntatori. Poniamo l'ultimo puntatore uguale a *0*.

X ripercorre ora *W* ponendo uguale a *0* ogni carattere *1*. Abbiamo separato così le stringhe che costituiscono i frammenti con caratteri zero in modo da poter applicare le funzioni del C per le stringhe.

```
static void errore (int n)
{switch(n) {case 1: puts("File non caricabile."); break;
case 2: puts("Ciclo infinito."); break;
case 3: puts("Ricostruzione non possibile."); break;
case 4: puts("Inizio non trovato.");}}
```

Questa funzione dà informazioni sul tipo degli eventuali errori.

```
static void rico (char **M)
{size_t n=luntotale(M),m; char *P,**PX,*A,a,*Fine; int ok;
P=new char[n+4]; Fine=P+n;
for (PX=M,ok=0;*PX;PX++) if (**PX=='+ ') {ok=1; break;}
if (!ok) {errore(4); return;}
memmove(P,*PX,strlen(*PX)+1); A=ultimo(P);
for (a=*A;a!='- ';A[1]=0) {if (A>=Fine) {errore(2); return;}
a=cont(P,A,M); if (!a) {errore(3); return;}
*(++A)=a;} puts(P);}
```

La funzione *rico* controlla l'algoritmo di ricostruzione. Nella quarta riga cerchiamo il frammento che inizia con +. Poi individuiamo la lettera *a* come *cont(P,A,M)* secondo l'algoritmo. La chiamata di *memmove* e' necessaria

perché i fattori sono delimitati da puntatori alle estremità e non dal vettore *0*.

```
static size_t luntotale (char **M)
{size_t n;
for (n=0;*M;M++) n+=strlen(*M); return n;}
```

Questa funzione calcola la somma delle lunghezze dei frammenti per usarla come limite superiore della lunghezza della stringa che verra' ricostruita.

```
static char *ultimo (char *P)
{if (*P==0) return 0; for (;*P;P++); return P-1;}
```

ultimo(P) é il puntatore all'ultimo carattere della stringa che inizia con *P*.

```
static char cont (char *P, char *A, char **M)
{char **PX,a;
for (a=0;!a&&(A>P);A--) a=cont(A,M); return a;}
```

```
static char cont (char *V, char **M)
{char **PX,*X,x,a; int trovato=0; size_t n=strlen(V);
for (PX=M;*PX;PX++) for (X=strstr(*PX,V);X;X=strstr(X+1,V))
{x=X[n]; if (!x) break;
if (!trovato) {trovato=1; a=x;} else if (a!=x) return 0;}
if (trovato) return a; return 0;}
```

Queste funzioni effettuano l'algoritmo di ricostruzione cercando il piú corto suffisso di un fattore che possiede un'unica continuazione nell'insieme dei pacchetti, determinando quella continuazione.

Calcolo dei pacchetti massimali

Il file *pacchetti.c* contiene all'inizio le dichiarazioni delle funzioni che verranno usate nel seguito:

```
static void d(char*,char*,char*,char**,char**), scrivipm(char*);
static char *pd(char*,char*), *pos(char*,char*,char*), *ps(char*,char*);
static int rip(char*,char*,char*);
```

La funzione *provapacchetti* chiede all'utente di indicare un file che contiene la parola di cui verranno calcolati i pacchetti massimali.

```
void provapacchetti()
{char a[80],b[80],*X,*Y,*Testo,*W,prefisso[]="Files/pacchetti/";
size_t lun,d; int commento;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) return;
Testo=new char[lun+4]; W=new char[lun+4];
if (!caricafile(b,Testo,lun)) {delete Testo; return;}
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n') commento=0; else if (*X=='#') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W;*X;X++) if (*X>32) *(Y++)=*X; *Y=0;
delete Testo; scrivipm(W); delete(W);}
```

Anche qui usiamo la tecnica vista precedentemente nella funzione *ricostruzione* del file *rico.c*; stavolta il compito è piú semplice perché non dobbiamo costruire il vettore di puntatori M.

Rappresenteremo nel seguito la parola w come una stringa normale del C, cioè tramite un puntatore W con la fine della stringa determinata dallo 0. I fattori $\neq \varepsilon$ di w, invece, vengono rappresentati da coppie *(A, B)* di puntatori di cui il primo punta al primo byte del fattore ed il secondo all'ultimo byte del fattore. Ciò ci permette di lavorare direttamente sul segmento di memoria che corrisponde alla parola w senza dover creare copie tranne in alcuni casi. Abbiamo usato questa tecnica in parte già nelle funzioni precedenti.

```
static void d (char *A, char *B, char *W, char **PX, char **PY)
// Continuazione destra di [A,B] in W.
{char *X;
*PX=X=pd(B,W); *PY=ps(X+1,W);}
```

Questa funzione ha per argomenti i puntatori A e B che definiscono il fattore α, il puntatore W che rappresenta la stringa w e i puntatori PX e PY che dalla funzione verranno modificati e i cui contenuti $*PX$ e $*PY$ daranno gli estremi del fattore $D\alpha$. Essa restituisce il fattore $D\alpha$, cioè la continuazione destra del fattore α.
La funzione si avvale delle funzioni *pd* (polo destro) e *ps* (polo sinistro).

```
static char *pd (char *A, char *W)
// X ... suffisso [X,A] semplice in W piu' corto di [W,A].
{char *X=A;
while (rip(X,A,W)) X--; return X;}
```

Chiamiamo qui *polo destro* il più corto suffisso semplice del segmento che va dall'inizio della parola ad A. Facendo riferimento alla teoria e precisamente alla definizione 5.20, intendiamo con polo destro di α il fattore semplice che era stato chiamato at dove t era ripetuto.
La funzione parte con $X = A$. Poi X inizia a spostarsi verso sinistra di una posizione per volta finchè il segmento *[X,A]* non è più ripetuto. A quel punto la funzione restituisce *X*. Nella definizione di *pd* si usa la funzione *rip*:

```
static int rip (char *A, char *B, char *W)
// 1 se [A,B] e' fattore ripetuto di W
{char *X;
X=pos(A,B,W); if (X==0) return 0;
X=pos(A,B,X+1); return (X!=0);}
```

Questa funzione restituisce il valore 1 se la parola f determinata dal segmento [A,B] in W è un fattore ripetuto, altrimenti 0. Se f non è fattore di w, X viene posto uguale a zero e si esce dalla funzione, altrimenti il puntatore X viene messo nella posizione della prima apparizione di f in w. Adesso cerchiamo lo stesso segmento a partire da X + 1: in caso di successo f deve essere un fattore ripetuto di w.

Per determinare la posizione di un segmento in una stringa usiamo la funzione pos:

```
static char *pos (char *A, char *B, char *W)
// Puntatore alla prima apparizione di [A,B] in W.
{char *U,*X;
U=new char[B-A+2];
memmove(U,A,B-A+1); U[B-A+1]=0;
X=strstr(W,U); delete U; return X;}
```

Il polo sinistro di un segmento, cioè il suo più corto prefisso semplice, si determina in modo analogo al polo destro.

```
static char *ps (char *A, char *W)
// X ... prefisso [A,X] semplice in W piu' corto di [A,-].
{char *X=A;
while (rip(A,X,W)) X++; return X;}
```

Ora siamo in grado di leggere la funzione d: essa restituisce la continuazione destra del segmento *[A,B]* in *W*. Più precisamente essa fornisce **PX* e **PY* che sono gli estremi di $D\alpha$, dove **PX* è il risultato della funzione *pd* applicata al segmento di w che va dall'inizio della parola a *B* e **PY* è il

risultato della funzione *ps* applicata al segmento di w che va dalla posizione *X+1* alla fine della parola.

Rimane ora solo l'ultima funzione *scrivipm*.

```
static void scrivipm (char *W)
// Scrive i pacchetti massimali di W.
{char *X,*Y;
nr(); puts(W); puts("\nPacchetti massimali:\n");
for (X=Y=W;;) {d(X,Y,W,&X,&Y); scrivi(X,Y); nr();
if (*Y=='- ') break;}}
```

Essa scrive nella prima riga w e poi, uno per riga, tutti i pacchetti massimali di w. Parte con $X = Y = W$, procede poi cambiando X e Y e applicando ad essi la funzione d e per ogni coppia di X ed Y scrive il risultato ottenuto. Quando infine $*Y$ assume il valore -, la funzione termina.

7. PROGRAMMAZIONE ORIENTATA AGLI OGGETTI IN PERL

Benché non esplicitamente visibile alla superficie, in Perl la programmazione orientata agli oggetti è particolarmente facile ed efficiente. In Perl sono separati tre aspetti elementari dalla strutturazione del codice sorgente: raccolta del codice su più files, suddivisione dello spazio dei nomi (packages), assegnazione di un oggetto ad una classe. Questa separazione dei tre componenti permette al programmatore disciplinato un'organizzazione versatile, efficace ed esteticamente gradevole del proprio lavoro. Presentiamo prima brevemente questi componenti che verranno successivamente discussi in dettaglio. Si noti quanto, a differenza di altri linguaggi della programmazione orientata agli oggetti, essi siano separati ed indipendenti tra loro.

(1) **Files.** Nel file principale che chiamiamo α e che inizierà con la riga *#! /usr/bin/perl -w*, vengono indicate le cartelle aggiuntive dove l'interprete cerca i files sorgente, con *use lib 'a, b, ... ';* e poi i files che intendiamo utilizzare con *use* α*; use* β*; ...* In Perl questi files si chiamano moduli e devono avere il nome α*.pm,* β*.pm, ...* (*pm* significa *Perl module*). I moduli $\underline{non}$ sono associati ad un particolare spazio di nomi.

(2) **Packages.** Ogni istruzione della forma $package\ \pi$; fa in modo che le variabili successivamente dichiarate (fino alla fine del file o fino alla prossima istruzione *package*) prendono il "cognome" (o nome di classe) π e all'esterno del package la variabile x deve essere chiamata $\pi::x$ (con il giusto prefisso quindi ad esempio $\$\pi::x$ per gli scalari). La suddivisione tematica per nomi di classe è completamente indipendente dai files in cui si trovano i nomi. Quando non è dichiarato esplicitamente un package, le variabili appartengono al package *main;* infatti *$a* è in tal caso lo stesso come *$main::a*. Ciò vale sia per il file principale sia per i moduli; ad esempio il file β*.pm* sia così strutturato

```
$a = 6;
package Mat;
$a = 5;
package Fis;
$a = 10;
package Mat;
$b = 9;
```

Allora in α con l'istruzione:

print "$a $Mat::a $Fis::a $Mat::b\n";

otteniamo l'output

6 5 10 9.

perché il primo *$a* nel modulo non essendo ancora definito un package è di *main*. In verità l'istruzione *$prova::a = 7* definisce una variabile di classe *prova* senza che debba essere dichiarato un pacchetto.

(3) **Oggetti.** Un riferimento (o puntatore) $r con $bless\ \$r$ diventa un oggetto del package in cui si trova l'istruzione (lo si può anche assegnare ad un altro package α con $bless\ \$r,\ "\alpha\ "$).
Usi tipici si hanno soprattutto nella definizione di costruttori

sub nuovo {bless @_}
sub nuovo {my $a = shift; bless $a}
sub nuovo {my ($x,$y,$z) = @_; bless {"x",$x, "y",$y, "z", $}}.

Adesso possiamo usare

$v = $Vettore::nuovo(3,6,10);
print $v→{z}; #output: 10

8. ESEMPI

Calcolo dei pacchetti massimali

```
# aaaxaaa
+aaaxaaa-
# Pacchetti massimali:
# +aaax
# xaaa-
```

```
# ababaa
+ababaa-
# Pacchetti massimali:
# +abab
# babaa
# aa-
```

```
# abba
+abba-
# Pacchetti massimali:
# +ab
# abb
# bba
# ba-
```

```
# abbabaab
# Inizio della successione di Morse.
+abbabaab-
# Pacchetti massimali:
# +abb
# bbab
# baba
# abaa
# aab-
```

 # abcdebcd

+abcdebcd-

 # Pacchetti massimali:

 # +a
 # abcde
 # ebcd-

 # abxaaa

+abxaaa-

 # Pacchetti massimali:

 # +ab
 # bx
 # xaaa
 # aaa-

 # arratia-426

 # R. Arratia/D. Martin/G. Reinert/M. Waterman:
 # Poisson process approximation for sequence repeats, and sequencing
 # by hybridization. J. Comp. Biol. 3 (1996), 425-463.
 # Pag. 426.

+gtgaccatggaagacttggaagtt-

 # Pacchetti massimali:

 # +gtg
 # gtga
 # tgacc
 # cca
 # cat
 # atggaaga
 # agact
 # cttg
 # ttggaagt
 # agtt
 # gtt-

 # arratia-452

 # R. Arratia/D. Martin/G. Reinert/M. Waterman:
 # Poisson process approximation for sequence repeats, and sequencing
 # by hybridization. J. Comp. Biol. 3 (1996), 425-463.
 # Pag. 452.

+aacgtagacgtatcgtg-

 # Pacchetti massimali:

 # +aa
 # aacgtag
 # aga
 # gacgtat
 # atc
 # tcgtg
 # tg-

 # carpi-152-1

 # A. Carpi/A. de Luca: Worlds and special factors.
 # Theor. Computer Sci. 259 (2001), 145-182.
 # Pag. 152.

+abccbabcab-

 # Pacchetti massimali:

 # +abcc
 # ccb
 # cba
 # babca
 # cab-

 # carpi-152-2

 # A. Carpi/A. de Luca: Worlds and special factors.
 # Theor. Computer Sci. 259 (2001), 145-182.
 # Pag. 152.

+abaababaaba-

 # Pacchetti massimali:

 # +abaabab
 # babaaba-

 # carpi-157-1

 # A. Carpi/A. de Luca: Worlds and special factors.
 # Theor. Computer Sci. 259 (2001), 145-182.
 # Pag. 157.

+abacbcbacba-

 # Pacchetti massimali:

 # +ab
 # abacbc
 # bcbac
 # cbacba
 # acba-

 # carpi-157-2

 # A. Carpi/A. de Luca: Worlds and special factors.
 # Theor. Computer Sci. 259 (2001), 145-182.
 # Pag. 157.

+abacbcbacbcbacba-

 # Pacchetti massimali:

 # +ab
 # abacbcbacbc
 # cbacbcbacba
 # acba-

 # varricchio-19

 # A. de Luca/S. Varricchio: Finiteness and regularity in
 # semigroups and formal languages. Springer 1999.
 # Pag. 19.

+aabaababbaaaba-

 # Pacchetti massimali:

 # +aabaa
 # abaab
 # baabab
 # babb
 # bbaaa
 # aaaba-

 # carpi-163

 # A. Carpi/A. de Luca: Worlds and special factors.
 # Theor. Computer Sci. 259 (2001), 145-182.
 # Pag. 163.

+babacbcabaccbb-

 # Pacchetti massimali:

 # +bab
 # babacb
 # acbc
 # bca
 # cabacc
 # ccbb
 # bb-

```
# carpi-173

# A. Carpi/A. de Luca: Worlds and special factors.
# Theor. Computer Sci. 259 (2001), 145-182.
# Pag. 173

+abbcbabbbccaab-

# Pacchetti massimali:

# +abbc
# abbcb
# cba
# babbb
# bbbcc
# cca
# caa
# aab-
```

```
# champernowne

# A. de Luca/S. Varricchio: Finiteness and regularity in
# semigroups and formal languages. Springer 1999.
# Pag. 25.
# Inizio della successione di Champernowne.

+0110111001011101111000-

# Pacchetti massimali:

# +0110
# 01101110
# 11011100
# 0111001
# 0010
# 01011101
# 11101111
# 1111000
# 000-
```

 # de-luca-24

 # A. de Luca: On the combinatorics of finite words.
 # Theor. Computer Sci. 218 (1999), 13-39.
 # Pag. 24.

+abbbbbaababaaab-

 # Pacchetti massimali:

 # +abb
 # abbbbb
 # bbbbba
 # bbaab
 # baaba
 # aabab
 # babaa
 # abaaa
 # aaab-

 # de-luca-29

 # A. de Luca: On the combinatorics of finite words.
 # Theor. Computer Sci. 218 (1999), 13-39.
 # Pag. 29.

+ababacbcabacba-

 # Pacchetti massimali:

 # +abab
 # babacbc
 # bca
 # cabacba
 # cba-

```
# de-luca-34

# A. de Luca: On the combinatorics of finite words.
# Theor. Computer Sci. 218 (1999), 13-39.
# Pag. 34.

+abaababaaba-

# Pacchetti massimali:

# +abaabab
# babaaba-
```

```
# de-luca-36

# A. de Luca: On the combinatorics of finite words.
# Theor. Computer Sci. 218 (1999), 13-39.
# Pag. 36.

+babaababaabab-

# Pacchetti massimali:

# +babaababa
# ababaabab-
```

```
# fibonacci

# A. de Luca/S. Varricchio: Finiteness and regularity in
# semigroups and formal languages. Springer 1999.
# Pag. 24.
# Inzio della successione di Fibonacci.

+abaababaabaababaababaabaababaabab-

# Pacchetti massimali:

# +abaababaabaababaabab
# babaababaabaababaabaa
# aabaababaabaab-
```

```
# gcatatcgcgattag
+gcatatcgcgattag-
# Pacchetti massimali:
# +gca
# cata
# atat
# tatc
# tcgc
# cgcg
# gcga
# gatt
# ttag
# ag-
```

```
# gusfield-41
# D. Gusfield: Algorithms on strings, trees, and sequences.
# Cambridge UP 1999.
# Pag. 41.
+abaabaabaabaabaab-
# Pacchetti massimali:
# +abaabaabaabaaba
# aabaabaabaabaab-
```

```
# morse

# A. de Luca/S. Varricchio: Finiteness and regularity in
# semigroups and formal languages. Springer 1999.
# Pag. 55.
# Inizio della successione di Morse.

+abbabaabbaababba-

# Pacchetti massimali:

# +abbab
# bbaba
# babaa
# abaabb
# aabbaa
# bbaaba
# aabab
# ababb
# babba-
```

```
# morse-3

# A. de Luca/S. Varricchio: Finiteness and regularity in
# semigroups and formal languages. Springer 1999.
# Pag. 55.
# Inizio della successione di Morse a tre lettere.

+abcacbabcbacabcacb-

# Pacchetti massimali:

# +abcacba
# acbab
# babcb
# bcbac
# baca
# acab
# cabcacb-
```

 # smillie

 # F. Smillie / W. Bains: Repetition structure of mammalian nuclear
 # DNA. J. Theor. Biol. 142 (1990), 463-471.
 # Pag. 466.
 # Questa successione e' particolarmente difficile da ricostruire.

+ggggctccggcgagagggcgggccccgggaacggcggcgggcggggcgggaggcggggc-

 # Pacchetti massimali:

 # +ggggct
 # tccggc
 # ccggcga
 # cgaga
 # agaggg
 # agggcgggc
 # gggcgggcc
 # gcccc
 # ccccg
 # cccggg
 # ccgggaa
 # aac
 # acggcggc
 # gcggcggg
 # cggcgggcg
 # cgggcgggg
 # gggcggggcg
 # ggggcgggа
 # gcgggag
 # ggaggc
 # aggcggggc-

Ricostruzione

```
# fra-1
# +babacbcabaccbb-
acbc bca +bab bb- cabacc ccbb babacb
```

```
# fra-2
# +gagttcatcgaagtttggattgaccattatataggaaattaggtaggatttatcata-
+gag gagttc ttcatc catcg cgaag aagttt gtttg
tttgg tggattg attga tgac acc ccatt cattat
attata ttatat atatag ataggaa ggaaa aaat aattag
ttaggt ggta gtaggat aggattt attta tttatc tatca
atcata cata-
```

```
# fra-3
# +babacbb-
acbc bca +bab bb- cabacc ccbb babacbb
```

```
# fra-4
# Ciclo infinito.
+aba baba ab- abab
```

```
# waterman-137
# M. Waterman: Introduction to computational biology. Maps,
# sequences and genomes. Chapman & Hall 1996.
# Pag. 137.
atat tatt ttat tata- +taat aata
# La ricostruzione con l'algoritmo di Carpi/de Luca non riesce.
# La più breve superstringa dei frammenti e' +taatattata-,
# i suoi pacchetti massimali sono
# +taa aatat atatt ttata tata-
```

Allineamenti

```
# all-1
gctga
.
aagggt
```

```
# all-2
gctgatatagct
.
gggtgattagct
```

```
# all-3
cgatagatatctgta
.
caattcgaatcaga
```

```
# all-4
cgatagatatctgtatagatatattcgaatcaatagatatct
.
caattcgaatcagatagatatctattcgaatca
```

```
# waterman-217
# M. Waterman: Introduction to computational biology. Maps,
# sequences and genomes. Chapman & Hall 1996.
# Pag. 217.
ccaatctactactgcttgca
.
gccactctcgctgtactgtg
# Generazione 1700, Rendimento -2:
# -ccaatctactactg-cttgca
# gccactct-c-gctgtactgtg
# === === = === ==
```

 # arratia-453

 # R. Arratia / D. Martin / G. Reinert / M. Waterman:
 # Poisson process approximation for sequence repeats, and sequencing
 # by hybridization. J. Comp. Biol. 3 (1996), 425-463.
 # Pag. 453.

ggcattggcataggt

.

ggaatcggcttaggt

 # Generazione 100, Rendimento 9:

 # ggcattggcataggt
 # ggaatcggcttaggt
 # == == === =====

 # crochemore-233

 # M. Crochemore / C. Hancart / T. Lecroq:
 # Algorithmique du texte. Vuibert 2001.
 # Pag. 233.

EAWACQGKL

.

ERDAWCQPGKWY

 # Generazione 100, Rendimento -3:

 # EAWA-CQ-GKL-
 # ERDAWCQPGKWY
 # = = == ==

 # histon-4

sgrgkggkglgkggakrhrkvlrdniqgitkpairrlarrggvkrisgliyeetrgvlkvflen
virdavtytehakrktvtamdvvyalkrqgrtlygfgg

.

sgrgkggkglgkggakrhrkvlrdniqgitkpairrlarrggvkrisgliyeetrgvlkiflen
virdavtyteharrktvtamdvvyalkrqgrtlygfgg

```
# lodish-236
# H. Lodish e.a.: Molecular cell biology. Freeman 2001.
# Pag. 236.
VVSQRFPQNSIGAVGSAMFLRFINPAIVSPYEAGILDKKPPPRIERGLKL
.
AASVNFPEYAYIAVGSFVFLRFIGPALVSPDSENIIIVTHAHDRKPFIT
# Generazione 3900, Rendimento -14:
# VVSQRFPQNSIGAVGSAMFLRFINPAIVSP-YEAGILDKKPPPRIERGLKL
# AASVNFPEYAYIAVGSFVFLRFIGPALVSPDSENIIIVTHAHDRKPFI-T-
# = == ==== ===== == === = = =
```

```
# waterman-194
# M. Waterman: Introduction to computational biology. Maps,
# sequences and genomes. Chapman & Hall 1996.
# Pag. 194.
gctgatatagct
.
gggtgattagct
# Generazione 100, Rendimento 5:
# -gctgatatagct
# gggtgat-tagct
# = ==== =====
```

 # gusfield-333

 # D. Gusfield: Algorithms on strings, trees, and sequences.
 # Cambridge UP 1999.
 # Pag. 333.

HCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR

.

HCILVVLARHCPGEFTPSAHAAMDKFLSKVATVLTSKYR

 # Generazione 100, Rendimento 15:

 # HCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
 # HCILVVLARHCPGEFTPSAHAAMDKFLSKVATVLTSKYR
 # == == == = = ==== == ==== = ========

 # pearson

 # W. Pearson/D. Lipman: Improved tools for biological sequence metrics.
 # Proc. Nat. Ac. USA 85 (1988), 2444-2448.
 # Pag. 2447.

gactcagaaagaacccaccatggtgctgtctcctgccgacaagaccaacgtcaaggccgcctggggta
aggtcggcgcgcacgctggcgagtatggtgcggaggccctggagagg

.

gactgagaaggaaccaccatggtgctgtctcccgctgacaagaccaacatcaagactgcctgggaaaag
atcggcagccacggtggcgagtatggcgccgaggccgtggagagg

 # Generazione 100, Rendimento 78:
#
 # gactcagaaagaacccaccatggtgctgtctcctgccgacaagaccaacgtcaaggccgcctggggtaagg
 # tcggcgcgcacgctggcgagtatggtgcggaggccctggagagg
 # gactgagaaggaac-caccatggtgctgtctcccgctgacaagaccaacatcaagactgcctgggaaaaga
 # tcggcagccacggtggcgagtatggcgccgaggccgtggagagg
 # ==== ==== ==== ================== == ============ ===== = ======= ===
 # ===== ==== ============ == ====== ========

9. IL METODO DELL'OTTIMIZZAZIONE GENETICA

Gli algoritmi genetici sono un insieme di tecniche di ottimizzazione che si ispirano all'evoluzione naturale. I sistemi biologici derivano da processi evolutivi basati sulla riproduzione selettiva degli individui migliori di una popolazione sottoposta a mutazioni e ricombinazione genetica. L'ambiente svolge un ruolo determinante nella selezione naturale in quanto solo gli individui più adatti tendono a riprodursi, mentre quelli le cui caratteristiche sono meno compatibili con l'ambiente tendono a scomparire. L'ottimizzazione genetica può essere applicata a problemi le cui soluzioni sono descrivibili mediante parametri codificabili capaci di rappresentarne le caratteristiche essenziali. Nell'ottimizzazione genetica il ruolo dell'ambiente viene assunto dalla funzione obiettivo che deve essere ottimizzata. Questo metodo presenta due vantaggi: non dipende da particolari proprietà matematiche e soprattutto la complessità è in generale praticamente lineare.

Negli algoritmi genetici, dopo la generazione iniziale di un insieme di possibili soluzioni (individui), alcuni individui sono sottoposti a mutazioni e a scambio di materiale genetico. La funzione di valutazione determina quali dei nuovi individui possono sostituire quelli originali. Nell'ordine di idee di prediligere gli individui migliori, gli algoritmi genetici si basano su tre operazioni fondamentali: il rinnovamento (introduzione di nuovi elementi nella popolazione), la mutazione e gli incroci. Nelle mutazioni il peggiore tra l'originale ed il mutante viene eliminato, negli incroci, invece, si segue il seguente principio: se nessuno dei due nuovi elementi è migliore di entrambi i vecchi, manteniamo i vecchi e scartiamo gli incroci; altrimenti scartiamo entrambi gli elementi vecchi e manteniamo solo gli incroci. Questo perchè selezionare solo gli individui migliori dando solo ad essi la possibilità di moltiplicarsi, tende a produrre una uniformità qualitativa i cui progressi possibili diventano sempre minori e meno probabili. Perciò non è conveniente procedere selezionando e moltiplicando ad ogni passo solo gli elementi migliori, agendo solo su di essi con mutazioni ed incroci. Così facendo infatti dopo breve tempo le soluzioni migliori risultano tutte imparentate tra loro ed è molto alto il rischio che l'evoluzione stagni in una situazione apparentemente ottimale e favorevole che non consente però ulteriori miglioramenti.

L'ottimizzazione genetica può essere di grande aiuto in molti problemi pratici in cui la funzione di valutazione è irregolare o complicata come quando ad esempio dipende in modo non lineare da molti parametri e non è accessibile ai metodi tradizionali. Questi al contrario sono da preferire nel

caso di funzioni molto regolari perchè approssimano la funzione più rapidamente e permettono una stima dell'errore. Tra le più attuali applicazioni possibili dell'ottimizzazione genetica in bioinformatica possiamo ricordare quella ai problemi di previsione della conformazione spaziale di proteine a partire dalla sequenza dei suoi aminoacidi.

Nel prossimo capitolo di questa tesi abbiamo applicato il metodo dell' ottimizzazione genetica al problema di allineamento di sequenze.

Useremo una popolazione di grandezza fissata 40. L'algoritmo consiste nei seguenti passi.

(1) Viene generata in modo casuale una popolazione P di 40 allineamenti.
(2) Per ciascun allineamento viene calcolato il rendimento.
(3) Gli elementi di P vengono ordinati in ordine decrescente secondo il rendimento (in ordine decrescente perché vogliamo massimizzare il rendimento, quindi gli elementi migliori sono quelli con rendimento maggiore).
(4) Gli elementi migliori vengono visualizzati sullo schermo. In questo punto l'algoritmo può essere interrotto dall'osservatore.
(5) I dieci elementi peggiori della popolazione vengono sostituiti da nuovi elementi generati in modo casuale.
(6) Incroci.
(7) Mutazioni.
(8) Si torna al punto (2).

10. OTTIMIZZAZIONE GENETICA DI ALLINEAMENTI

In questo capitolo descriviamo il programma per l'ottimizzazione genetica di allineamenti. Cominciamo con le dichiarazioni:

```
class allineamento {public: char *S1,*S2; int n; double rendimento;
   void calcolarendimento(), copia(allineamento*),
   muta(), nuovo(), riduci();};
static void aggiusta(char*,int), allinea(char*,char*,char*),
   impostapuntatori(), incroci(),
   incrocio(allineamento*,allineamento*), liberapuntatori(),
   mutazioni(), nuovi(int,int), ordina();
static int migliore(void*,void*),
migliore(allineamento*,allineamento*),
   paroledafile(), prepara(allineamento*), uno(char*), visualizza(int);
static double val2(char*,char*);

static char *W1,*W2;
static char *A1,*A2;
static int n1,n2,m,maxn1n2;
static char sep='- ';
static allineamento allineamenti[40],*Puntatori[41],
   confronto1,confronto2;
```

È qui dichiarata la classe *allineamento* che contiene come dati le stringhe *S1* e *S2* che corrispondono agli *schemi* (o *maschere*) che rappresentano l'allineamento, la lunghezza n dell'allineamento e il rendimento che viene calcolato secondo la formula delineata nell'introduzione ($+1$ per ogni corrispondenza, -1 per lettere differenti, -2 risp. -4 per uno o due trattini nella stessa posizione). I metodi della classe effettuano il calcolo del rendimento e le operazioni di copiatura, di mutazione, di riduzione (cioè l'eliminazione di trattini doppi che quindi dovrebbe rendere superflua la valutazione -4).

Al di fuori della classe le stringhe *W1* e *W2* contengono le due parole da allineare, *A1* e *A2* invece l'allineamento reale che viene costruito dalle maschere.

Assumiamo ad esempio che *W1* corrisponda alla stringa *ACTGA*. Se allora *S1* è uguale a *011001011*, la prima componente dell'allineamento diventa *-AC–T-GA*; sostituiamo cioè ogni 0 con un trattino (il separatore *sep*), ogni 1 con la prossima lettera della parola. È chiaro che ci devono essere tanti 1 quante sono le lettere di *W1*. Le operazioni genetiche (ad esempio le

mutazioni) vengono effettuate solo sulle maschere che vengono trasformate negli allineamenti quando serve.

Gli allineamenti *confronto1* e *confronto2* vengono utilizzati per contenere i nuovi esemplari durante le operazioni di mutazione e di incrocio.

Vengono indicati con *n1* ed *n2* le lunghezze delle parole *W1* e *W2*, con *m* la somma *n1 + n2 + 4* che è la massima lunghezza possibile di un allineamento e con *maxn1n2* la lunghezza massima tra n1 ed n2.

Veniamo ora alla definizione della funzione *ottimizzazione* che viene chiamata dal menu principale e che governa le funzioni secondarie del programma. Tra queste la prima ad essere chiamata è *paroledafile* che legge le coppie di parole da allineare da un file che può essere scelto dall'utente.

```
void ottimizzazione ()
{int t,dt=100;
if (!paroledafile()) {printf("\nFile non caricabile.\n"); return;}
impostacasuali(); impostapuntatori(); nuovi(0,39);
for (t=1;;t++) {ordina();
if (t%dt==0) if (!visualizza(t)) break;
nuovi(30,39); incroci(); mutazioni();}
liberapuntatori();}
```

```
static int paroledafile ()
{char a[80],b[80],*X,*Y,*Testo,prefisso[]="Files/allineamenti/";
size_t lun,d; int commento;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) return 0;
Testo=new char[lun+4]; W1=new char[lun+4];
if (!caricafile(b,Testo,lun)) return 0;
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n ') commento=0; else if (*X=='# ') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W1;*X!='. ';X++)
if (*X>32) *(Y++)=*X; *Y=0;
for (X++;*X<=32;X++);
for (W2=++Y;*X;X++) if (*X>32) *(Y++)=*X; *Y=0;
delete Testo; return 1;}
```

La tecnica usata in questa funzione è molto simile a quella già vista nelle funzioni di lettura da file del capitolo sesto. Qui le due stringhe da allineare

sul file sono separate da un punto e ciò viene applicato nella quintultima riga della funzione.

```
static void impostapuntatori ()
{int k;
n1=strlen(W1); n2=strlen(W2); m=n1+n2+4;
if (n1<n2) maxn1n2=n2; else maxn1n2=n1;
A1=new char[m+1]; A2=new char[m+1];
for (k=0;k<40;k++) {Puntatori[k]=&allineamenti[k];
prepara(&allineamenti[k]);} Puntatori[k]=0;
prepara(&confronto1); prepara(&confronto2);}
```

Questa funzione imposta i puntatori come richiesto dalla funzione *quick-sort*; in particolare l'ultimo puntatore del vettore di puntatori *Puntatori* deve essere il puntatore zero. La funzione calcola *n1*, *n2* ed *m* come esposto all'inizio.

```
static int prepara (allineamento *P)
// il risultato potrebbe essere utilizzato per un
// controllo sullo spazio disponibile
{char *X;
X=P→S1=new char[m+1]; if (!X) return 0;
X=P→S2=new char[m+1]; if (!X) return 0;
P→S1[m]=P→S2[m]=0; P→n=m; return 1;}
```

Questa funzione prepara lo spazio per i due allineamenti; l'avremmo potuto anche includere tra i metodi della classe *allineamento*.

```
static void nuovi (int a, int b)
{int k;
for (k=a;k<=b;k++)
{Puntatori[k]→nuovo(); Puntatori[k]→calcolarendimento();}}
```

```
void allineamento::nuovo ()
{int i,p;
for (i=0;i<m;i++) S1[i]=S2[i]='0 ';
n=m; S1[n]=S2[n]=0;
aggiusta(S1,n1); aggiusta(S2,n2);
riduci();}
```

Il metodo *nuovo* della classe crea i due schemi *S1* ed *S2* entrambi di lunghezza *m* ponendo prima tutti caratteri uguali a zero per applicare poi la funzione *aggiusta* per selezionare (in modo casuale) le posizioni in cui la maschera contiene 1. Il metodo *riduci* elimina le posizioni con due trattini sovrapposti.

```
void allineamento::riduci ()
{char *U,*X,*V,*Y; int k,j;
for (U=X=S1,V=Y=S2,j=k=0;k<n;X++,Y++,k++)
{if ((*X=='0 ') && (*Y=='0 ')) continue;
*(U++)=*X; *(V++)=*Y; j++;} *U=*V=0; n=j;}
```

```
static void aggiusta (char *S, int p)
{int q,i,lun;
q=uno(S); if (p==q) return;
lun=strlen(S);
while (q<p) {i=dado(lun)-1; if (S[i]=='0 ') {q++; S[i]='1 ';}}
while (q>p) {i=dado(lun)-1; if (S[i]=='1 ') {q--; S[i]='0 ';}}}
```

La funzione *aggiusta* fa in modo che ci siano tanti 1 nelle maschere dell'allineamento quante sono le lettere nelle due stringhe a cui le maschere corrispondono. La funzione sceglie a caso le posizioni in cui scrivere o togliere un 1.

```
static int uno (char *A)
{int p;
for (p=0;*A;A++) if (*A=='1 ') p++; return p;}
```

Questa funzione semplicemente conta il numero degli 1 presenti in una maschera.

```
void allineamento::calcolarendimento ()
{allinea(S1,W1,A1); allinea(S2,W2,A2); rendimento=val2(A1,A2);}
```

Questa funzione calcola il rendimento di un allineamento. Come si vede, dalle due maschere *S1* e *S2* vengono prima costruite le due stringhe che costituiscono l'allineamento il cui valore viene successivamente calcolato dalla funzione *val2*.

```
static void allinea (char *S, char *W, char *A)
/ / Scrive in A la parola W allineata secondo lo schema S.
{for (;*S;S++,A++) {if (*S=='0 ') *A=sep; else *A=*(W++);} *A=0;}
```

Abbiamo già descritto all'inizio come a una maschera corrisponde una stringa in cui ogni 1 della maschera viene sostituito da una lettera della parola da allineare, ogni zero da un trattino.

```
static double val2 (char *A, char *B)
{double val=0; char a,b; int vuoti;
for (;*A;A++,B++) {a=*A; b=*B; vuoti=(a==sep)+(b==sep);
if (vuoti) {val-=(vuoti+vuoti); continue;}
if (a==b) val+=1; else val-=1;} return val;}
```

Questo è l'algoritmo di valutazione già altre volte descritto.

```
static void ordina ()
{quicksort((void**)Puntatori,migliore);}
```

La funzione *ordina* si avvale della funzione *quicksort*. Il *quicksort* è considerato l'algoritmo generico di ordinamento più efficiente ed è qui realizzato in una funzione contenuta nel file ausiliario *quicksort.c*.

```
static int visualizza (int t)
{allineamento *P=Puntatori[0]; char a[40],*X,*Y;
allinea(P→S1,W1,A1); allinea(P→S2,W2,A2);
printf("\nGenerazione %d, Rendimento %.0f:\n\n",t,P→rendimento);
puts(A1); puts(A2); for (X=A1,Y=A2;*X;X++,Y++)
if (*X==*Y) printf("="); else printf(" "); puts("");
printf("\nVuoi continuare? "); input(a,40);
if(us(a,"no")) return 0; return 1;}
```

Ogni 100 generazioni l'allineamento migliore viene visualizzato; le posizioni in cui i caratteri delle due stringhe coincidono vengono sottolineate da un segno di uguaglianza (=). L'utente, rispondendo *no* alla domanda del programma, può terminare l'ottimizzazione.

```
static void incroci ()
{int k; allineamento *P,*Q;
for (k=0;k<38;k+=2) {P=Puntatori[k]; Q=Puntatori[k+1];
incrocio(P,Q);
confronto1.calcolarendimento(); confronto2.calcolarendimento();
if ((migliore(&confronto1,P)&&migliore(&confronto2,Q))||
migliore(&confronto1,Q)&&migliore(&confronto2,P))
{confronto1.copia(P); confronto2.copia(Q);}}}
```

In un incrocio da due allineamenti vengono creati due altri e confrontati con quelli originali. Se uno dei nuovi allineamenti è migliore di entrambi gli originali, i nuovi allineamenti sostituiscono quelli vecchi. Nell'introduzione abbiamo spiegato perché anche nel caso che uno degli originali fosse migliore di uno dei due allineamenti nuovi, bisogna sostituire entrambi gli originali.

```
static void incrocio (allineamento *P, allineamento *Q)
{int alfa1,alfa2,i; char *X,*Y;
alfa1=P→n/2; alfa2=Q→n/2;
for (X=P→S1,Y=confronto1.S1,i=0;i<alfa1;X++,Y++,i++) *Y=*X;
for (X=P→S2,Y=confronto1.S2,i=0;i<alfa1;X++,Y++,i++) *Y=*X;
for (X=Q→S1,Y=confronto2.S1,i=0;i<alfa2;X++,Y++,i++) *Y=*X;
for (X=Q→S2,Y=confronto2.S2,i=0;i<alfa2;X++,Y++,i++) *Y=*X;
for (X=P→S1+alfa1,Y=confronto2.S1+alfa2,i=alfa1;i<P→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=P→S2+alfa1,Y=confronto2.S2+alfa2,i=alfa1;i<P→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=Q→S1+alfa2,Y=confronto1.S1+alfa1,i=alfa2;i<Q→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=Q→S2+alfa2,Y=confronto1.S2+alfa1,i=alfa2;i<Q→n;
   X++,Y++,i++) *Y=*X; *Y=0;
confronto1.n=alfa1+Q→n-alfa2; confronto2.n=alfa2+P→n-alfa1;
aggiusta(confronto1.S1,n1); aggiusta(confronto1.S2,n2);
aggiusta(confronto2.S1,n1); aggiusta(confronto2.S2,n2);
confronto1.riduci(); confronto2.riduci();}
```

L'idea usata negli incroci è che i due allineamenti originali vengono tagliati a metà e i pezzi incrociati. Naturalmente a questo punto le due maschere non conterranno più il giusto numero di 1 (che deve essere uguale a *n1* nella prima maschera e uguale a *n2* nella seconda), perciò viene chiamata ancora

la funzione *aggiusta* che, in modo casuale, aggiunge o toglie gli 1 mancanti o in soprannumero.

```
static int migliore (void *P, void *Q)
{allineamento *A=(allineamento *)P, *B=(allineamento *)Q;
return (A→rendimento>B→rendimento);}
```

```
static int migliore (allineamento *P, allineamento *Q)
{return (P→rendimento>Q→rendimento);}
```

Abbiamo usato due funzioni *migliore* per la leggera differenza nel tipo dei parametri; infatti la funzione *quicksort* richiede che *migliore* abbia come argomenti due puntatori generici, cioè di tipo *void**.

```
void allineamento::copia (allineamento *P)
{char *X,*Y;
for (X=S1,Y=P→S1;*X;X++,Y++) *Y=*X; *Y=0;
for (X=S2,Y=P→S2;*X;X++,Y++) *Y=*X; *Y=0;
P→n=n; P→rendimento=rendimento;}
```

Questa funzione crea una copia di un allineamento.

```
static void mutazioni ()
{int k; allineamento *P;
for (k=0;k<40;k++) {P=Puntatori[k]; P→copia(&confronto1);
confronto1.muta(); confronto1.calcolarendimento();
if (migliore(&confronto1,P)) confronto1.copia(Puntatori[k]);}}
```

```
void allineamento::muta ()
{int nn,i,j,k; char x,y;
nn=dado(maxn1n2,m); S1[nn]=S2[nn]=0;
if (nn<n) {aggiusta(S1,n1); aggiusta(S2,n2);}
else for (i=n;i<nn;i++) S1[i]=S2[i]='0 '; n=nn;
for (k=0;k<20;k++) {i=dado(n)-1; j=dado(n)-1;
x=S1[i]; y=S1[j]; if (x!=y) {S1[i]=y; S1[j]=x;}
i=dado(n)-1; j=dado(n)-1;
x=S2[i]; y=S2[j]; if (x!=y) {S2[i]=y; S2[j]=x;}}
riduci();}
```

La mutazione di una maschera consiste in modifiche casuali precedute da una modifica anch'essa casuale della lunghezza della maschera.

```
static void liberapuntatori ()
{int k;
for (k=0;k<40;k++)
{delete allineamenti[k].S1; delete allineamenti[k].S2;}
delete confronto1.S1; delete confronto1.S2;
delete confronto2.S1; delete confronto2.S2;
delete W1; delete A1; delete A2;}
```

Vengono liberati gli spazi riservati ai puntatori.

11. LISTATI IN C++

```
# Makefile per il C++

librerie = -lc -lm

VPATH=Oggetti

make: alfa.o allineamenti.o aus.o casuali.o files.o pacchetti.o\
    quicksort.o rico.o
g++ -o alfa Oggetti/*.o $(librerie)

%.o: %.c alfa.h
g++ -o Oggetti/$*.o -c $*.c
```

```
// alfa.c
# include "alfa.h"

int main();
////////////////////////////////////////////////////
int main()
{char a[80];
impostacasuali();
for (;;) {printf("\nScelta: "); input(a,40);
if (us(a,"fine")) goto fine;
if (us(a,"all")) ottimizzazione(); else
if (us(a,"pac")) provapacchetti(); else
if (us(a,"ric")) ricostruzione();}
fine: exit(0);}
```

```
// casuali.c
# include "alfa.h"

void impostacasuali()
{unsigned int u;
u=time(0)+rand(); srand(u);}

int dado (int n)
{long x;
if (n<=1) return 1; x=rand(); x%=n; return x+1;}

int dado (int a, int b)
{return a-1+dado(b-a+1);}

double casualereale (double a, double b, double dx)
{return a+dado(0,(int)((b-a)/dx))*dx;}
```

```
// alfa.h
# include <math.h>
# include <stdarg.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <time.h>
////////////////////////////////////////////////////
// allineamenti.c
void ottimizzazione();
////////////////////////////////////////////////////
// out.c
void input(char*,int), nr(int=1), scrivi(char*,char*);
int uis(char*,char*), us(char*,char*);
////////////////////////////////////////////////////
// casuali.c
void impostacasuali();
int dado(int), dado(int,int);
double casualereale(double,double,double);
////////////////////////////////////////////////////
// files.c
int aggiungifile(char*,char*), caricafile(char*,char*,size_t),
   scrivifile(char*,char*);
size_t lunghezzafile(char*);
////////////////////////////////////////////////////
// pacchetti.c
void provapacchetti();
////////////////////////////////////////////////////
// quicksort.c
void quicksort(void**,int(*)(void*,void*));
////////////////////////////////////////////////////
// rico.c
void ricostruzione();
```

```
// allineamenti.c
# include "alfa.h"

class allineamento {public: char *S1,*S2; int n; double rendimento;
   void calcolarendimento(), copia(allineamento*),
   muta(), nuovo(), riduci();};

static void aggiusta(char*,int), allinea(char*,char*,char*),
   impostapuntatori(), incroci(),
   incrocio(allineamento*,allineamento*), liberapuntatori(),
   mutazioni(), nuovi(int,int), ordina();
static int migliore(void*,void*),
migliore(allineamento*,allineamento*),
   paroledafile(), prepara(allineamento*), uno(char*), visualizza(int);
static double val2(char*,char*);

static char *W1,*W2;
static char *A1,*A2;
static int n1,n2,m,maxn1n2;
static char sep='- ';
static allineamento allineamenti[40],*Puntatori[41],
   confronto1,confronto2;
////////////////////////////////////////////////////////
void ottimizzazione ()
{int t,dt=100;
if (!paroledafile()) {printf("\nFile non caricabile.\n"); return;}
impostacasuali(); impostapuntatori(); nuovi(0,39);
for (t=1;;t++) {ordina();
if (t%dt==0) if (!visualizza(t)) break;
nuovi(30,39); incroci(); mutazioni();}
liberapuntatori();}
////////////////////////////////////////////////////////
static void aggiusta (char *S, int p)
{int q,i,lun;
q=uno(S); if (p==q) return;
lun=strlen(S);
while (q<p) {i=dado(lun)-1; if (S[i]=='0 ') {q++; S[i]='1 ';}}
while (q>p) {i=dado(lun)-1; if (S[i]=='1 ') {q--; S[i]='0 ';}}}

static void allinea (char *S, char *W, char *A)
// Scrive in A la parola W allineata secondo lo schema S.
{for (;*S;S++,A++) {if (*S=='0 ') *A=sep; else *A=*(W++);} *A=0;}
```

```
static void impostapuntatori ()
{int k;
n1=strlen(W1); n2=strlen(W2); m=n1+n2+4;
if (n1<n2) maxn1n2=n2; else maxn1n2=n1;
A1=new char[m+1]; A2=new char[m+1];
for (k=0;k<40;k++) {Puntatori[k]=&allineamenti[k];
prepara(&allineamenti[k]);} Puntatori[k]=0;
prepara(&confronto1); prepara(&confronto2);}

static void incroci ()
{int k; allineamento *P,*Q;
for (k=0;k<38;k+=2) {P=Puntatori[k]; Q=Puntatori[k+1];
incrocio(P,Q);
confronto1.calcolarendimento(); confronto2.calcolarendimento();
if ((migliore(&confronto1,P)&&migliore(&confronto2,Q))||
migliore(&confronto1,Q)&&migliore(&confronto2,P))
{confronto1.copia(P); confronto2.copia(Q);}}}

static void incrocio (allineamento *P, allineamento *Q)
{int alfa1,alfa2,i; char *X,*Y;
alfa1=P→n/2; alfa2=Q→n/2;
for (X=P→S1,Y=confronto1.S1,i=0;i<alfa1;X++,Y++,i++) *Y=*X;
for (X=P→S2,Y=confronto1.S2,i=0;i<alfa1;X++,Y++,i++) *Y=*X;
for (X=Q→S1,Y=confronto2.S1,i=0;i<alfa2;X++,Y++,i++) *Y=*X;
for (X=Q→S2,Y=confronto2.S2,i=0;i<alfa2;X++,Y++,i++) *Y=*X;
for (X=P→S1+alfa1,Y=confronto2.S1+alfa2,i=alfa1;i<P→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=P→S2+alfa1,Y=confronto2.S2+alfa2,i=alfa1;i<P→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=Q→S1+alfa2,Y=confronto1.S1+alfa1,i=alfa2;i<Q→n;
   X++,Y++,i++) *Y=*X; *Y=0;
for (X=Q→S2+alfa2,Y=confronto1.S2+alfa1,i=alfa2;i<Q→n;
   X++,Y++,i++) *Y=*X; *Y=0;
confronto1.n=alfa1+Q→n-alfa2; confronto2.n=alfa2+P→n-alfa1;
aggiusta(confronto1.S1,n1); aggiusta(confronto1.S2,n2);
aggiusta(confronto2.S1,n1); aggiusta(confronto2.S2,n2);
confronto1.riduci(); confronto2.riduci();}
```

```
static void liberapuntatori ()
{int k;
for (k=0;k<40;k++)
{delete allineamenti[k].S1; delete allineamenti[k].S2;}
delete confronto1.S1; delete confronto1.S2;
delete confronto2.S1; delete confronto2.S2;
delete W1; delete A1; delete A2;}

static void mutazioni ()
{int k; allineamento *P;
for (k=0;k<40;k++) {P=Puntatori[k]; P→copia(&confronto1);
confronto1.muta(); confronto1.calcolarendimento();
if (migliore(&confronto1,P)) confronto1.copia(Puntatori[k]);}}

static void nuovi (int a, int b)
{int k;
for (k=a;k<=b;k++)
{Puntatori[k]→nuovo(); Puntatori[k]→calcolarendimento();}}

static void ordina ()
{quicksort((void**)Puntatori,migliore);}

static int migliore (void *P, void *Q)
{allineamento *A=(allineamento *)P, *B=(allineamento *)Q;
return (A→rendimento>B→rendimento);}

static int migliore (allineamento *P, allineamento *Q)
{return (P→rendimento>Q→rendimento);}

static int paroledafile ()
{char a[80],b[80],*X,*Y,*Testo,prefisso[]="Files/allineamenti/";
size_t lun,d; int commento;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) return 0;
Testo=new char[lun+4]; W1=new char[lun+4];
if (!caricafile(b,Testo,lun)) return 0;
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n ') commento=0; else if (*X=='# ') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W1;*X!='. ';X++)
if (*X>32) *(Y++)=*X; *Y=0;
for (X++;*X<=32;X++);
for (W2=++Y;*X;X++) if (*X>32) *(Y++)=*X; *Y=0;
delete Testo; return 1;}
```

```
static int prepara (allineamento *P)
// il risultato potrebbe essere utilizzato per un
// controllo sullo spazio disponibile
{char *X;
X=P→S1=new char[m+1]; if (!X) return 0;
X=P→S2=new char[m+1]; if (!X) return 0;
P→S1[m]=P→S2[m]=0; P→n=m; return1;}

static int uno (char *A)
{int p;
for (p=0;*A;A++) if (*A=='1 ') p++; return p;}

static int visualizza (int t)
{allineamento *P=Puntatori[0]; char a[40],*X,*Y;
allinea(P→S1,W1,A1); allinea(P→S2,W2,A2);
printf("\nGenerazione %d, Rendimento %.0f:\n\n",t,P→rendimento);
puts(A1); puts(A2); for (X=A1,Y=A2;*X;X++,Y++)
if (*X==*Y) printf("="); else printf(" "); puts("");
printf("\nVuoi continuare? "); input(a,40);
if(us(a,"no")) return 0; return 1;}

static double val2 (char *A, char *B)
{double val=0; char a,b; int vuoti;
for (;*A;A++,B++) {a=*A; b=*B; vuoti=(a==sep)+(b==sep);
if (vuoti) {val-=(vuoti+vuoti); continue;}
if (a==b) val+=1; else val-=1;} return val;}
////////////////////////////////////////////////////////////
void allineamento::calcolarendimento ()
{allinea(S1,W1,A1); allinea(S2,W2,A2); rendimento=val2(A1,A2);}

void allineamento::copia (allineamento *P)
{char *X,*Y;
for (X=S1,Y=P→S1;*X;X++,Y++) *Y=*X; *Y=0;
for (X=S2,Y=P→S2;*X;X++,Y++) *Y=*X; *Y=0;
P→n=n; P→rendimento=rendimento;}
```

```
void allineamento::muta ()
{int nn,i,j,k; char x,y;
nn=dado(maxn1n2,m); S1[nn]=S2[nn]=0;
if (nn<n) {aggiusta(S1,n1); aggiusta(S2,n2);}
else for (i=n;i<nn;i++) S1[i]=S2[i]='0 ';
n=nn;
for (k=0;k<20;k++) {i=dado(n)-1; j=dado(n)-1;
x=S1[i]; y=S1[j]; if (x!=y) {S1[i]=y; S1[j]=x;}
i=dado(n)-1; j=dado(n)-1;
x=S2[i]; y=S2[j]; if (x!=y) {S2[i]=y; S2[j]=x;}}
riduci();}

void allineamento::nuovo ()
{int i,p;
for (i=0;i<m;i++) S1[i]=S2[i]='0 ';
n=m; S1[n]=S2[n]=0;
aggiusta(S1,n1); aggiusta(S2,n2);
riduci();}

void allineamento::riduci ()
{char *U,*X,*V,*Y; int k,j;
for (U=X=S1,V=Y=S2,j=k=0;k<n;X++,Y++,k++)
{if ((*X=='0 ') && (*Y=='0 ')) continue;
*(U++)=*X; *(V++)=*Y; j++;} *U=*V=0; n=j;}
```

```
// pacchetti.c
# include "alfa.h"

static void d(char*,char*,char*,char**,char**), scrivipm(char*);

static char *pd(char*,char*), *pos(char*,char*,char*), *ps(char*,char*);

static int rip(char*,char*,char*);
////////////////////////////////////////////////////////
void provapacchetti()
{char a[80],b[80],*X,*Y,*Testo,*W,prefisso[]="Files/pacchetti/";
size_t lun,d; int commento;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) return;
Testo=new char[lun+4]; W=new char[lun+4];
if (!caricafile(b,Testo,lun)) {delete Testo; return;}
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n ') commento=0; else if (*X=='# ') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W;*X;X++) if (*X>32) *(Y++)=*X; *Y=0;
delete Testo; scrivipm(W); delete(W);}
////////////////////////////////////////////////////////
static void d (char *A, char *B, char *W, char **PX, char **PY)
// Continuazione destra di [A,B] in W.
{char *X;
*PX=X=pd(B,W); *PY=ps(X+1,W);}

static char *pd (char *A, char *W)
// X ... suffisso [X,A] semplice in W piu' corto di [W,A].
{char *X=A;
while (rip(X,A,W)) X--; return X;}

static char *pos (char *A, char *B, char *W)
// Puntatore alla prima apparizione di [A,B] in W.
{char *U,*X;
U=new char[B-A+2];
memmove(U,A,B-A+1); U[B-A+1]=0;
X=strstr(W,U); delete U; return X;}

static char *ps (char *A, char *W)
// X ... prefisso [A,X] semplice in W piu' corto di [A,-].
{char *X=A;
while (rip(A,X,W)) X++; return X;}
```

```
static int rip (char *A, char *B, char *W)
// 1 se [A,B] e' fattore ripetuto di W
{char *X;
X=pos(A,B,W); if (X==0) return 0;
X=pos(A,B,X+1); return (X!=0);}

static void scrivipm (char *W)
// Scrive i pacchetti massimali di W.
{char *X,*Y;
nr(); puts(W); puts("\n# Pacchetti massimali:\n");
for (X=Y=W;;) {d(X,Y,W,&X,&Y); printf("# "); scrivi(X,Y); nr();
if (*Y=='-') break;}}
```

// ***files.c***

```
# include "alfa.h"

int aggiungifile (char *A, char *B)
{FILE *File;
File=fopen(B,"a"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File); return 1;}

int caricafile (char *A, char *B, size_t n)
{FILE *File; int z,tutti=0; size_t k;
File=fopen(A,"r");
if (File==0) {*B=0; return 0;} for (k=0;k<n;k++,B++)
{z=getc(File); if (z==EOF) {tutti=1; break;} *B=z;}
fclose(File); *B=0; if (tutti) return 1; return -1;}

int scrivifile (char *A, char *B)
{FILE *File;
File=fopen(B,"w"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File); return 1;}

size_t lunghezzafile (char *A)
{FILE *File;
File=fopen(A,"r"); if (File==0) return 0;
fseek(File,0,SEEK_END); return ftell(File);}
```

```
// rico.c
# include "alfa.h"

static void errore(int), rico(char**);
static char cont(char*,char**),cont(char*,char*,char**),*ultimo(char*);
static size_t luntotale(char**);
////////////////////////////////////////////////////////////////
void ricostruzione()
// I frammenti devono essere separati da spazi bianchi.
{char a[80],b[80],prefisso[]="Files/frammenti/";
char *X,*Y,**M,*Testo,*W; int k,commento; size_t lun;
printf("\nFile: "); input(a,40);
sprintf(b,prefisso); sprintf(b+strlen(prefisso),a);
lun=lunghezzafile(b); if (lun==0) {errore(1); return;}
Testo=new char[lun+5]; W=new char[lun+4];
if (!caricafile(b,Testo,lun)) {errore(1); delete Testo; return;}
for (X=Testo,commento=0;*X;X++)
{if (*X=='\n ') commento=0; else if (*X=='# ') commento=1;
if (commento) *X=1;}
for (X=Testo,Y=W+1;*X;X++) if (*X>1) *(Y++)=*X; *Y=0;
for (W[0]=1, X=W;*X;X++) if (*X<=32) *X=1;
for (X=W,k=0;X;X=strpbrk(X,"\1"))
{X+=strspn(X,"\1"); k++;} M=new (char*)[k+1];
for (X=W,k=0;X;X=strpbrk(X,"\1"))
{X+=strspn(X,"\1"); M[k]=X; k++;} M[k]=0;
for (X=W;*X;X++) if (*X==1) *X=0;
rico(M); delete Testo; delete W; delete M;}
////////////////////////////////////////////////////////////////
static void errore (int n)
{switch(n) {case 1: puts("File non caricabile."); break;
case 2: puts("Ciclo infinito."); break;
case 3: puts("Ricostruzione non possibile."); break;
case 4: puts("Inizio non trovato.");}}
```

```
static void rico (char **M)
{size_t n=luntotale(M),m; char *P,**PX,*A,a,*Fine; int ok;
P=new char[n+4]; Fine=P+n;
for (PX=M,ok=0;*PX;PX++) if (**PX==’+ ’) {ok=1; break;}
if (!ok) {errore(4); return;}
memmove(P,*PX,strlen(*PX)+1); A=ultimo(P);
for (a=*A;a!=’- ’;A[1]=0) {if (A>=Fine) {errore(2); return;}
a=cont(P,A,M); if (!a) {errore(3); return;}
*(++A)=a;} puts(P);}

static char cont (char *P, char *A, char **M)
{char **PX,a;
for (a=0;!a&&(A>P);A--) a=cont(A,M); return a;}

static char cont (char *V, char **M)
{char **PX,*X,x,a; int trovato=0; size_t n=strlen(V);
for (PX=M;*PX;PX++) for (X=strstr(*PX,V);X;X=strstr(X+1,V))
{x=X[n]; if (!x) break;
if (!trovato) {trovato=1; a=x;} else if (a!=x) return 0;}
if (trovato) return a; return 0;}

static char *ultimo (char *P)
{if (*P==0) return 0; for (;*P;P++); return P-1;}

static size_t luntotale (char **M)
{size_t n;
for (n=0;*M;M++) n+=strlen(*M); return n;}
```

```
// quicksort.c
# include "alfa.h"

static void quicksortinterno (void**,int,int,int(*)(void*,void*));
////////////////////////////////////////////////////////////////////
void quicksort (void **Dati, int (*migliore)(void*,void*))
{int n; void **Ultimo;
for (Ultimo=Dati,n=0;*Ultimo;Ultimo++,n++);
quicksortinterno(Dati,0,n-1,migliore);}
////////////////////////////////////////////////////////////////////
static void quicksortinterno (void **Dati, int sin, int des,
   int (*migliore)(void*,void*))
{void *Conf,*X; int p,i,m;
if (sin>=des) return; m=(sin+des)/2;
Conf=Dati[m]; Dati[m]=Dati[sin]; p=sin+1;
for (i=p;i<=des;i++) if ((*migliore)(Dati[i],Conf))
{X=Dati[p]; Dati[p]=Dati[i]; Dati[i]=X; p++;}
p--; Dati[sin]=Dati[p]; Dati[p]=Conf;
quicksortinterno(Dati,sin,p-1,migliore);
quicksortinterno(Dati,p+1,des,migliore);}

// aus.c
# include "alfa.h"

void input (char *A, int n)
{if (n<1) n=1; fgets(A,n+1,stdin);
for (;*A;A++); A--;
if (*A=='\n ') *A=0;}

void nr (int n=1)
// Nuova riga, n volte.
{int k;
for (k=0;k<n;k++) puts("");}

void scrivi (char* A, char *B)
// scrive [A,B].
{for (;A<=B;A++) printf("%c",*A);}

int uis (char *A, char *B)
{return (strncmp(A,B,strlen(A))==0);}

int us (char *A, char *B)
{return (strcmp(A,B)==0);}
```

12. LISTATO IN PERL

```
1; # base.pm
package base;

use overload ".",\&concat, "/",\&segmento, "| ",\&segmentoinv,
   "+",\&intd, "-",\&ints, "++",\&st;

sub n {my $a=shift; bless \$a}
########################################
sub cd # completamento destro di f in w
   {my ($f,$w)=@_; my $i=$f→ind($w); $w/[$i,$w→lun-$i+1]}

sub cs # completamento sinistro di f in w
   {my ($f,$w)=@_; $w/[1,$f→ind($w)+$f→lun-1]}

sub d # continuazione destra
   {my ($f,$w)=@_; my $u=$f→pd($w)→cd($w);
   ($u/1).($u+1)→ps($w)}

sub ind # prima apparizione di f in w a partire da pos
   {my ($f,$w,$pos)=@_; $pos=1 if not defined $pos;
   index($$w,$$f,$pos-1)+1}

sub lun # lunghezza
   {my $w=shift; length($$w)}

sub pacmass # pacchetti massimali di w
   {my $w=shift; my $a=n("+"); my @pm=();
   while (not $a→suffchiuso)
   {$a=$a→d($w); push(@pm,$a)} @pm}

sub pd # polo destro del fattore semplice f in w
   {my ($f,$w)=@_; $w=$f if not defined $w;
   my $m=$f→lun; my $s=$f/$m--;
   while ($s→rip($w)) {$s=$f/$m--.$s} $s}

sub ps # polo sinistro del fattore semplice f in w
   {my ($f,$w)=@_; $w=$f if not defined $w;
   my $m=1; my $p=$f/$m++;
   while ($p→rip($w)) {$p=$p.$f/$m++} $p}

sub rip # 1 se f e' fattore ripetuto di w
   {my ($f,$w)=@_; my $i=$f→ind($w);
   return 0 if $i==0; $i=$f→ind($w,$i+1);
   return 0 if $i==0; 1}
```

```
sub suffchiuso # 1 se f termina con -
   {my $w=shift; return 1 if substr($$w,-1) eq "-"; 0}

########################################
sub concat # concatenazione
   {my ($a,$b)=@_; n($$a.$$b)}

sub intd # interno destro
   {my $a=shift; $a/[2,$a→lun-1]}

sub ints # interno sinistro
   {my $a=shift; $a/[1,$a→lun-1]}

sub segmento # w(i,k)
   {my ($w,$ik)=@_; return n(substr($$w,$ik-1,1)) if not ref($ik);
   my ($i,$k)=($$ik[0],$$ik[1]); n(substr($$w,$i-1,$k))}

sub segmentoinv # w(n-i+1,k)
   {my ($w,$ik)=@_; return n(substr($$w,-$ik)) if not ref($ik);
   my ($i,$k)=($$ik[0],$$ik[1]); n(substr($$w,-$i,$k))}

sub st # stampa sullo schermo
   {my $w=shift; print "$$w\n"}
```

BIBLIOGRAFIA

S. Altschul: Sequence comparison and alignment.
In Bishop/Rawlings 1997, 137-167.

A. Apostolico/J. Hein(ed.): CPM97. Springer LN CS 1264 (1997).

R. Arratia/D. Martin/G. Reinert/M. Waterman: Poisson process approximation for sequence repeats, and sequencing by hybridization.
J. Comp. Biol. 3 (1996), 425-463.

M. Attimonelli/G. Pesole/E. Quagliarello/G. Saccone: Principi di bioinformatica. Gnocchi 1997.

M. Bishop/C. Rawlings(ed.): DNA and protein sequence analysis.
Oxford UP 1997.

J. Breckow/R. Greinert: Biophysik. De Gruyter 1994.

A. Carpi/A. de Luca: Words and repeated factors. Sém. Lothar. 42 (1988).

A. Carpi/A. de Luca: Words and special factors.
Theor. Comp. Sci. 259 (2001), 145-182.

M. Crochemore/C. Hancart/T. Lecroq: Algorithmique du texte.
Vuibert 2001.

M. Crochemore/R. Vérin: On compact directed acyclic word graphs.
In Mycieski/Rozenberg/Salomaa 1997, 192-211.

M. Crochemore/R. Vérin: Direct construction of compact directed acyclic word graphs. In Apostolico/Hein 1997, 116-129.

A. de Luca: On the combinatorics of finite words.
Theor. Comp. Sci. 218 (1999), 13-39.

A. de Luca/S. Varricchio: Finiteness and regularity in semigroups and formal languages. Springer 1999.

J. Epplen/O. Riess: Repetitive sequences in DNA.
In Bishop/Rawlings 1997, 185-195.

Furtwangen: Materiali dal corso di bioinformatica. Settembre 2001.

C. Gibas/P. Jambeck: Developing bioinformatics computer skills.
O'Reilly 2001.

D. Gusfield: Algorithms on strings, trees, and sequences. Cambridge UP 1999.

H. Lodish e.a.: Molecular cell biology. Freeman 2001.

J. Mycieski/G. Rozenberg/A. Salomaa (ed.): Structures in logic and computer science. Springer LN CS 1261 (1997).

C. Notredame: Utilisation des algorithmes génétiques pour l'analyse de séquences biologiques. Thèse Univ. Paul Sabatier 1998.

R. Parsons/S. Forrest/ C. Burks: Genetic algorithms operators, and DNA fragment assembly. Machine Learning 21/1-2 (1995), 11-33

E. Passarge: Taschenatlas der Genetik. Thieme 1994.

P. Pevzner: Computational molecular biology. MIT Press 2000.

M. Raffinot: On maximal repeats in strings.
Inf. Processing Letters 80 (2001), 165-169.

D. Sankoff/J. Kruskal (ed.): Time warps, string edits, and macromolecules. The theory and practice of sequence comparison. Addison-Wesley 1983.

J. Setubal/J. Meidanis: Introduction to computational biology. PWS 1997.

J. Sim/C. Iliopoulos/K. Park/ W. Smyth: Approximate periods of strings. Theor. Comp. Sci. 262 (2001), 557-568.

W. Smyth: Repetitive perhaps, but certainly not boring.
Theor. Comp. Sci. 249 (2000), 343-355.

F. Smillie/W. Bains: Repetition structure of mammalian nuclear DNA.
J. Theor. Biol. 142 (1990), 463-471.

J. Tisdall: Beginning Perl for bioinformatics. O'Reilly 2001.

R. Vérin: Algorithmes de récherche de motifs dans les séquences d'ADN.
Thèse Univ. Marne La Vallée 1998.

L. Wall/T. Christiansen/J. Orwant: Programming Perl. O'Reilly 2000.

M. Waterman: Introduction to computational biology.
Chapman & Hall 1996.