Fonaments computacionals de la Bioinformàtica

Departament de Ciències Experimentals
i de la Salut
Universitat Pompeu Fabra
Curs 2008/2009

Robert Castelo
robert.castelo@upf.edu

Tema 1

Alfabets, paraules i llenguatges

1.1 Alfabets i paraules

Definició 1.1 *Alfabet*
Conjunt finit d'elements no buit, com ara

$$\Sigma = \{a, b, c, \ldots, z\},$$

on els elements s'anomenen *simbols* o *lletres*

Un exemple d'alfabet podrien ser les quatre lletres que utilitzem en una seqüència d'ADN,

$$\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{T}, \mathrm{G}\}.$$

Definició 1.2 *Paraula*
Seqüència finita de símbols d'un alfabet.

Per exemple, la següent paraula w estaria formada a partir de de l'alfabet donat com a exemple anteriorment,

$$w = \mathrm{ACCTGT}.$$

Definició 1.3 *Paraula buida*
Paraula que no té cap símbol, la denotarem amb la lletra grega λ.

Definició 1.4 *Concatenació de paraules*
La concatenació de dues paraules w i w', denotada per ww', correspon a la paraula formada pels símbols de la paraula w seguits dels símbols de la paraula w'.

Per exemple, si $w = \mathrm{ATTC}$ i $w' = \mathrm{CTCA}$, la seva concatenació serà:

$$ww' = \mathrm{ATTCGTCA}.$$

Definició 1.5 *Longitut d'una paraula*
Nombre de símbols d'una paraula. Ho denotarem amb dues barres verticals als costats de la paraula, com ara $|w|$.

A l'exemple anterior per la paraula $ww' = \mathrm{ATTCGTCA}$,

$$|ww'| = 8.$$

Definició 1.6 *Revessat d'una paraula*
El revessat d'una paraula $w = a_1 a_2 \ldots a_n$, denotat per w^R, és la paraula formada per la seqüencia de símbols de w en ordre invertit $w^R = a_n a_{n-1} \ldots a_1$, o mes formalment:

1. w si $w = \lambda$.
2. $v^R a$ si $w = av$, on $a \in \Sigma$ i v és una paraula.

Per exemple, el revessat de $w = \mathrm{ACT}$ és

$$w^R = (\mathrm{CT})^R \mathrm{A} = T^R \mathrm{CA} = \lambda^R \mathrm{TCA} = \lambda \mathrm{TCA} = \mathrm{TCA}.$$

Definició 1.7 *Palíndrom*
Anomenarem palíndrom a aquelles paraules w tal que

$$w = w^R.$$

Definició 1.8 *Potència d'una paraula*
La potència d'una paraula w es defineix com:

1. $w^0 = \lambda$.
2. $w^{n+1} = ww^n$.

Per exemple, per una paraula $w = \mathrm{ACCT}$,

$$w^0 = \lambda, \quad w^1 = \mathrm{ACCT}, \quad w^2 = \mathrm{ACCTACCT}.$$

Definició 1.9 *Subparaula*
Una paraula w és subparaula de una paraula v si[1]

$$\exists\, x,y \text{ tal que } v = xwy.$$

1.2 Llenguatges

Definició 1.10 *Llenguatge*
Conjunt de paraules sobre un alfabet.

Per exemple, donat un alfabet $\Sigma = \{\mathrm{A},\mathrm{C},\mathrm{T},\mathrm{G}\}$, un possible llenguatge seria:

$$L = \{A, CT, AG, ACTTG\}.$$

Definició 1.11 *Concatenació de llenguatges*
La concatenació de dos llenguatges L_1 i L_2, denotada per L_1L_2, correspon al llenguatge:

$$L_1L_2 = \{w_1w_2 \,|\, w_1 \in L_1 \wedge w_2 \in L_2\}.$$

Per exemple, donats dos llenguatges

$$L_1 = \{\mathrm{A},\mathrm{CT},\mathrm{AG}\} \quad L_2 = \{\mathrm{G},\mathrm{T},\mathrm{C}\}.$$

la seva concatenació és

$$L_1L_2 = \{\mathrm{AG},\mathrm{AT},\mathrm{AC},\mathrm{CTG},\mathrm{CTT},\mathrm{CTC},\mathrm{AGG},\mathrm{AGT},\mathrm{AGC}\}.$$

Definició 1.12 *Potència d'un llenguatge*
Donat un llenguatge L la i-èssima potència d'un llenguatge es defineix com:

1. $L^0 = \{\lambda\}$.
2. $L^i = LL^{i-1}$.

Per exemple, per un llenguatge $L = \{\mathrm{AT},\mathrm{GC}\}$,

$$L^0 = \{\lambda\},\ L^1 = L = \{\mathrm{AT},\mathrm{GC}\},\ L^2 = \{\mathrm{ATAT},\mathrm{ATGC},\mathrm{GCAT},\mathrm{GCGC}\}.$$

Definició 1.13 *Clausura positiva d'un llenguatge*
Donat un llenguatge L la seva clausura positiva, denotada per L^+ es defineix com

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

[1] El símbol matemàtic $\exists$ es llegeix com *existeix*.

Figura 1.1: Reconeixedor de llenguatges.

Definició 1.14 *Estrella de Kleene d'un llenguatge*
Donat un llenguatge L la seva estrella (o també, clausura) de Kleene, denotada per L^* es defineix com

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup \bigcup_{i=1}^{\infty} L^i = \{\lambda\} \cup L^+.$$

Definició 1.15 Llenguatge universal
El llenguatge universal, denotat per Σ^*, és el llenguatge que correspon a *totes* les paraules que es poden formar a partir d'un alfabet Σ, es a dir[2]:

1. $\lambda \in \Sigma^*$.
2. si $w \in \Sigma^*$ llavors $aw \in \Sigma^*, \forall a \in \Sigma$.
3. no hi ha mes paraules que les generades per (1) i (2).

1.3 Reconeixedors de llenguatges

Un reconeixedor de llenguatges és un dispositiu que ens diu si una paraula donada w pertany, o no a un determinat llenguatge L. La Figura 1.1 ens mostra aquest concepte gràficament.
Exemples de reconeixedors de llenguatges són:

- Autòmats finits.
- Màquines de Turing.

Definició 1.16 *Autòmat finit determinista (AFD/DFA)*
Un autòmat finit determinista (AFD/DFA) és un vector de 5 elements:

$$M = (Q, \Sigma, \delta, q_0, q_F) \quad \text{on}$$

[2] El símbol matemàtic $\forall$ es llegeix com *per a tot*.

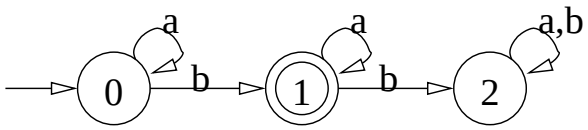


Figura 1.2: Exemple d'un AFD que receoneix paraules sobre un alfabet $\Sigma = \{a, b\}$ que contenen una única lletra b.

- Q és un conjunt d'estats.
- Σ és un alfabet de símbols.
- $\delta : Q \times \Sigma \to Q$ és una funció de transició que donat un estat i un símbol ens retorna un nou estat.
- q_0 estat inicial.
- q_F estat final[3].

Definició 1.17 *Configuració de l'autòmat*
Anomenem una configuració de l'autòmat al parell (q, w) on $q \in Q$ és un estat i $w \in \Sigma^*$ és una paraula.

Es diu que una paraula w es *acceptada*, o *reconeguda*, per M si a partir de la configuració (q_0, w) arribem a la configuració (q_F, λ).

Els AFD's reconeixen els anomenats *llenguatges regulars*. Per exemple, construïm un AFD que reconegui el llenguatge regular format per les paraules sobre l'alfabet $\Sigma = \{a, b\}$ que tenen una única b. Necessitem 3 estats $Q = \{0, 1, 2\}$ on l'estat inicial $q_0 = 0$ i l'estat final $q_F = 2$. La funció de transició $\delta : Q \times \Sigma \to Q$ seria la següent:

δ	a	b
0	0	1
1	1	2
2	2	2

[3]De fet pot haver mes d'un estat final, però per simplicitat ilustrarem aqui el concepte d'AFD amb un únic estat final.

Una forma més intuitiva de representar l'AFD es mitjançant un graf dirigit com el de la Figura 1.2. En aquesta representació gràfica, el conjunt d'estats és $Q = \{0, 1, 2\}$ i els representem dins un cercle. Les fletxes entre els cercles representen les transicions entre estats. Cada fletxa te un o més símbols associats que corresponen als símbols que porten a la transició d'estat corresponent. D'aquesta forma podem representar gràficament la funció de transició δ. L'estat inicial està indicat amb una fletxa que l'apunta i que no surt de cap altre estat, en aquest cas doncs $q_0 = 0$. L'estat final es representa amb un doble cercle, en aquest cas doncs $q_F = 1$.

Per tal de veure si la paraula *abaaa* seria reconeguda per l'AFD anterior començarem per la configuració inicial $(0, abaaa)$, i hem de veure si arribem a la configuració final $(1, \lambda)$, de la següent forma:

$$(0, abaaa)$$

$$(0, baaa)$$

$$(1, aaa)$$

$$(1, aaa)$$

$$(1, aa)$$

$$(1, a)$$

$$(1, \lambda)$$

Per tant, la paraula *abaaa* és reconeguda per l'AFD. Ara fem el mateix per la paraula *abab*:

$$(0, abab)$$

$$(0, bab)$$

$$(1, ab)$$

$$(1, b)$$

$$(2, \lambda)$$

Per tant, la paraula *abab* **no** és reconeguda per l'AFD.

1.4 Expressions Regulars

S'anomena expressió regular sobre un alfabet Σ a tota expressió que satisfà la definició recursiva següent:

1. $\emptyset, \lambda$ són expressions regulars.
2. a és una expressió regular, $\forall a \in \Sigma$.
3. Si E_1 i E_2 són expressions regulars, aleshores $E_1 + E_2$ i E_1E_2 són expressions regulars.
4. Si E és una expressió regular, aleshores E^* es una expressió regular.

Definició 1.18 *Llenguatge associat a una expressió regular*
El llenguatge associat a una expressió regular, denotat per $L(E)$, és el llenguatge definit pel següent conjunt de regles:

1. $L(\emptyset) = \emptyset$.
2. $L(\lambda) = \{\lambda\}$.
3. $L(a) = \{a\}$.
4. $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
5. $L(E_1E_2) = L(E_1)L(E_2)$.
6. $L(E^*) = L(E)^*$.

Per exemple, considerem l'alfabet $\Sigma = \{\mathrm{A},\mathrm{C},\mathrm{T},\mathrm{G}\}$, llavors

$$\lambda \quad \mathrm{A} \quad \mathrm{A}+\mathrm{C}+\mathrm{TG} \quad \text{ó} \quad (\mathrm{A}+\mathrm{C}+\mathrm{T}+\mathrm{G})^*,$$

són expressions regulars.

Per tal de trobar el llenguatge associat a una expressió regular, s'han utilitzar les regles especificades a la Definició 1.18. Per exemple, quin és el llenguatge associat a l'expressió regular $(\mathrm{A}+\mathrm{C}+\mathrm{T}+\mathrm{G})^*$?

$$L((\mathrm{A}+\mathrm{C}+\mathrm{T}+\mathrm{G})^*) = (L(\mathrm{A}+\mathrm{C}+\mathrm{T}+\mathrm{G}))^* = \{\mathrm{A},\mathrm{C},\mathrm{T},\mathrm{G}\}^* = \Sigma^*,$$

es a dir, el llenguatge universal sobre l'alfabet Σ. Quin és el llenguatge associat a l'expressió regular $(b+ba)^*$?

$$L((b+ba)^*) = (L(b+ba))^* = \{b,ba\}^* =$$

$$= \{w \,|\, w \text{ comenca per } b \text{ i no te dues } a \text{ consecutives}\}.$$

El llenguatge associat a una expressió regular és un llenguatge regular i per tant el podem reconéixer amb un AFD. Les expressions regulars s'utilitzen a la Biologia, per exemple, per al reconeixement de patrons de consens.[4]

1.5 Les sintaxis de les expressions regulars

La sintaxi que hem descrit a les seccions anteriors, referent a l'us de llenguatges i expressions regulars, no correspon exactament a la sintaxi que s'utilitza amb els programes que manipulen expressions regulars, com ara la comanda `grep` al sistema operatiu Unix. Una de les raons perquè aquesta sintaxi sigui diferent és, per exemple, la impossibilitat d'escriure amb codi ASCII[5] un símbol elevat a una potència tal i com ho fem en llenguatge matemàtic.

Aquest es un petit resum de la sintaxi que utilitza la comanda `grep` del Unix i el llenguatge de programació Perl (els exemples estan construïts sobre un alfabet format pel conjunt de símbols de l'ADN A,C,G,T):

Operació	**Sintaxi `grep` (exemple)**	**Sintaxi matemàtica (exemple)**
Conjunció de símbols	AC	AC
Disjunció de símbols	$[ACGT]$	$(A+C+G+T)$
Clausura positiva	$A+$ $[AC]+$	A^+ $(A+C)^+$
Estrella de Kleene	$A*$ $[AC]*$	A^* $(A+C)^*$
Disjunció d'expressions regulars	$AA\|TG$ $(AC\|TT)+$	$AA+TG$ $(AC+TT)^+$

A mes a mes, la comanda `grep` del Unix, o el llenguatge de programació Perl, incorporen sintaxi addicional per poder representar la paraula buida λ en diverses circumstancies, alhora que permet utilitzar una sintaxi mes econòmica per a determinades expressions regulars:

[4] Com ara a `http://www.expasy.ch/prosite`.

[5] El codi ASCII és el conjunt estandaritzat de 256 símbols que utilitzen generalment els ordinadors per intercanviar informació. Un símbol ASCII està codificat per 8 bits ($2^8 = 256$).

Sintaxi	Descripció	Exemple	Exemple (Sn. mat.)
$\backslash <$	paraula buida concatenada al principi	$\backslash < AAT$	λAAT
$\backslash >$	paraula buida concatenada al final	$AAT \backslash >$	$AAT\lambda$
$[s_1 - s_2]$	un de tots els símbols entre s_1 i s_2	$[A-C]$	$(A+B+C)$
$[\hat{}s_1 \ldots s_n]$	un símbol diferent de $s_1, s_2, \ldots, s_n$	$[\hat{}ACTG]$	$(B+D+E+\ldots)$

1.6 Exercicis

Exercici 1.1 (Avaluació Formativa, Febrer 2002)
Donades les següents 6 seqüències d'ADN:

```
(1) ATGTCTCAGG
(2) GGCGCTATGC
(3) GCTGACGATC
(4) GATCTTTCTC
(5) CCGAGACGGA
(6) GGCGACGACT
```

Assumint que estem utilitzant la sintaxi de la comanda `grep` del Unix:

(a) Quines seqüències encaixen amb l'expressió regular:

$$\backslash < [ACTG]*CTTTC[ACTG]* \backslash >$$

(b) Quines seqüències encaixen amb l'expressió regular:

$$\backslash < [ACG]* \backslash >$$

(c) Quines seqüències encaixen amb l'expressió regular:

$$\backslash < (TAT|TAC)+ \backslash >$$

(d) Doneu una expressió regular que només encaixi amb les seqüències 2 i 6 des del seu començat fins al final, i que no sigui una disjunció d'ambdues, es a dir, (...|...).

Exercici 1.2 (Examen de Setembre, 2002)
Considereu un alfabet format pels símbols A, C, G, T:

(a) Considereu la paraula buida λ i la paraula $w = TATA$. Quina serà la longitut de la paraula resultant de la concatenació λw?

(b) Donada l'expressió regular $(A+C+G+T)^*$, escriviu una paraula generada a partir d'aquesta expressió regular tal que la seva longitut sigui la més petita possible.

(c) Escriviu una expressió regular que reconegui paraules que contenen la subparaula *TATA* un o més cops des del començament fins al final.

Exercici 1.3 (PEM, 2002)
Donada la seqüència d'ADN `ATTGAT`, quina de les següents expressions regulars **genera** aquesta seqüència (assumim sintaxi `grep`):

(a) $[TG]*$
(b) $A[TG]AT$
(c) $[ATG]+$
(d) $AT[TG]+T$
(e) $(ATT|GAT)[GA]*$

Exercici 1.4 (PEM, 2002)
Donada la seqüència d'ADN `ACTGATCG`, quina de les següents expressions regulars **no genera** aquesta seqüència (assumim sintaxi `grep`):

(a) $[ACGT]*$
(b) $[ACT]+[ACGT]+$
(c) $ACTGATCG$
(d) $AC[AGT]*[CG]+$
(e) $A[TGA]+CG$

Exercici 1.5 (PEM, 2002)
Donada l'expressió regular $(A+C)^*(T+G)^+$, quina, o quines, de les següents seqüències d'ADN **pertany(en)** al llenguatge regular generat per l'expressió donada?

(a) AT
(b) TG
(c) les dues anteriors
(d) AC
(e) totes les anteriors

Exercici 1.6 (PEM, 2002)
Donada l'expressió regular $(A+C+T)^+$, quina de les següents seqüències d'ADN **no pertany** al llenguatge regular generat per l'expressió donada?

(a) λ (paraula buida)

(b) *AATC*
(c) *CTA*
(d) *ATATAT*
(e) *CCCCC*

Exercici 1.7 (Avaluació Formativa, Febrer 2003)
Donat un conjunt de seqüències de proteïnes, escriviu una expressió regular que encaixi amb aquelles seqüències que, o be comencen amb els aminoàcids `MALD`, o be acaben amb els aminoàcids `DD`.

Exercici 1.8 (Avaluació Formativa, Febrer 2003)
Donat un conjunt de seqüències d'ADN corresponents a introns humans, escriviu una expressió regular que encaixi amb aquelles seqüències que acaben amb una o més bases que no siguin `A` ni `T`, seguides d'un o més dinucleòtids `CT`, seguits finalment del dinucleòtid `AG`.

Exercici 1.9 (PEM, 2003)
Donada la seqüència d'ADN `CTCTTCTC`, quina de les següents expressions regulars **no la pot generar** (assumim sintaxi `grep`):

(a) $[ACGT]*$
(b) $[ACGT]+$
(c) $(CT|TC)+$
(d) $\backslash <(CT)+\backslash >$
(e) $\backslash <[CT]*\backslash >$

Exercici 1.10 (PEM, 2003)
Donada la seqüència d'ADN `TTTCCTTTTTT`, quina de les següents expressions regulars **la pot generar** (assumim sintaxi `grep`):

(a) $T[TC]CCT+$
(b) $T+C*T*$
(c) $[AC]+$
(d) $(CT)+$
(e) $(TC)+$

Exercici 1.11 (PEM, 2003)
Donada l'expressió regular C^*TA^*, quina de les següents seqüències d'ADN **pertany** al llenguatge regular generat per l'expressió donada?

(a) *AA*

(b) *TT*
(c) *T*
(d) *CA*
(e) *CC*

Exercici 1.12 (PEM, 2003)
Donada l'expressió regular $(C+T)^+$, quin/es de les següents seqüències d'ADN **no pertany/en** al llenguatge regular generat per l'expressió donada?

(a) *CCC*
(b) *TTT*
(c) Les dues anteriors
(d) $\lambda\lambda$
(e) Totes les anteriors

Exercici 1.13 (PEM, 2003)
Senyaleu quina de les següents afirmacions és correcta:

(a) La paraula buida és una paraula formada per 0 símbols.
(b) La paraula buida és una paraula formada per 0 ó 1 símbols.
(c) La paraula buida no es pot concatenar amb cap paraula.
(d) La paraula buida és una paraula grega.
(e) La paraula buida no existeix.

Exercici 1.14 (PEM, 2004)
Donada l'expressió regular $(A+C+G+T)^*(AG+AC)\lambda$, quina de les següents seqüències d'ADN **no pertany** al llenguatge regular generat per l'expressió donada?

(a) *ACGCTCGAG*
(b) *GCGCGCAG*
(c) *AAAAAC*
(d) *AAAAGG*
(e) *TCGCTGCAC*

Exercici 1.15
Escriviu una expressió regular sobre l'alfabet dels aminoàcids que encaixi amb seqüències de proteïnes que contenen cinc o més Glutamines (Q) seguides.

Exercici 1.16
Utilitzant la sintaxi de la comanda del Unix `grep` (vegueu Secció 1.5) escriviu una

expressió regular sobre l'alfabet dels aminoàcids que encaixi amb seqüències de proteïnes que contenen cinc o més Glutamines (Q) seguides on un dels 3 aminoàcids del mig poguès ser qualsevol aminoàcid.

Exercici 1.17 (PEM, 2005)
Quina de les següents expressions regulars ens permetria seleccionar seqüències de proteïnes que tinguessin el domini N-terminal `MTPHRLLPPL`, i no seleccionar cap altra proteïna:

(a) $[MTPHRL]+$
(b) $MTPHRLLPPL\backslash >$
(c) $\backslash < MTPHR$
(d) $\backslash < MTPHRLLPPL$
(e) $[MTPHRL]*$

Exercici 1.18 (PEM, 2005)
Donada l'expressió regular `[CT]*[AG]+`, quina(es) de les següents seqüències d'ADN **pertany(en)** al llenguatge regular generat per l'expressió donada ?

(a) CA
(b) GA
(c) les dues anteriors
(d) CT
(e) totes les anteriors

Exercici 1.19 (PEM, 2006)
Tenim una proteina que anomenarem P amb un domini d'unió a ARN amb afinitat per seqüències riques en uridines (U) admetent eventualment un nucleòtid `A` entremig de la seqüència d'uridines en l'ARN. Donat un conjunt de seqüències precursores de l'ARN missatger, quina de les següents expressions regulars ens permetria seleccionar (encaixar amb) **només** aquelles que contenen un lloc d'unió a la proteina P? (assumim sintaxi `grep`)

(a) $U+A*U+$
(b) $U+A+U+$
(c) $U+|U+AU+$
(d) $U*A*U*$
(e) $U*|U*AU*$

Exercici 1.20 (PEM, 2006)
Tenim l'expressió regular `[CT]+`, quines de les següents seqüències d'ADN **no pertany** al llenguatge regular generat per l'expressió donada:

(a) $CCCCCCCC$
(b) $CTCTCTCT$
(c) $TCTCTCTC$
(d) $TTTTTTTT$
(e) λ

Exercici 1.21 (PEM, 2006)
Volem escriure una expressió regular que ens permeti seleccionar **només** (encaixar només amb) seqüències d'un o més dinucleòtids CT, quina de les següents seria correcta?

(a) $[CT]+$
(b) $(CT)+$
(c) $C+T+$
(d) $CTCT$
(e) $C*T*$

Exercici 1.22 (PEM, 2007)
Tenim l'expressió regular $U*AAU*$, quines de les següents seqüències d'ADN **pertany** al llenguatge regular generat per l'expressió donada?

(a) UAU
(b) UU
(c) $UAAAU$
(d) UA
(e) AA

Exercici 1.23 (PEM, 2007)
Donada la seqüència d'ADN `GCTTAG`, quina de les següents expressions regulars **genera** aquesta seqüència?

(a) $[CGT]+$
(b) $[CG]+T*AG$
(c) $G[CT]AG$
(d) $[CG]*T[AG]+$
(e) $G[CT]+G$

Exercici 1.24 (PEM, 2008)
Els pèptids anomenats en anglès *Tachykinins* són un grup de neuropèptids biològicament actius que exciten neurones, provoquen respostes de comportament i

són potents vasodilatadors. Concretament, els darrers cinc residus de l'extrem C-terminal, els quals estan conservats d'amfibis a mamífers i són essencials per l'activitat biològica d'aquests pèptids, són una Fenilalanina seguida de, o be una Isoleucina, o be una Valina, o be una Fenilalanina, o be una Tirosina, seguida d'una Glicina, seguida de, o be una Leucina, o be una Meteonina, seguida d'una Meteonina. Quina de les següents expressions regulars ens permetria recuperar els pèptids *Tachykinins* a partir de qualseol conjunt donat de pèptids?

(a) $[FIVY]+[GLM]+M$
(b) $\backslash <[FIVFY]+[GLM]+M$
(c) $F[IVFY]G[LM]M\backslash >$
(d) $\backslash <F[IVFY]G[LM]M$
(e) $F[IVFY]G[LM]M$

Solucions

Exercici 1.1: (a) seqüència 4, (b) seqüència 5, (c) cap, (d) una possible solució seria $\backslash <GGCG[ACGT]*\backslash >$

Exercici 1.2: (a) 4, (b) λ (la paraula buida), (c) $\lambda(A+C+G+T)^*TATA(A+C+G+T)^*\lambda$

Exercici 1.3: c

Exercici 1.4: e

Exercici 1.5: c

Exercici 1.6: a

Exercici 1.7: $\backslash <MALD|DD\backslash >$

Exercici 1.8: $[CG](CT)+AG\backslash >$

Exercici 1.9: d

Exercici 1.10: b

Exercici 1.11: c

Exercici 1.12: d

Exercici 1.13: a

Exercici 1.14: d

Exercici 1.15: QQQQQ

Exercici 1.16: $Q[A-Z]QQQ|QQ[A-Z]QQ|QQQ[A-Z]Q$

Exercici 1.17: d

Exercici 1.18: c

Exercici 1.19: c

Exercici 1.20: e

Exercici 1.21: b

Exercici 1.22: e

Exercici 1.23: b

Exercici 1.24: c

Tema 2

Introducció als algorismes

2.1 Noció d'algorisme

Definició 2.1 *Acció*
Una acció és un aconteixement que té lloc en un periode de temps finit i produeix un resultat ben definit i previst.

El fet de que una acció duri un temps finit vol dir que es pot trobar l'instant del temps en què comença i l'instant del temps en què acaba.

Definició 2.2 *Informació*
Per tal de poder entendre el resultat de l'algorisme, és necessari que durant el temps entre l'instant en què comença l'algorisme i l'instant en què acaba, poguéssim observar o obtenir informació sobre el que està passant.

Definició 2.3 *Estat*
Conjunt d'informació observat en un instant de temps donat, entre el principi i el final.

Definició 2.4 *Algorisme*
Un algorisme és una seqüència d'accions que ens porta d'un estat inicial a un estat final en el qual obtenim el resultat.

La construcció d'un algorisme consisteix en *descobrir* quines accions *elementals* cal *organitzar* en el temps i, també, consisteix en *escollir* la *forma* d'organitzar-les per tal d'obtenir el resultat. Això requereix:

- Capacitat d'*abstracció* a differents nivells, fent el que es coneix com *disseny descendent*.
- Utilització d'un *llenguatge* amb notació i sintaxi precises.
- Capacitat de formular i *analitzar* problemes.

Definició 2.5 *Programa*
És la implementació d'un algorisme en un llenguatge de programació determinat (per exemple el *Perl*).

2.2 Estructures algorísmiques bàsiques

Definició 2.6 *Constants*
Les constants són valors explícits que no varien en el curs d'execució de l'algorisme. També es coneixen com a *literals*.

Exemples de constants són:

```
1, 2, 10, 3.1416, "hola"
```

Definició 2.7 *Variable*
Una variable és un nom associat a un espai a la memòria de l'ordinador on s'emmagatzema una constant o be una estructura de dades.

Considerarem dues classes de variables: les variables *singulars* i les variables *plurals*. Les variables que anomenarem *singulars* seran aquelles que estan associades a un únic valor, mentre que les variables que anomenarem *plurals* seran aquelles que estan associades a dos o mes valors.

Ens podem referir a una variable escrivint simplement el seu nom, com ara x, v o i. No obstant, sovint utilitzarem la notació del llenguatge Perl que distingeix explícitament el caràcter singular o plural d'una variable, de la següent forma:

- variables *singulars*: escrivirem el símbol del Dólar `$` davant el nom de la variable, com ara `$x, $i` o `$n`.
- variables *plurals*: escrivirem el símbol de l'arroba `@` davant el nom de la variable, com ara `@v, @t` o `@h`.

Definició 2.8 *Assignació*
Direm que a una variable li *assignem* un valor quan especifiquem quin valor volem que prengui la variable. L'assignació l'especificarem amb l'operador = on a l'esquerra hi ficarem la variable a la qual li volem assignar el valor, i a la dreta hi ficarem el valor que ha de passar a ser el nou contingut de la variable. Per exemple:

```
a = 2
```

que ho llegirem com "a pren el valor 2".

En el llenguatge de programació Perl especificarem l'assignació anterior com `$a=2`. Altres exemples d'assignacions en llenguatge Perl són:

```
$a = 2 + 1
$a = $a + 1
```

Les assignacions prèvies les llegirem com: "a pren el valor 3", "a pren el valor a més 1". En aquests darrers exemples d'assignacions, el valor assignat era en realitat el resultat de l'avaluació d'una expressió.

Definició 2.9 *Expressió*
Anomenarem expressió a l'especificació sintàcticament correcta d'una o més operacions sobre un conjunt de variables, constants i funcions.

Podrem especificar operacions aritmètiques en llenguatge Perl mitjançant el següent conjunt d'operadors aritmètics:

Operador	Exemple	Resultat
Suma	`$a + $b`	suma de `a` i `b`
Resta	`$a - $b`	resta de `a` menys `b`
Multiplicació	`$a * $b`	producte de `a` i `b`
Divisió (quocient)	`$a / $b`	divisió de `a` per `b`
Divisió (residu, mòdul)	`$a % $b`	residu de dividir `a` per `b`

Exemples d'expressions són:

```
1+1
$a+2
3*log($a)
($a+$b)*2
```


Les *expressions regulars* són un tipus no aritmètic d'expressió on malgrat alguns dels operadors poden coincidir amb els aritmètics la seva semàntica es completament different. Penseu, per exemple, en el que fa l'operador especificat amb un asterisc "*" a una expressió aritmetica i a una expressió regular.

Un altre tipus d'expressió que haurem d'utilitzar sovint, són les expressions *lògiques*. L'avaluació d'una expressió lògica comporta com a resultat un valor de veritat: *cert* o *fals*. Les expressions lògiques les anomenarem també *condicions* i les construirem mitjançant els següents operadors de comparació (sintaxi Perl):

Comparació	Numèrica	Alfabètica
Igual	`==`	`eq`
No igual	`!=`	`ne`
Més petit que	`<`	`lt`
Més gran que	`>`	`gt`
Més petit o igual que	`<=`	`le`
Més gran o igual que	`>=`	`ge`

Exemples d'expressions lògiques formades utilitzant els operadors de comparació anteriors són:

```
$a == 2

$z eq 'b'

$x != 3.5

$j >= 10

$d ne ' '
```

Podem construïr expressions lògiques més complexes utilitzant a mes a mes operadors lògics, que són els següents (sintaxi Perl):

Operador	Significat
`expr1 && expr2`	conjunció `expr1 i expr2`
`expr1 || expr2`	disjunció `expr1 ó expr2`
`!expr`	negació `no expr`

on `expr`, `expr1` i `expr2` fan referència a expressions lògiques. Al igual que amb les expressions aritmètiques podem imbricar expressions lògiques utilitzant els parèntesis, "(" i ")".

Definició 2.10 *Composició seqüencial*
Anomenarem composició seqüencial a un conjunt d'accions que s'executen incondicionalment seguint un ordre pre-establert entre elles.

Un exemple de composició seqüencial podria ser el següent:

```
$a = 2;
$b = $a;
$b = $b + $a * 4;
print $b;
```

On hem utilitzat la notació del llenguatge de programació Perl per especificar variables que emmagatzemen un únic valor (ficant el símbol del `$` davant del nom de la variable) i delimitar el final de cada acció amb un punt i coma ";". Quin valor imprimirà l'algorisme anterior?

Definició 2.11 *Composició alternativa (o condicional)*
Anomenarem composició alternativa (o condicional) a dos conjunts d'accions dels quals nomes un d'ells s'executa depenent del valor de veritat de la condició especificada.

Un exemple de composició alternativa podria ser el següent:[1]

```
$a = 4;
$b = 7;
$c = ($a * $b) + 1;

if ($a % 2 == 0) {
  print "parell\n";
}
else {
  print "senar\n";
}
```

Què imprimirà l'algorisme anterior, "parell" o "senar" ?

Definició 2.12 *Composició iterativa*
Anomenarem composició iterativa a un conjunt d'accions que s'executa repetidament fins que la condició especificada pren un valor de veritat determinat.

[1] Fixeu-vos que la composició alternativa forma part d'una composició seqüencial.

Un exemple de composició iterativa podria ser el següent:

```
$i = 0;
$s = 0;
while ($i < 10) {
  $i = $i + 1;
  $s = $s + $i;
}
print $s;
```

Tota composició iterativa compleix una condició inicial (abans de començar a *iterar*) i una condició final (quan ha acabat d'*iterar*). A mes a mes, tota composició iterativa compleix una condicio general que no varia (per això també s'anomena *invariant*) durant l'execució de la composició iterativa.
Per exemple, a l'algorisme anterior la condició inicial es que les variables `i` i `s` tenen el valor 0. I la condició general que no varia (l'invariant) es que en tot moment la variable `s` pren el valor de la variable `s` més el valor de la variable `i`.

Quina és la condició final? Què fa l'algorisme anterior ?

2.3 Exercicis

Exercici 2.1 Feu un programa en Perl que sumi tots els nombres parells i negatius entre -200 i -100, ambdos inclosos. Penseu previament les condicions inicial, final i l'invariant de l'algorisme.

Exercici 2.2 (Examen de Setembre, 2002)
Feu un programa en Perl que compti quants nombres sencers entre 1 i 100 són divisibles per 3.

Exercici 2.3 Feu un programa en Perl que calculi el factorial d'un nombre sencer no-negatiu donat i enregistrat en una variable `$n`. Penseu previament les condicions inicial, final i l'invariant de l'algorisme.

Exercici 2.4 (PEM, 2002)
Considereu el següent programa escrit en Perl:

```
$x = 1;
$i = 0;
while ($i < 3) {
```

```
    $x = $x * 2;
    $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 5
(b) 3
(c) 1
(d) 8
(e) 2

Exercici 2.5 Quines són les condicions inicial i final, i quin és l'invariant al programa de l'Exercici 2.4?

Exercici 2.6 Feu un programa en Perl que calculi el resultat d'elevar un nombre sencer positiu `$b` (base) a un altre nombre sencer no-negatiu `$e` (exponent). Penseu previament les condicions inicial, final i l'invariant de l'algorisme.

Exercici 2.7 (Assaig, 2002)
Direm que un nombre sencer positiu és perfecte si és igual a la suma de tots els seus divisors excepte ell mateix. Per exemple, el 6 és perfecte ja que els seus divisors són 1, 2 i 3; el 28 és perfecte ja que els seus divisors són 1, 2, 4, 7 i 14. Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui si és perfecte o no (mostrant algun tipus de missatge amb la instrucció `print`).

Exercici 2.8 (Avaluació Formativa, Febrer 2003)
Un nombre sencer i positiu és primer si, i només si, els seus únics divisors són 1 i ell mateix. Per exemple, el 5 és divisible per 1 i 5 però no per 2, 3 o 4, per tant és primer. En canvi, el 6, a més a més del 1 i del 6, també és divisible per 3 i per 2, per tant no és un nombre primer. Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui si és primer o no (mostrant algun tipus de missatge amb la instrucció `print`).

Exercici 2.9 (PEM, 2003)
Considereu el següent programa escrit en Perl:

```
$x = 0;
$i = 0;
while ($i < 4) {
```



```
    $x = $x * 2;
    $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 2
(b) 8
(c) 16
(d) 4
(e) 0

Exercici 2.10 (Assaig, 2003)
Direm que dos nombres sencers positius són **relativament primers** si l'únic divisor que tenen en comú és el 1. Per exemple, els divisors del 4 són el 1, el 2 i el 4, mentre que els divisors del 6 són el 1, el 2, el 3 i el 6. Com que el 2 és un divisor comú a tots dos, el 4 i el 6 **no són** relativament primers. Els divisors del 9 són el 1, el 3 i el 9, per tant el 4 i el 9 **sí són** relativament primers. Feu un programa en Perl que, donat dos nombres sencers positius enregistrats en dues variables `$x` i `$y`, respectivament, ens digui si els nombres que hi ha a `$x` i `$y` són relativament primers o no (mostrant algun tipus de missatge amb la instrucció `print`).

Exercici 2.11 (Avaluació Formativa, Febrer 2004)
Direm que dos nombres sencers positius són **amics** si tenen el mateix quocient al dividir, per a cadascun d'ells, la suma dels seus divisors per ell mateix. Per exemple, els divisors del 6 són el 1, 2, 3, i el 6, i per tant sumen 12; mentre que els divisors del 28 són el 1, 2, 4, 7, 14 i 28, i per tant sumen 56; com que el quocient de dividir la suma dels divisors de 6 per ell mateix, 12/6 = 2, es igual al quocient de dividir la suma dels divisors de 28 per ell mateix, 56/28 = 2, el 6 i el 28 son **amics**. Si considerem el 12, els seus divisors són el 1, 2, 3, 4, 6, i el 12, que sumen 28, i per tant 28/12=14/6=7/3 que al no ser 2, el 12 no pot ser amic ni del 6 ni del 28.

Feu un programa en Perl que, donats dos nombres sencers positius enregistrats en dues variables `$x` i `$y`, respectivament, ens digui si els nombres que hi ha a `$x` i `$y` són amics o no (mostrant algun tipus de missatge amb la instrucció `print`).

Exercici 2.12 (PEM, 2004)
Considereu el següent programa escrit en Perl:

```
$n = 0;
$i = 0;
```

```
while ($i < 3) {
  $j = 2;
  while ($j >= 0) {
    $k = 0;
    while ($k < $i * $j) {
      $n = $n + 1;
      $k = $k + 1;
    }
    $j = $j - 1;
  }
  $i = $i + 1;
}
```

Quin serà el valor de la variable `$n` quan s'acabi d'executar el programa?

(a) 3
(b) 6
(c) 2
(d) 36
(e) 9

Exercici 2.13 (PEM, 2004)
L'operador aritmètic del residu d'una divisió s'escriu en Perl amb el símbol del tant per cent %. Direm que un nombre `$x` és divisible per un altre `$y` si el residu de dividir `$x` per `$y` és 0. Considereu el següent programa escrit en Perl on assumim que tenim un nombre sencer i positiu enregistrat en una variable `$x`:

```
$i = 2;
$n = 2;

while ($i < $x) {

  if ($x % $i == 0) {
    $n = $n + 1;
  }

  $i = $i + 1;
}

if ($n > 2) {
```



```
  print "no es primer\n";
} else {
  print "es primer\n";
}
```

Què fa aquest programa?

(a) Comprova si `$x` és divisible per algun nombre sencer entre 2 i `$x`-1 (ambdòs inclosos), i mostra el missatge "no" si és divisible entre algun d'aquests nombres.
(b) Comprova si `$x` és un nombre primer i mostra el missatge "si", si ho és.
(c) Les dues coses anteriors
(d) Comprova si `$x` és un nombre perfecte (es a dir, divisible entre la suma de tots els seus divisors excepte ell mateix) i mostra el missatge "si", si ho és.
(e) Totes les coses anteriors.

Exercici 2.14 (PEM, 2005)
Considereu el següent programa escrit en Perl:

```
$x = 1;
$i = 0;
while ($i < 3) {
  $x = $x + $i;
  $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 0
(b) 4
(c) 3
(d) 6
(e) 8

Exercici 2.15 (PEM, 2005)
Considereu el següent programa escrit en Perl:

```
$x = 0;
$i = 1;
while ($i <= 10) {
  if ($i > 4 && $i <= 8) {
    $x = $x + $i;
```

```
      $i = $i + 1;
    }
    $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 12
(b) 26
(c) 30
(d) 18
(e) 22

Exercici 2.16 (PEM, 2005)
L'operador aritmètic del residu d'una divisió s'escriu en Perl amb el símbol del tant per cent %. Direm que un nombre `$x` és divisible per un altre `$y` si el residu de dividir `$x` per `$y` és 0. Considereu el següent programa escrit en Perl on assumim que tenim un nombre sencer i positiu enregistrat en una variable `$x`:

```
$i = 1;
$s = 0;
while ($i <= $x) {
  if ($x % $i == 0) {
    $s = $s + $i;
  }
  $i = $i + 1;
}

if ($s % $x == 0) {
  print "si\n";
} else {
  print "no\n";
}
```

Assenyaleu per a quins dels següents valors de `$x` el programa anterior mostrarà el missatge "si":

(a) 6
(b) 28
(c) els dos valors anteriors
(d) 1
(e) tots els valors anteriors

Exercici 2.17 (Avaluació Formativa, 2005)
Direm que dos nombres sencers positius són **amics** si tenen el mateix quocient al dividir, per a cadascun d'ells, la suma dels seus divisors per ell mateix. Malgrat amb aquesta definició podem decidir si dos nombres donats són amics, no existeix cap criteri que ens pugui garantir que un nombre donat en té de nombres amics. Direm que un nombre sencer positiu és **solitari** si no té cap amic i se sap que aquesta circumstància es dona en aquells nombres sencers positius per als que el màxim comú divisor (MCD) entre la suma dels seus divisors i ell mateix és 1. Recordeu que el MCD(x,y) de dos nombres sencers positius x i y és el divisor més gran comú a x i y.

Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui, segons aquest criteri, **si és solitari o podria tenir algun amic** (mitjançant algun missatge amb la instrucció `print`). Per exemple, el 6 no potser solitari perquè, apart de que el MCD(6+3+2+1,6)=MCD(12,6)=6$\neq$1, se sap que és amic del 28. En canvi, el 5 és solitari perquè el MCD(5+1,5)=MCD(6,5)=1. Hi ha casos com el del 10 al qual encara avui dia no se li coneix cap amic però no es pot descartar que en tingui donat que MCD(10+5+2+1,10)=MCD(18,10)=2$\neq$1. El nostre programa ens ha de dir per al 5 que és solitari i per al 6 o el 10 que potser tenen algun amic.

Exercici 2.18 (PEM, 2006)
Considereu el següent programa escrit en Perl:

```
$x = 1;
$i = 0;
while ($i < 3) {
  $x = $x * $i;
  $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 0
(b) 1
(c) 2
(d) 3
(e) 4

Exercici 2.19 (PEM, 2006)
Considereu el següent programa escrit en Perl:

```
$x = 0;
$i = 0;
while ($i < 10) {
  if ($i > 3 && $i <= 6) {
    $x = $x + $i;
    $i = $i + 2;
  }
  $i = $i + 1;
}
```

Quin serà el valor de la variable `$x` quan s'acabi d'executar el programa?

(a) 10
(b) 15
(c) 18
(d) 7
(e) 4

Exercici 2.20 (PEM, 2006)
L'operador aritmètic del residu d'una divisió s'escriu en Perl amb el símbol del tant per cent %. Direm que un nombre `$x` és divisible per un altre `$y` si el residu de dividir `$x` per `$y` és 0. Considereu el següent programa escrit en Perl on assumim que tenim un nombre sencer i positiu enregistrat en una variable `$x`:

```
$i = 1;
$s = 0;

while ($i <= $x) {

  if ($x % $i == 0) {
    $s = $s + 1;
  }

  $i = $i + 1;
}

if ($s > 2) {
  print "no\n";
} else {
  print "si\n";
}
```


Assenyaleu per a quins dels següents valors de `$x` el programa anterior mostrarà el missatge "si":

(a) 1
(b) 7
(c) Els dos anteriors
(d) 4
(e) Tots els anteriors.

Exercici 2.21 (Avaluació Formativa, 2006)
Direm que un nombre sencer positiu és **guay** si aquest nombre és igual a la suma de qualsevol seqüència consecutiva i creixent de nombres sencers i positius més petits o iguals que ell mateix, començant a partir del 1. Per exemple, el 1 és guay ja que l'únic nombre sencer i positiu més petit o igual que ell és el propi 1; el 2 no és guay perquè ni 1 ni 1+2 són iguals a 2, el 3 en canvi si que és guay perquè 3=1+2. Fins al 10, el 4, 5, 7, 8 i 9 no són guays però, en canvi, el 6=1+2+3 i el 10=1+2+3+4 si que ho són, de guays.

Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui si és guay o no mitjançant la instrucció `print`. **Nota:** l'única operació aritmètica que necessiteu és la suma.

Exercici 2.22 (Avaluació Formativa, 2007)
Direm que un nombre sencer positiu és **superguay** si aquest nombre és igual a la suma de qualsevol seqüència consecutiva i creixent de nombres sencers i positius més petits que ell mateix, començant a partir de qualsevol número sencer positiu (es a dir, més gran que 0). Per exemple, el 3 és superguay ja que 3=1+2; el 4 no és superguay perquè $1+2 \neq 4$, $1+2+3 \neq 4$, $2+3 \neq 4$; el 5 és superguay perquè 5=2+3.

Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui si és superguay o no mitjançant la instrucció `print`. L'única operació aritmètica que necessiteu és la suma. **No feu anar vectors**.

Exercici 2.23 (Avaluació Formativa, 2008)
Direm que un nombre sencer positiu és **txatxi** si la suma dels seus divisors es més gran que la suma dels divisors de **cadascun** dels nombres sencers positius mes petits que ell. Per exemple, el 5 **no** és txatxi perquè els seus divisors son el 1 i el 5, per tant la suma dels seus divisors es 5+1=6 i aquest valor és inferior a la suma dels divisors del 4, que és 4+2+1=7. En canvi, el 8 si que és txatxi perque no hi ha cap nombre sencer positiu inferior a ell què la suma dels seus divisors sigui superior a 8+4+2+1=15.

Feu un programa en Perl que donat un nombre sencer positiu enregistrat en una variable `$x` ens digui si és txatxi o no mitjançant la instrucció `print`.

Tema 3

Disseny d'algorismes iteratius

Definició 3.1 *Vector*
És un tipus de variable plural que pot emmagatzemar més d'un valor i que permet l'accés indexat dels seus valors. També es coneix sovint com a *taula* o *array*.

Per exemple:

```
@v=(1,2,3,4,5);

print $v[0];
```

emmagatzema a un vector que es diu "v" els valors de l'u al cinc, i la darrera instrucció mostra el primer valor del vector, es a dir un 1. Teniu en compte que els valors d'un vector en llenguatge Perl van indexats desde la posició 0 fins a la $n-1$, per un vector amb n valors. Utilitzem un altre cop la notació de Perl, ficant el símbol `@` per especificar que "v" conté més d'un valor.

Els algorismes iteratius consisteixen principalment d'una composició algorísmica en la que una o més accions poden executar-se repetidament sense have d'especificar-les mes d'un cop. Per exemple, utilitzarem un algorisme iteratiu per sumar els valors d'un vector:

```
@v=(1,2,3,4,5);
$i=0;
$s=0;

while ($i < 5) {
  $s = $s + $v[$i];
```



```
  $i = $i + 1;
}
print $s;
```

En la composició iterativa d'aquest algorisme la condició inicial és que no hem sumat cap valor del vector `v`, es a dir el valor de la variable `i` és 0 i per tant el valor de la suma també ho és (`s` val 0). La condició final és que hem sumat tots els elements del vector, és a dir, la variable `i` té el valor 5 i per tant la variable `s` tindrà el valor de la suma dels elements del vector . L'invariant d'aquesta composició iterativa serà que la suma parcial a calcular a cada iteració correspondrà la suma parcial anterior més l'element del vector corresponent a la iteració, es a dir `s=s+v[i]`.

Suposem que al vector "v" tenim una seqüència de lletres *a*'s i *b*'s acabada en un espai. Feu un programa que imprimeixi la paraula "si", si el vector "v" conté una única lletra *b*, i "no" en qualsevol altre cas. Primera idea, fer un programa que simuli el AFD/DFA del llenguatge associat a l'expressió regular a^*ba^* sobre l'alfabet $\Sigma = \{a,b\}$. Recordeu que aquest autòmat el tenim a la Figura 1.2.

```
$i=0;
$e=0;

while ($v[$i] ne ' ') {
  if ($v[$i] eq 'b') {
    if ($e == 0) {
      $e = 1;
    }
    else {
      if ($e == 1) {
        $e = 2;
      }
    }
  }
  $i=$i+1;
}

if ($e == 1) {
  print "si";
}
else {
  print "no";
}
```

L'algorisme anterior fa anar una variable `e` per emmagatzemar l'estat de l'autòmat i una variable `i` per recórrer les posicions del vector `v`. Fixeu-vos que l'algorisme

només contempla aquelles circumstàncies en les què l'estat de l'autòmat canvia donat que si l'estat no canvia no s'ha de fer res mes que avançar de posició al vector `v`. Penseu ara una forma una mica més senzilla de fer aquest algorisme (pista: només cal un `if` dins el `while`). Solució:

```
$i=0;
$n=0;

while ($v[$i] ne ' ') {
  if ($v[$i] eq 'b') {
    $n = $n + 1;
  }
  $i = $i + 1;
}

if ($n == 1) {
  print "si\n";
}
else {
  print "no\n";
}
```

Es a dir, només cal recórrer el vector i contar quantes lletres "b" hem vist. Finalment, comprovar si hem vist una única "b" o no. Ara penseu un algorisme iteratiu per comptar paraules. Suposem que emmagatzemem en un vector "v" cadascun dels caràcters (lletres, espais, signes de puntuació) d'una frase que acaba amb un punt "." i on les paraules estan separades per un o mes espais, per exemple:

```
"tronc   de nadal   pixa vi  blanc  ."
```

es a dir `$v[0]='t', $v[1]='r', $v[2]='o', ..., $v[34]=' ', $v[35]='.'`

Quina és la idea principal? ...

... doncs que cada paraula comença per una lletra que apareix immediatament després d'un espai. Una possible solució al problema seria la següent:

```
$i=0;
$n=0;

while ($v[$i] ne '.') {
```



```
  if ($i == 0 && $v[$i] ne ' ') {
    $n = $n + 1;
  }
  else {
    if ($v[$i] ne ' ' && $v[$i-1] eq ' ') {
      $n = $n + 1;
    }
  }
  $i = $i + 1;
}
print $n,"\n";
```

3.1 Exercicis

Exercici 3.1 Dissenyeu un algorisme iteratiu que donada una seqüència d'ADN en un vector "v", ens imprimeixi per pantalla la seva seqüència complementària (A-T, G-C). Recordeu que les cadenes d'ADN s'escriuen de 5' a 3'.

Exercici 3.2 (Avaluació Formativa, Febrer 2002)
Donat el següent programa escrit en Perl:

```
@v=('G','A','T','C','T','T','T','C','T','C','T','C',
    'G','G','G','T');

$x=0;

$i=0;

while ($i < scalar(@v)) {

  if (($v[$i] eq 'G') || ($v[$i] eq 'C')) {

    $x = $x + 1;

  }

  $i = $i + 1;

}

print "$x\n";
```

(a) A què correspon el valor `$x` ?

(b) Quin és aquest valor en acabar el programa ?

Nota: la última instrucció del programa mostra per pantalla el valor de la variable `$x`.
La funció `scalar()` calcula el nombre d'elements d'un vector donat.

Exercici 3.3 (PEM, 2002)
Considereu el següent programa escrit en Perl:

```
@v=('A','T','T','G','C','C','T','A');
$i=0;
$n=0;
while ($i < 8) {

  if ($v[$i] eq 'T') {
    $n = $n + 1;
  }

  $i = $i + 1;
}
```

Què és el que fa aquest programa en relació amb la variable `$n`?

(a) compta el nombre de símbols del vector `@v`.
(b) compta el nombre de subseqüències que comencen amb `'T'`.
(c) compta el nombre de símbols `'T'` del vector `@v`.
(d) mostra els símbols del vector `@v` per pantalla.
(e) comprova si al vector `@v` hi ha 8 símbols.

Exercici 3.4 (PEM, 2002)
Considereu el següent programa escrit en Perl:

```
@v=('A','G','T','G','G','C','A','G','C','T','G','A');
$i=0;
$n=0;
$f=0;
while ($i < 12) {

  if ($v[$i] eq 'G' && $f == 1) {
    $n = $n + 1;
  }
```



```
  if ($v[$i] eq 'T' && $f == 0) {
    $f = 1;
  }
  else {
    if ($v[$i] eq 'T' && $f == 1) {
      $f = 0;
    }
  }

  $i = $i + 1;
}
```

Quin serà el valor de la variable `$n` al final de l'execució d'aquest programa?

(a) 5
(b) 2
(c) 12
(d) 7
(e) 3

Exercici 3.5 (PEM, 2002)
Considereu el següent programa escrit en Perl:

```
@v=([4,3,5,2,
    [6,7,8,4,
     2,1,3,5,
     8,3,5,3]);
$i = 0;
$n = 0;
while ($i < 4) {

  $j = 0;
  while ($j < 4) {

    if ($v[$i][$j] > $n && $i == $j) {
      $n = $v[$i][$j];
    }
    $j = $j + 1;
  }

  $i = $i + 1;
}
```

Quin serà el valor de la variable `$n` al final de l'execució d'aquest programa?

(a) 17
(b) 8
(c) 7
(d) 5
(e) cap dels anteriors

Exercici 3.6 Feu un programa en Perl que donada una seqüència d'ADN enregistrada en un vector `@v` ens doni la proporcio de dinucleòtids `CT` dins aquesta seqüència.

Exercici 3.7 (PEM, 2003)
Considereu el següent programa escrit en Perl:

```
@v=('A','T','T','G','C','C','G','T','A','C');
$i=0;
$n=0;
while ($i < 10) {

  if ($v[$i] eq 'G' || $v[$i] eq 'C') {
    $n = $n + 1;
  }

  $i = $i + 1;
}
```

Què és el que fa aquest programa en relació amb la variable `$n`?

(a) mostra els símbols del vector `@v` per pantalla.
(b) compta el nombre de símbols `'G'` en el vector `@v`.
(c) compta el nombre de símbols `'C'` en el vector `@v`.
(d) compta el nombre de dinucleòtids `'GC'` en el vector `@v`.
(e) compta el nombre de símbols `'G'` i `'C'` en el vector `@v`.

Exercici 3.8 (PEM, 2003)
Considereu el següent programa escrit en Perl:

```
$v[0] = 1;
$v[1] = 1;
my $i = 0;

while ($i < 10) {
```



```
  if ($i > 1) {
    $v[$i] = $v[$i-2] + $v[$i-1];
  }

  print "$v[$i], ";

  $i = $i + 1;
}
```

Quina de les següents sèries de 10 nombres sencers ens mostrarà el programa per pantalla?

(a) 1,1,2,4,8,16,32,64,128,256,
(b) 1,1,2,3,5,8,13,21,34,55,
(c) 1,1,1,1,1,1,1,1,1,1,
(d) donarà un error per intentar accedir a una posició negativa del vector `@v`.
(e) 1,1,-2,-5,-8,-11,-14,-17,-20,-23

Exercici 3.9 (PEM, 2003)
Considereu el següent programa escrit en Perl:

```
@v=([5,4,6,7],
    [4,6,8,6],
    [2,3,6,5],
    [2,1,5,9]);
$i = 0;
$n = 0;
while ($i < 4) {

  $j = 0;
  while ($j < 4) {

    if ($v[$i][$j] > $n && $i != $j) {
      $n = $v[$i][$j];
    }
    $j = $j + 1;
  }

  $i = $i + 1;
}
```

Quin serà el valor de la variable `$n` al final de l'execució d'aquest programa?

(a) 8
(b) 9
(c) 5
(d) 16
(e) cap dels anteriors

Exercici 3.10 (PEM, 2003)
Considereu el següent programa escrit en Perl:

```
@v=([4,3,5,2],
    [6,7,8,4],
    [2,1,3,5],
    [8,3,5,3]);
$i = 0;
$n = 10;
while ($i < 4) {

  $j = 0;
  while ($j < 4) {

    if ($v[$i][$j] < $n && $i == $j) {
      $n = $v[$i][$j];
    }
    $j = $j + 1;
  }

  $i = $i + 1;
}
```

Quin serà el valor de la variable `$n` al final de l'execució d'aquest programa?

(a) 17
(b) 3
(c) 1
(d) 5
(e) 4

Exercici 3.11 (PEM, 2005)
L'operador de comparació `eq` en Perl s'avalua com a cert quan tots dos valors comparats són iguals. L'operador lògic `&&` en Perl correspon a la conjunció d'expressions lògiques. Considereu el següent programa escrit en Perl:


```
@v=('A','T','G','G','C','C','G','T','G','C');
$i = 0;
$n = 0;
$m = 0;
while ($i < 9) {
  if ($v[$i] eq 'G') {
    if ($v[$i] eq 'G' && $v[$i+1] eq 'C') {
      $n = $n + 1;
    }
    $m = $m + 1;
  }
  $i = $i + 1;
}
```

Si volguèssim mostrar la freqüència relativa de nucleòtids `C` que apareixen a continuació d'un nucleòtid `G` al vector `@v` utilitzant el programa anterior, quina de les següents instruccions li afegirieu al final:

medskip (a) `print $n/9;`
(b) `print $m/9;`
(c) `print $n/$m;`
(d) `print $m/$n;`
(e) `print ($m+$n)/9;`

Exercici 3.12 (PEM, 2005)
L'operador lògic `||` en Perl correspon a la disjunció d'expressions lògiques. Considereu el següent programa escrit en Perl:

```
@v=([4,3,5,2],
    [5,6,9,4],
    [2,1,3,5],
    [7,3,8,3]);
$i = 0;
$n = 0;
while ($i < 4) {
  $j = 0;
  while ($j < 4) {
    if ($v[$i][$j] > $n && ($i > 2 || $j < 2)) {
      $n = $v[$i][$j];
    }
    $j = $j + 1;
```

```
    }
    $i = $i + 1;
  }
```

Quin serà el valor de la variable `$n` al final de l'execució d'aquest programa ?

(a) 9
(b) 3
(c) 6
(d) 7
(e) 8

Exercici 3.13 (Examen de Setembre, 2005)
Feu un programa en Perl que donada una seqüència d'ADN enregistrada en un vector `@seq` i un motiu funcional de `$m` nucleòtids enregistrat en un vector `@motiu` ens mostri per pantalla amb la instrucció `print` a quines posicions de la seqüència es troba el motiu i quin percentatge de `G+C` té el motiu en la seva composició nucleotídica, **sense utilitzar expressions regulars**.
Nota: La funció `scalar(@v)` ens calcula el nombre délements que conté el vector `@v`. Podeu assumir que tots els nucleòtids estan enregistrats amb lletres majúscules.

Exercici 3.14 (Assaig, 2005)
Un *stem-loop* és una estructura secundària de l'ARN que es forma quan una molècula d'ARN d'una sola hebra es plega sobre sí mateixa per formar una doble hèlix complementària (*stem*) encapçalada per un bucle (*loop*) com si fos una piruleta. Un exemple de *stem-loop* seria el següent:

```
                      A A
                    G     A
5' CAGGAAACUG 3'      G-C
                      A-U
                      C-G
                    5'   3'
```

Assumint que els únics emparellaments de bases admissibles en el *stem* són `G-C` i `A-U` la seqüència d'ARN que forma el *stem* ha de ser el que s'anomena un *palíndrom* d'ARN: la seqüència de 5' a 3' ha de ser la mateixa que llegint el seu revessat complementari de 3' a 5', com ara `AAUU` o `CCAGUACUGG` de les quals si fem el seu revessat complementari obtindrem la mateixa seqüència i que per tant tindrà sempre un nombre parell de bases. Naturalment, en un *stem-loop* tenim un *loop* enmig de les dues parts complementaries del palíndrom que forma el *stem*.

(a) Feu un programa en Perl que donada una seüència d'ARN enregistrada en un vector `@v` ens digui si pot formar un *stem-loop* amb un loop de longitut 0, es a dir, sense *loop*, mostrant per pantalla un missatge `si` o `no` amb la instrucció `print`. Per exemple, per la seqüència `AAUU` ens ha de dir `si` i per la seqüència `AAUG` ens ha de dir `no`.

(b) Feu un programa en Perl que donada una seqüència d'ARN enregistrada en un vector `@v` ens digui si pot formar un *stem-loop* amb un *loop* de longitut 1.

(c) Feu un programa en Perl que donada una seqüència d'ARN enregistrada en un vector `@v` ens digui si pot formar un *stem-loop* amb un *loop* de longitut qualsevol enregistrada en una variable `$n`.

Nota: Indiqueu clarament a quin apartat correspon cadascun dels programes que feu. Escriviu el codi identant amb espais els blocs associats a composicions iteratives o alternatives. **En cap cas podeu utilitzar expressions regulars**.
Consell: Formalitzeu per escrit la noció de palíndrom d'ARN (a quines posicions hem de comprovar la complementarietat de les bases a cada moment).

Exercici 3.15 (PEM, 2006)
Considereu el següent programa escrit en Perl:

```
@v = ('A','T','T','G','C','C','G','T','A','C');
$i = 1;
$n = 0;
while ($i < 10) {

  if ($v[$i-1] eq 'A' || $v[$i] eq 'T') {
    $n = $n + 1;
  }

  $i = $i + 1;
}
```

Què és el que fa aquest programa en relació amb la variable `$n`?

(a) compta el nombre de nucleòtids `'A'` en el vector `@v`
(b) compta el nombre de nucleòtids `'T'` en el vector `@v`
(c) compta el nombre de nucleòtids `'A'` i `'T'` en el vector `@v`
(d) compta el nombre de dinucleòtids `'AT'` en el vector `@v`
(e) no fa res ja que no modifica el seu valor inicial en cap moment.

Exercici 3.16 (Examen de Setembre, 2006)
Feu un programa en Perl que donada una seqüència d'ADN enregistrada en un vector `@seq` ens mostri per pantalla amb la instrucció `print` quants codons de stop (TAG, TGA i TAA) es troben a la seqüència per **cada** possible pauta de lectura en el sentit enregistrat de la seqüència (es a dir, no cal per la seqüència complementaria inversa), i **sense utilitzar expressions regulars**.

Nota: La funció `scalar(@v)` ens calcula el nombre d'elements que conté el vector `@v`. Podeu assumir que tots els nucleòtids estan enregistrats amb lletres majúscules.

Exercici 3.17 (PEM, 2008)
Considereu el següent programa escrit en Perl:

```
@v=('A','T','G','G','C','C','G','T','G','C');
$i = 0;
$n = 0;
while ($i < 10) {
  if ($v[$i] eq 'G') {
    $n = $n + 1;
  }
  $i = $i + 1;
}
```

Si volguéssim mostrar la freqüència relativa (o proporció) de nucleòtids `'G'` que apareixen al vector `@v` utilitzant el programa anterior, quina de les següents instruccions li afegirieu al final:

(a) `print $i/10;`

(b) `print $n/10;`

(c) `print $n;`

(d) `print $i;`

(e) `print $i/$n;`

Exercici 3.18 (Assaig, 2008)
Feu un programa en Perl què donada una seqüència d'ADN enregistrada en un vector anomenat `@seq` ens mostri per pantalla, amb un missatge mitjançant la instrucció `print`, el nombre de cops que ocorre la subseqüència `'CNC'`, on `N` fa referència a qualsevol nucleòtid, dins el vector `@v`.

Exercici 3.19 (Examen de Setembre, 2008)
Feu un programa en Perl què, donada la seqüència d'ADN enregistrada en el vector `@v` que trobareu a sota, ens calculi i mostri per pantalla, amb la instrucció `print`, la freqüència relativa (es a dir, el percentatge) de dinucleòtids `'CT'` que apareixen a la seqüència, **sense** utilitzar expressions regulars.

```
@v=('A','T','C','T','C','C','C','T','C','T');
```

Tema 4

Eficiència dels algorismes

Tal i com hem vist a l'exemple de reconéixer paraules sobre l'alfabet $\Sigma = \{a, b\}$ que contenen una sola lletra "b", podem trobar més d'una manera d'escriure un algorisme per al mateix problema.

Dos algorismes differents per al mateix problema poden ser també differents en com de ràpid resolen el problema. Aleshores es diu que un algorisme és *més eficient* que l'altre.

Per exemple, considerem el problema de cercar una paraula a un diccionari. La forma més simple de cercar una paraula a un diccionari és llegir la primera paraula del diccionari i comprovar si és la paraula que estem cercant. Si ho és, haurem acabat. Si no ho és, llegirem la segona paraula. Si la segona paraula és la que estem cercant, haurem acabat. Si no ho és, llegirem la tercera, i així repetidament fins trobar la paraula que busquem.

Si, per exemple, busquem d'aquesta forma la paraula `tronc`, quan l'haurem trobat haurem llegit totes les paraules del diccionari entre la lletra `A` i la lletra `T` a mes a mes de les que comencen per `T` i són alfabèticament anteriors a `tronc`. Qualsevol diccionari de butxaca sol tenir unes 50,000 paraules, per tant per buscar la paraula `tronc` podem haver llegit fàcilment unes 40,000 paraules.

Ara pensem com fem realment nosaltres la cerca de paraules al diccionari. La manera en què ho fem es similar al que es coneix com *cerca binaria*.

Algorisme de cerca binaria. Considerem un rang de paraules, per exemple les que hi ha de la lletra `A` a la lletra `H`, que anomenarem *finestra*. Inicialment la finestra correspon al rang de paraules de la `A` a la `Z`, es a dir, el diccionari sencer. Tal i com l'algorisme va executant-se, la finestra s'anirà escurçant, de vegades per l'esquerra, de vegades per la dreta (o per dalt i per baix si us imagineu la finestra verticalment).

Arribarem a un punt en el qual o be la finestra conté una única paraula que és la que estem buscant, o be la finestra és buida i vol dir que la paraula no és al diccionari. Aquesta podria ser una especificació vàlida d'aquest algorisme:

```
@dic = ("Genis", "Mar", "Pep", "Robert", "Roderic",
        "Sergi", "Xavi");
$par = "Pep";
$min = 0;
$max = 6;
$trobada = 0;

while ($min <= $max && !$trobada) {

  $mig = int( ($min+$max)/2 );

  if ($dic[$mig] lt $par) {
    $min = $mig + 1;
  }
  else {

    if ($dic[$mig] gt $par) {
      $max = $mig - 1;
    }
    else {
      $trobada = 1;
    }

  }

}

if ($trobada == 1) {
  print "paraula trobada\n";
}
else {
  print "paraula no trobada\n";
}
```

En l'exemple anterior on buscavem la paraula `tronc` al diccionari de butxaca,

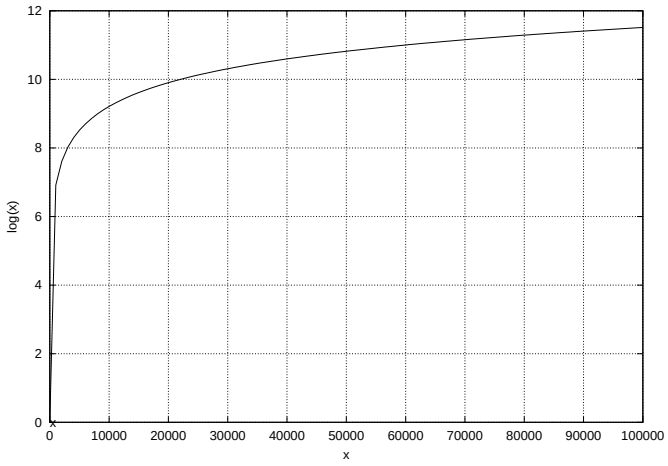


Figura 4.1: Funció de creixement logarítmic.

haviem de llegir unes 40,000 paraules d'un diccionari de 50,000. Quantes hauriem de llegir, fent el mateix, si el diccionari tingués el doble de paraules, es a dir, 100,000?

Assumint que el tamany del diccionari creix proporcionalment per cada lletra, hauriem de llegir unes 80,000 paraules per trobar `tronc`. En un cas com aquest un diu que l'algorisme té un *ordre de creixement lineal*, i ho anotarem $O(N)$. També es sol dir que la *complexitat* de l'algorisme és d'ordre lineal. Això implica que el nombre d'operacions que l'algorisme ha de fer es proporcional al nombre d'elements que ha de tractar l'algorisme.

Ara pensem quin esforç extra hauria de fer l'algorisme de cerca binaria si en lloc de utilitzar-lo amb un diccionari de 50,000 paraules l'utilitzéssim amb un de 100,000.

Doncs, simplement, una volta addicional de la composició iterativa (el `while`). La cerca binaria pertany a la classe d'algorismes que que tenen un *ordre de creixement logarítmic*, que l'anotarem com $O(\log N)$. A la Figura 4.1 podem veure dibuixada la funció logarítmica per valors de l'u fins al cent-mil. Com podeu apreciar el rang de valors de l'eix de la funció (l'eix vertical) arriba fins al valor 12. Ara penseu que per una funció de creixement lineal aquest rang arribaria fins al valor 100,000.

És important adonar-se'n que la noció de complexitat, o ordre de creixement, s'interpreta en termes del comportament d'un algorisme *en general*. Podem pensar

fàcilment d'algun cas en el que la cerca simple al diccionari sigui més eficient que la cerca binària, i no per aixo l'algorisme de cerca binaria deixa de ser més "eficient" que l'algorisme de cerca simple (per exemple amb la paraula `abad` la qual possiblement estigui en la 3a o 4a posició de tot el diccionari).

Per tal de tenir una idea més precisa de fins a quin punt l'ordre de creixement determina l'eficiència dels algorismes, a la següent taula trobarem diversos ordres (funcions) de creixement amb el seu valor per $N = 1,000,000$.

Funció de creixement	**Valor per** $N = 1,000,000$
1	1
$\log N$	13.8
$\sqrt{N}$	1,000
N	1,000,000
$N \log N$	13,815,510
N^2	1,000,000,000,000
N^3	1,000,000,000,000,000,000
2^N	un número amb 693,148 dígits

Considerem la següent matriu de 4×4 elements, que en llenguatge Perl l'especificarem com a un vector de vectors.

```
@v=([11,12,13,14],
    [21,22,23,24],
    [31,32,33,34],
    [41,42,43,44]);
```

La forma en la que ens referim a un element en concret d'una matriu en Perl es mitjançant la notació `$v[2][3]` on, en aquest cas, el 2 es refereix a la tercera posició a la primera dimensió i el 3 es refereix a la quarta posició de la segona dimensió, es a dir, l'element 34.

Penseu ara un algorisme que, donada la matriu "v" ens imprimeix la suma dels elements en cadascuna de les quatres "files" corresponents a la primera dimensió. Es a dir:

$$11+12+13+14$$

$$21+22+23+24$$

$$31+32+33+34$$

$$41+42+43+44$$

L'algorisme doncs, podria ser el seguent:

```
$i=0;

while ($i < 4) {
  $s=0;
  $j=0;
  while ($j < 4) {
    $s = $s + $v[$i][$j];
    $j = $j + 1;
  }
  print $s,"\n";
  $i=$i+1;
}
```

Quina és la complexitat (ordre de creixement) d'aquest algorisme respecte a la dimensió de la matriu?

(a) $O(N)$ lineal ?

(b) $O(N^2)$ quadràtica ?

(c) $O(N^3)$ cúbica ?

(d) $O(2^N)$ exponencial ?

Teniu en compte que si la matriu en lloc de ser de dimensions $n \times n$ (quadrada), fos $n \times m$ amb $n \neq m$ (rectangular), aleshores escriuriem l'ordre de creixement com:

$$O(nm)$$

El qual segueix sent d'ordre quadràtic. Donada una funció de creixement qualsevol formada per termes amb ordres de creixement diferents, direm que el terme de complexitat més alta *domina* la funció i que per tant la complexitat, o ordre de creixement, de l'algorisme serà el del terme amb complexitat més alta. Per exemple, un algorisme amb ordre de creixement:

$$O(n^2+m+1)$$

direm que és de complexitat quadràtica perquè de tots tres termes, n^2, m i 1, el terme n^2 és el que *domina* els altres dos.

4.1 Exercicis

Exercici 4.1 (PEM, 2002)
Quina és la complexitat (o ordre de creixement) del programa del Exercici 3.5?

(a) lineal
(b) quadràtica
(c) cúbica
(d) exponencial
(e) logarítmica

Exercici 4.2 (PEM, 2003)
La complexitat (o ordre de creixement) del programa del Exercici 3.9 és:

(a) Més gran que quadràtica però més petita que exponencial
(b) Més petita que quadràtica
(c) Més gran que lineal però més petita que cúbica
(d) Cúbica
(e) Exponencial

Exercici 4.3 (PEM, 2004)
La complexitat (o ordre de creixement) del programa del Exercici 2.12 és:

(a) Logarítmica
(b) Lineal
(c) Quadràtica
(d) Cúbica
(e) Exponencial

Tema 5

Correspondència entre seqüències de símbols

Motivació. En seqüències d'ADN, ARN o aminoàcids, un grau similaritat alt entre seqüències normalment implica també un grau alt de similaritat estructural i/o funcional. Sovint, un grau de similaritat entre seqüències prou significatiu indica que totes dues tenen un ancestre comú, es a dir, que són *homòlogues*. Donades dues seqüències de símbols (per exemple d'ADN):

```
t={AATGC}
p={AGGC}
```

anomenarem com a *problema de la correspondència entre seqüències de símbols* (en anglés, *string matching problem*) al problema de trobar si `t` apareix dins de `p`, o `p` dins de `t`, o en quina mesura `t` i `p` són similars.

Aquest problema té dues variants principals:

- **la correspondència exacta**: on buscarem si una de les seqüències es troba replicada exactament dins de l'altra.

- **la correspondència inexacta**: on calcularem en quin grau les dues seqüències són similars.

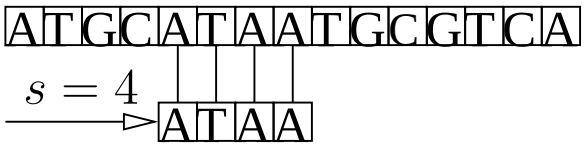


Figura 5.1: Correspondència exacta d'una seqüencia respecte a una altra.

5.1 Correspondència exacta

Considerem una seqüència

```
t={ATGCATAATGCGTCA}
```

i una seqüència

```
p={ATAA}
```

En aquest cas la seqüència `p` es troba dins de `t` a partir del cinqué símbol i llavors direm que la seqüència `p` *apareix amb desplaçament s* o que *apareix a partir de la posició* $s+1$ on, en aquest cas, $s=4$. També anomenarem a `p` el patró a buscar dins de `t`. En la Figura 5.1 podem veure gràficament la situació que acabem de descriure.

En general, suposem que `t` té n símbols i `p` en té m. El problema de la correspondència exacta entre seqüències de símbols correspon al problema de trobar tots els possibles desplaçaments s que fan que els símbols

```
t[s+1]t[s+2]t[s+3]...t[s+m]
```

siguin exactament els de `p`, es a dir:

```
t[s+1]==p[0]
t[s+2]==p[1]
t[s+3]==p[2]
...
t[s+m]==p[m-1]
```

Quants possibles desplaçaments s com a màxim podem arribar a trobar ? Per què ?

$$n-m+1$$

Quin seria l'algorisme més senzill per trobar tots els desplaçaments en els quals la seqüència `p` apareix exactament dins de `t`?

```
$s=0;

while ($s <= $n-$m) {
  $i=0;

  while ($i < $m  &&  $t[$s+$i] eq $p[$i]) {
    $i = $i + 1;
  }

  if ($i == $m) {
    print "patro trobat amb desplacament ",$s,"\n";
  }

  $s = $s + 1;
}
```

Quina és la complexitat de l'algorisme anterior ?

$$O((n-m+1)m)$$

Aquest algorisme *simple* no és el més eficient, existeixen algorismes per aquest problema amb ordre de creixement $O(n+m)$.

5.2 Correspondència inexacta

Trobar una correspondència inexacta, o aproximada, de dues seqüències de símbols és de tant o més interés que intentar trobar una correspondència exacta que moltes vegades no existeix. Per exemple, si intentem trobar una correspodència exacta entre les dues seqüències de sota, no en podrem trobar cap. Però, si per l'alineament indicat, anotem les correspondències entre els parells de símbols, podem arribar a trobar un nombre suficientment gran de correspondències com per a considerar-ho una informació rellevant des d'un punt de vista biològic.


```
AASRPRSGVPAQSDSDPCQNLAATPIPSRPPSSQSADARQGRWGP
      || | |       |   |   | |   || |
      SGAPGQRGEPGPQGHAGAPGPPGPPGSD
```

Idea fonamental. Assignar una puntuació (en anglès, *score*) a cadascuna de les possibles correspondències inexactes. Aquesta puntuació serà la suma de puntuacions individuals de cada parell de símbols que estan a la mateixa posició. Per exemple, suposem que puntuem amb 1 quan dos símbols a la mateixa posició són iguals, i amb 0, si no ho són:

```
ATCGCA
 ATGTA
000101 = 2
```

Buscarem, aleshores, la correspondència que ens maximitza la puntuació. Respecte a l'exemple anterior, podriem trobar una correspondència inexacta millor si introduïm un salt (en anglès, *gap*) dins de la seqüència més curta:

```
ATCGCA
AT-GTA
110101 = 4
```

Un altre exemple podria ser el següent:

```
AASRPRSGVPAQSDSDPCQNLAATPIPSRPPSSQSCQKCRADARQGRWGP
      || | |       |   |   | |   || |
      SGAPGQRGEPGPQGHAGAPGPPGPPGSDGSPARKG

AASRPRSGVPAQSDSDPCQNLAATPIPSRPPSSQSCQKCRADARQGRWGP
      || | |       |   |   | |   || |          || |
      SGAPGQRGEPGPQGHAGAPGPPGPPGSDG-----SPARKG
```

on hem trobat tres correspondències més introduint un salt de longitut 5.

El problema de trobar una correspondència inexacta *òptima* entre dues seqüències de símbols, introduïnt salts, es coneix a la Biologia com el problema de l'*alineament de seqüències* (en anglès, *sequence alignment problem*).
Una forma de trobar aquesta correspondència inexacta òptima seria per *força bruta*:

- Tenim dues seqüències de longitut N entre les quals volem trobar la seva correspondència inexacta òptima.
- Considerem totes les formes possibles inserir salts en una seqüència de longitut N:

```
A G T T C
A G T T-C
A G T-T C
A G T-T-C
A G-T T C
A G-T T-C
A G-T-T C
A G-T-T-C
...
```

- Per cadascuna de les dues seqüències inicials, generem totes les seqüències que resulten d'inserir totes les combinacions de salts possibles. Distingirem entre les seqüències generades a partir de l'una o de l'altra seqüència inicial, dividint-les en dos grups.
- Per cada parell de seqüències, on hi ha una de cada grup, calculem les puntuacions per cadascuna de les possibles correspondències inexactes.
- Finalment seleccionem aquella correspondència inexacta que maximitza la puntuació.

Per tenir una idea de la complexitat d'aquest algorisme penseu que, per una seqüència de N símbols, hi ha approximadament 2^N formes diferents d'inserir salts dins la seqüència.

Per cada parell de seqüències, hem de generar tots els possibles alineaments, que son N. Per cada alineament hem de calcular la seva puntuació que requerirà de l'ordre de N operacions. Es a dir, per cada parell haurem d'executar un conjunt d'accions amb una complexitat d'ordre $O(N^2)$, i tenim $2^N \times 2^N = 2^{2N}$ parells. Globalment, la complexitat del problema d'alinear seqüències es, en principi:

$$O(2^{2N}N^2)$$

Més concretament, s'ha de tenir en compte que molts d'aquests alineaments (on es fan correspondre salts amb salts) no tenen sentit des d'un punt de vista biològic,

amb lo qual la complexitat real del problema de l'alineament de seqüències (o correspondència inexacta de seqüències) és (Waterman, 1984):

$$O\left(\frac{2^{2N}}{4\sqrt{N\pi}}\right)$$

Per exemple, per dues seqüències de longitut $N = 1000$ hauriem de fer de l'ordre de 10^{600} operacions!!

Aquest problema té una solució algorísmica més eficient (ordre $O(nm)$) mitjançant el que es coneix com *programació dinàmica* (en anglés, *dynamic programming*). On el terme *programació* no es refereix al fet de crear un *programa d'ordinador*, sino al fet de formular un *programa matemàtic*, i la utilització de la paraula *programació* es una mera coincidència entre ambdues terminologies. Uns altres tipus de programació matemàtica són, per exemple, la *programació lineal*, la *programació quadràtica* o la *programació entera*.

El programes matemàtics s'apliquen a problemes d'optimització i, més concretament, la programació dinàmica s'aplica a aquells problemes que tenen una *subestructura òptima*. Això passa quan el problema el podem dividir en subproblemes on les seves respectives solucions òptimes ens condueixen a la solució òptima del problema sencer.

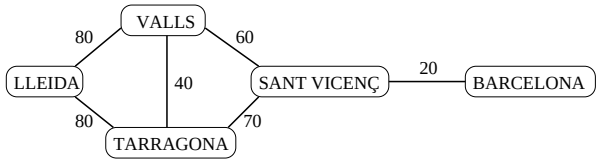
Considerem que, a sota, alineament de l'esquerra és òptim. La puntuació (*score*) total d'aquest alineament és la puntuació de les primeres 4 posicions més la puntuació de la última posició, tal i com s'especifica al mig:

```
AATGC          AATG   C          AATGC
|  ||     +    |  |  |     =    |  ||
AG-GC          AG-G   C          A-GGC
```

Idea fonamental. Si l'alineament de totes les posicions és òptim, aleshores l'alineament de les primeres quatre posicions també ho és.

Si això no fos veritat de forma que, per exemple, alineant G i T donés una puntuació més alta, aleshores l'alineament de la dreta tindria una puntuació global més alta que el de l'esquerra, contradint el supòsit inicial.

Aquest argument és equivalent a considerar, per exemple, les següents distàncies en Km. dels diferents trajectes que podem fer amb tren entre les ciutats de Lleida, Valls, Tarragona, San Vicenç i Barcelona:



i adonar-nos que si el trajecte més curt entre Lleida i Barcelona, passa per Valls, forçosament el trajecte més curt entre Lleida i Sant Vicenç també ha de passar per Valls. Plantejeu-vos, per exemple, que si no fos així i el trajecte més curt entre Lleida i San Vicenç passés per Tarragona, que implicaria això respecte al trajecte entre Lleida i Barcelona que passa per Valls?

Tornant al problema de l'alineament de seqüències, es tracta d'explotar el fet de que la puntuació d'un alineament és la suma dels alineaments individuals de cada parell de símbols:

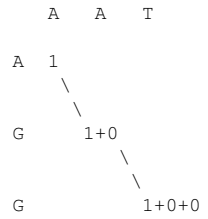
```
AATGC            A   A   T   G   C
|  ||            | +   +   + | + |
AG-GC            A   G   -   G   C
```

Si ens fixem be, fent l'alineament símbol per símbol, ens adonarem que a cada pas només tenim 3 possibilitats:

- Alinear el següent símbol de la seqüència 1 amb el següent símbol de la seqüència 2:

  ```
  seq 1            A   A   T
                   | +   +
  seq 2            A   G   G
  ```

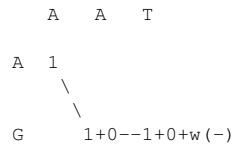
  tenim alineats els dos primers símbols de les dues seqüències, i aliniem el tercer. Això es representa també amb la següent matriu on la dimensió horitzontal l'associatrem a la sequència 1, i la vertical a la seqüència 2. Els alineaments entre símbols s'anoten diagonalment propagant la puntuació al llarg d'aquesta diagonal i sumant cada cop la puntuació de correspondència entre els dos símbols que aliniem.

- alinear el següent símbol de la seqüència 1 amb un salt a la seqüència 2:


  ```
  seq 1            A   A   T
                   | +   +
  seq 2            A   G   -
  ```

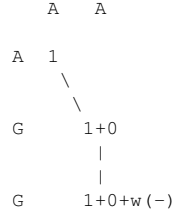
  tenim alineats els dos primers símbols de les dues seqüències, i aliniem el tercer símbol de la seqüència 1 amb un salt (gap). Això ho expressarem a la matriu com un desplaçament en la dimensió del símbol que alinearem amb el salt, en aquest cas la dimensió *horitzontal*. La puntuació es propaga al llarg d'aquest desplaçament i es suma una quantitat $w(-)$ que *penalitza* la inserció del salt.

- alinear el següent símbol de la seqüència 2 amb un salt a la seqüència 1:

  ```
  seq 1            A   A   -
                   | +   +
  seq 2            A   G   G
  ```

  tenim alineats els dos primers símbols de les dues seqüències, i aliniem el tercer símbol de la seqüència 2 amb un salt (gap). Això ho expressarem a la matriu com un desplaçament en la dimensió del símbol que alinearem amb el salt, en aquest cas la dimensió *vertical*. La puntuació es propaga al llarg d'aquest desplaçament i es suma una quantitat $w(-)$ que *penalitza* la inserció del salt.

L'algorisme de programació dinàmica per la correspondència inexacta, o alineament, de seqüències consisteix aleshores en construir la matriu sencera d'una seqüència contra l'altra. Abans de començar a omplir cel.les de la matriu, hem d'afegir el símbol del salt al començament de cada sequència per tal de poder considerar alineaments que poden començar, o acabar, alineant símbols d'una de les dues seqüències amb un salt.

Començarem per la fila 1 columna 1 i, d'esquerra a dreta, i de dalt a baix, a cada posició considerarem les tres possibles propagacions. De les tres possibles propagacions calcularem quines puntuacions generen cadascuna, i escollirem la propagació de màxima puntuació.

Un cop hem construït la matriu sencera, començarem per la posició de la última fila i la última columna, i resseguint el camí de màxima puntuació arribarem a la posició de la primera fila i primera columna, recuperant l'alineament òptim al temps que escrivim els alineaments individuals d'acord amb el tipus de propagació dut a terme (diagonal, desplaçament vertical o desplaçament horitzontal).

Per les seqüències anteriors `AATGC` i `AGGC`, intenteu construïr la matriu per trobar l'alineament òptim ficant la seqüència `AATGC` a la dimensió horitzontal i la seqüència `AGGC` a la vertical. Considereu una penalització de salt nul.la $w(-) = 0$, i una puntuació de 1 quan dos símbols són iguals i de 0 quan no ho són. La matriu en qüestió la trobem a la Figura 5.2.

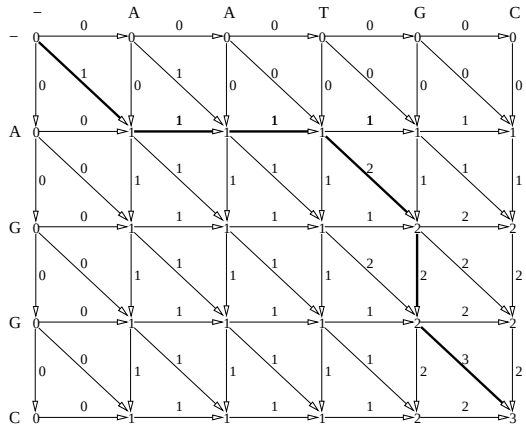


Figura 5.2: Matriu de propagació de puntuacions dels alineaments de dues seqüències. Les fletxes més dobles ens indiquen el camí òptim.

La reconstrucció de l'alineament òptim és fa de la següent forma:

```
          C
          |        diagonal
          C

        - C
          |        desp. vertical
        G C

      G - C
          |        diagonal
      G G C

    T G - C
          |        desp. horitzontal
    - G G C

  A T G - C
          |        desp. horitzontal
  - - G G C

A A T G - C
          |        diagonal (inici)
A - - G G C
```

Com podem apreciar, el cost de tot el procés és a la construcció de la matriu, que té una complexitat $O(nm)$ donat que les puntuacions es van reutilitzant tal i com anem construïnt la matriu.

Per tal de poder alinear el primer símbol de qualsevol de les dues seqüències amb un salt, hauriem d'incorporar a totes dues seqüències un salt com a primer símbol, i fer la resta del procés com s'ha explicat.

Cal emfatitzar que si haguéssim donat un valor negatiu a la funció de penalització $w(-)$ hauriem trobat un altre alineament, possiblement aquest:

```
A A T G C
A G - G C
```

Finalment, també cal dir que no necessariament l'òptim és únic, pot haver diversos camins que ens proporcionin la puntuació màxima.