Part 1

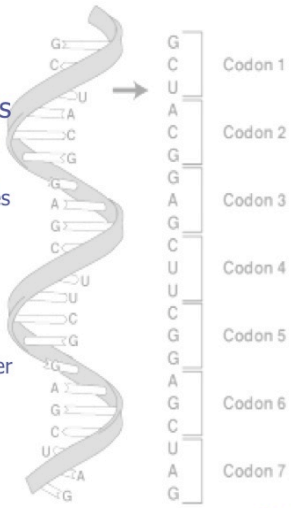
Genome and Sequences

The Genetic Code

(1) 4 Letters – ATGC

(2) A triplet code for 20 amino acids
- Four letters allow 64 different *ordered* triplets
- Several amino acids have multiple codes
- Further control codes
- Code is not universal, evolutionary distant species use different codes
- Each code has a specific translator molecule (tRNA) matching 3 bases on one side and an amino acid on the other side

(3) Start position and direction matter!



Genes

(1) Elementary coding regions on the DNA
- Sections coding particular proteins (or parts of proteins or RNA molecules)
- Some species have lots of non-coding „junk DNA"

(2) Reading process must be activated
- Specialized cells only activate a subset of the genes
- By complex regulation processes (through „docking" at the DNA, other proteins can promote or inhibit)
- DNA double-helix needs to be unfolded and unzipped

(3) A gene is a structured sequence
- Exons are sections that are decoded
- Introns are spliced out after reading and thus are ignored
- Promotors are distant regions relevant for regulation

The Use of Models

(1) The genetic code is a model
- Sequence is reduced to letters
- Simplified, abstract view of the DNA molecule and its role in the organism
- Sequence of amino acids uniquely identifies a protein
- Analysis and predictions can be made on the model (-> bioinformatics)

(2) The idea of a „gene" is a model
- Genome is reduced to set of genes with functional annotation
- The idea is „something that is responsible for a set of proteins"
- There is nothing universal on the DNA saying „gene starts here"
- Actual gene performance (splicing, regulation) much more complex
- A catalog of genes/proteins helps structuring knowledge

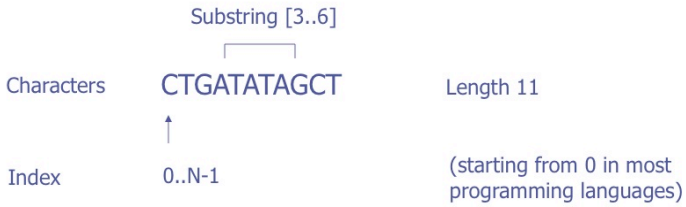
How to find and compare genes?

human alpha globin vs. human beta globin

human alpha globin vs. leghaemoglobin from yellow lupin
(evolutionary and functionally related)

human alpha globin vs. F11G11.2 (unrelated)

String representation



The string model for genetic analysis

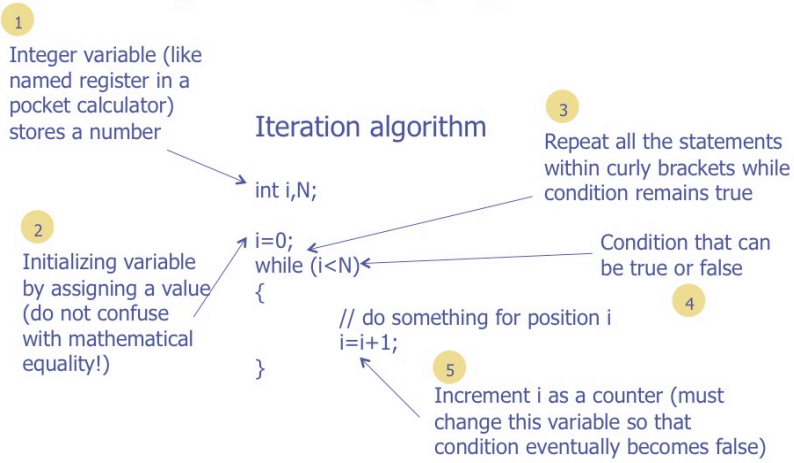
(1) What can you do with a genomic sequence string?

- Annotate a sequence with a gene (-> database)
- Collect the whole genome as a string
- Count the number/frequency of letters/bases
- Find the location of a gene in the whole sequence
- Find a degree of similarity of sequences
- Assemble overlapping sequence fragments (-> Genome Project)
- Establish differences between organisms by (dis)similarity

(2) What would you like to do with a genomic sequence string?

- Find the location of genes in a sequence
- Identify patterns that represent specific (regulatory) functions
- Identify functional sites in the encoded protein
- Predict the 3D structure of the encoded protein
- ...

Describing and executing algorithms



Language: groovy.codehaus.org

Elementary string algorithms (1)

Compare strings a and b of length N:

```
string a,b;
1 int i,N;
boolean same;

2 i=0;
while (i<N && a[i]==b[i]) 3
{
        i=i+1; 4
}
if (i>=N)
        same = true;
5 else
        same = false;
```


Programming nomenclature

string a,b;
Defining two variables of type *string*
Value could be set by
a=„TTAG"

&&
The boolean „and" operator that checks whether both conditions are true

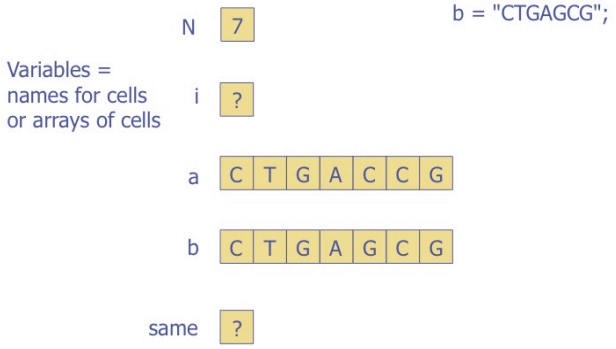
==
The mathematical „equal" operator, checking whether two numbers are the same

while (*some condition*) { *a group of statements to be repeated* }
The general loop statement. While condition is true, statements are executed over and over

if (*some condition*) *statement1* else *statement2*
A conditional statement that, depending on the condition, executes either one or the other

Computer memory

```
N = 7;
a = "CTGACCG";
b = "CTGAGCG";
```



Each memory cell contains a number or character code, which is initially undefined

Elementary string algorithms (2)

Search string b (length M) in a (length N):

```
int i, N, M;
boolean found = false;
int offset = 0;      // first try for the start pos of b in a

while  (N-M >= offset && found == false)
{
          i=0;
          while (i<M && a[i+offset]==b[i])
          {
            i=i+1;
          }
          if (i>=M)
                    found = true;
          else
                    offset=offset+1;
}
```

Algorithmic complexity

(1) *O* (F(n)) means runtime on large n is always less than cF(n)
- There may be a constant overhead that is relevant for small n
- c is an arbitrary constant

(2) String comparison is *O* (N)
- Linear time, c is roughly the time needed for one loop

(3) Simple string search is *O* (MN)
- For large N and small M, may need to try all N positions and compare with M characters
- On the average, only half of N needs to be checked, so c could be half the time for one inner loop

(4) Polynomial time complexity is still „efficient"
- Exponential time is no longer computationally tractable

More efficient string search

(1) Boyer-Moore skips some intermediate positions
- Compares from right to left
- If there is a mismatch, it is a property of the pattern how many positions it can advance for the next comparison
- CT**GAT**ATAGCT
- TA**GAT**
- We can advance the pattern by 4 positions as GAT does not appear in the pattern again (always if there is a mismatch at position 2)

(2) Usually trade time for space
- Preprocessing the pattern (useful if pattern is much smaller than the search string)
- Array of numbers for each position in the pattern

1.1

Sequence alignment

Sequence alignment

(1) Biological sequences (DNA, protein) may only be similar instead of equal
- Sequencing errors
- Mutations (changes, insertions, deletions)
- Comparison across evolutionary changes
- Comparison with functionally related sequences

(2) How to describe similarity?
- Amino acids might be replaced by chemically similar ones
- Long gaps might not be relevant (spliced away, hidden in a functionally inactive area of the protein)

Pairwise alignment

```
GFTGATATAGFT
GGGTGATTAGFT
```
only 6 matches

```
_GFTGATATAGFT
GGGTGAT_TAGFT
```
10 matches when gaps are allowed

```
__GFTGATATAGFT
GGG_TGAT_TAGFT
```
Same number of matches, more gaps

Could be more probable if the F-by-G substition disturbes function more than a gap

Examples for pairwise alignment

human alpha globin vs. human beta globin

human alpha globin vs. leghaemoglobin from yellow lupin (evolutionary and functionally related)

human alpha globin vs. F11G11.2 (unrelated)

Scoring

(1) Numerical function to decide which alignment is the most relevant
- Higher scores = higher probability

(2) Scoring function is a model of the biophysical situation
- Which amino acids are similar with respect to molecular forces?
- Which amino acids could be replaced without changing the protein structure?
- Which amino acids are less relevant for the overall structure (i.e. which are not involved in an active binding domain)?

(3) Scores can be derived from biological observations
- Depending on the frequency of pairs of characters occurring in alignments that are biologically „approved" as relevant
- Sort of „machine learning"

Scoring scheme

(1) Score
- $\sum s(a_i,b_i)$ where s(a,b) is the substitution matrix entry for the amino acids a and b
- The substitution matrix contains the degree of similarity of the two amino acids with respect to their effect on the protein function

(2) Gap penalty
- Fixed value for each gap position (additive)
- Penalizes long gaps
- Better approximation: fixed value for first gap character and smaller addition for every further one

(3) Simple algorithm
- Enumerate all possible alignments and calculate the score

Substitution matrices

(1) Matrix contains likelihood p of each aligned pair compared to random appearance (frequency q)
- log $(p_{ab}/q_a q_b)$ scaled and rounded to the nearest integer

BLOSUM50 matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5																			
R	-2	7																		
N	-1	-1	7																	
D	-2	-2	2	8																
C	-1	-4	-2	-4	13															
Q	-1	1	0	0	-3	7														
E	-1	0	0	2	-3	2	6													
G	0	-3	0	-1	-3	-2	-3	8												
H	-2	0	1	-1	-3	1	0	-2	10											
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5										
L	-2	-3	-4	-4	-2	-2	-3	-2	-3	2	5									
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6								
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7							
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8						
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10					
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5				
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5			
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15		
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

The problem of optimum alignment

(1) The scores are only a crude approximation
- We do not know the internal forces and mechanisms
- Co-evolution is also an effect when scoring evolutionary conservation
- We treat amino acids independently

(2) The number of possible alignments is exponential with the length of the sequence!
- It is not possible to just go through all the possible alignments, score them, and keep the best one.

(3) How do we find out where to put gaps
- The solution is a special algorithm that can do this provided the score is additive
- Additive score means that the scores for all pairs of amino acids or amino acids with gaps are added up
- An additive score is not able to model neighbour effects, such as „A" is less probable if the neighbour is „T".

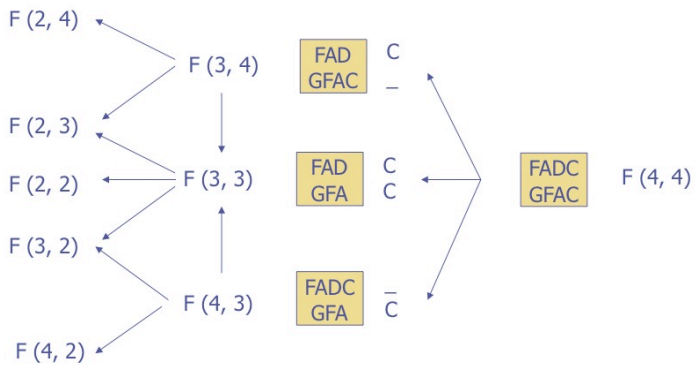
Dynamic programming

(1) Best alignment can be calculated based on the best alignments of prefixes of the two strings
- Uses additive property of score
- Let F (i, j) be the score of the best alignment between a[0..i-1] and b[0..j-1]

(2) Three possible cases
- s: substitution matrix, d: gap penalty
- a[i-1] and b[j-1] match:
  F (i,j) = F (i-1, j-1) + s (a[i-1], b[j-1])
- a[i-1] is aligned to a gap:
  F (i,j) = F (i-1, j) - d
- b[j-1] is aligned to a gap:
  F (i,j) = F (i, j-1) - d

(3) Iteratively calculate F as the maximum of the three cases

Example



Needleman-Wunsch algorithm

```
for (i in 0..M) F [i][0] = -i * d;
for (j in 0..N) F [0][j] = -j * d;
for (i in 1..M)
        for (j in 1..N)
                F [i][j] = max (
                        F [i-1][j-1] + s (a [i-1], b[j-1]),
                        F [i-1][ j ] - d,
                        F [i]  [j-1] - d);
```

a = HEAGAWGHEE
b = PAWHEAE

		H	E	A	G	A	W	G	H	E	E
	0	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80
P	-8	-2	-9	-17	-25	-33	-41	-49	-57	-65	-73
A	-16	-10	-3	-4	-12	-20	-28	-36	-44	-52	-60
W	-24	-18	-11	-6	-7	-15	-5	-13	-21	-29	-37
H	-32	-14	-18	-13	-8	-9	-13	-7	-3	-11	-19
E	-40	-22	-8	-16	-16	-9	-12	-15	-7	3	-5
A	-48	-30	-16	-3	-11	-11	-12	-12	-15	-5	2
E	-56	-38	-24	-11	-6	-12	-14	-15	-12	-9	1

Example: global alignment

F(4,1):

F(4,0) = - 32 d = - 8
F(3,1) = - 17 d = - 8
F(3,0) = - 24 s(P,G) = - 2;

F(4,1) = max(- 40,- 25,- 26) = - 25

F(6,3):

F(5,3) = - 15 d = - 8
F(6,2) = - 28 d = - 8
F(5,2) = - 20 s(W,W) = 15

F(6,3) = max(- 5,- 36,- 23) = - 5

Optimal global alignment:

HEAGAWGHE -E
- - P -AW- HEAE

		H	E	A	G	A	W	G	H	E	E
	0	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80
P	-8	-2	-9	-17	-25	-33	41	-49	-57	-65	-73
A	-16	-10	-3	-4	-12	-20	-28	-36	-44	-52	-60
W	-24	-18	-11	-6	-7	-15	-5	-13	-21	-29	-37
H	-32	-14	-18	-13	-8	-9	-13	-7	-3	-11	-19
E	-40	-22	-8	-16	-16	-9	-12	-15	-7	3	-5
A	-48	-30	-16	-3	-11	-11	-12	-12	-15	-5	2
E	-56	-38	-24	-11	-6	-12	-14	-15	-12	-9	1

Recovering the alignment

(1) Backtracking from F(M, N)

- Going back to the cell from which the maximum was calculated
- Depending on the three cases, add either a gap on one side or a matched pair
- Either use recursion or build backwards
- Exercise: write down the algorithm

(2) Multiple solutions possible

- Whenever maximum of the three choices is not unique

Overlap matches

(1) No penalty for gaps at the beginning or end

- When only overlaps are assumed
- For example, when assembling DNA fragments (shotgun sequencing)

(2) Small algorithmic differences

- Initialization of top and left border to 0 (no penalty)
- Take the maximum from the right and bottom border (instead of the bottom-right corner) to find the best partial match
- Exercise: write down the modified algorithm

Local alignment (Smith-Waterman)

```
for (i in 0..M) F [i][0] = 0;
for (j in 0..N) F [0][j] = 0;
for (i = 1; i <= length (a); i++)
        for (j = 1; j <= length (b); j++)
                F (i, j) = max (0,
                        F (i-1, j-1) + s (a[i-1], b[j-1]),
                        F (i-1, j) – d),
                        F (i, j-1) – d));
```

Alignment ends in maximum F

Regard only local matches with positive scores (Random s(a,b) must be negative, at least one must be positive)

Optimal local alignment: AWGHE / AW -HE

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	5	0	0	0	0	0
W	0	0	0	0	2	0	20	12	4	0	0
H	0	10	2	0	0	0	12	18	22	14	6
E	0	2	16	8	0	0	4	10	18	28	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

Repeated local alignment

```
F (0, j) = 0;
for (i = 1; i <= length (a); i++)
        F (i, 0) = max (F (i-1, 0), F (i-1, j) – T);
        for (j = 1; j <= length (b); j++)
                F (i, j) = max (F(i, 0),
                        F (i-1, j-1) + s (a[i], b[j]),
                        F (i-1, j) – d),
                        F (i, j-1) – d));
```

Positive threshold T (= 20)

Asymmetric

Going back from F(n+1,0)

HEAGAWGHEE
HEA AW -HE

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	1	1	1	1	1	3	9
P	0	0	0	0	1	1	1	1	1	3	9
A	0	0	0	5	1	6	1	1	1	3	9
W	0	0	0	0	2	1	21	13	5	3	9
H	0	10	2	0	1	1	13	19	23	15	9
E	0	2	16	8	1	1	5	11	19	29	21
A	0	0	8	21	13	6	1	5	11	21	28
E	0	0	6	13	18	12	4	1	5	17	27

Affine gap scores

(1) Lower penalty for longer gaps

- d = initial gap penalty, e = penalty for each extension

(2) Dynamic programming algorithm with $O(n^2)$ possible

- M (i, j): best score given that a[i] and b[j] match
- Ia (i, j): best score given that a[i] is matched to a gap
- Ib (i, j): best score given that b[j] is matched to a gap
- M (i, j) = max (M (i-1, j-1) + s (a[i], b[j]),
  Ia (i-1, j-1) + s (a[i], b[j]),
  Ib (i-1, j-1) + s (a[i], b[j]))
- Ia (i, j) = max (M (i-1, j) – d,
  Ia (i-1, j) – e);
- Ib (i, j) = max (M (i, j-1) – d,
  Ib (i, j-1) – e);