Searching (Sub-)Strings

Ulf Leser

This Lecture

- Exact substring search
  - Naïve
  - Boyer-Moore
- Searching with profiles
  - Sequence profiles
  - Ungapped approximate search
  - Statistical evaluation of search results

„Searching Strings“ (aka Pattern Matching)

- Exact matching
  - Given strings s and t: Find all occurrences of s in t
  - Given S and t: Find all occurrences of any s∈S in t
- Approximate matching
  - Given s and t: Find all approximate occurrences of s in t
    - With or without gaps? With or without specific replacement scores?
  - Given s and t: Find s’, t’ such that s’ similar to t’ and s’ is a substring of s and t’ is a substring of t
  - Given s and T
    - Find all t∈T that are similar to s
    - Find all t∈T containing a t’ similar to a s’ contained in s
- Many more variants …

Strings

- A string (or sequence) S is an ordered list of characters from an alphabet Σ
  - |S| is the length of S
  - S[i] is the character at position i in S
  - S[i..j] is the substring from position i to position j in S
  - S[i..j] is an empty string if i > j
  - S[1..i] is a prefix of S ending at position i
  - S[i..] is a suffix of S starting at position i
- Alphabet
  - Usually: Σ={A, C, G, T}
  - Often, we need blanks: Σ'={A, C, G, T, _}
- Lower/upper case: S may denote a set of strings, or a sequence of characters (a string)

Exact Matching

- Given P, T with |P| << |T|
- Find all occurrences of P in T
- Example of application: Restriction enzymes
  - Cut at precisely defined sequence motifs of length 4-10
  - Are used to generate fragments (for later sequencing)
  - Example: Eco RV - GATATC

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaataaattaaaaataatggaacatggcacattttcctaaactcttcacagattgctaatgat
tattaattaaagaataaatgttataatttttatggtaacggaatttcctaaaatattaattcaagcaccatggaatgcaaataagaaggactctgttaattggtactat
tcaactcaatgcaagtggaactaagttggtattaatactctttttacatatatatgtagttattttaggaagcgaaggacaatttcatctgctaataaagggattacga
aaaactttttaataacaaagttaaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaatttttctttttgtattac
ttcaatgaaggtaacgcaacaagattagagtatatatggccaataaggtttgctgtaggaaaattattctaaggagatacgcgagagggcttctcaaatttattcagaga
tggatgtttttagatggtggtttaagaaaagcagtattaaatccagcaaaactagaccttaggtttattaaagcgaggcaataagttaattggaattgtaaaagatatct
aattcttcttcatttgttggaggaaaactagttaacttcttaccccatgcagggccatagggtcgaatacgatctgtcactaagcaaaggaaaatgtgagtgtagacttt
aaaccattttttattaatgactttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatatatta
ttcagttacatagagattatagctggtctattcttagttataggacttttgacaagatagcttagaaaataagattatagagcttaataaaagagaacttcttggaatta
gctgcctttggtgcagctgtaatggctattggtatggctccagcttactggttaggttttaatagaaaaattccccatgattgctaattatatctatcctattgagaaca
acgtgcgaagatgagtggcaaattggttcattattaactgctggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaaatattg
ctaactggtgctggtagggtttggggattggattatttcctctacaagaaatttggtgtttactgatatccttataaataatagagaaaaaattaataaagatgatat
```

How to do it?

- The straight-forward way (naïve algorithm)
  - We use two counter: t, p
  - One (outer, t) runs through T
  - One (inner, p) runs through P
  - Compare characters at position T[t+p] and P[p]

```
for t = 1 to |T| - |P| + 1
        match := true;
        p := 1;
        while ((match) and (p <= |P|))
                if (T(t + p - 1) <> P(p)) then
                        match := false;
                else
                        p := p + 1;
        end while;
        if (match) then
                -> OUTPUT t
end for;
```

Examples

Typical case

```
T   ctgagatcgcgta
P   gagatc
     gagatc
      gagatc
       gagatc
        gagatc
         gatatc
          gatatc
           gatatc
```

Worst case

```
T   aaaaaaaaaaaaaa
P   aaaaat
     aaaaat
      aaaaat
       aaaaat
           ...
```

- How many comparisons do we need in the worst case?
  - t runs through T
  - p runs through the entire P for every value of t
  - Thus: |P|*|T| comparisons
  - Indeed: The algorithm has worst-case complexity O(|P|*|T|)

Other Algorithms

- Exact substring search has been researched for decades
  - Boyer-Moore, Z-Box, Knuth-Morris-Pratt, Karp-Rabin, Shift-AND, …
  - All have WC complexity O(|P| + |T|)
  - Real performance depends much on size of alphabet and composition of strings (most have their strength in certain settings)
- In practice, our naïve algorithm is quite competitive for random strings and non-trivial alphabets (e.g., DNA)
- But we can do better: Boyer-Moore
  - We present a simplified form
  - BM is among the fastest algorithms in practice
- Note: Much better performance possible if T maybe preprocessed (up to O(|P|))

This Lecture

- Exact substring search
  - Naïve
  - Boyer-Moore
- Searching with profiles
  - Sequence profiles
  - Ungapped approximate search
  - Statistical evaluation of search results

Boyer-Moore Algorithm

- R.S. Boyer /J.S. Moore. „A Fast String Searching Algorithm“, Communications of the ACM, 1977
- Main idea
  - Again, we use two counters (inner loop, outer loop)
  - Inner loop runs from right-to-left
  - If we reach a mismatch, we know
    - The character in T we just haven't seen
      - This is captured by the bad character rule
    - The suffix in P we just have seen
      - This is captured by the good suffix rule
- Use this knowledge to make longer shifts in T

Bad Character Rule

- Setting 1
  - We are at position t in T and compare right-to-left
  - Let i by the position of the first mismatch in P
    - We saw n-i+1 matches before
  - Let x be the character at the corresponding pos (t-n+i) in T
  - Candidates for matching x in P
    - Case 1: x does not appear in P at all – we can move t such that t-n+i is not covered by P anymore

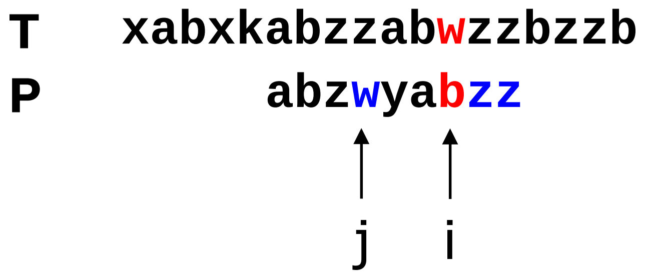
```
T    xabxfabzzabxzzbzzb
P    abwxyabzz
            ←
```

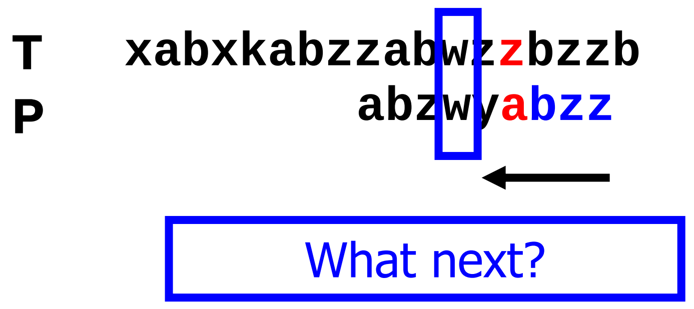
```
T    xabxfabzzabwzzbzzb
P         abwxyabzz
                ←
```

What next?

Bad Character Rule 2

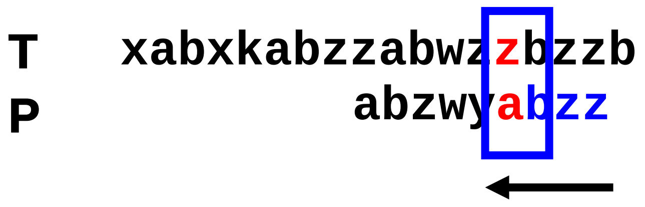
- Setting 2
  - We are at position t in T and compare right-to-left
  - Let i by the position of the first mismatch in P
  - Let x be the character at the corresponding pos (t-n+i) in T
  - Candidates for matching x in P
    - Case 1: x does not appear in P at all
    - Case 2: Let j be the right-most appearance of x in P and let j<i – we can move t such that j and i align





Bad Character Rule 3

- Setting 3
  - We are at position t in T and compare right-to-left
  - Let i by the position of the first mismatch in P
  - Let x be the character at the corresponding pos (t-n+i) in T
  - Candidates for matching x in P
    - Case 1: x does not appear in P at all
    - Case 2: Let j be the right-most appearance of x in P and let j<i
    - Case 3: As case 2, but j>i – we need some more knowledge



Preprocessing 1

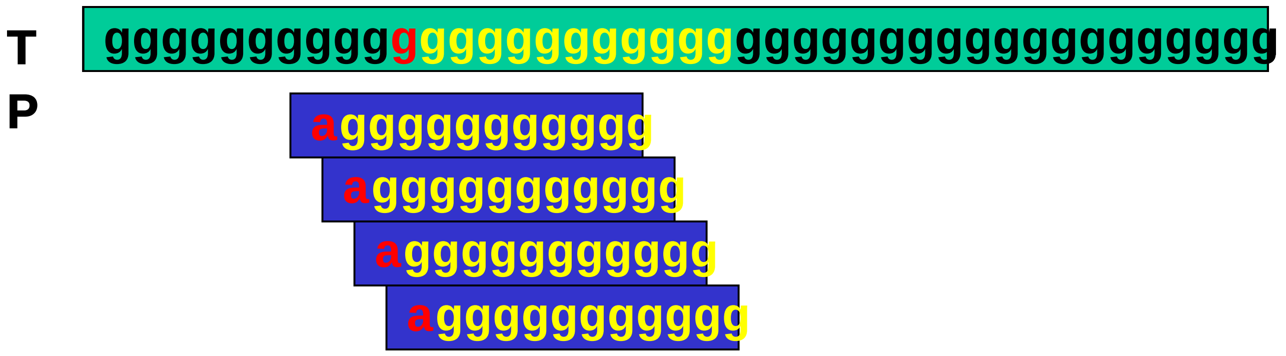
- In case 3, there are some “x” right from position i
  - For small alphabets (DNA), this will almost always be the case
  - Thus, this case 3 is the usual one
- These are irrelevant – we need the right-most x left of i
- This can (and should!) be pre-computed
  - Build a two-dimensional array A[|Σ|,|P|]
  - Run through P from left-to-right (pointer i)
  - If character c appears at position i, set all A[c,j]:=i for all j>=i
  - Needs time (complexity?), but negligible because
    - P is small
    - Complexity is independent from T
- Array: Constant lookup, needs some space (lists …)

(Extended) Bad Character Rule

- Simple, effective for larger alphabets
- For random DNA, average shift-length is 4
  - Expected distances to the next match using EBCR
  - Thus, n# of comparisons down to |P|*|T|/4
- Worst-Case complexity does not change
  - Why?

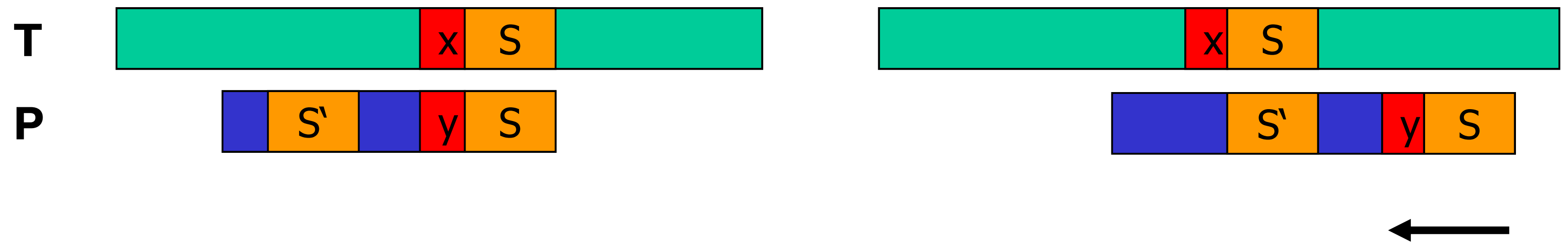
(Extended) Bad Character Rule

- Simple, effective for larger alphabets
- For random DNA, average shift-length should be 4
  - Thus, n# of comparisons down to |P|*|T|/4
- Worst-Case complexity does not change
  - Why?



Good-Suffix Rule

- Recall: If we reach a mismatch, we know
  - The character in T we just haven't seen
  - The suffix in P we just have seen
- Good suffix rule
  - We have just seen some matches in P (S)
  - Where else does S appear in P?
  - If we know the right-most appearance S' of S in P, we can immediately align S' with the current match in T
  - If S does not appear once more in P, we can shift t by |P|

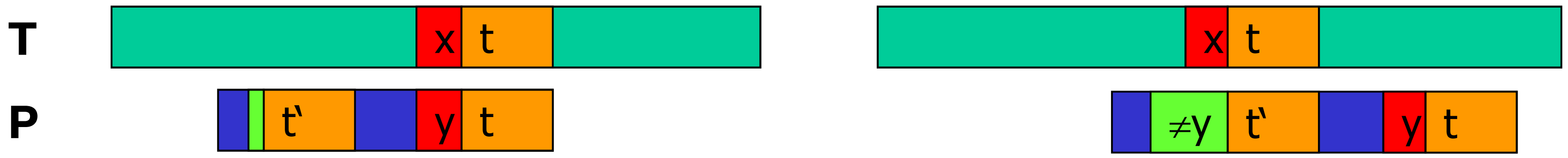


Good-Suffix Rule – One Improvement

- Actually, we can do a little better
- Not all S‘ are of interest to us

Good-Suffix Rule – One Improvement

- Actually, we can do a little better
- Not all S‘ are of interest to us

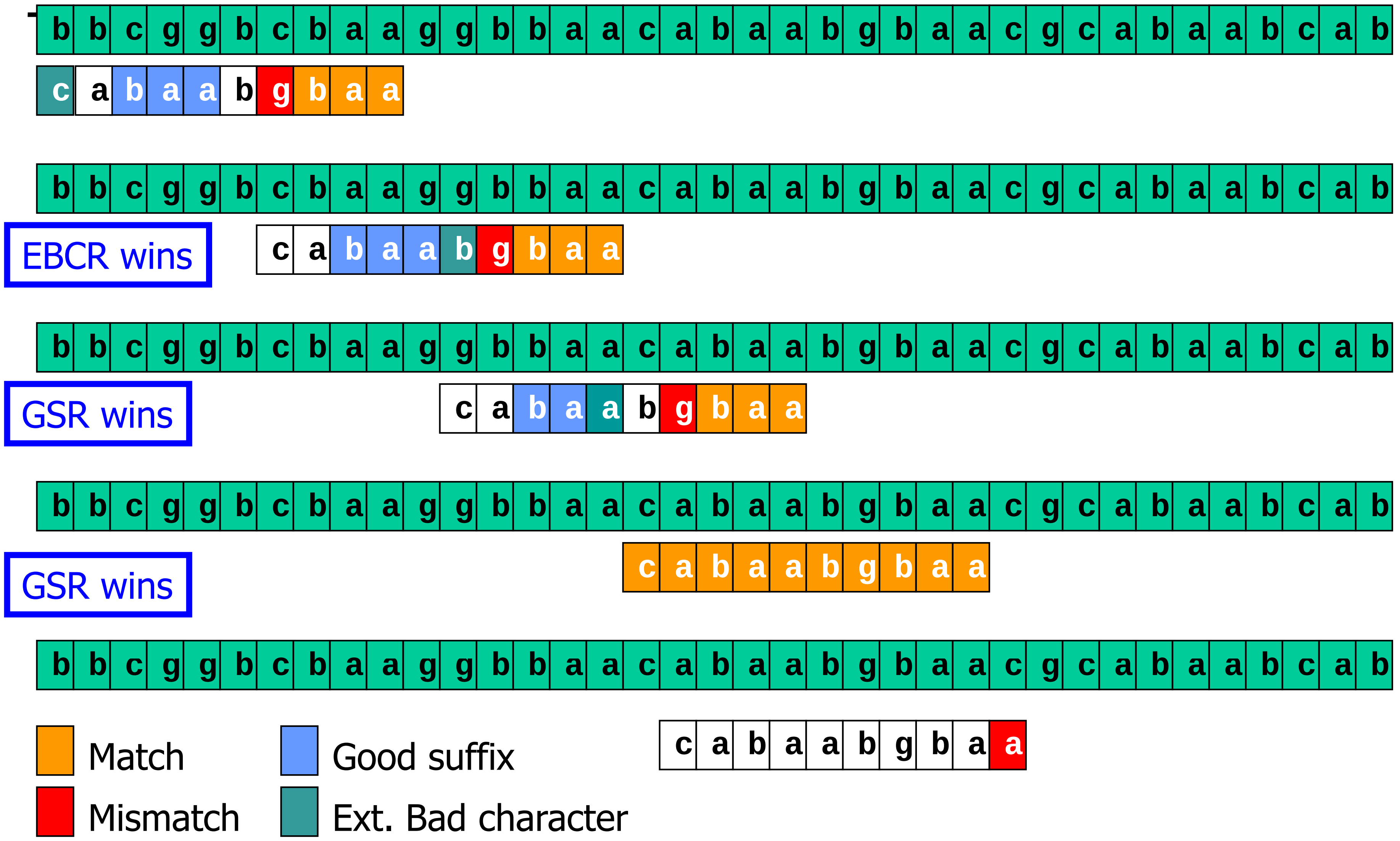


- We only need S‘ whose next character to the left is not y
- Why don‘t we directly require that this character is x?
  - Of course, this could be used for further optimization

Concluding Remarks

- Preprocessing 2
  - For the GSR, we need to find all occurrences of all suffixes of P in P
  - This can be solved using our naïve algorithm for each suffix
  - Or, more complicated, in linear time (not this lecture)
- WC complexity of Boyer-Moore is still O(|P|*|T|)
  - But average case is sub-linear
  - WC complexity can be reduced to linear (not this lecture)
- Faster variants
  - Often, using the GSR does not pay-off
  - BM-Horspool: Instead of looking at the mismatch character x, always look at the symbol in T aligned to the last position of P
    - Generates longer shifts on average (i is maximal)

Example


b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b
c a b a a b g b a a
EBCR wins
GSR wins
GSR wins
Match
Mismatch
Good suffix
Ext. Bad character

This Lecture

- Exact substring search
  - Naïve
  - Boyer-Moore
- Searching with profiles
  - Splicing
  - Position Specific Weight Matrices
  - Likelihood scores

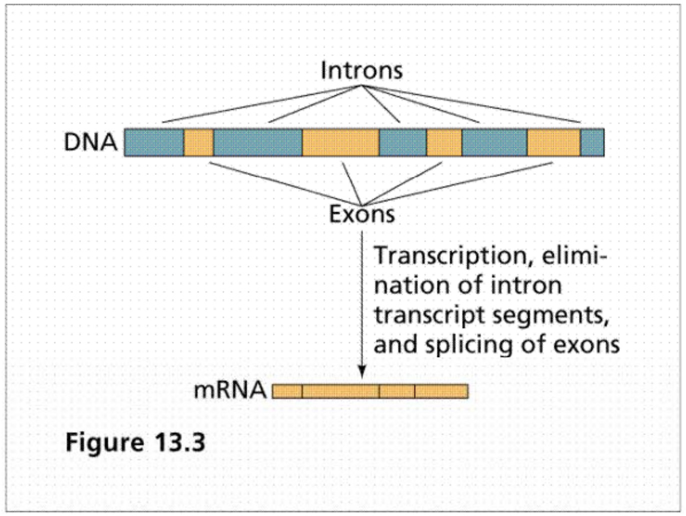
Approximate Search (First Instantiation)

- Requiring an exact match is too strict in many applications
  - And in most bioinformatics applications
- More often, one is interested in matches similar to P
  - Or can describe P only vaguely
- Many definitions of "similar" are possible

- For now: Searching with Position Specific Weight Matrices
  - Also called profiles
  - Powerful tool for many bioinformatics applications
  - We develop the idea using an example taken from Spang et al. "Genome Statistics", Lecture 2003/2005, FU Berlin

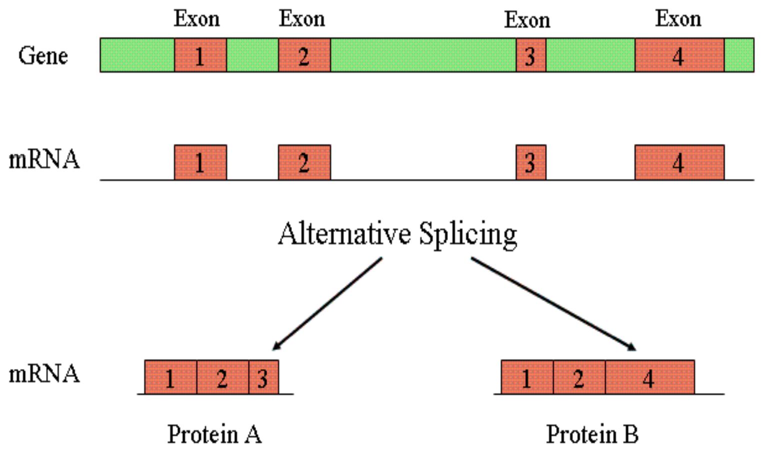
Splicing

- Not all DNA of a “gene” are translated into amino acid
- Splicing: Removal of introns
- Alternative splicing: Removal of (some) exons

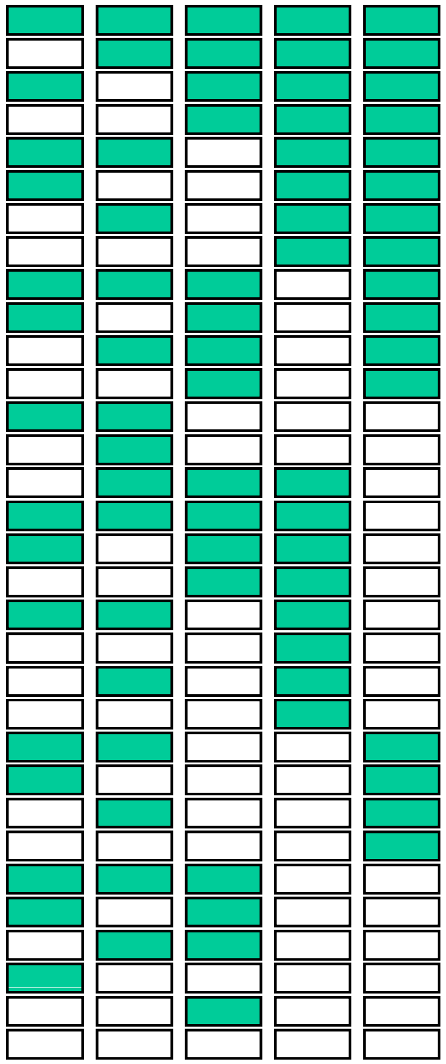


Introns
DNA
Exons
Transcription, elimination of intron transcript segments, and splicing of exons
mRNA

Figure 13.3



Diversity



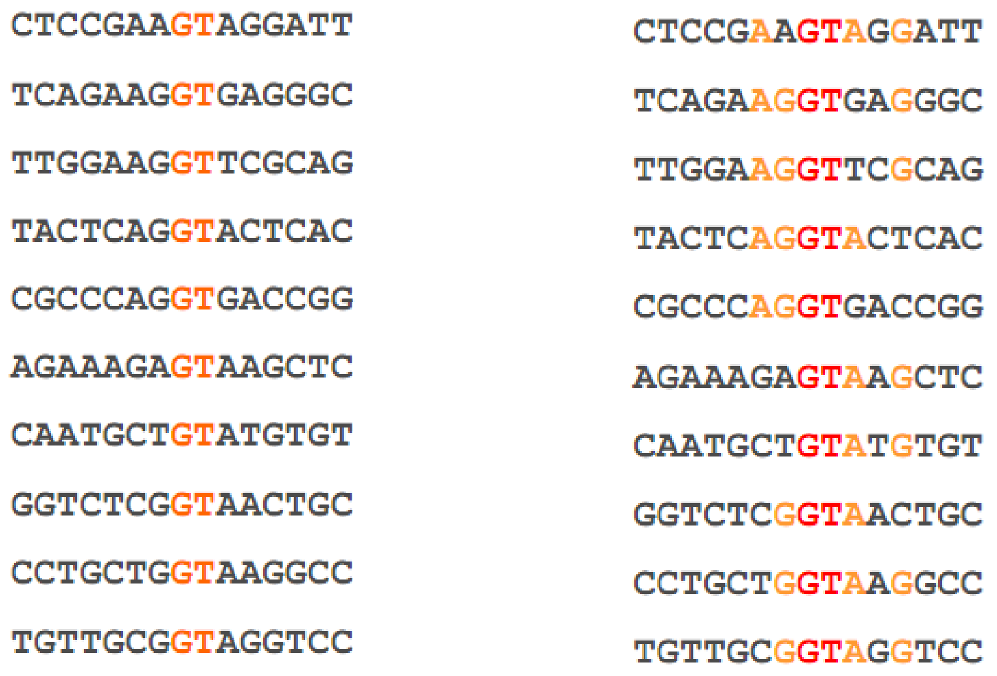
- From a gene with n exons, alternative splicing can create 2^n-1 proteins
- Example: Troponin T (muscle protein)
  - 18 exons
  - 64 different isoforms
  - 10 exons present in all isoforms

- Source: Eurasnet, „Alternative Splicing"

Recognizing Splice Sites

- A special enzyme (spliceosome) very precisely recognizes exon-intron boundaries in mRNA
- To this end, it scans the sequences and is triggered by certain motifs
- How are these motifs characterized? Can we find them?
  - Very often, introns start with GT (GU) and end with AG
  - But that is not specific enough - why?
  - In random sequences, we expect a GT (AT) at every 16th position
  - Thus, the average distance between a GT and an AT is 16, and we find such pairs very often
  - But: Introns typically are larger than 100 bases

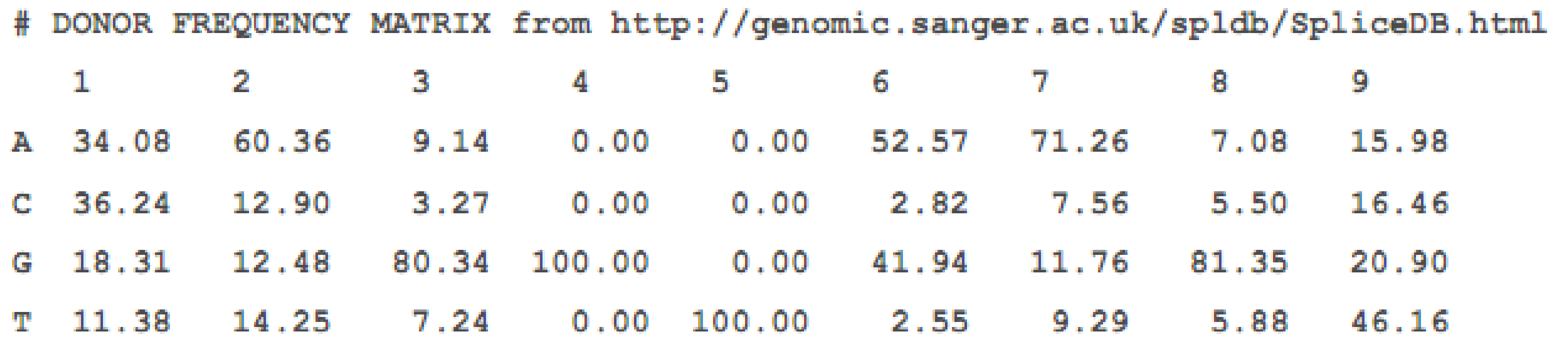
Context of a Splice Site



```
CTCCGAAGTAGGATT     CTCCGAAGTAGGATT
TCAGAAGGTGAGGGC     TCAGAAGGTGAGGGC
TTGGAAGGTTCGCAG     TTGGAAGGTTCGCAG
TACTCAGGTACTCAC     TACTCAGGTACTCAC
CGCCCAGGTGACCGG     CGCCCAGGTGACCGG
AGAAAGAGTAAGCTC     AGAAAGAGTAAGCTC
CAATGCTGTATGTGT     CAATGCTGTATGTGT
GGTCTCGGTAACTGC     GGTCTCGGTAACTGC
CCTGCTGGTAAGGCC     CCTGCTGGTAAGGCC
TGTTGCGGTAGGTCC     TGTTGCGGTAGGTCC
```

- Observing real splice sites, we find no crisp context
- But: columns are not composed at random either
- How can we capture this knowledge?

Position-Specific Weight Matrices



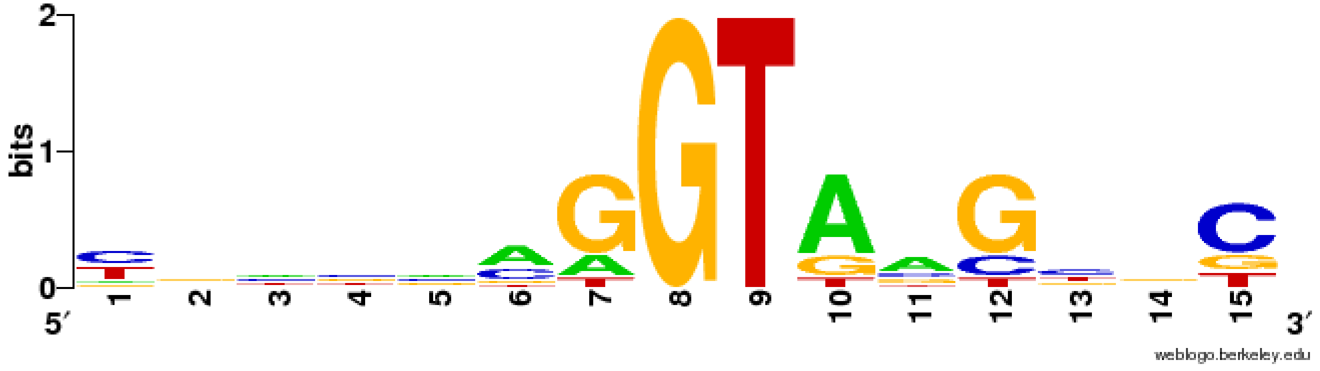
```
# DONOR FREQUENCY MATRIX from http://genomic.sanger.ac.uk/spldb/SpliceDB.html
     1        2        3       4        5        6        7        8        9
A  34.08    60.36     9.14    0.00     0.00    52.57    71.26     7.08    15.98
C  36.24    12.90     3.27    0.00     0.00     2.82     7.56     5.50    16.46
G  18.31    12.48    80.34  100.00     0.00    41.94    11.76    81.35    20.90
T  11.38    14.25     7.24    0.00   100.00     2.55     9.29     5.88    46.16
```

- Count in every column the frequencies of all bases
- Store the relative frequencies in an array of size |P|*|Σ|
  - With |P| being the size of the context around the splice sites
- At "GT", all values except one are 0% and one is 100%
  - Actually, GT is not perfectly conserved in real sequences
- In random sequences, all values should be 25%

Vizualization: Sequence Logos

- Very popular
- Based on information content of each base at each position
  - Which, in turn, is based on the entropy of the columns

```
CTCCGAAGTAGGATT
TCAGAAGGTGAGGGC
TTGGAAGGTTCGCAG
TACTCAGGTACTCAC
CGCCCAGGTGACCGG
AGAAAGAGTAAGCTC
CAATGCTGTATGTGT
GGTCTCGGTAACTGC
CCTGCTGGTAAGGCC
TGTTGCGGTAGGTCC
```

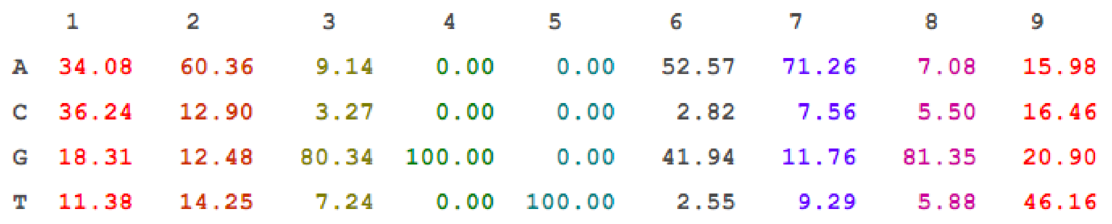


Scoring with a PSWM

- Eventually, we want to find potential splice sites in a genome G (e.g. to do gene prediction)
- We need a way to decide, given a sequence S and a PSWM A (both of the same length): Does S match A?
  - We want to assign a score to S given A
  - Knowing this, we can score all subsequences of length |A| in G
  - Subsequences above a given threshold are considered candidates
- We give this question a probabilistic interpretation
  - Assume, for each column, a dice which four faces; each face is thrown with the relative frequency as given in A for this column
  - How high is the probability that this dice generates S?

Examples

- In random sequences, all values in A are 25%, and all possible S would get the same probability: ¼$^{|S|}$
- But



	1	2	3	4	5	6	7	8	9
A	34.08	60.36	9.14	0.00	0.00	52.57	71.26	7.08	15.98
C	36.24	12.90	3.27	0.00	0.00	2.82	7.56	5.50	16.46
G	18.31	12.48	80.34	100.00	0.00	41.94	11.76	81.35	20.90
T	11.38	14.25	7.24	0.00	100.00	2.55	9.29	5.88	46.16

1. P (AAGGTACGT) ≈ 0.34*0.6*0.8*1*1*0.53*0.71*0.81*0.46 =0.023
2. P (CCCGTCCCC) ≈ 0.36*0.13*0.03*1*1*0.03*0.08*0.05*0.16 =2.7e-08
3. P (CTGGTCCGA) ≈ 0.36*0.14*0.8*1*1*0.03*0.08*0.81*0.16 =1.25e-05
4. P (TACCTCCGT) = 0

- 1st sequence (S) matches A much better than the others do

This Lecture

- Exact substring search
  - Naïve
  - Boyer-Moore
- Searching with profiles
  - Splicing
  - Position Specific Weight Matrices
  - Likelihood scores

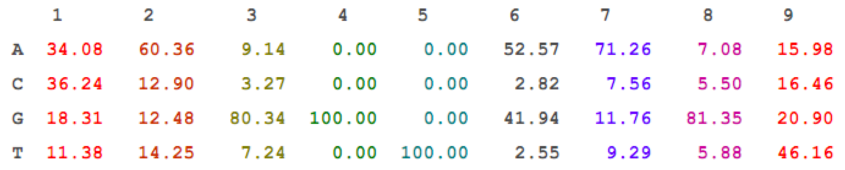
I am not Convinced (yet)

- Is S actually a match for A?
- Observations
  - The first S from the previous slide is about as good as it can get: The best possible sequence would get a score of 0.025 (compared to 0.023)
  - If S is not a splice site, it is an “ordinary” sequence. How likely is it that S is generated under this “zero model”?
    - “Zero model” means: Equal probability for all bases
    - p(S|”zero”) = $¼^9$ ~ 3.8E-6
    - Thus, is it much more likely (app. 6000 times more likely) that S was generated under the “A model” than that is was generated under the “zero model”

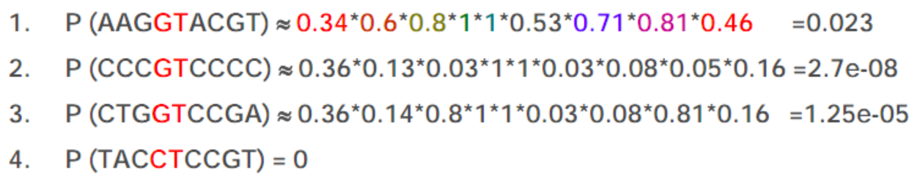
Likelihood (Odds) Ratios

- Given two models A, Z. The likelihood ratio score s of a sequence S is the ratio of p(S|A) / p(S|Z)

  - s(AAGGTACGT) ~ 6000
  - S(CCCGTCCCC) ~ 140
  - s(CTGGTCCGA) ~ 3
  - S(TCCGTCCCC) < 1



	1	2	3	4	5	6	7	8	9
A	34.08	60.36	9.14	0.00	0.00	52.57	71.26	7.08	15.98
C	36.24	12.90	3.27	0.00	0.00	2.82	7.56	5.50	16.46
G	18.31	12.48	80.34	100.00	0.00	41.94	11.76	81.35	20.90
T	11.38	14.25	7.24	0.00	100.00	2.55	9.29	5.88	46.16



1. P (AAGGTACGT) ≈ 0.34*0.6*0.8*1*1*0.53*0.71*0.81*0.46 =0.023
2. P (CCCGTCCCC) ≈ 0.36*0.13*0.03*1*1*0.03*0.08*0.05*0.16 =2.7e-08
3. P (CTGGTCCGA) ≈ 0.36*0.14*0.8*1*1*0.03*0.08*0.81*0.16 =1.25e-05
4. P (TACCTCCGT) = 0

- Also called odds score

Matching with a PSWM

- Given G, A, Z: find all S in G with s(S)>t
- Straight-forward: Compute all S of length |A|, compute s(S) for each
  - This requires |G|*|A| divisions and multiplications
  - Divisions can be saved easily (how?)
- Can we do better?
  - Not easily
  - Trick: The number of match-situations are limited. Pre-compute all possible matches between q-grams and lookup values during the scan

More Stable and Faster

- Values get quite small (close to 0) for longer A
- This yields problems with numeric stability in programs
- Better: Compute log-likelihood score s'=$\log_2$(s)
  - Also faster: Replace multiplication with addition

$$s'(S) = \log\left(\frac{p(S \mid A)}{p(S \mid Z)}\right) = \log\left(\frac{p(S_1 \mid A_1) * \ldots * p(S_n \mid A_n)}{p(S_1 \mid Z_1) * \ldots * p(S_n \mid Z_n)}\right)$$

$$= \log\left(\frac{p(S_1 \mid A_1)}{p(S_1 \mid Z_1)}\right) + \ldots + \log\left(\frac{p(S_n \mid A_n)}{p(S_n \mid Z_n)}\right)$$

Beware

- Assume a perfectly conserved motif of length 8
  - The chance for a given S to match is 0.000015 – low
  - But |G|=3.000.000.000
  - Only by change, we will have ~45000 matches of S in G
- For PSWM, the chances for finding false hits depend on the setting of the threshold t
  - Higher t: Stricter search, less false hits, but may incur misses
  - Lower t: Less strict, less misses, but many false hits
- A match is only an hypothesis that needs further analysis
  - By additional knowledge (e.g.: is S part of a gene?)
  - By experimentation (can we find an isoform spliced at S)?

Pattern Matching

- We discussed exact matching and matching with a PSWM
- But motifs also may look quite differently
  - Motifs (domains) in protein sequences
  - Some important positions and much “glue” of unspecified length
  - Pattern here may be: [AV].*[QSA]FGK.*[IV]…
  - Which positions in S should we compare to which columns in P?
  - How can we compute P given S_1-S_6?

S_1: M---AIDE----NKQKALAAALGQ--KQFGKGSIMRLGEDR-SMDVETISTGSLSLDI
S_2: MSDN--------KKQQALELALKQI-KQFGKGSIMKLGDG-ADHSIEAIPSGSIALDI
S_3: M----AINTDTSGKQKALTMVLNQIERSFGKGAIMRLGDA-TRMRVETISTGALTLDL
S_4: M-----------DRQKALEAAVSQ--RAFGKGSIM-LGGKD---ETEVVSTRILGLDV
S_5: M------DE---NKKRALAAALGQI-KQFGKGAVMRMGDHE-RQAIPAISTGSLGLDI
S_6: MD---------------------K-EKSFGKGSIMRMGEE-VVEQVEVIPTGSIA---

Further Reading

- On string matching algorithms
  - Gusfield
- On sequence logos and TFBS-identification
  - Christianini & Hahn, chapter 10
  - Merkl & Waack, chapter 10