Searching (Sub-)Strings
• Normally, here is a short repetition of the last lecture
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 \( s \in S \) 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 \( t \in T \) that are similar to s
    - Find all \( t \in T \) 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 $\Sigma$
  - $|S|$ 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: $\Sigma = \{A, C, G, T\}$
  - Often, we need blanks: $\Sigma' = \{A, C, G, T, _\}$

- 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
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

<table>
<thead>
<tr>
<th>T</th>
<th>ctgagatcgctga</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>gagatc</td>
</tr>
<tr>
<td></td>
<td>gagatc</td>
</tr>
<tr>
<td></td>
<td>gagatc</td>
</tr>
<tr>
<td></td>
<td>gagatc</td>
</tr>
<tr>
<td></td>
<td>gagatc</td>
</tr>
<tr>
<td></td>
<td>gatatc</td>
</tr>
<tr>
<td></td>
<td>gatatc</td>
</tr>
</tbody>
</table>

Worst case

<table>
<thead>
<tr>
<th>T</th>
<th>aaaaaaaaaaaaaa</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

- 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

- 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

\[
\begin{array}{c}
T \quad \textbf{xabxfabzzabxxzbzzb} \\
\textbf{P} \quad \textbf{abwxyabzz} \\
\end{array}
\]

\[
\begin{array}{c}
T \quad \textbf{xabxfabzzabwzzbzzb} \\
\textbf{P} \quad \textbf{abwxyabzz} \\
\end{array}
\]

What next?
Bad Character Rule 2

- Setting 2
  - We are at position $t$ in $T$ and compare right-to-left
  - Let $i$ be 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

What next?
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 $A[|\Sigma|,|P|]$
  - Run through $P$ from left-to-right (pointer $i$)
  - If character $c$ appears at position $i$, set all $A[c,j] := i$ for all $j \geq i$
  - 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 should be 3
  - Thus, n# of comparisons down to $|P|*|T|/3$
- Worst-Case complexity does not change
  - Why?
(Extended) Bad Character Rule

- Simple, effective for larger alphabets
- For random DNA, **average shift-length should be 3**
  - Thus, n# of comparisons down to \( |P| \times |T| / 3 \)
- 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 \( O(|P| \cdot |T|) \)
  - 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

```
bbccggbbcbaaggbbbaaccabaaabgbbaaccgcabaaabcab

cababgbaba

bbccggbbcbaaggbbbaaccabaaabgbbaaccgcabaaabcab

EBCR wins
cababgbaba

bbccggbbcbaaggbbbaaccabaaabgbbaaccgcabaaabcab

GSR wins
cababgbaba

bbccggbbcbaaggbbbaaccabaaabgbbaaccgcabaaabcab

GSR wins
cababgbaba

bbccggbbcbaaggbbbaaccabaaabgbbaaccgcabaaabcab
```

- **Match**
- **Good suffix**
- **Mismatch**
- **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
Diversity

• From a gene with n exons, alternative splicing can create $2^n$ 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 16\textsuperscript{th} 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

| CTCCGAAGTACGTT | CTCCGAAGTACGTT |
| TCAGAAGTGACGAGG | TCAGAAGTGACGAGG |
| TTGGAGTGCTCGAC | TTGGAGTGCTCGAC |
| TACTCAGGTACTCAC | TACTCAGGTACTCAC |
| CGCCAGTGACCGG | CGCCAGTGACCGG |
| AGAAAGAGTAAGCTG | AGAAAGAGTAAGCTG |
| CAATGCTGATGTGT | CAATGCTGATGTGT |
| GTCTCGTGACTGCG | GTCTCGTGACTGCG |
| CCTGCTGTAAGGCC | CCTGCTGTAAGGCC |
| TGTTCGCGTACGTCC | TGTTCGCGTACGTCC |

- 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

<table>
<thead>
<tr>
<th>DONOR FREQUENCY MATRIX from <a href="http://genomic.sanger.ac.uk/spldb/SpliceDB.html">http://genomic.sanger.ac.uk/spldb/SpliceDB.html</a></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>T</td>
</tr>
</tbody>
</table>

- Count in every column the frequencies of all bases
- Store the **relative frequencies** in an array of size $|P| \times |\Sigma|$  
  - 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

CTCCGAAAGTAGGATT
TCAGAAGGTGAGGGC
TTGGAAGGTTCGGCAG
TACTCAGGTACTCAC
CGCCCAGGTGACCGG
AGAAAGAGTAAGCTC
CAATGCTGTATGTGT
GGTCTCGGTAACTGC
CCTGCTGGTAAGGCC
TGTTCGCGTAGGTCC
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: $\frac{1}{4} |S|$
- But

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

1. $P(\text{AAGGTACGT}) \approx 0.34 \times 0.6 \times 0.8 \times 1 \times 1 \times 0.53 \times 0.71 \times 0.81 \times 0.46 \approx 0.023$
2. $P(\text{CCCCGCCCCC}) \approx 0.36 \times 0.13 \times 0.03 \times 1 \times 1 \times 0.03 \times 0.08 \times 0.05 \times 0.16 \approx 2.7 \times 10^{-8}$
3. $P(\text{CTGTCGCG}) \approx 0.36 \times 0.14 \times 0.8 \times 1 \times 1 \times 0.03 \times 0.08 \times 0.81 \times 0.16 \approx 1.25 \times 10^{-5}$
4. $P(\text{TACCTCGGT}) = 0$

- 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
    - \( p(S|\text{"zero"}) = \frac{1}{4^9} \approx 3.8E-6 \)
    - 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 $p(S|A) / p(S|Z)$
  - $s(\text{AAGGTACGT}) \sim 6000$
  - $S(\text{CCCGTCCCC}) \sim 140$
  - $s(\text{CTGGTCCGA}) \sim 3$
  - $S(\text{TCCGTCCCC}) < 1$

- Also called odds score
Matching with a PSWM

- Given G, A, Z: find all S in G with s(S)>t
- Naïve: Compute all S of length |A|, compute s(S) for each
  - This requires |G|*|A| divisions and multiplications
  - Divisions can be saved easily (how?), multiplications remain
- Can we do better?
  - Even if we are not allowed to preprocess G
Observation

\[ s(S) = \frac{p(S \mid A)}{p(S \mid Z)} = \frac{p(S_1 \mid A_1) \cdot p(S_2 \mid A_2) \cdots p(S_n \mid A_n)}{p(S_1 \mid Z_1) \cdot p(S_2 \mid Z_2) \cdots p(S_n \mid Z_n)} = \frac{p(S_1 \mid A_1)}{p(S_1 \mid Z_1)} \cdot \frac{p(S_2 \mid A_2)}{p(S_2 \mid Z_2)} \cdots \frac{p(S_n \mid A_n)}{p(S_n \mid Z_n)} \]

- We can compute \( s(S) \) iteratively: Scan \( G \) base-by-base, and at every step divide a running \( s \) by the score of the base left from the new position of \( S \) and multiply by the score of the new base.

\[
\begin{align*}
G & : \quad \text{ACTGTCGCGAGGTAAGTTTCATG...} \\
s_0 & : \quad \text{ACTGTCGCG} \\
s_1 & : \quad \text{CTGTCGCGA} \\
s_2 & : \quad \text{TGTCGCGAG} \\
\vdots & \\
\end{align*}
\]
More Stable and Faster

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

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

$$= \log\left(\frac{p(S_1 \mid A_1)}{p(S_1 \mid Z_1)}\right) + \ldots + \log\left(\frac{p(S_n \mid A_n)}{p(S_n \mid Z_n)}\right)$$
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 $\sim 45000$ 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 be some misses?
  – Lower $t$: Less strict, less changes to miss anything, but potentially 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)?
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 \(S_1-S_6\)?

\[
\begin{align*}
S_1: \ & M---AIDE----NKQKALAALGQ--QFGKGSIMRLGEDR-SMDVETISTGSLLDI \\
S_2: \ & MSDN--------KKQQALELAKQI-QFGKGSIMKLGDG-ADHSIEAIPSGSTALDI \\
S_3: \ & M-----AINTDTSGKQKALTVMNQIERSFKGAIMRLGDA-TRMRVETISTGALTLDI \\
S_4: \ & M-----------DRQKALEAVS--QAFGKGSIM-LGKD---ETEVSTRILGLDV \\
S_5: \ & M------DE---NKKRALAAALGQI-QFGKGAVMRMDHE-RQAIPAISTGSLGLDI \\
S_6: \ & MD-------------------K-EKSFCKGSIMRMGEE-VVEQVEVPTGSLI---
\end{align*}
\]
Further Reading

- On string matching algorithms
  - Gusfield

- On sequence logos and TFBS-identification
  - Christianini & Hahn, chapter 10
  - Merkl & Waack, chapter 10