## Lecture 2

### Sequence Alignment and Dynamic Programming

#### Module 1: Aligning and modeling genomes

- **Module 1: Computational foundations**
  - Dynamic programming: exploring exponential spaces in poly-time
  - Introduce Hidden Markov Models (HMMs): Central tool in CS
  - HMM algorithms: Decoding, evaluation, parsing, likelihood, scoring

- **This week: Sequence alignment / comparative genomics**
  - Local/global alignment: infer nucleotide-level evolutionary events
  - Database search: scan for regions that may have common ancestry

- **Next week: Modeling genomes / exon / CpG island finding**
  - Modeling class of elements, recognizing members of a class
  - Application to gene finding, conservation islands, CpG islands

---

### Alignment: all species/genes share common ancestry

![Tree of Life](image)

*Slide credit: Serafim Batzoglou*
Genome-wide alignments reveal orthologous segments

Comparative genomics reveals conserved regions

Comparative genomics can reveal functional elements
- For example: exons are deeply conserved to mouse, chicken, fish
- Many other elements are also strongly conserved: exons / regulatory?

Develop methods for estimating the level of constraint
- Count the number of edit operations, number of substitutions and gaps
- Estimate the number of mutations (including estimate of back-mutations)
- Incorporate information about neighborhood: conservation ‘windows’
- Estimate the probability of a constrained ‘hidden state’: HMMs next week
- Use phylogeny to estimate tree mutation rate, or ‘rejected substitutions’
- Allow different portions of the tree to have different rates: phylogenetics

Evolutionary signatures for diverse functions

Protein-coding genes
- Codon Substitution Frequencies
- Reading Frame Conservation

RNA structures
- Compensatory changes
- Silent G-U substitutions

microRNAs
- Shape of conservation profile
- Structural features: loops, pairs
- Relationship with 3'UTR motifs

Regulatory motifs
- Mutations preserve consensus
- Increased Branch Length Score
- Genome-wide conservation

Alignment: Evolution preserves functional elements!

Today’s goal:
How do we actually align two genes?

We can ‘read’ evolution to reveal functional elements

**Goal: Sequence Alignment / Dynamic Programming**

1. **Introduction to sequence alignment**
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments

2. **Introduction to principles of dynamic programming**
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back

3. **DP for sequence alignment**
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry \( \diamond \) prefix alignment score; path \( \diamond \) alignment

4. **Advanced topics: Dynamic Programming variants**
   - Linear-time bounded DP (heuristic). Better than \( O(n^2) \)?
   - Linear-space DP: Four-Russians algorithm. Total time?

---

**Genomes change over time**

```
begin
ACGT CATCA
ACGT GATCA
A X GTG X TCA
AGTG TCA
T AGT GTCA
end
```

**From Bio to CS: Formalizing the problem**

- Define set of evolutionary operations (insertion, deletion, mutation)
  - Symmetric operations allow time reversibility (part of design choice)

- Define optimality criterion (min number, min cost)
  - Impossible to infer exact series of operations (Occam’s razor: find min)

- Design algorithm that achieves that optimality (or approximates it)
  - Tractability of solution depends on assumptions in the formulation

**Formulation 1: Longest common substring**

- Given two possibly related strings \( S_1 \) and \( S_2 \)
  - What is the longest common substring? (no gaps)

**Formulation 2: Longest common subsequence**

- Given two possibly related strings \( S_1 \) and \( S_2 \)
  - What is the longest common subsequence? (gaps allowed)

---

**Goal of alignment: Infer edit operations**

```
begin
ACGT CATCA
?
end
TAGT GTCA
```

---

**Note:** Not all decisions are conflicting (some are both relevant and tractable)
(e.g. Pevzner vs. Sankoff and directionality in chromosomal inversions)
**Formulation 3: Sequence alignment**

- Allow gaps (fixed penalty)
  - Insertion & deletion operations
  - Unit cost for each character inserted or deleted
- Varying penalties for edit operations
  - Transitions (Pyrimidine ⇔ Pyrimidine, Purine ⇔ Purine)
  - Transversions (Purine ⇔ Pyrimidine changes)
  - Polymerase confuses Aw/G and Cw/T more often

<table>
<thead>
<tr>
<th>Score</th>
<th>A</th>
<th>G</th>
<th>T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>+1</td>
<td>-½</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Mismatch(A,G)</td>
<td>-½</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Mismatch(C,T)</td>
<td>-½</td>
<td>+1</td>
<td>-½</td>
<td>+1</td>
</tr>
<tr>
<td>All other operations</td>
<td>-1</td>
<td>-½</td>
<td>+1</td>
<td>+1</td>
</tr>
</tbody>
</table>

Transitions:
- A ⇔ G is common
- Transitions have lower penalty

**Formulation 4: Varying gap cost models**

1. Linear gap penalty
   - Same as before
2. Affine gap penalty
   - Big initial cost for starting or ending a gap
   - Small incremental cost for each additional character
3. General gap penalty
   - Any cost function
   - No longer computable using the same model
4. Frame-aware gap penalty
   - Multiples of 3 disrupt coding regions
5. Seek duplicated regions, rearrangements, …
   - Etc

**How many alignments are there?**

- Longest ‘non-boring’ alignment: n+m entries
  - Otherwise a gap will be aligned to a gap
  - Condense
- Alignment is equivalent to gap placement
  - (n+m choose n) ways to choose S1 placement
  - At each position yes/no answer of placing character
  - Exponential number of possible placements
- Exponential number of sequence alignment
  - Enumerating and scoring each of them not an option
  - Need faster solution for finding best alignment

Need polynomial algorithm to find best alignment amongst an exponential number of possible alignments!

**Goal: Sequence Alignment / Dynamic Programming**

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments
2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back
3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry ⇔ prefix alignment score; path ⇔ alignment
4. Advanced topics: Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Better than O(n^2)?
   - Linear-space DP: Four-Russians algorithm. Total time?

**Computing Fibonacci Numbers**

- Fibonacci numbers

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_n</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

F_n = F_{n-1} + F_{n-2}

**A simple introduction to the principles of Dynamic Programming**

Turning exponentials into polynomials

F_6 = F_5 + F_4 = (F_4 + F_3) + (F_3 + F_2) = ... = (3+2) + (2+1) = 5 + 3 = 8

Fibonacci numbers

F_0 = 0
F_1 = 1
F_2 = 1
F_3 = 2
F_4 = 3
F_5 = 5
F_6 = 8

F_n = F_{n-1} + F_{n-2}
Fibonacci numbers are ubiquitous in nature

Computing Fibonacci numbers: Top down

- Fibonacci numbers are defined recursively:
  - Python code
  ```python
def fibonacci(n):
    if n==1 or n==2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
  ```

- Goal: Compute $n^{th}$ Fibonacci number.
  - $F(0)=1, F(1)=1, F(n)=F(n-1)+F(n-2)$
  - $1,1,2,3,5,8,13,21,34,55,89,144,233,377,\ldots$

- Analysis:
  - $T(n) = T(n-1) + T(n-2) = (\ldots) = O(2^n)$

Computing Fibonacci numbers: Bottom up

- Bottom up approach
  - Python code
  ```python
def fibonacci(n):
    fib_table[1] = 1
    fib_table[2] = 1
    for i in range(3,n+1):
      fib_table[i] = fib_table[i-1]+fib_table[i-2]
    return fib_table[n]
  ```

- Analysis: $T(n) = O(n)$

Lessons from iterative Fibonacci algorithm

- What did the iterative solution do?
  - Reveal identical sub-problems
  - Order computation to enable result reuse
  - Systematically filled-in table of results
  - Expressed larger problems from their subparts

- Ordering of computations matters
  - Naïve top-down approach very slow
    - results of smaller problems not available
    - repeated work
  - Systematic bottom-up approach successful
    - Systematically solve each sub-problem
    - Fill-in table of sub-problem results in order.
    - Look up solutions instead of recomputing

Dynamic Programming in Theory

- Hallmarks of Dynamic Programming
  - **Optimal substructure**: Optimal solution to problem (instance) contains optimal solutions to sub-problems
  - **Overlapping subproblems**: Limited number of distinct subproblems, repeated many times

- Typically for optimization problems (unlike Fib example)
  - Optimal choice made locally: max( subsolution score)
  - Score is typically added through the search space
  - Traceback common, find optimal path from indiv. choices

- Middle of the road in range of difficulty
  - Easier: greedy choice possible at each step
  - DynProg: requires a traceback to find that optimal path
  - Harder: no opt. substr., e.g. subproblem dependencies

Hallmarks of optimization problems

1. **Optimal substructure**
   An optimal solution to a problem (instance) contains optimal solutions to subproblems.

2. **Overlapping subproblems**
   A recursive solution contains a “small” number of distinct subproblems repeated many times.

3. **Greedy choice property**
   Locally optimal choices lead to globally optimal solution

Greedy Choice is not possible
Globally optimal solution requires traceback through many choices
**Dynamic Programming in Practice**

- Setting up dynamic programming
  1. Find ‘matrix’ parameterization (# dimensions, variables)
  2. Make sure sub-problem space is finite! (not exponential)
    - If not all subproblems are used, better off using memoization
    - If reuse not extensive, perhaps Dynamic Programming is not right solution!
  3. Traversal order: sub-results ready when you need them
    - Computation order matters! (bottom-up, but not always obvious)

- Recursion formula: larger problems = F(subparts)
  1. Systematically fill in table of results, find optimal score
  2. Trace-back from optimal score, find optimal solution

**Goal: Sequence Alignment / Dynamic Programming**

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments

2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space,
     (3) traversal order, (4) recursion formula, (5) trace-back

3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry Ù prefix alignment score; path Ù alignment

4. Advanced topics: Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Better than O(n^2)?
   - Linear-space DP: Four-Russians algorithm. Total time?

---

**Key insight #1: Score is additive, smaller to larger**

- Compute best alignment recursively
  - For a given aligned pair (i, j), the best alignment is:
    - Best alignment of S[1..i] and S[1..j]
    - Best alignment of S[1..i..n] and S[1..j..m]
  - Proof: cut-and-paste argument (see 6.046)

- This allows a single recursion (top-left to bottom-right) instead of two recursions (middle-to-outside top-down)

---

**Key insight #2: compute scores recursively**

- Compute alignment of CGT vs. TG exactly once

---

**Key insight #3: sub-problems are repeated ➔ reuse!**

- Identical sub-problems! We can reuse our work!
Solution #1 – Memoization
- Create a big dictionary, indexed by aligned seqs
  - When you encounter a new pair of sequences
  - If it is in the dictionary:
    - Look up the solution
  - If it is not in the dictionary
    - Compute the solution
    - Insert the solution in the dictionary
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!

Top down approach

Solution #2 – Dynamic programming
- Create a big table, indexed by (i,j)
  - Fill it in from the beginning all the way till the end
  - You know that you’ll need every subpart
  - Guaranteed to explore entire search space
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!
  - Very simple computationally!

Bottom up approach

Key insight #4: Optimal prefix almt score $\leftrightarrow$ Matrix entry

<table>
<thead>
<tr>
<th></th>
<th>$S_1[1..i]$</th>
<th>i</th>
<th>$S_2[1..j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_2[1..j]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_1[1..n]$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key insight #5: Optimal alignment $\leftrightarrow$ Matrix path

DP approach: iteratively grow best alignment soltn

Computing alignments recursively: $M[i,j]=F($smaller$)$
- **Local update rules**, only look at neighboring cells:
  - Compute next alignment based on previous alignment
  - Just like Fibonacci numbers: $F[i] = F[i-1] + F[i-2]$ 
  - Table lookup avoids repeated computation
- Computing the score of a cell from smaller neighbors
  - $M(i,j) = \max \{ M(i-1,j-1) + \text{score} \}$
  - Only three possibilities for extending by one nucleotide: a gap in one species, a gap in the other, a (mis)match
- Compute scores for prefixes of increasing length
  - Start with prefixes of length 1, extend by one each time, until all prefixes have been computed
  - When you reach bottom right, alignment score of $S_1[1..m]$ and $S_2[1..n]$ is alignment of full $S_1$ and full $S_2$
  - (Can then trace back to construct optimal path to it)
Dynamic Programming for sequence alignment

- Setting up dynamic programming
  1. Find 'matrix' parameterization
     • Prefix parameterization. Score(S1[1..i],S2[1..j]) → M(i,j)
     • (i,j) only prefixes vs. (i,j,k) all substrings → simpler 2-d matrix
  2. Make sure sub-problem space is finite! (not exponential)
     • It's just n^2, quadratic (which is polynomial, not exponential)
  3. Traversal order: sub-results ready when you need them
  4. Recursion formula: larger problems = Func(subparts)
     • Need formula for computing M[i, j] as function of previous results
     • Single increment at a time, only look at M[i-1,j], M[i,j-1], M[i-1,j-1]
       corresponding to 3 options: gap in S1, gap in S2, char in both
     • Score in each case depends on gap/match/mismatch penalties
  5. Remember choice: F() typically includes min() or max()
     • Remember which of three cells (top,left,diag) led to maximum

Step 1: Setting up the scoring matrix M[i,j]

Initialization:
• Top left: 0
Update Rule:
M(i,j) = max{
  M(i-1, j) - 2 gap
  M(i, j-1) - 2 gap
  M(i-1, j-1) -1 mismatch
  M(i-1, j-1) +1 match
}
Termination:
• Bottom right

Step 2: Filling in the optimal scores from top left

Initialization:
• Top left: 0
Update Rule:
M(i,j) = max{
  M(i-1, j) - 2 gap
  M(i, j-1) - 2 gap
  M(i-1, j-1) -1 mismatch
  M(i-1, j-1) +1 match
}
Termination:
• Bottom right

Step 3: Trace back pointers to construct alignment

Initialization:
• Top left: 0
Update Rule:
M(i,j) = max{
  M(i-1, j) - 2 gap
  M(i, j-1) - 2 gap
  M(i-1, j-1) -1 mismatch
  M(i-1, j-1) +1 match
}
Termination:
• Bottom right

Path segments that lead to the globally optimal solution
Path segments that lead to locally optimal choices

Genome alignment in an excel spreadsheet

https://www.dropbox.com/s/ksh4qfl5eb182p6/Lecture02_DP%20Alignment%20In%20Excel.xlsx?dl=0

Path segment that lead to the optimal choice
What is missing? (5) Returning the actual path!

- We know how to compute the best score
  - Simply the number at the bottom right entry
- But we need to remember where it came from
  - Pointer to the choice we made at each step
- Retrace path through the matrix
  - Need to remember all the pointers

Time needed: $O(m*n)$
Space needed: $O(m*n)$

Goal: Sequence Alignment / Dynamic Programming

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments
2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back
3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry $\Rightarrow$ prefix alignment score; path $\Rightarrow$ alignment
4. Advanced topics: Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Better than $O(n^2)$?
   - Linear-space DP: Four-Russians algorithm. Total time?

Goal: Sequence Alignment / Dynamic Programming

If time permits...

(4) Extensions to basic DP solution

Bounded Dynamic Programming

Initialization:
$F(i,0), F(0,j)$ undefined for $i, j > k$

Iteration:
For $i = 1 \ldots M$
  For $j = \max(1, i - k) \ldots \min(N, i+k)$
  
  $F(i, j) = \max$
  
  $F(i - 1, j - 1) + s(x_i, y_j)$
  $F(i, j - 1) - d$, if $j > i - k(N)$
  $F(i - 1, j) - d$, if $j < i + k(N)$

Termination: same

Slides credit: Serafim Batzoglou

Can we do better than $O(n^2)$ in the general case?

- Reduced Orthogonal Vectors to PATTERN
- Reduced PATTERN to EDIT DISTANCE
- Proved EDIT DISTANCE is a SETH-hard problem

Faster edit dist. algorithm probably not a good term project
Linear space alignment

It is easy to compute $F(M, N)$ in linear space

- Allocate (column[1])
- Allocate (column[2])
- For $i = 1 \ldots M$
  - If $i > 1$, then:
    - Free(column[$i-2$])
    - Allocate(column[$i$])
- For $j = 1 \ldots N$
  - $F(i, j) = \ldots$

What about the pointers?

Finding the best back-pointer for current column

- Now, using 2 columns of space, we can compute for $k = 1 \ldots M$, $F(M/2, k)$, $F'(M/2, N-k)$
- PLUS the backpointers

Best forward-pointer for current column

- Now, we can find $k'$ maximizing $F(M/2, k) + F'(M/2, N-k)$
- Also, we can trace the path exiting column $M/2$ from $k'$

Recursively find midpoint for left & right

- Iterate this procedure to the left and right!

Total time cost of linear-space alignment

- Dynamic programming
  - Reuse of computation
  - Order sub-problems. Fill table of sub-problem results
  - Read table instead of repeating work (ex: Fibonacci)
- Sequence alignment
  - Edit distance and scoring functions
  - Dynamic programming matrix
  - Matrix traversal path $\Rightarrow$ Optimal alignment
- Thursday: Variations on sequence alignment
  - Local/global alignment, affine gaps, algo speed-ups
  - Semi-numerical alignment, hashing, database lookup
- Recitation:
  - Dynamic programming applications
  - Probabilistic derivations of alignment scores
**Goal: Sequence Alignment / Dynamic Programming**

1. **Introduction to sequence alignment**
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments

2. **Introduction to principles of dynamic programming**
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back

3. **DP for sequence alignment**
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry \( \leftrightarrow \) prefix alignment score; path \( \leftrightarrow \) alignment

4. **Advanced topics: Dynamic Programming variants**
   - Linear-time bounded DP (heuristic). Better than \( O(n^2) \)?
   - Linear-space DP: Four-Russians algorithm. Total time?