Pairwise Sequence alignment
Basic Algorithms
Some Complexity Bounds

Most pairwise sequence alignment problems can be solved in $O(mn)$ time.

Space requirement can be reduced to $O(m+n)$, while keeping run-time fixed [Hirshberg, 1988].

Highly similar sequences can be aligned in $O(dn)$ time, where $d$ measures the distance between the sequences [Landau, 1986]

The best known alignment algorithms have time complexity $O(n^2/log n)$ [Crochemore, Landau, Ziv-Ukelson, 2003]

For Discrete Scoring Schemes: [Masek and Paterson, 1980]
Outline

• LCS computation viewed as a problem of computing heaviest paths in an alignment graph
• Reminder: MergeSort
• Finding the middle point in the alignment matrix in linear space
• Linear space sequence alignment
\[
c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\
\text{max}(c[\text{prefix } \alpha, \beta], c[\alpha, \text{prefix } \beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta), \\
c[\text{prefix } \alpha, \text{prefix } \beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), 
\end{cases}
\]

- Keep track of \(c[\alpha, \beta]\) in a table of \(nm\) entries:
  - top/down: increasing row order
  - within each row left-to-right: increasing column order
\[ c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\
\min(\max(c[prefix_\alpha, prefix_\beta], c[\alpha, prefix_\beta]) + 1) & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\
\max(c[prefix_\alpha, \beta], c[\alpha, prefix_\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). 
\end{cases} \]

Time Complexity: \(O(nm)\).
Space Complexity: \(O(nm)\).

Can the space complexity be improved?
Edit Graph for LCS Problem

Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence

LCS Problem: Find a path with maximum number of diagonal edges
\[ c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty}, \\
c[\text{prefix } \alpha, \text{prefix } \beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\
\max(c[\text{prefix } \alpha, \beta], c[\alpha, \text{prefix } \beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). 
\end{cases} \]

Time Complexity: \( O(nm) \).
Sequence Alignment Runtime and Memory

- It takes $O(nm)$ time to fill in the $nxm$ dynamic programming matrix.

- Memory? Naively $O(nm)$ – we next show how to reduce it to $O(n)$ by combining divide and conquer with dynamic programming.
Reducing space requirements

- $O(mn)$ tables are often the limiting factor in computing large alignments
- There is a linear space technique that only doubles the time required [Hirschberg, 1977]
### IDEA: We only need the previous row to calculate the next
Why Do We Need So Much Space?

To compute \( V[n,m] = d(s[1..n], t[1..m]) \), we need only \( O(\min(n,m)) \) space:

- Compute \( V(i,j) \), column by column, storing only two columns in memory (or row by line if rows are shorter).

Note, however, that

This "trick" fails when we need to reconstruct the optimizing sequence.

Trace back information requires \( O(mn) \) memory bytes.
Linear-space Alignments

\[ mn + \frac{1}{2} mn + \frac{1}{4} mn + \frac{1}{8} mn + \frac{1}{16} mn + \ldots = 2 \, mn \]
Computing Alignment Path Requires Quadratic Memory

Alignment Path
- Space complexity for computing alignment path for sequences of length \( n \) and \( m \) is \( O(nm) \)
- We need to keep all backtracking references in memory to reconstruct the path (backtracking)

Can the space complexity be improved if we just compute the length of the LCS and do not need to recover the actual LCS?

In this case we only need to compute the alignment score.
Computing Alignment Score with Linear Memory

Alignment Score

- Space complexity of computing just the score itself is $O(n)$
- We only need the previous column to calculate the current column, and we can then throw away that previous column once we’re done using it
Computing Alignment Score: Recycling Columns

Only two columns of scores are saved at any given time

memory for column 1 is used to calculate column 3

memory for column 2 is used to calculate column 4
Computing Alignment Score with Linear Memory

Alignment Score

- Space complexity of computing just the score itself is $O(n)$
- We only need the previous column to calculate the current column, and we can then throw away that previous column once we’re done using it
Why Do We Need So Much Space?

To compute $V[n,m]=d(s[1..n],t[1..m])$, we need only $O(\min(n,m))$ space:

- Compute $V(i,j)$, column by column, storing only two columns in memory (or line by line if lines are shorter).

Note, however, that

This “trick” fails when we need to reconstruct the optimizing sequence.

Trace back information requires $O(mn)$ memory bytes.
Computing Alignment Path Requires Quadratic Memory

Alignment Path

- Space complexity for computing alignment path for sequences of length $n$ and $m$ is $O(nm)$
- We need to keep all backtracking references in memory to reconstruct the path (backtracking)

Can the space complexity be improved if need to recover the actual LCS?

Yes, but for this we will have to combine dynamic programming with divide and conquer....
Divide & Conquer Algorithms
Divide and Conquer Algorithms

• **Divide** problem into sub-problems
• **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion
• **Combine** the solutions of sub-problems into a solution of the original problem (tricky part)
Sorting Problem Revisited

• Given: an unsorted array

\[
\begin{array}{cccccccc}
5 & 2 & 4 & 7 & 1 & 3 & 2 & 6 \\
\end{array}
\]

• Goal: sort it

\[
\begin{array}{cccccccc}
1 & 2 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
Mergesort: Divide Step

Step 1 – Divide

How many divisions to split an array of size $n$ into single elements?

$\log(n)$ divisions to split an array of size $n$ into single elements
Mergesort: Conquer Step

Step 2 – Conquer

\[
\begin{array}{cccccccc}
  5 & 2 & 4 & 7 & 1 & 3 & 2 & 6 \\
  2 & 5 & 4 & 7 & 1 & 3 & 2 & 6 \\
  2 & 4 & 5 & 7 & 1 & 2 & 3 & 6 \\
  1 & 2 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
Mergesort: Combine Step

Step 3 – Combine

- 2 arrays of size 1 can be easily merged to form a sorted array of size 2
- 2 sorted arrays of size $n$ and $m$ can be merged in $O(n+m)$ time to form a sorted array of size $n+m$
Mergesort: Combine Step

Combining 2 arrays of size 4

Etcetera…
Merge Algorithm

1. **Merge**\((a, b)\)
2. \(n1 \leftarrow\) size of array \(a\)
3. \(n2 \leftarrow\) size of array \(b\)
4. \(a_{n1+1} \leftarrow \infty\)
5. \(a_{n2+1} \leftarrow \infty\)
6. \(i \leftarrow 1\)
7. \(j \leftarrow 1\)
8. **for** \(k \leftarrow 1\) **to** \(n1 + n2\)
9. **if** \(a_i < b_j\)
10. \(c_k \leftarrow a_i\)
11. \(i \leftarrow i + 1\)
12. **else**
13. \(c_k \leftarrow b_j\)
14. \(j \leftarrow j + 1\)
15. **return** \(c\)
Mergesort: Conquer Step

Step 2 – Conquer

How many iterations? How long does each iteration take?

\( \log n \) iterations, each iteration takes \( O(n) \) time. **Total Time:** \( O(n \log n) \)
Mergesort: Example

Divide

Conquer
MergeSort Algorithm

1. MergeSort(c)
2. \( n \leftarrow \text{size of array } c \)
3. if \( n = 1 \)
4. \hspace{1em} return \( c \)
5. \( \text{left} \leftarrow \text{list of first } n/2 \text{ elements of } c \)
6. \( \text{right} \leftarrow \text{list of last } n - n/2 \text{ elements of } c \)
7. \( \text{sortedLeft} \leftarrow \text{MergeSort(left)} \)
8. \( \text{sortedRight} \leftarrow \text{MergeSort(right)} \)
9. \( \text{sortedList} \leftarrow \text{Merge(sortedLeft, sortedRight)} \)
10. return \( \text{sortedList} \)
MergeSort: Running Time

• The problem is simplified to baby steps
  • for the $i$th merging iteration, the complexity of the problem is $O(n)$
  • number of iterations is $O(\log n)$
  • running time: $O(n \log n)$
Computing LCS Alignment Path in Less than Quadratic Memory

• The output itself (alignment path) requires only linear space, as the number of edges/vertices participating in this path is bounded by $n + m$.

• So is there some way to store, throughout the $O(n \cdot m)$ time computation, only the $O(n + m)$ vertices and edges participating in the LCS path?
Space Efficient Version: Outline

**Input:** Sequences $s[1,n]$ and $t[1,m]$ to be aligned.

**Idea:** perform divide and conquer

1. If $n=1$ align $s[1,1]$ and $t[1,m]$
2. Else, find position $(n/2, j)$ at which some best alignment crosses a midpoint

Construct alignments

1. $A = s[1,n/2]$ vs $t[1,j]$
2. $B = s[n/2+1,n]$ vs $t[j+1,m]$

- Return $AB$
Divide and conquer sequence alignment

Source:
Jones and Pevzner
Finding the Midpoint

The score of the best alignment that goes through \( j \) equals:

\[
d(s[1,n/2],t[1,j]) + d(s[n/2+1,n],t[j+1,m])
\]

- Thus, we need to compute these two quantities for all values of \( j \)
Finding the Midpoint (Algorithm)

Define

- \( F[i,j] = d(s[1,i],t[1,j]) \)
- \( B[i,j] = d(s[i+1,n],t[j+1,m]) \)

- \( F[i,j] + B[i,j] = \text{score of best alignment through } (i,j) \)

- We compute \( F[i,j] \) as we did before
- We compute \( B[i,j] \) in exactly the same manner, going “backward” from \( B[n,m] \)
Divide and Conquer Approach to LCS

\textbf{Path}(source, sink)

- if(source \& sink are in consecutive columns)
  - output the longest path from source to sink
- else
  - \textit{middle} ← middle vertex between source \& sink
  - \textbf{Path}(source, middle)
  - \textbf{Path}(middle, sink)
Divide and Conquer Approach to LCS

**Path**(source, sink)

- if(source & sink are in consecutive columns)
- output the longest path from source to sink
- else
- middle ← middle vertex between source & sink
- Path(source, middle)
- Path(middle, sink)

The only problem left is how to find this “middle vertex”!
We want to calculate the longest path from \((0,0)\) to \((n,m)\) that passes through \((i, m/2)\) where \(i\) ranges from 0 to \(n\) and represents the \(i\)-th row.

Define

\[
\text{length}(i)
\]

as the length of the longest path from \((0,0)\) to \((n,m)\) that passes through vertex \((i, m/2)\).
Define \((mid, m/2)\) as the vertex where the longest path crosses the middle column.

\[
\text{length}(mid, m/2) = \text{optimal length} = \max_{0 \leq i \leq n} \text{length}(i)
\]
Computing Prefix($i$)

- $\text{prefix}(i)$ is the length of the longest path from (0,0) to ($i$, $m/2$)
- Compute $\text{prefix}(i)$ by dynamic programming in the left half of the matrix

![Diagram](image-url)
Computing Suffix(i)

- \( \text{suffix}(i) \) is the length of the longest path from \((i, m/2)\) to \((n, m)\)
- \( \text{suffix}(i) \) is the length of the longest path from \((n, m)\) to \((i, m/2)\) with all edges reversed
- Compute \( \text{suffix}(i) \) by dynamic programming in the right half of the “reversed” matrix

![Diagram showing the process of computing Suffix(i)](image-url)
\[ \text{Length}(i) = \text{Prefix}(i) + \text{Suffix}(i) \]

- Add \textit{prefix}(i) and \textit{suffix}(i) to compute \textit{length}(i):
  - \textit{length}(i) = \text{prefix}(i) + \text{suffix}(i)

- You now have a middle vertex of the maximum path \((i, m/2)\) as maximum of \textit{length}(i)
Finding the Middle Point

<table>
<thead>
<tr>
<th>0</th>
<th>m/4</th>
<th>m/2</th>
<th>3m/4</th>
<th>m</th>
</tr>
</thead>
</table>

Diagram showing a path connecting points from 0 to m, passing through m/4 and m/2, with a heart marking the midpoint.
Finding the Middle Point again

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>m/4</th>
<th>m/2</th>
<th>3m/4</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A diagram illustrating the process of finding the middle point again.
And Again

0  m/8  m/4  3m/8  m/2  5m/8  3m/4  7m/8  m
Time = Area: First Pass

- On first pass, the algorithm covers the entire area

Area = $n \times m$
Time = Area: First Pass

- On first pass, the algorithm covers the entire area

\[ \text{Area} = nm \]
Time = Area: Second Pass

- On second pass, the algorithm covers only $\frac{1}{2}$ of the area.

Area/2
Time = Area: Third Pass

- On third pass, only 1/4th is covered.
Geometric Reduction At Each Iteration

1 + \( \frac{1}{2} \) + \( \frac{1}{4} \) + ... + \((\frac{1}{2})^k\) \(\leq\) 2

- Runtime: \( O(\text{Area}) = O(nm) \)

Diagram:
- First pass: 1
- Second pass: 1/2
- Third pass: 1/4
- Fourth pass: 1/8
- Fifth pass: 1/16
Time Complexity Analysis

• Finding the mid-point:
  1. find $F[n/2, j]$ for all $1 \leq j \leq m$ \(O(n \cdot m/2))\)
  2. find $B[n/2, j]$ for all $1 \leq j \leq m$ \(O(n \cdot m/2))\)

Complexity:

- Time: $O(nm)$
- Space: $O(n)$
Size of recursive sub-problems: 
(n/2,j) and (n/2,m-j-1), hence

Complexity:
Time: $O(mn/2)$
Space: $O(n/2)$

• Remark: Time should be added

• Space required for previous sub-problem can be erased
Time Complexity Analysis (cont.)

- Total time complexity:
  \[ mn + mn/2 + mn/4 + \ldots < 2mn = O(mn) \]
- Total space complexity: \( O(n) \)

Thus, space complexity is **linear** in size of the problem.

At worst, twice the cost of the regular solution.