Divide & Conquer Algorithms
Outline

1. MergeSort
2. Finding the middle vertex
3. Linear space sequence alignment
4. Block alignment
5. Four-Russians speedup
6. LCS in sub-quadratic time
Section 1: MergeSort
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.

- **Goal**: Sort it.

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

• Step 1: DIVIDE

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

- Step 2: CONQUER

  5 2 4 7 1 3 2 6  \(O(n)\)

  2 5 4 7 1 3 2 6  \(O(n)\)

  2 4 5 7 1 2 3 6  \(O(n)\)

  1 2 2 3 4 5 6 7  \(O(n)\)

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

• Step 3: **COMBINE**

\[
\begin{array}{c}
5 \\
2 \\
\end{array} \rightarrow \begin{array}{c}
2 \\
5 \\
\end{array}
\]

• 2 arrays of size 1 can be easily merged to form a sorted array of size 2.

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

Divide

Conquer
MergeSort Algorithm

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

• The problem is simplified to baby steps:
  • For the $ith$ merging iteration, the complexity of the problem is $O(n)$.
  • Number of iterations is $O(\log n)$.
  • **Running Time:** $O(n \log n)$. 
Divide and Conquer Approach to LCS

1. \textbf{Path}(\textit{source, sink})
2. \textbf{if}(\textit{source} \& \textit{sink} are in consecutive columns)
3. \textbf{else}
4. \textbf{output the longest path from source to sink}
5. \textbf{middle} ← middle vertex between \textit{source} \& \textit{sink}
6. \textbf{Path}(\textit{source, middle})
7. \textbf{Path}(\textit{middle, sink})
Divide and Conquer Approach to LCS

1. \textbf{Path}(source, sink)
2. \textbf{if}(source \& sink are in consecutive columns)
3. \hspace{1em} output the longest path from source to sink
4. \textbf{else}
5. \hspace{1em} middle \leftarrow \text{middle vertex between source \& sink}
6. \hspace{1em} \textbf{Path}(source, middle)
7. \hspace{1em} \textbf{Path}(middle, sink)

- The only problem left is how to find this “middle vertex”!
Section 2: Finding the Middle Vertex
Alignment Score Requires Linear Memory

- Space complexity of computing the *alignment score* is just $O(n)$.

- 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 re-used to calculate column 3

Memory for column 2 is re-used to calculate column 4
Alignment Path Requires Quadratic Memory

- Space complexity for computing an alignment path for sequences of length \( n \) and \( m \) is \( O(nm) \).

- The reason is that we need to keep all backtracking references in memory to reconstruct the path (backtracking).
Crossing the Middle Line

- 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 length(i) as the length of the longest path from (0,0) to (n,m) that passes through vertex (i, m/2).
Crossing the Middle Line

- 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 ith 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).
Crossing the Middle Line

- Define \((mid, m/2)\) as the vertex where the longest path crosses the middle column.

- \(\text{length}(mid) = \text{optimal length} = \max_{0 \leq i \leq n} \text{length}(i)\)
Crossing the Middle Line

• Define \((mid, m/2)\) as the vertex where the longest path crosses the middle column.

\[
\text{length}(mid) = \text{optimal length} = \max_{0 \leq i \leq n} \text{length}(i)
\]
Computing $\text{prefix}(i)$

- $\text{prefix}(i)$ 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 showing dynamic programming](image-url)
Computing $\text{suffix}(i)$

- $\text{suffix}(i)$ 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.

Store $\text{suffix}(i)$ column
\[ \text{length}(i) = \text{prefix}(i) + \text{suffix}(i) \]

- Add \( \text{prefix}(i) \) and \( \text{suffix}(i) \) to compute \( \text{length}(i) \):
- \( \text{length}(i) = \text{prefix}(i) + \text{suffix}(i) \)
- You now have a middle vertex of the maximum path \((i, m/2)\) as maximum of \( \text{length}(i) \).
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 the middle points and their corresponding values.
Finding the Middle Point Again

0  m/4  m/2  3m/4  m
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.

\[ \text{Area} = n \cdot m \]
Time = Area: First Pass

- On first pass, the algorithm covers the entire area.

\[ \text{Area} = n \times m \]
Time = Area: Second Pass

- On second pass, the algorithm covers only 1/2 of the area
Time = Area: Third Pass

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

- $1 + \frac{1}{2} + \frac{1}{4} + \ldots + (\frac{1}{2})^k \leq 2$
- Runtime: $O(\text{Area}) = O(nm)$
Can We Align Sequences in Subquadratic Time?

- Dynamic Programming takes $O(n^2)$ for global alignment.

- Can we do better?

- Yes, use *Four-Russians Speedup*. 
Section 4: Block Alignment
Partitioning Sequences into Blocks

- Partition the $n \times n$ grid into blocks of size $t \times t$.

- We are comparing two sequences, each of size $n$, and each sequence is sectioned off into chunks, each of length $t$.

- Sequence $u = u_1 \ldots u_n$ becomes
  
  $|u_1 \ldots u_t| |u_{t+1} \ldots u_{2t}| \ldots |u_{n-t+1} \ldots u_n|$

  and sequence $v = v_1 \ldots v_n$ becomes

  $|v_1 \ldots v_t| |v_{t+1} \ldots v_{2t}| \ldots |v_{n-t+1} \ldots v_n|$
Partitioning Alignment Grid into Blocks

\[ \text{partition} \]

\[ \frac{n}{t} \]
Block Alignment

- **Block alignment** of sequences $u$ and $v$.
  1. An entire block in $u$ is aligned with an entire block in $v$.
  2. An entire block is inserted.
  3. An entire block is deleted.

- **Block path**: a path that traverses every $t \times t$ square through its corners.
Block Alignment: Examples

valid

invalid
Block Alignment Problem

- **Goal**: Find the longest block path through an edit graph.

- **Input**: Two sequences, \( u \) and \( v \) partitioned into blocks of size \( t \). This is equivalent to an \( n \times n \) edit graph partitioned into \( t \times t \) subgrids.

- **Output**: The block alignment of \( u \) and \( v \) with the maximum score (longest block path through the edit graph).
Constructing Alignments within Blocks

• To solve: Compute alignment score $BlockScore_{i,j}$ for each pair of blocks $|u_{(i-1)t+1}…u_{it}|$ and $|v_{(j-1)t+1}…v_{jt}|$.

• How many blocks are there per sequence?
  • $(n/t)$ blocks of size $t$

• How many pairs of blocks for aligning the two sequences?
  • $(n/t) \times (n/t)$

• For each block pair, solve a mini-alignment problem of size $t \times t$
Constructing Alignments within Blocks

Block pair represented by each square

Solve mini-alignment problems

\( n/t \)
Block Alignment: Dynamic Programming

• Let $s_{i,j}$ denote the optimal block alignment score between the first $i$ blocks of $u$ and first $j$ blocks of $v$.

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j} - \sigma_{\text{block}} \\
  s_{i,j-1} - \sigma_{\text{block}} \\
  s_{i-1,j-1} + \text{BlockScore}(i, j) 
\end{cases}$$

• $\sigma_{\text{block}}$ is the penalty for inserting or deleting an entire block.

• $\text{BlockScore}(i, j)$ is the score of the pair of blocks in row $i$ and column $j$. 
Block Alignment Runtime

• Indices $i, j$ range from 0 to $n/t$.

• Running time of algorithm is
  \[ O(\ [n/t]*[n/t]) = O(n^2/t^2) \]
  if we don’t count the time to compute each $BlockScore(i, j)$. 
Block Alignment Runtime

- Computing all $BlockScore_{i,j}$ requires solving $(n/t) \times (n/t)$ mini block alignments, each of size $(t \times t)$.

- So computing all $\beta_{i,j}$ takes time
  $$O([n/t] \times [n/t] \times t \times t) = O(n^2)$$

- This is the same as dynamic programming.

- How do we speed this up?
Section 5: Four Russians Speedup
Four Russians Technique

• Let $t = \log(n)$, where $t$ is the block size, $n$ is the sequence size.

• Instead of having $(n/t) \times (n/t)$ mini-alignments, construct $4^t \times 4^t$ mini-alignments for all pairs of strings of $t$ nucleotides (huge size), and put in a lookup table.

• However, size of lookup table is not really that huge if $t$ is small. Let $t = (\log n)/4$. Then $4^t \times 4^t = n$. 
Each sequence has $t$ nucleotides.

Lookup table “Score”:

Size is only $n$, instead of $(n/t) \times (n/t)$.
New Recurrence

• The new lookup table \( Score \) is indexed by a pair of \( t \)-nucleotide strings, so

\[
\begin{align*}
    s_{i,j} &= \max \left\{ s_{i-1,j} - \sigma_{\text{block}} , s_{i,j-1} - \sigma_{\text{block}} , s_{i-1,j-1} + \text{Score}(i^{\text{th}} \text{ block of } v, j^{\text{th}} \text{ block of } u) \right\}
\end{align*}
\]

• **Key Difference:** The \( Score \) function is taken from the hash table rather than computed by dynamic programming as before.
Four Russians Speedup Runtime

• Since computing the lookup table $Score$ of size $n$ takes $O(n)$ time, the running time is mainly limited by the $(n/t)*(n/t)$ accesses to the lookup table.

• Each access takes $O(\log n)$ time.

• Overall running time: $O( [n^2/t^2]*\log n)$

• Since $t = \log n$, substitute in:
  • $O( [n^2/\{\log n\}^2]*\log n) \geq O( n^2/\log n )$
Section 6: LCS in Sub-Quadratic Time
So Far…

- We can divide up the grid into blocks and run dynamic programming only on the corners of these blocks.

- In order to speed up the mini-alignment calculations to under $n^2$, we create a lookup table of size $n$, which consists of all scores for all $t$-nucleotide pairs.

- Running time goes from quadratic, $O(n^2)$, to subquadratic: $O(n^2/\log n)$
Four Russians Speedup for LCS

• Unlike the block partitioned graph, the LCS path does not have to pass through the vertices of the blocks.
Block Alignment vs. LCS

• In block alignment, we only care about the corners of the blocks.

• In LCS, we care about all points on the edges of the blocks, because those are points that the path can traverse.

• Recall, each sequence is of length $n$, each block is of size $t$, so each sequence has $(n/t)$ blocks.
Block Alignment vs. LCS: Points Of Interest

Block alignment has \((n/t)^2(n/t) = (n^2/t^2)\) points of interest

LCS alignment has \(O(n^2/t)\) points of interest
Traversing Blocks for LCS

- Given alignment scores $s_{i,*}$ in the first row and scores $s_{*,j}$ in the first column of a $t \times t$ mini square, compute alignment scores in the last row and column of the minisquare.

- To compute the last row and the last column score, we use these 4 variables:
  1. Alignment scores $s_{i,*}$ in the first row.
  2. Alignment scores $s_{*,j}$ in the first column.
  3. Substring of sequence $u$ in this block ($4^t$ possibilities).
  4. Substring of sequence $v$ in this block ($4^t$ possibilities).
Traversing Blocks for LCS

- If we used this to compute the grid, it would take quadratic, $O(n^2)$ time, but we want to do better.
Four Russians Speedup

• Build a lookup table for all possible values of the four variables:
  1. All possible scores for the first row $s_{*,j}$
  2. All possible scores for the first column $s_{*,j}$
  3. Substring of sequence $u$ in this block ($4^t$ possibilities).
  4. Substring of sequence $v$ in this block ($4^t$ possibilities).

• For each quadruple we store the value of the score for the last row and last column.
  • This will be a huge table, but we can eliminate alignments scores that don’t make sense.
Reducing Table Size

- Alignment scores in LCS are monotonically increasing, and adjacent elements can’t differ by more than 1.

- Example: 0,1,2,2,3,4 is ok; 0,1,2,4,5,8, is not because 2 and 4 differ by more than 1 (and so do 5 and 8).

- Therefore, we only need to store quadruples whose scores are monotonically increasing and differ by at most 1.
Efficient Encoding of Alignment Scores

- Instead of recording numbers that correspond to the index in the sequences $u$ and $v$, we can use binary to encode the differences between the alignment scores.
Reducing Lookup Table Size

• $2^t$ possible scores ($t = \text{size of blocks}$)

• $4^t$ possible strings
  • Lookup table size is $(2^t \times 2^t) \times (4^t \times 4^t) = 2^{6t}$

• Let $t = (\log n)/4$;
  • Table size is: $2^{6((\log n)/4)} = n^{(6/4)} = n^{(3/2)}$

• Time = O( $[n^2/t^2]\times\log n$ )

• O( $[n^2/(\log n)^2]\times\log n$ ) $\geq$ O( $n^2/\log n$ )
Summary

• We take advantage of the fact that for each block of $t = \log(n)$, we can pre-compute all possible scores and store them in a lookup table of size $n^{(3/2)}$.

• We used the Four Russian speedup to go from a quadratic running time for LCS to subquadratic running time: $O(n^2 / \log n)$. 