Lecture 12:
Divide and Conquer Algorithms

Study Chapter 7.1 – 7.4
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
Sorting Problem Revisited

Given: an unsorted array

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

Goal: sort it

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

Step 1 – Divide

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

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

Step 2 – Conquer

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

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

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

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

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

Merge

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

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

- 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: Merge

Merge 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. \textbf{if } a_i < b_j
10. \hspace{1cm} c_k \leftarrow a_i
11. \hspace{1cm} i \leftarrow i + 1
12. \textbf{else}
13. \hspace{1cm} c_k \leftarrow b_j
14. \hspace{1cm} j \leftarrow j + 1
15. \textbf{return } c
MergeSort: Example

Divide

Conquer
MergeSort Algorithm

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

- Given by a recurrence relation

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \]
\[ T(1) = O(1) \]

with solution

\[ T(n) = O(n \lg n) \]

Now for a biological problem
Alignments Require Quadratic Memory

Alignment Path

- Space complexity for computing alignment path for sequences of length $n$ and $m$ is $O(nm)$
- We keep a table of all scores and backtracking references in memory to reconstruct the path (backtracking)
Computing Alignment Score with Linear Memory

Alignment Score

- However, the space complexity of just computing the score itself is only $O(n)$
- For example, we only need the previous column to calculate the current column, and we can 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.
D&C Sequence Alignment

Find the best scoring path aligning two sequences

\[
\text{Path}(\text{source, sink})
\]

1. if(\text{source} & \text{sink} are in consecutive columns)
2. output the longest path from \text{source} to \text{sink}
3. else
4. \text{middle} ← vertex with largest score from \text{source} to \text{sink}
5. \text{Path}(\text{source, middle})
6. \text{Path}(\text{middle, sink})

The only problem left is how to find this “middle vertex”!
Computing the Alignment Path

We want to find the longest (i.e. best) path from (0,0) to (n,m) as it passes through column $m/2$

For $0 \leq i \leq n$

$\text{length}(i)$

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.

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

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

![Matrix Diagram]
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 \( \text{suffix}(i) \) 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
Finding the Middle Point again

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

9/30/2014
And Again
Time = Area: First Pass

- On first pass, the algorithm touches the entire area

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

- On second pass, the algorithm covers only 1/2 of the area

\[ \text{Area/2} \]
Time = Area: Second Pass

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

\[ \frac{\text{Area}}{2} \]

Regardless of \( i \)'s value!
Time = Area: Third Pass

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

\[ 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \left(\frac{1}{2}\right)^k \leq 2 \]

- Runtime: \( O(\text{Area}) = O(nm) \)
- Total Space: \( O(n) \) for score computation, \( O(n+m) \) to store the optimal alignment

First pass: 1

Second pass: \( \frac{1}{2} \)

Third pass: \( \frac{1}{4} \)

Fourth pass: \( \frac{1}{8} \)

Fifth pass: \( \frac{1}{16} \)

Total Space: \( O(n) \) for score computation, \( O(n+m) \) to store the optimal alignment
**Can We Do Even Better?**

- Align in Subquadratic Time?
- Dynamic Programming takes $O(nm)$ for global alignment, which is quadratic assuming $n \approx m$
- Yes, using the *Four-Russians Speedup*
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

Partitioning a grid of size $n^2$ into blocks of size $t^2$ results in $n/t$ blocks.
Block Alignment

• **Block alignment** of sequences $u$ and $v$:
  1. An entire segment of length $t$ in $u$ is aligned with an entire segment of length $t$ in $v$
  2. An entire segment of length $t$ in $u$ is deleted
  3. An entire segment of length $t$ in $v$ 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 $\beta_{i,j}$ for each pair of blocks $|u_{(i-1)*t+1…i*t}|$ and $|v_{(j-1)*t+1…j*t}|$
- 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 $t \times t$, which requires $t \times t = O(t^2)$ effort
- Total cost? $O((n/t)^2 t^2) = O(n^2)$

No improvement!
Constructing Alignments within Blocks

Block pair represented by each small 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 $v$

$\begin{align*}
    s_{i,j} &= \max \left\{ 
        s_{i-1,j} - \sigma_{\text{block}}, \\
        s_{i,j-1} - \sigma_{\text{block}}, \\
        s_{i-1,j-1} + \beta_{i,j}
    \right\}
\end{align*}$

- $\sigma_{\text{block}}$ is the penalty for inserting or deleting an entire segment of length $t$
- $\beta_{i,j}$ is score of 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
  \[ O\left( \frac{n}{t} \right)^2 \cdot O(\beta_{i,j}) = O\left( \frac{n^2}{t^2} \right) \]
- Computing all $\beta_{i,j}$ requires solving $(n/t)\cdot(n/t)$ mini block alignments, each of size $(t\cdot t)$
- So computing all $\beta_{i,j}$ takes time
  \[ O\left( \frac{n^2}{t^2} \cdot t^2 \right) = O\left( n^2 \right) \]
- Looks like a wash, but is it?
Recall Our Bag of Tricks

- A key insight of dynamic programming was to reuse repeated computations by storing them in a tableau.
- Are there any repeated computations in Block Alignments?
- Let’s check out some numbers…
  - Let’s assume \( n = m = 4000 \) and \( t = 4 \)
  - \( n/t = 1000 \), so there are 1,000,000 blocks.
  - How many possible blocks are there?
    - Assume we are aligning DNA with DNA, so the sequences are over an alphabet of \{A,C,G,T\}.
    - Possible sequences are \( 4^t = 4^4 = 256 \),
    - Possible alignments are \( 4^t \times 4^t = 65536 \).
  - There are fewer possible alignments than blocks, thus we must be frequently solving the same alignments!
Four Russians Technique

• The trick is in how to pick \( t \) relative to \( n \)

• choose \( t = \log_2(n)/4 \)

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

• However, size of lookup table is not really that huge if \( t \) is small.

• Choose \( t = (\log_2 n)/4 \). Then

\[
4^t \times 4^t = \sqrt{n} \times \sqrt{n} = n
\]

which is much less than \( \frac{n}{t} \times \frac{n}{t} \)
Look-up Table for Four Russians Technique

Each sequence has $t$ nucleotides

Lookup table "Score"

size is $n$, which is much smaller than $(n/t)^*(n/t) \rightarrow$ repeats

Rather than precomputing this table you could actually use a hash table and compute it lazily.

You can also order the sequences (alphabetize them) to exploit the symmetry, thus cutting the table size in half.
New Recurrence

- The new lookup table $Score$ is indexed by a pair of $t$-nucleotide strings, so

\[
s_{i,j} = \max \begin{cases} 
  s_{i-1,j} - \sigma_{\text{block}} \\
  s_{i,j-1} - \sigma_{\text{block}} \\
  s_{i-1,j-1} + \text{Score}(i^{th} \text{ block of } v, j^{th} \text{ block of } u) 
\end{cases}
\]
Four Russians Speedup Runtime

- Since computing the lookup table Score of size $n$ takes $O(n)$ time, the running time is dominated by the $(n/t) \times (n/t)$ accesses to the lookup table.
- Overall running time: $O\left( \frac{n^2}{t^2} \right)$
- Since $t = \frac{\log_2 n}{4}$, substitute in:
  - $O\left( \frac{n^2}{\log_2 n} \right)$ accesses to the lookup table
- How much time for each access?
  - Total lookup table size is $4^t \times 4^t = n$ so safe answer is $O(\log_2 n)$ using some sort of binary tree.
- Total time is therefore
  - $O\left( \frac{n^2}{\log_2 n} \right) \cdot (\log_2 n) = O\left( \frac{n^2}{\log_2 n} \right)$
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 is not restricted to pass through the vertices of the blocks.

block alignment

longest common subsequence
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 
\((n/t)\times(n/t) = (n^2/t^2)\) 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 (cont’d)

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

Given these $2t - 1$ scores, we can calculate these $2t - 3$ scores

$t \times t$ block
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,\textcolor{red}{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.

<table>
<thead>
<tr>
<th>Original Encoding</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 2 3 4</td>
<td>1 1 1 0 1 1</td>
</tr>
</tbody>
</table>
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 $t = \frac{\log n}{4}$;
  - Table size is: $2^{6((\log n)/4)} = n^{(6/4)} = n^{(3/2)}$
- Table construction time $t^2 \cdot 2^{6t} = (n^{1.5} (\log_2 n)^2)$ which is $o(n^2 / \log n)$, i.e. dominated by the block alignment time
Summary

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

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

- Graph Algorithms