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 (tricky part)
• 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

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

Step 2 – Conquer

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

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

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

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

\(\log n\) 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

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

- 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…
1. **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. **for** \(k \leftarrow 1 \text{ to } n1 + n2\)
9. \(\text{if } a_i < b_j\)
10. \(c_k \leftarrow a_i\)
11. \(i \leftarrow i + 1\)
12. \(\text{else}\)
13. \(c_k \leftarrow b_j\)
14. \(j \leftarrow j + 1\)
15. **return** \(c\)
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 $i$'th merging iteration, the complexity of the problem is $O(n)$
  - number of iterations is $O(\log n)$
  - running time: $O(n \log n)$

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)
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

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

memory for column 1 is used to calculate column 3

memory for column 2 is used to calculate column 4
Find the best scoring path aligning two sequences

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

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

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

\[\begin{array}{ccc}
0 & m/2 & m \\
\end{array}\]

store \(\text{prefix}(i)\) column
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 $\text{suffix}(i)$ in the right half of the “reversed” matrix

store $\text{suffix}(i)$ column
**Length(i) = Prefix(i) + Suffix(i)**

- Add *prefix(i)* and *suffix(i)* to compute *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

Diagram showing a line path from 0 to m with labels m/4, m/2, 3m/4, and m along the way.
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>
</table>

The diagram shows a path starting from 0 and moving to the right, passing through points m/4, m/2, 3m/4, and m, respectively.
And Again
Time = Area: First Pass

- On first level, the algorithm touches the entire area

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

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

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

\[ \text{Area}/2 \]

\[ i \]

\[ m/2 \]

*Regardless of i’s value!*
Time = Area: Third Pass

- On third pass, only 1/4th is covered.

\[
\text{Area}/4
\]
$\sum_{k=0}^{\infty} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \left(\frac{1}{2}\right)^k \leq 2$

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

- Total Space: $O(n)$ for score computation, $O(n+m)$ to store the optimal alignment

Geometric Reduction At Each Iteration:

- First pass: 1
- Second pass: $\frac{1}{2}$
- Third pass: $\frac{1}{4}$
- Fourth pass: $\frac{1}{8}$
- Fifth pass: $\frac{1}{16}$
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
  
  $\begin{array}{c}
  |u_1\ldots u_t| \\
  |u_{t+1}\ldots u_{2t}| \\
  \vdots \\
  |u_{n-t+1}\ldots u_n|
  \end{array}$

  and sequence $v = v_1\ldots v_n$ becomes
  
  $\begin{array}{c}
  |v_1\ldots v_t| \\
  |v_{t+1}\ldots v_{2t}| \\
  \vdots \\
  |v_{n-t+1}\ldots v_n|
  \end{array}$
Partitioning Alignment Grid into Blocks

\[ n \]

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

\[ \text{partition} \]
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 $\beta_{i,j}$ for each pair of blocks $|u_{(i-1)\cdot t+1} \ldots u_{i\cdot t}|$ and $|v_{(j-1)\cdot t+1} \ldots v_{j\cdot t}|$

• 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

• Looks like a wash \(O((n/t)^2 \cdot t^2) = O(n^2)\), but is it?
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 $v$

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

- $\sigma_{\text{block}}$ is the penalty for inserting or deleting an entire block
- $\beta_{i,j}$ 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

$$O\left(\frac{n}{t} \times \frac{n}{t} \times O(\beta_{i,j})\right) = O\left(\frac{n^2}{t^2}\right)$$

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

• So computing all $\beta_{i,j}$ takes time

$$O\left(\frac{n^2}{t^2} \times t^2\right) = O(n^2)$$

• 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 there 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 revisiting alignments!
Four Russians Technique

• The trick is in how to pick $t$ relative to $n$
• If we pick $t = \log_2(n)/4$
• 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.
• If $t = (\log_2 n)/4$. Then $4^t \times 4^t = 4^{\sqrt{n^2}} \times 4^{\sqrt{n^2}} = n$
Look-up Table for Four Russians Technique

Each sequence has $t$ nucleotides

<table>
<thead>
<tr>
<th></th>
<th>AAAA</th>
<th>AAAAC</th>
<th>AAAAG</th>
<th>AAAAT</th>
<th>AAAACA</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAA</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AAAAAC</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>AAAAAG</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>AAAAAT</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>AAAACA</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

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 \( \text{Score} \) is indexed by a pair of \( t \)-nucleotide strings, so

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

• Since computing the lookup table $\text{Score}$ of size $n$ takes $O(n)$ time, the running time is dominated by the $(n/t)*(n/t)$ accesses to the lookup table.

• Overall running time: $O\left( \frac{n^2}{t^2} \right)$

• Since $t = (\log_2 n)/4$, substitute in:

• $O\left( \frac{n^2}{(\log_2 n)^2} \right) = O\left( \frac{n^2}{\log(n \log n)} \right)$
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: $O(n^2 / \log(n \log n))$. 
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 \((n/t)^2(n/t) = (n^2/t^2)\) 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 (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; 0,1,2,4,5,8, 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

\[
\begin{array}{cccccc}
0 & 1 & 2 & 2 & 3 & 4 \\
1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

original encoding

binary encoding
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 = $O\left(\frac{n^2}{t^2}\right) \times \log n$
- $O\left(\frac{n^2}{(\log n)^2}\right) \log n = O\left(\frac{n^2}{\log n}\right)$
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: $O(n^2 / \log(n \log n))$

• Next Time: Graph Algorithms