Is It Possible to Align Sequences in Subquadratic Time?

- Dynamic Programming takes O(n^2) for global alignment
- Can we do better?
- Yes, use *Four-Russians Speedup*

Partitioning Sequences into Blocks

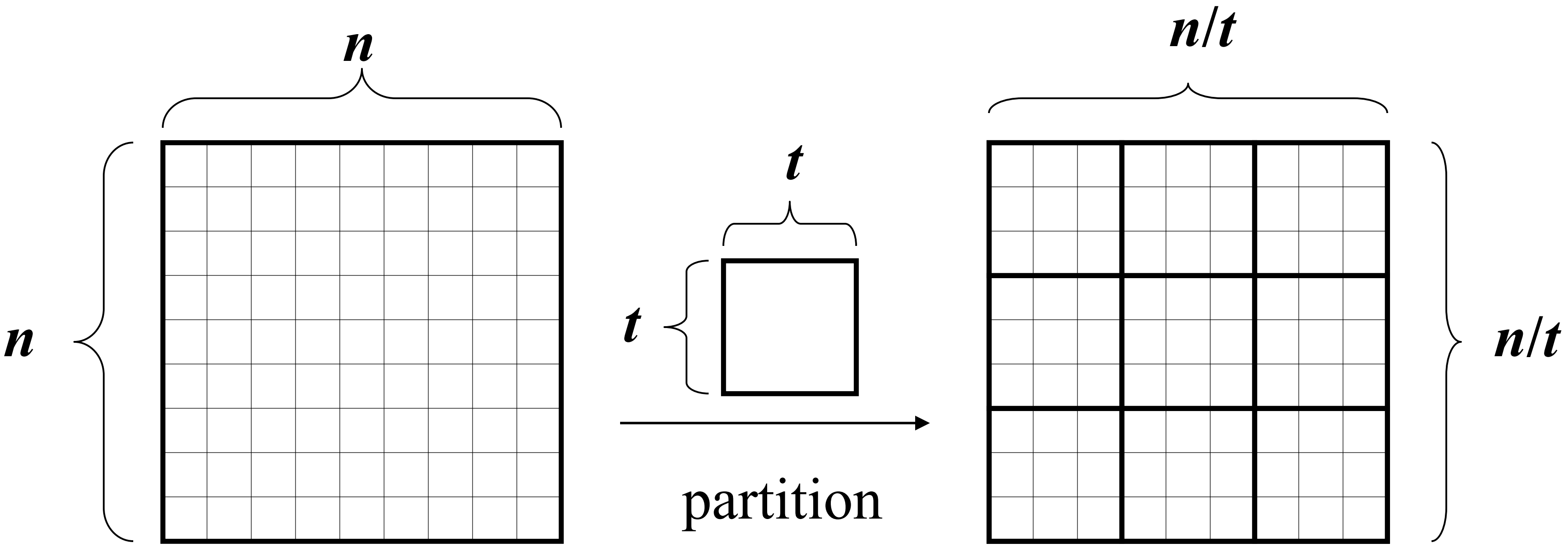
- Partition the *n* x *n* grid into blocks of size *t* x *t*
- We are comparing two sequences, each of size *n*, and each sequence is sectioned off into chunks, each of length *t*
- Sequence $\boldsymbol{u} = u_1 \ldots u_n$ becomes

$$|u_1 \ldots u_t| \; |u_{t+1} \ldots u_{2t}| \; \ldots \; |u_{n-t+1} \ldots u_n|$$

and sequence $\boldsymbol{v} = v_1 \ldots v_n$ becomes

$$|v_1 \ldots v_t| \; |v_{t+1} \ldots v_{2t}| \; \ldots \; |v_{n-t+1} \ldots v_n|$$

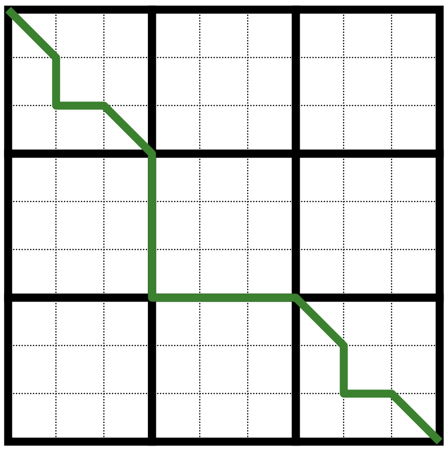
Partitioning Alignment Grid into Blocks



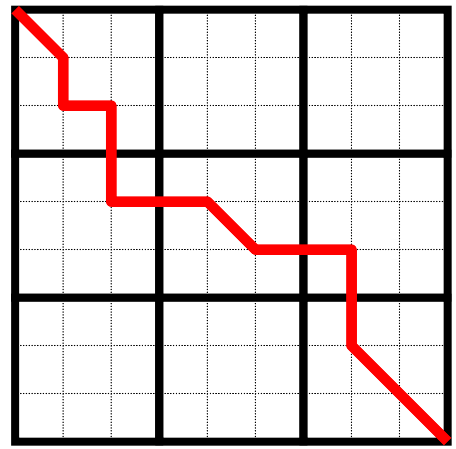
Block Alignment

- **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 x 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* x *n* edit graph partitioned into *t* x *t* subgrids
- Output: The block alignment of ***u*** and ***v*** with the maximum score (longest block path through the edit graph

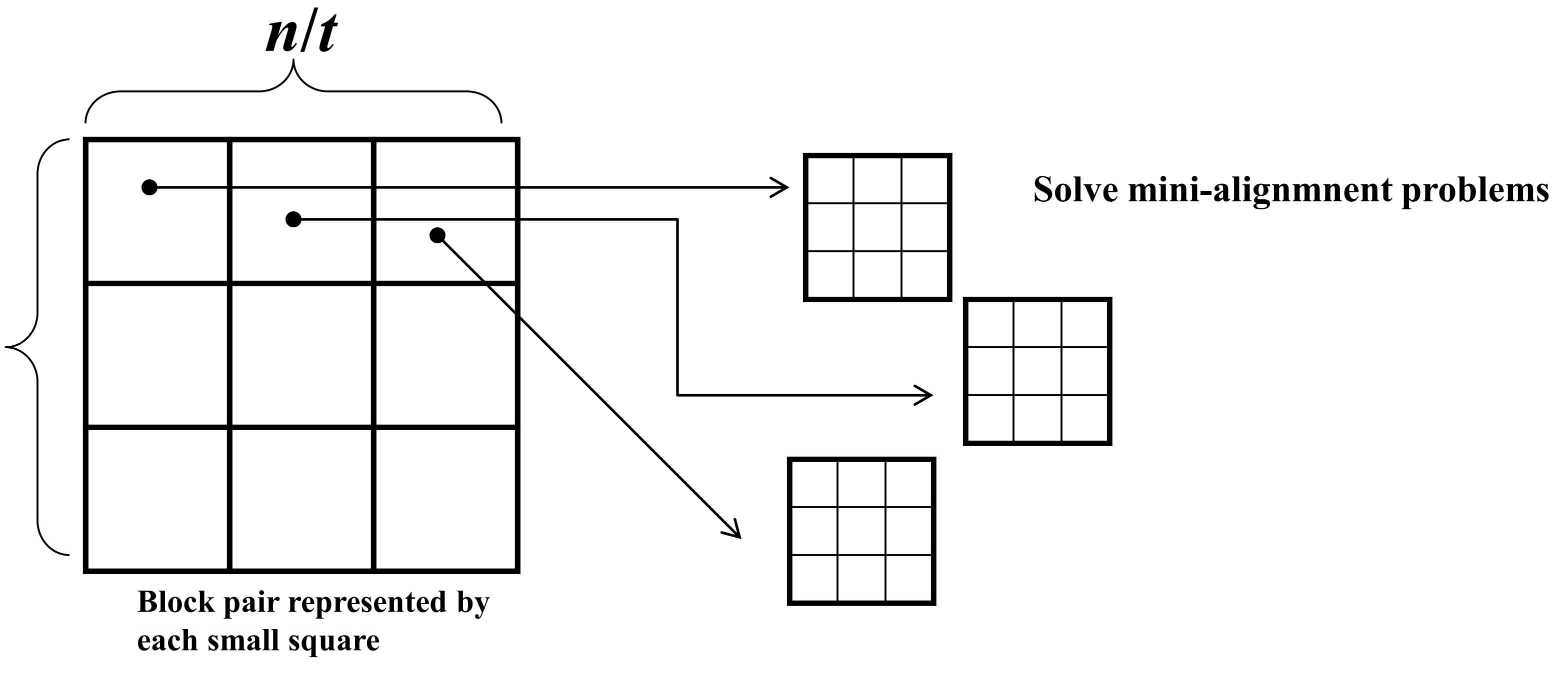
Constructing Alignments within Blocks

- To solve: compute alignment score $ß_{i,j}$ for each pair of blocks $|u_{(i-1)*t+1} \ldots u_{i*t}|$ and $|v_{(j-1)*t+1} \ldots v_{j*t}|$
- 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) x (n/t)
- For each block pair, solve a mini-alignment problem of size t x t

Constructing Alignments within Blocks



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***

$$s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} - \beta_{i,j} \end{array} \right\}$$

σ_{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

$$O(\ [n/t]*[n/t]) = O(n^2/t^2)$$

if we don't count the time to compute each $\beta_{i,j}$

Block Alignment Runtime (cont'd)

- Computing all $\beta_{i,j}$ requires solving (n/t)*(n/t) mini block alignments, each of size $(t*t)$
- So computing all $\beta_{i,j}$ takes time

$$O([n/t]*[n/t]*t*t) = O(n^2)$$

- This is the same as dynamic programming
- How do we speed this up?

Four Russians Technique

- Let $t = \log(n)$, where t is block size, n is sequence size.
- Instead of having $(n/t)*(n/t)$ mini-alignments, construct $4^t \times 4^t$ mini-alignments for all pairs of strings of t nucleotides, and put in a lookup table.
- However, size of lookup table is not really that huge if t is small. Let $t = (\log \underline{n})/4$. Then $4^t \times 4^t = n$

Look-up Table for Four Russians Technique

each sequence has t nucleotides

Lookup table “*Score*”

	AAAAAA	AAAAAC	AAAAAG	AAAAAT	AAAACA	...
AAAAAA						
AAAAAC						
AAAAAG						
AAAAAT						
AAAACA						
...						

size is only n, instead of $(n/t)*(n/t)$

New Recurrence

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

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{\text{block}} \\ s_{i,j-1} - \sigma_{\text{block}} \\ s_{i-1,j-1} - Score(i^{\text{th}} \text{ block of } v,\ j^{\text{th}} \text{ block of } u) \end{cases}$$

Four Russians Speedup Runtime

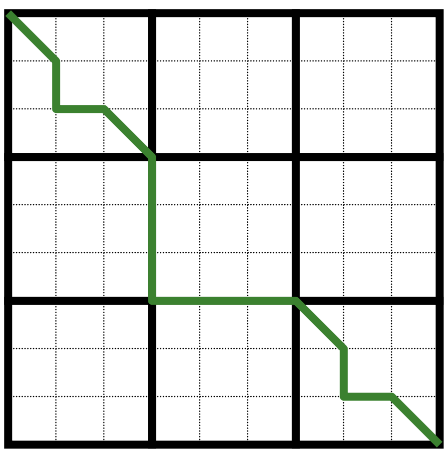
- Since computing the lookup table *Score* of size n takes $O(n)$ time, the running time is mainly limited by the $(n/t)*(n/t)$ accesses to the lookup table
- Each access takes $O(\log n)$ time
- Overall running time: $O([n^2/t^2]*\log n)$
- Since $t = \log n$, substitute in:
- $O([n^2/\{\log n\}^2]*\log n) \geq O(n^2/\log n)$

So Far…

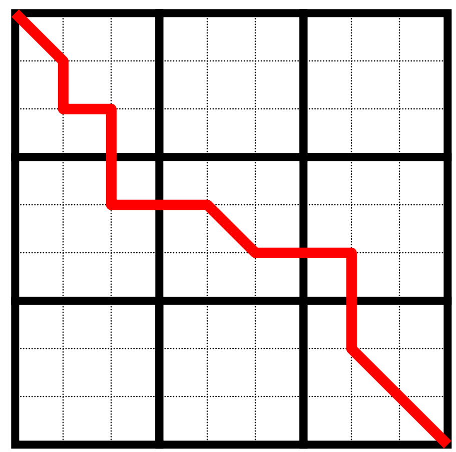
- We can divide up the grid into blocks and run dynamic programming only on the corners of these blocks
- In order to speed up the mini-alignment calculations to under n^2, we create a lookup table of size n, which consists of all scores for all t-nucleotide pairs
- Running time goes from quadratic, $O(n^2)$, 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

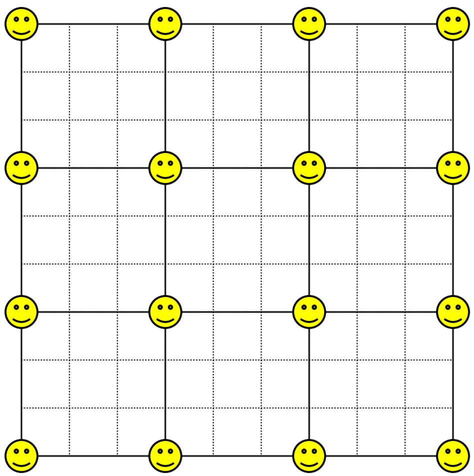


longest common subsequence

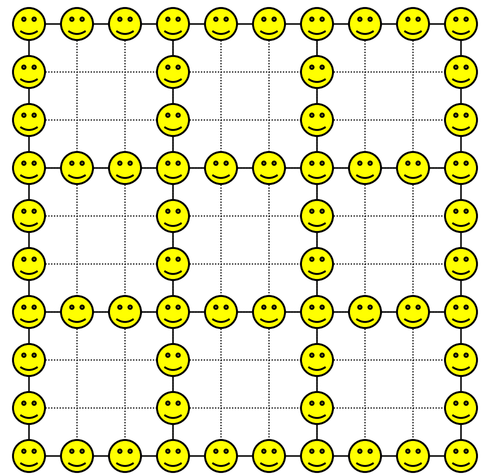
Block Alignment vs. LCS

- 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)*(n/t) = (n^2/t^2) points of interest



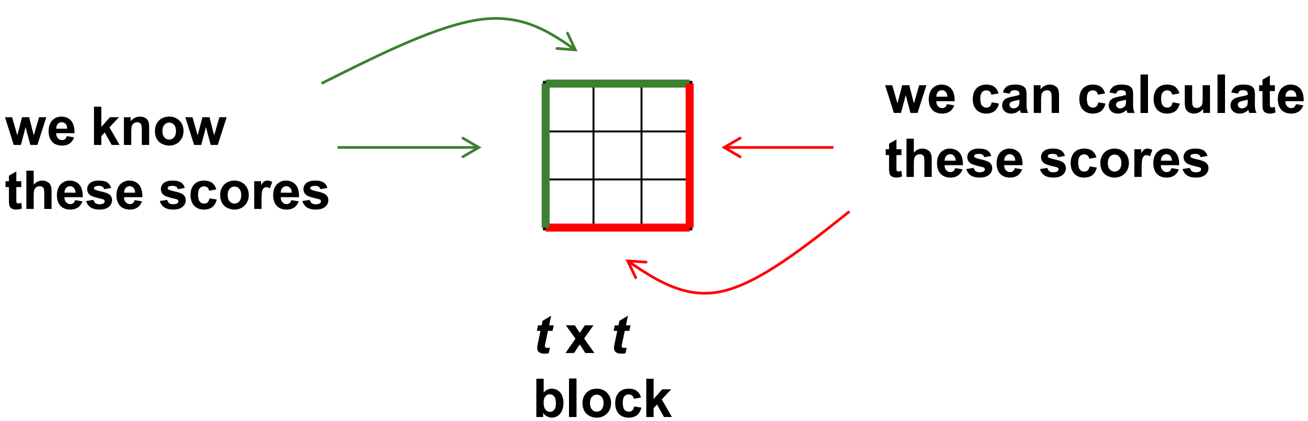
LCS alignment has $O(n^2/t)$ points of interest

Traversing Blocks for LCS

- Given alignment scores $s_{i,*}$ in the first row and scores $s_{*,j}$ in the first column of a t x 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.



Four Russians Speedup

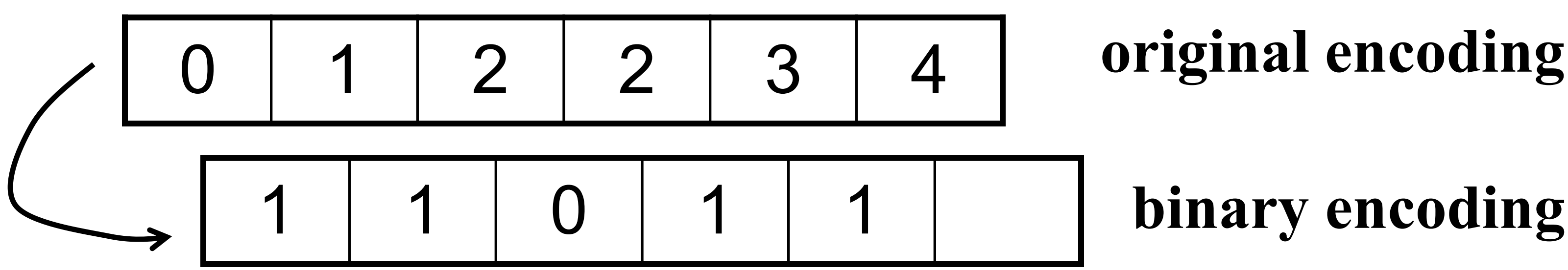
- Build a lookup table for all possible values of the four variables:
  1. all possible scores for the first row $s_{*,j}$
  2. all possible scores for the first column $s_{*,j}$
  3. substring of sequence u in this block (4^t possibilities)
  4. substring of sequence v in this block (4^t possibilities)
- 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 = size of blocks)
- 4^t possible strings
  - Lookup table size is $(2^t * 2^t)*(4^t * 4^t) = 2^{6t}$
- 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]*log$n$)
- O([$n^2/\{\log n\}^2$]*log$n$) $\geq$ O($n^2/\log n$)

Main Observation

Within a rectangle of the DP matrix,

values of D depend only

on the values of A, B, C,

and substrings $x_{l...l'}$, $y_{r...r'}$

Definition:

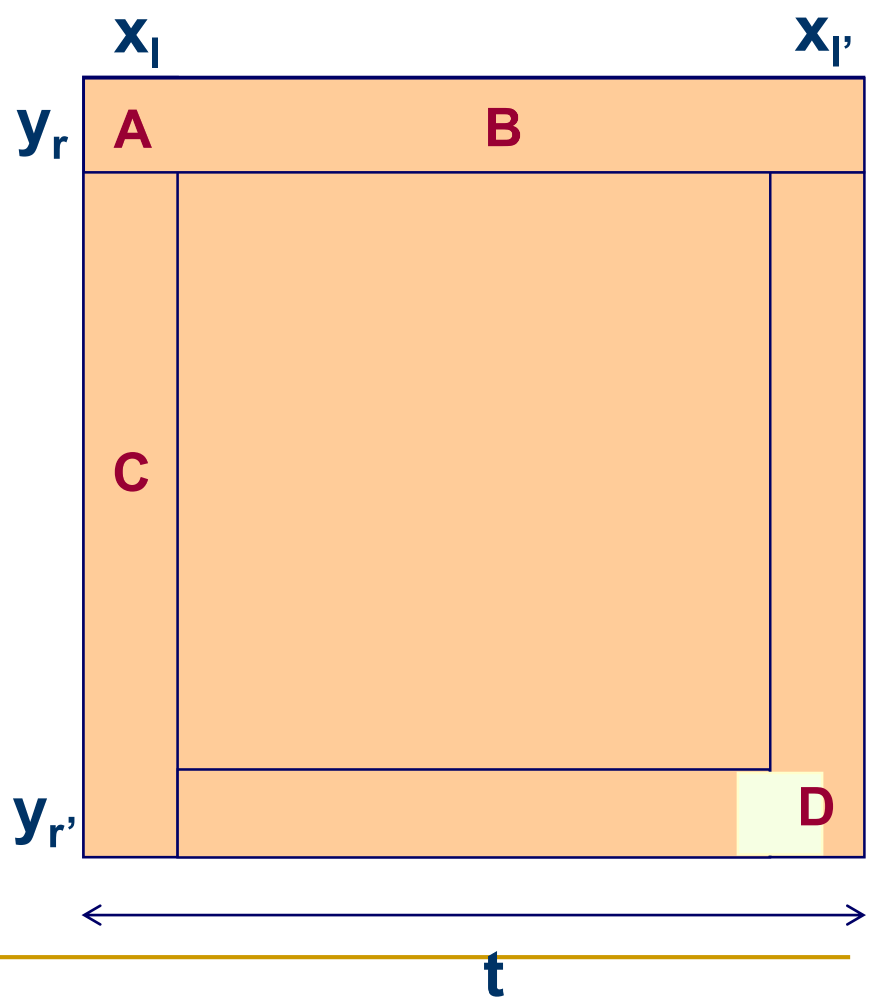
A t-block is a $t \times t$ square of the DP matrix

Idea:

Divide matrix in t-blocks,

Precompute t-blocks

Speedup: O(t)



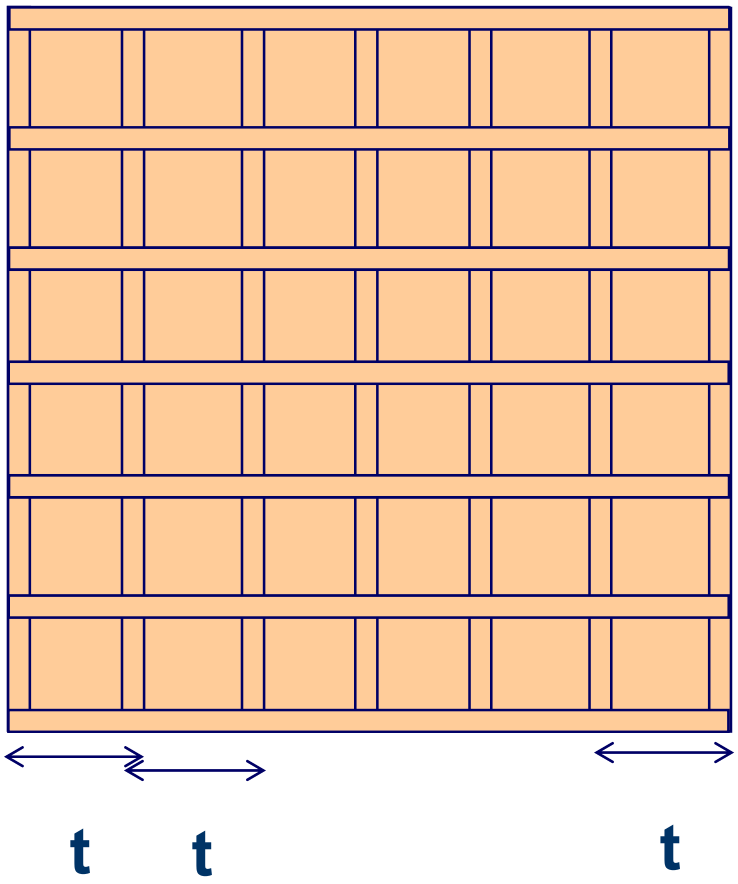
The Four-Russians Algorithm

Main structure of the algorithm:

- Divide N×N DP matrix into K×K $\log_2 N$-blocks that overlap by 1 column & 1 row

- For i = 1……K
- For j = 1……K
- Compute $D_{i,j}$ as a function of $A_{i,j}$, $B_{i,j}$, $C_{i,j}$, $x[l_i \ldots l'_i]$, $y[r_j \ldots r'_j]$

Time: $O(N^2 / \log^2 N)$



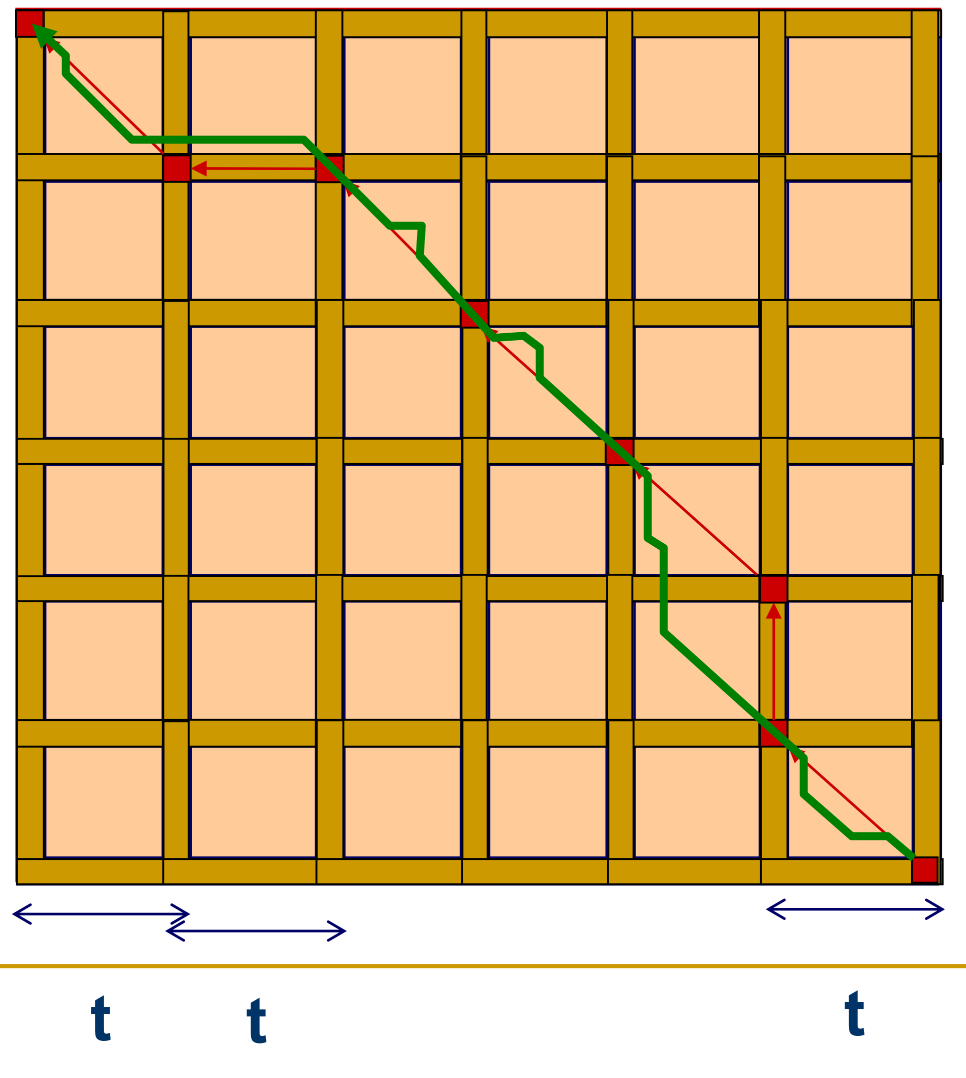
The Four-Russians Algorithm

Four-Russians Algorithm: (Arlazarov, Dinic, Kronrod, Faradzev)

1. Cover the DP table with t-blocks
2. Initialize values F(.,.) in first row & column
3. Row-by-row, use offset values at leftmost column and top row of each block, to find offset values at rightmost column and bottom row
4. Let Q = total of offsets at row n; F(n, n) = Q + F(n, 0) = Q + n

Runtime: $O(n^2 / \log n)$

The Four-Russians Algorithm



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 $n^{(3/2)}$
- Four Russians speedup: from a quadratic running time for LCS to subquadratic running time: $O(n^2/\log \underline{n})$