CS481/CS583: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/

APPROXIMATE STRING MATCHING: BANDED ALIGNMENT

Limiting gaps

- We know how to calculate global and local alignments in O(mn) time
- What if the problem definition limits the gaps to w, where $w<<n$ and $w<<m$?
  - Can we improve run time?

Limiting gaps

		A	C	C	A	C	A	C	A
	0								
A		1							
C			2						
A				1					
C					0				
C						1			
A							2		
T								1	
A									2

Example: Limit gaps to *w*=2

Banded global alignment

		A	C	C	A	C	A	C	A
	0	-2	-4						
A	-2	1	-1	-3					
C	-4	-1	2	0	-2				
A		-3	0	1	1	-1			
C			-2	1	0	2	0		
C				-1	0	1	1	1	
A					0	-1	2	0	2
T						-1	0	1	0
A							0	-1	2

- Example
  - w=2
- What's the running time?

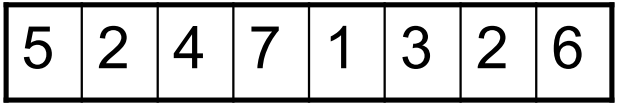
DP IN LINEAR SPACE & DIVIDE AND CONQUER ALGORITHMS

Divide and Conquer Algorithms

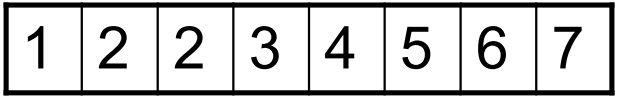
- **Divide** problem into sub-problems
- **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion
- **Combine** the solutions of sub-problems into a solution of the original problem (tricky part)

Sorting Problem

- Given: an unsorted array

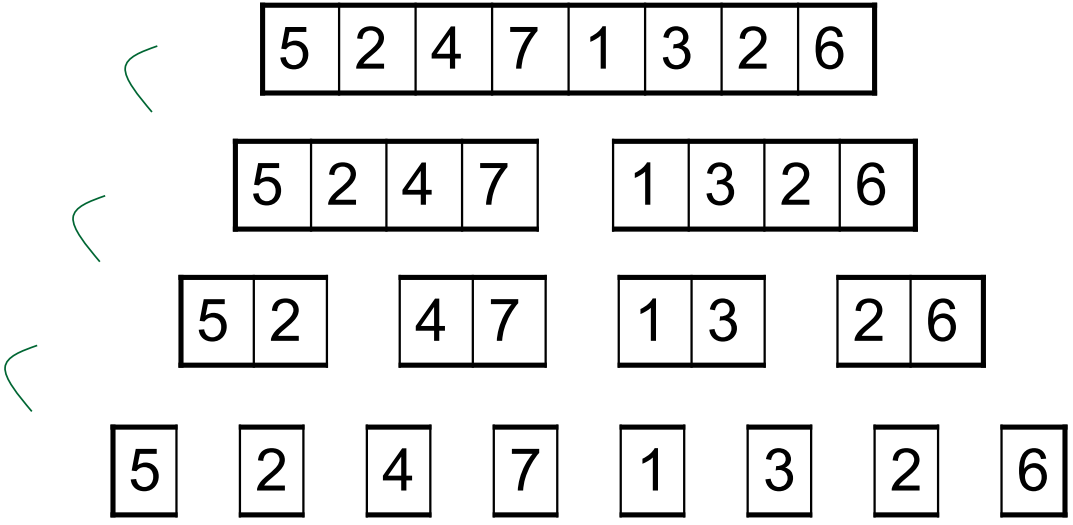


- Goal: sort it



Mergesort: Divide Step

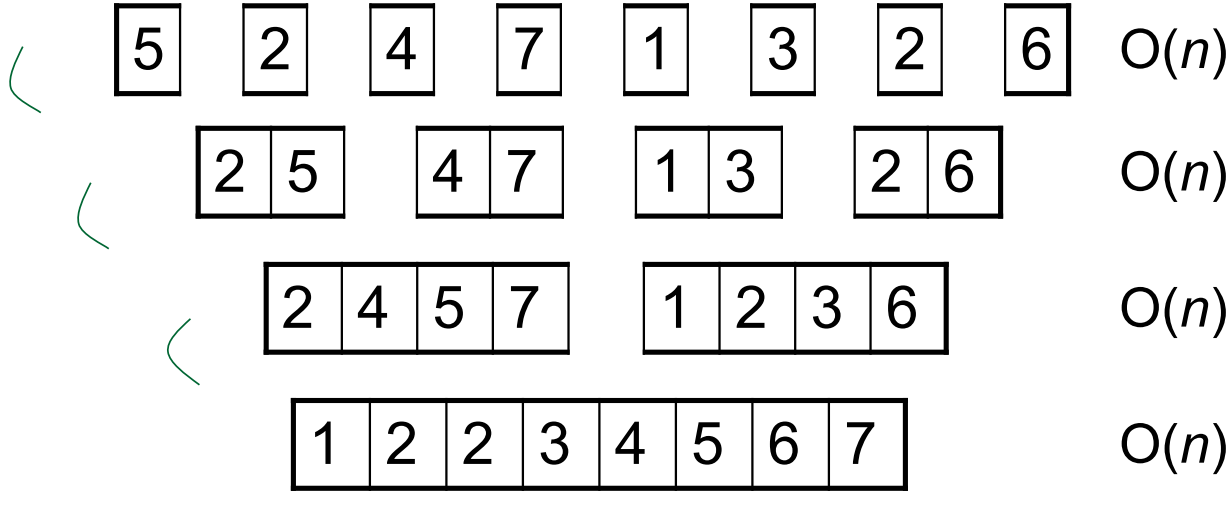
Step 1 – Divide



log(*n)* divisions to split an array of size *n* into single elements

Mergesort: Conquer Step

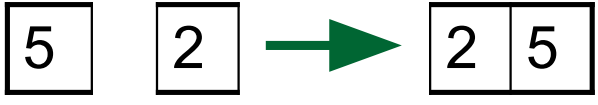
Step 2 – Conquer



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

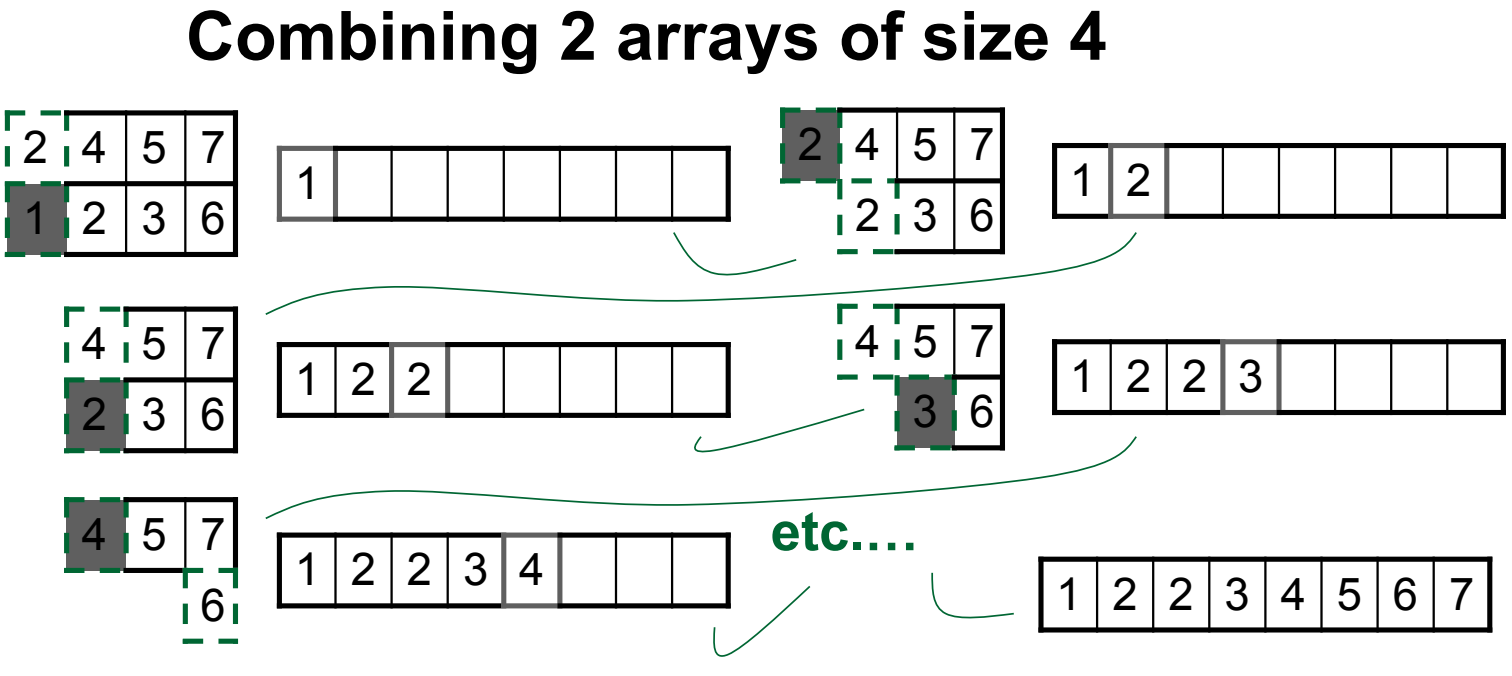
Mergesort: Combine Step

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*

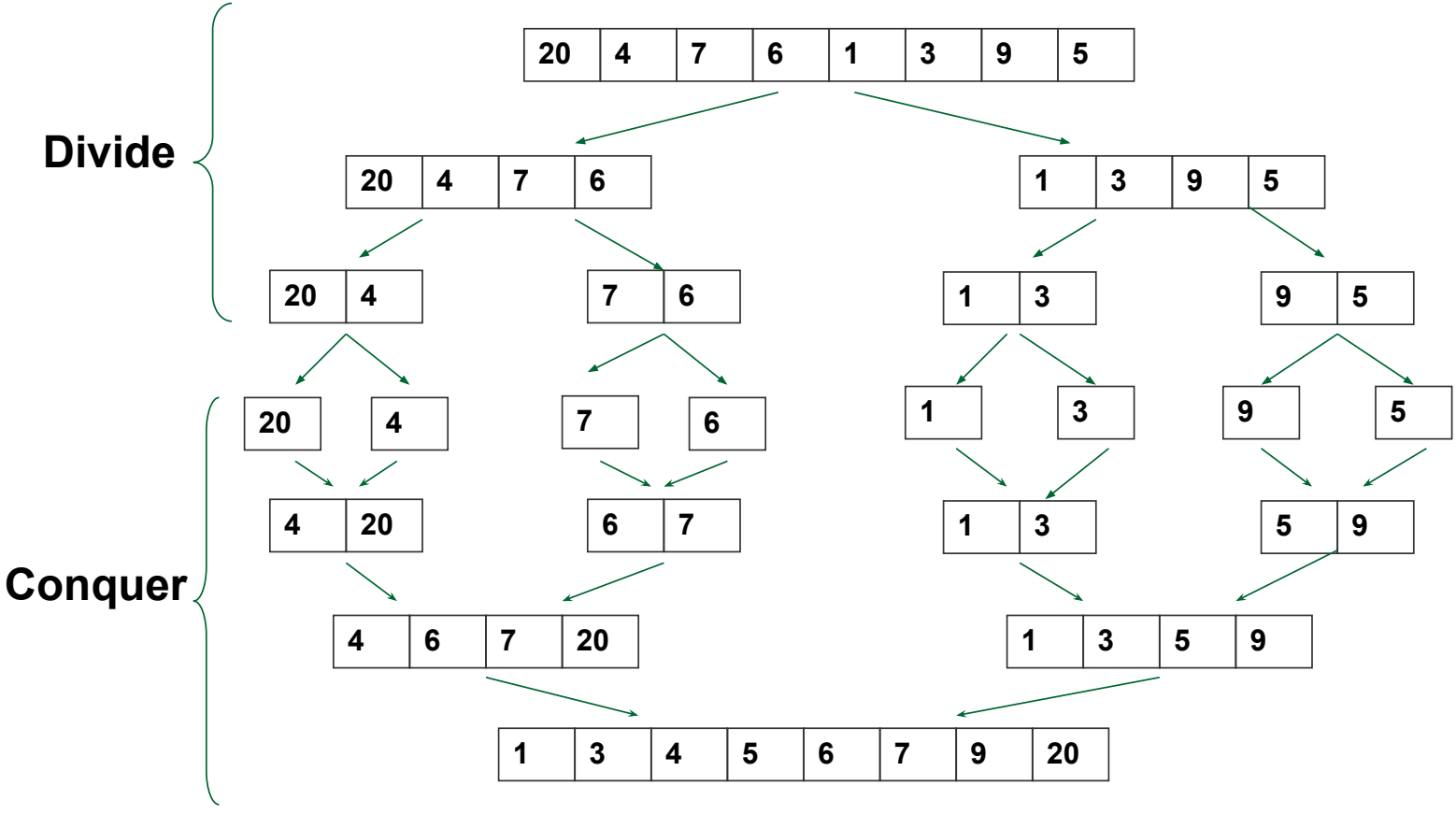
Mergesort: Combine Step



Merge Algorithm

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

Mergesort: Example



MergeSort Algorithm

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

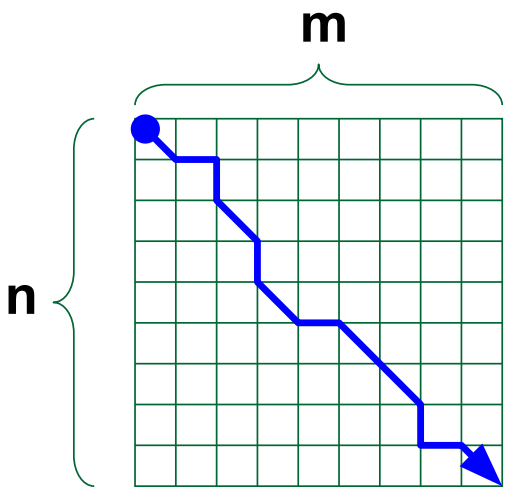
MergeSort: Running Time

- The problem is simplified to smaller 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 Alignment Path Requires Quadratic Memory

Alignment Path

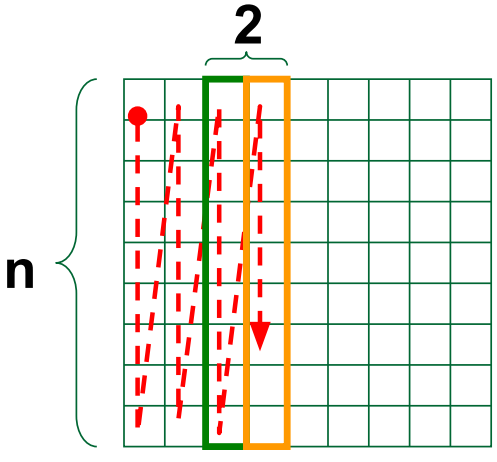
- Space complexity for computing alignment path for sequences of length *n* and *m* is O(*nm*)
- We need to keep all backtracking references in memory to reconstruct the path (backtracking)



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



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*)

Divide and Conquer Approach to LCS

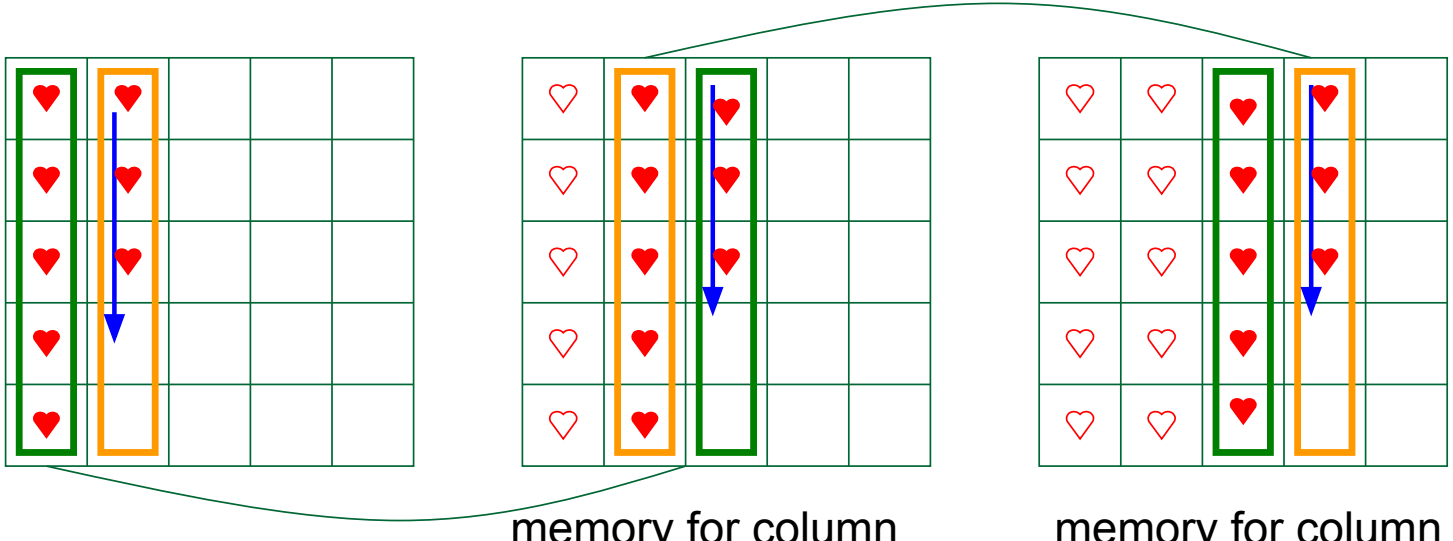
Path(*source, sink*)

- **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"!

Computing Alignment Score: Recycling Columns

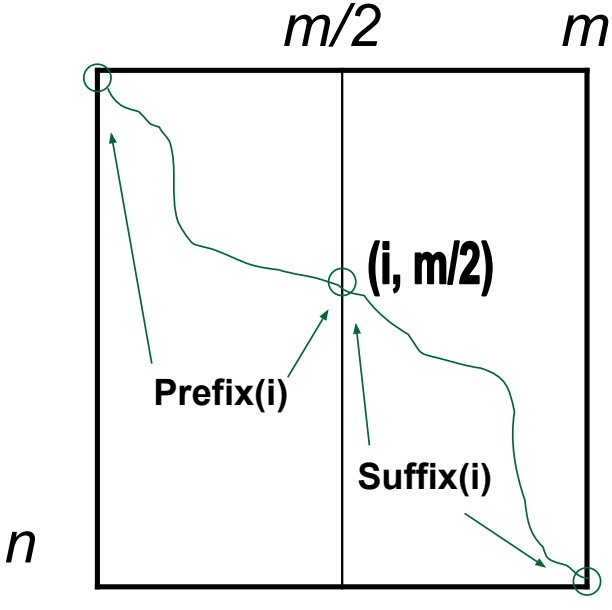
Only two columns of scores are saved at any given time



memory for column 1 is used to calculate column 3

memory for column 2 is used to calculate column 4

Crossing the Middle Line



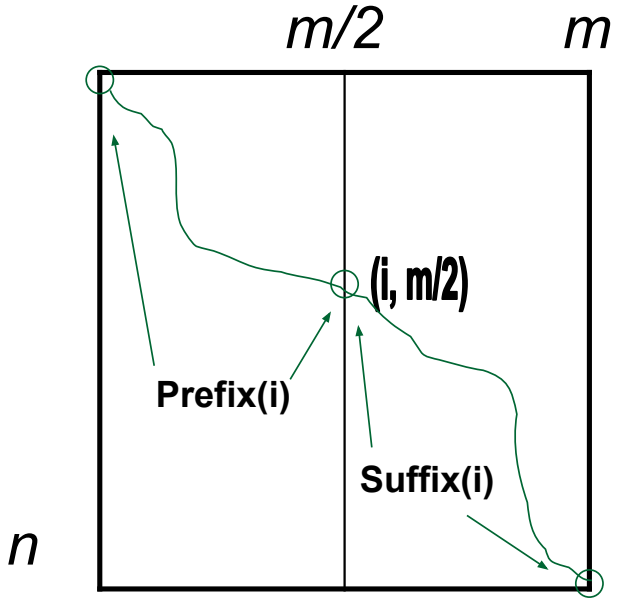
We want to calculate the longest path from (0,0) to (*n*,*m*) that passes through (*i*,*m*/2) where *i* ranges from 0 to *n* and represents the *i*-th row

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

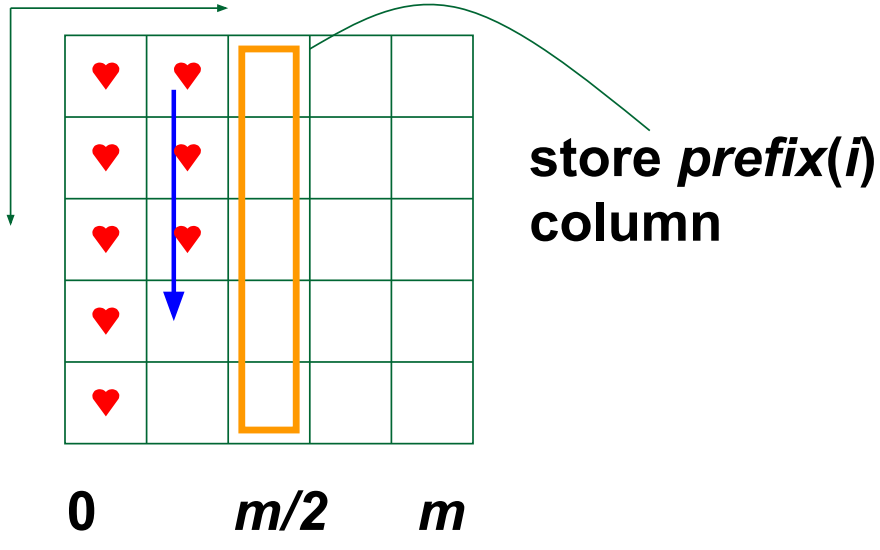


Define (*mid*,*m*/2) as the vertex where the longest path crosses the middle column.

$$length(mid) = \text{optimal length} = \max_{0 \le i \le n} length(i)$$

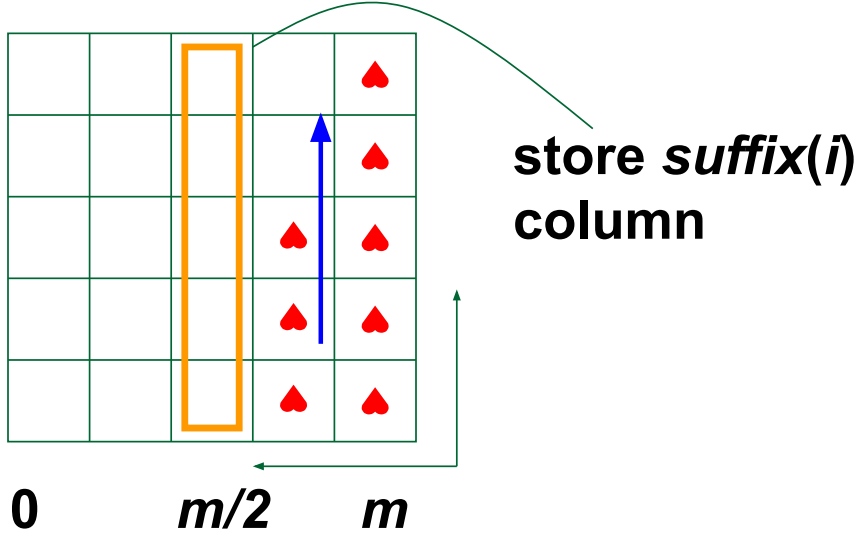
Computing Prefix(*i*)

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



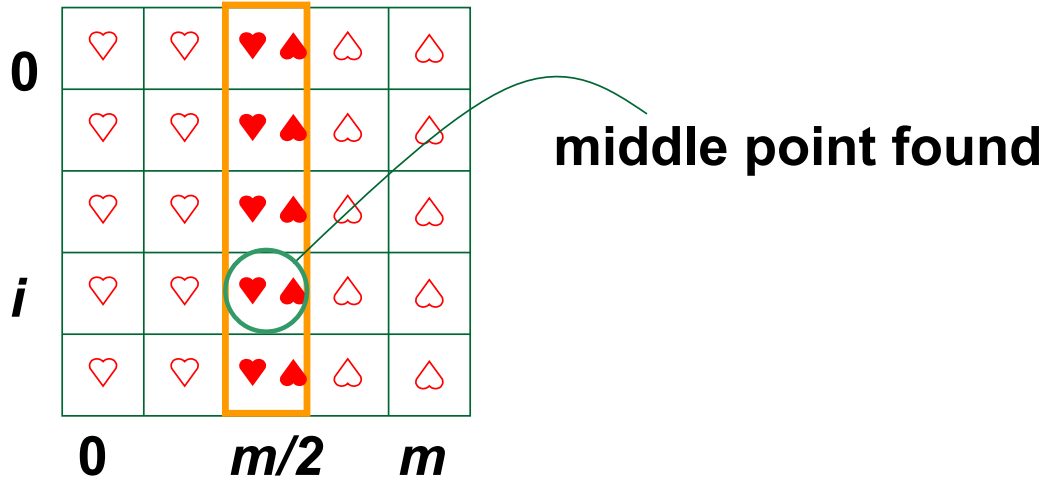
Computing Suffix(*i*)

- *suffix*(*i*) is the length of the longest path from (*i*,*m*/2) to *(n,m)*
- *suffix*(*i*) is the length of the longest path from (*n*,*m*) to (*i*,*m*/2) with all edges reversed
- Compute *suffix*(*i*) by dynamic programming in the right half of the "reversed" matrix

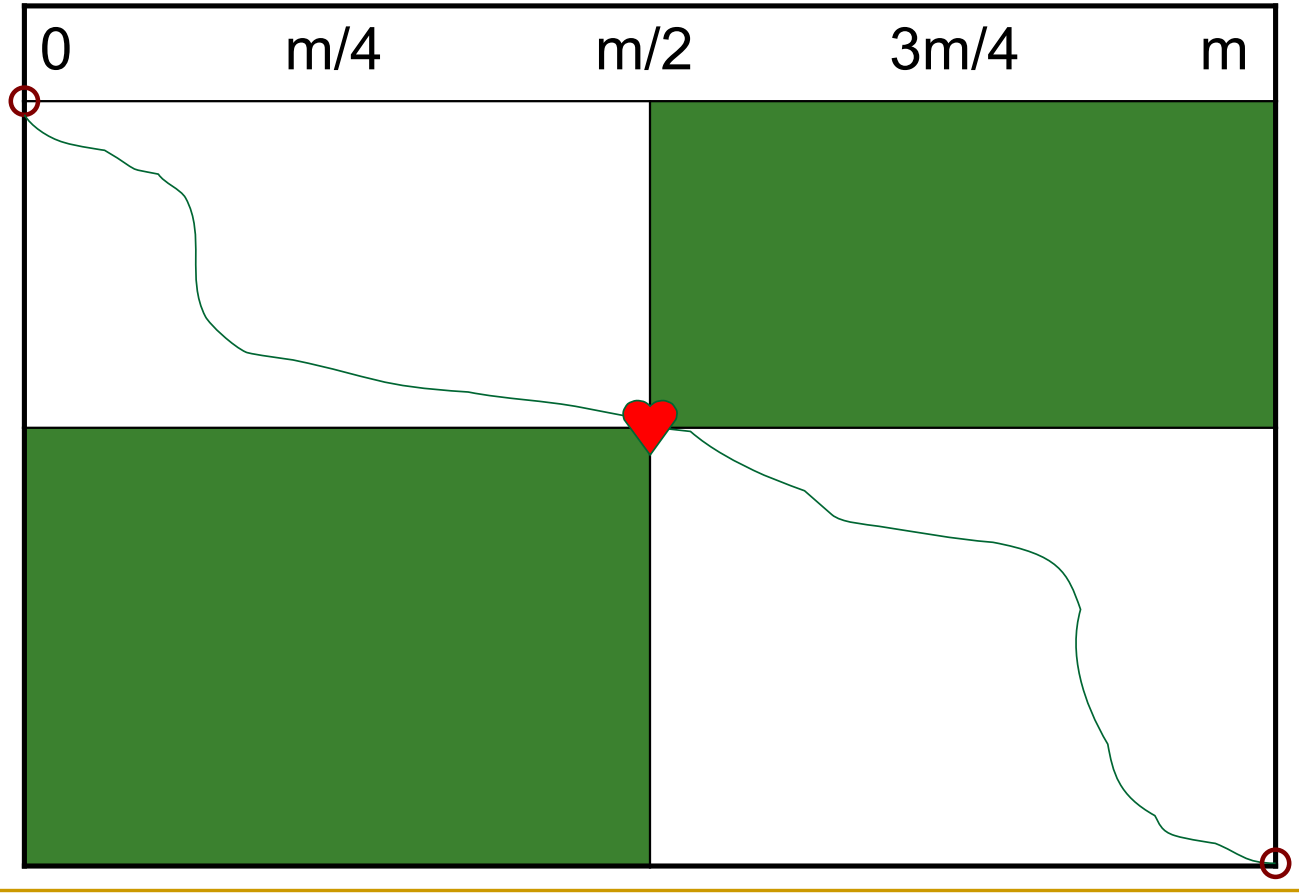


Length(i) = Prefix(i) + Suffix(i)

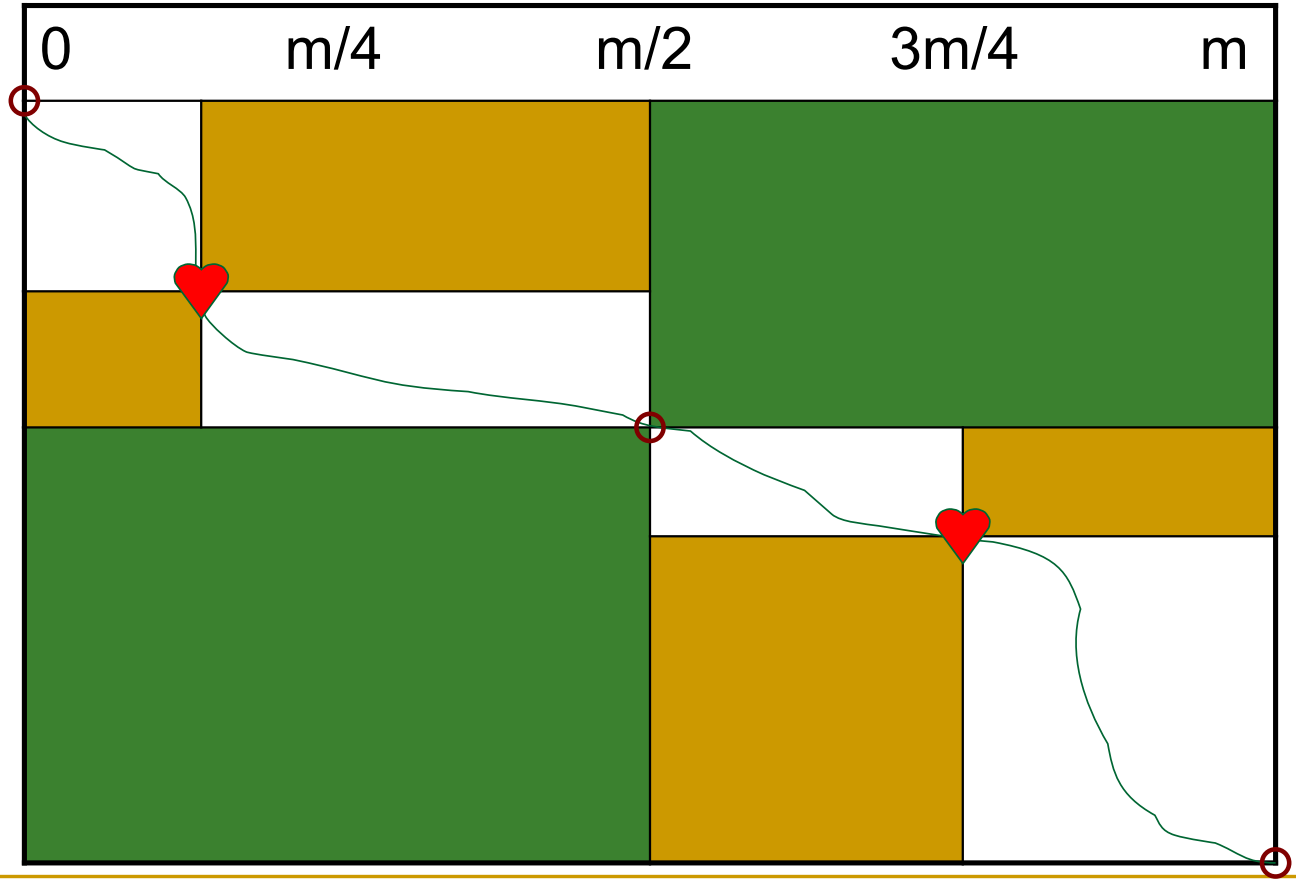
- Add *prefix*(*i*) and *suffix*(*i*) to compute *length(i):*
  - *length*(*i*)=*prefix*(*i*) + *suffix*(*i*)
- You now have a middle vertex of the maximum path (*i,m*/2) as maximum of *length(i)*



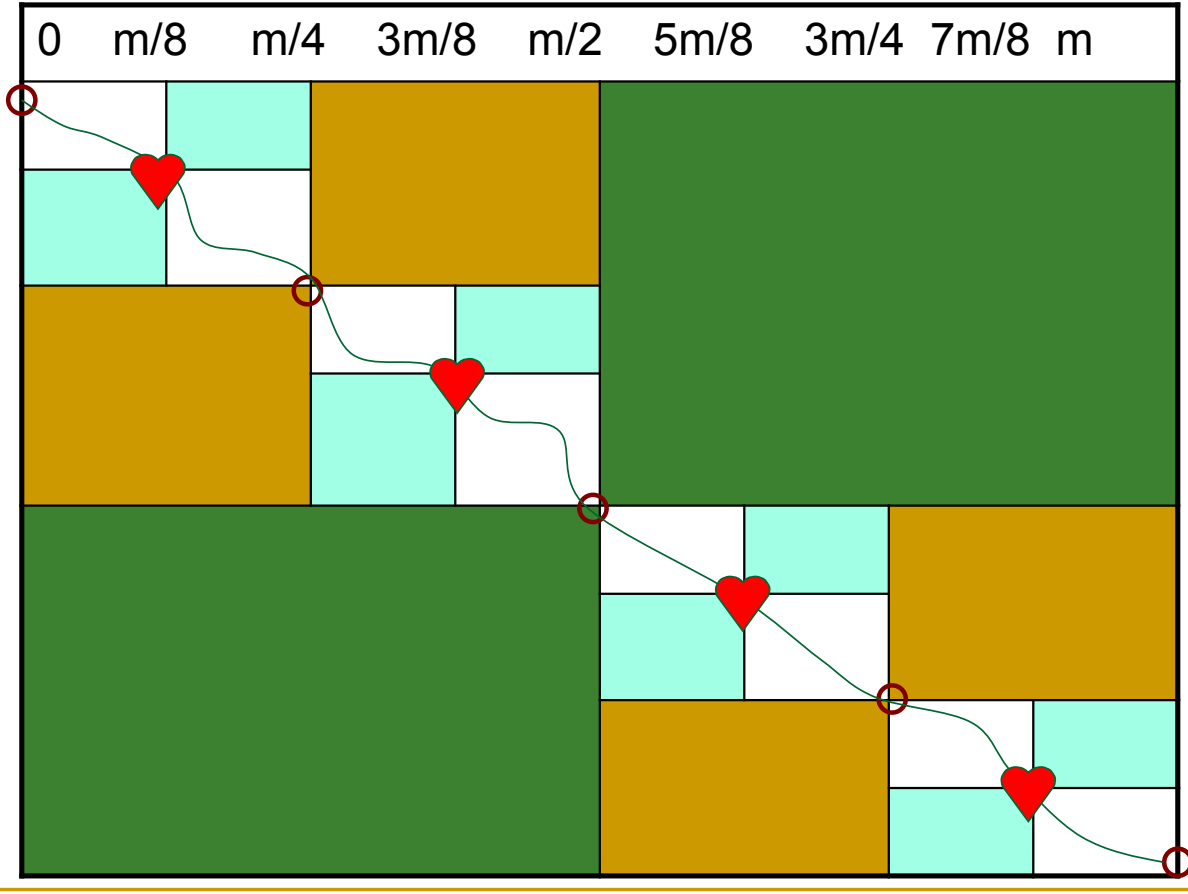
Finding the Middle Point



Finding the Middle Point again



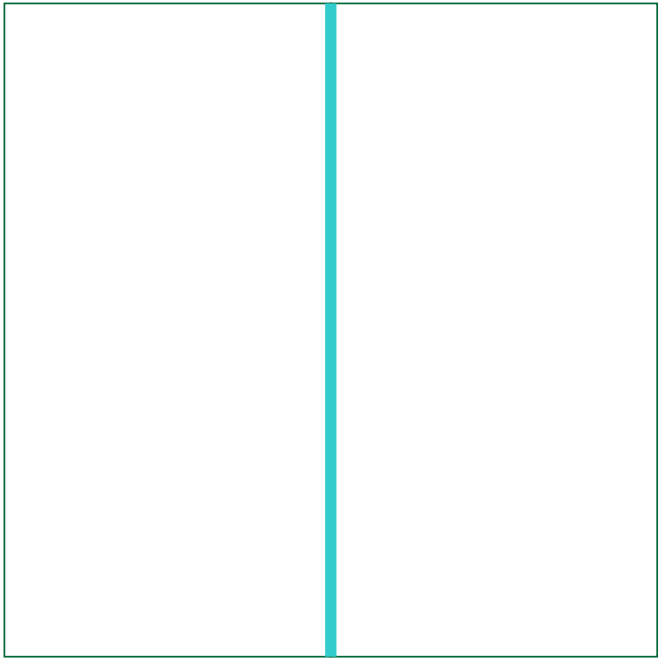
And Again



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

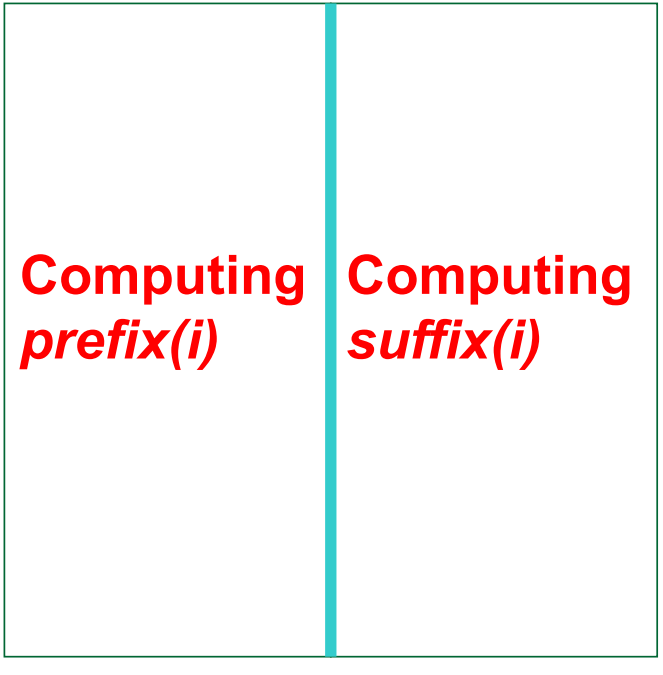
Area = $n \cdot m$



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

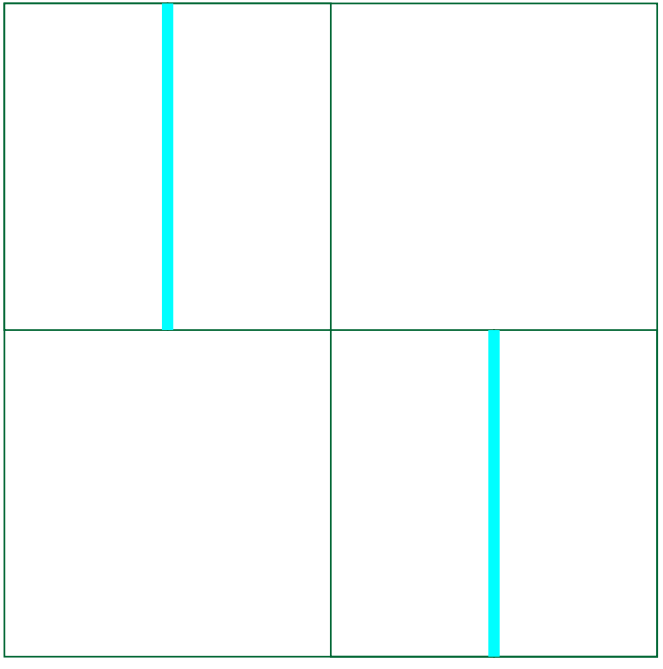
Area = $n \cdot m$



Time = Area: Second Pass

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

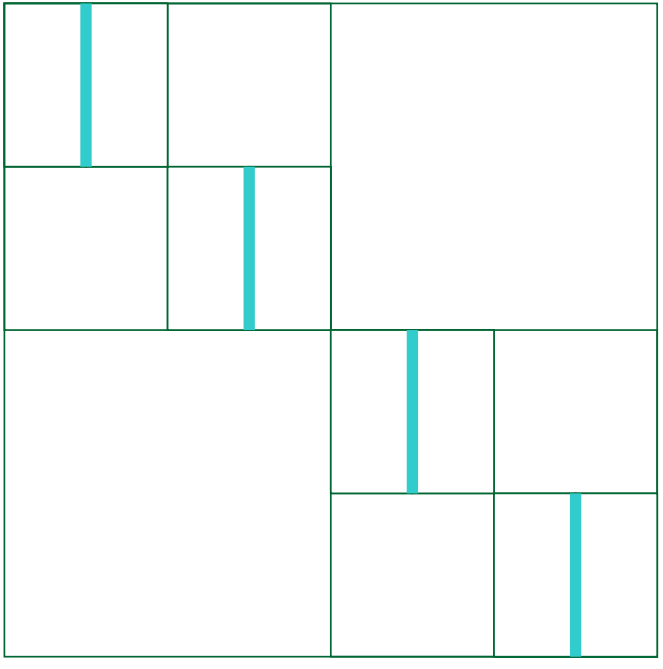
Area/2



Time = Area: Third Pass

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

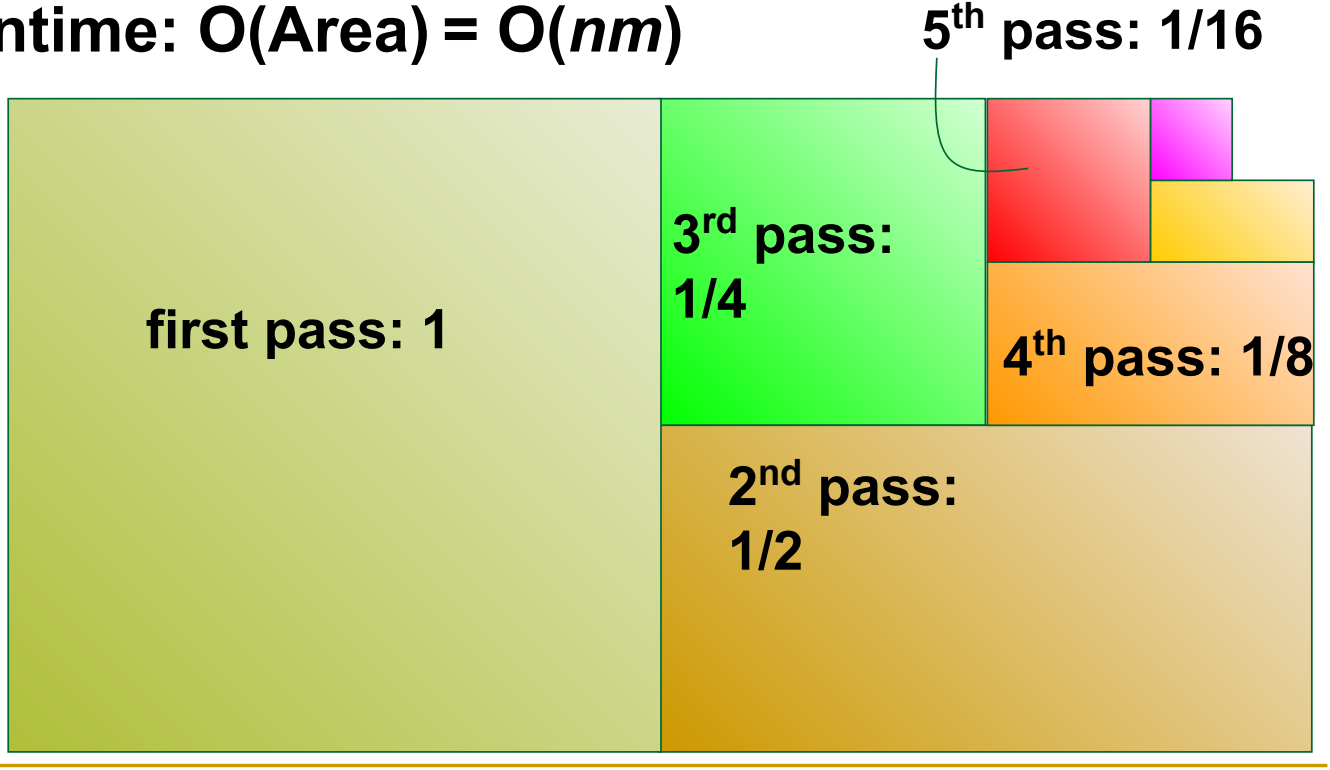
Area/4



Geometric Reduction At Each Iteration

$$1 + ½ + ¼ + \ldots + (½)^k \leq 2$$

- **Runtime: O(Area) = O(*nm*)**



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

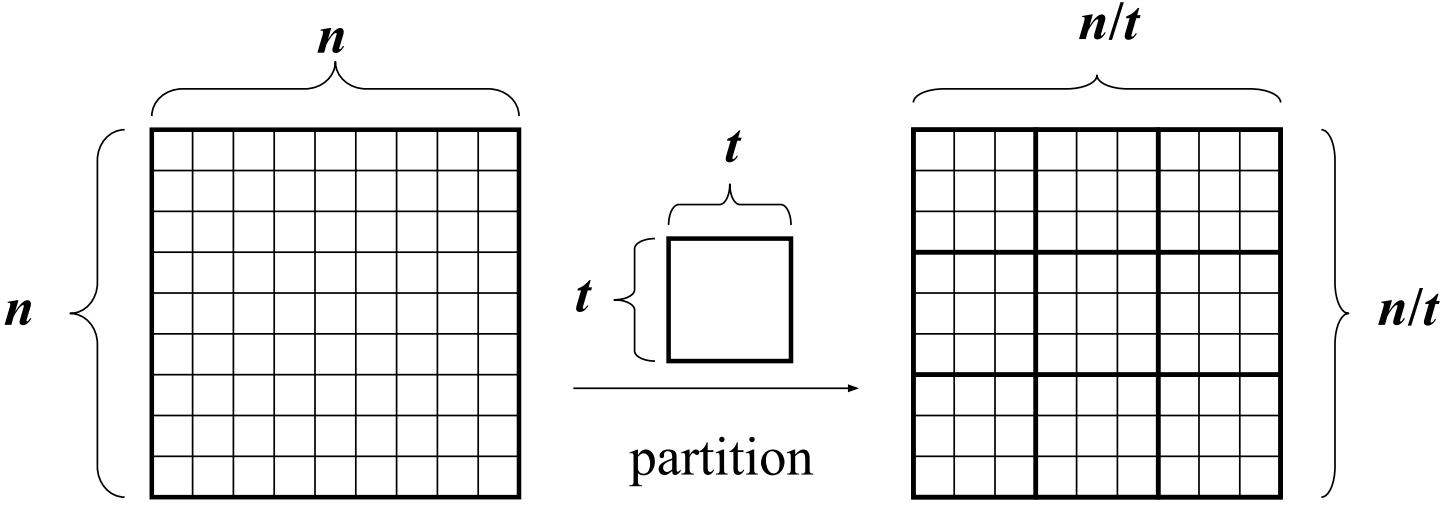
- Partition the n x n grid into blocks of size t x t
- We are comparing two sequences, each of size n, and each sequence is sectioned off into chunks, each of length t
- Sequence $\boldsymbol{u} = u_1 \ldots u_n$ becomes

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

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

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

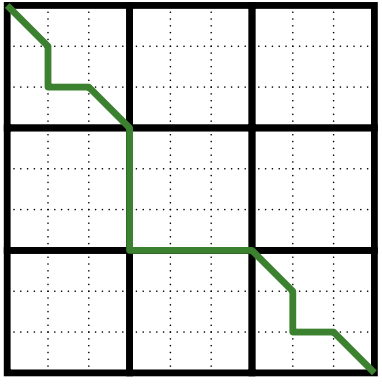
Partitioning Alignment Grid into Blocks



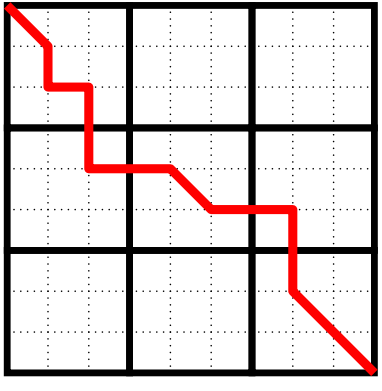
Block Alignment

- **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

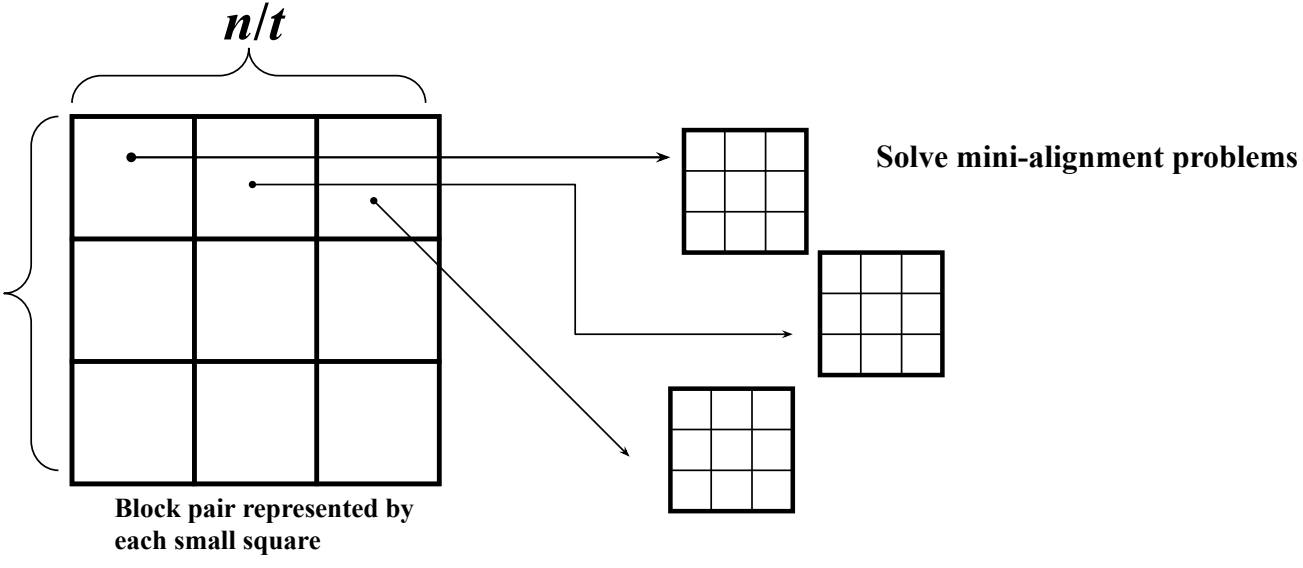
Constructing Alignments within Blocks

- 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***

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

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

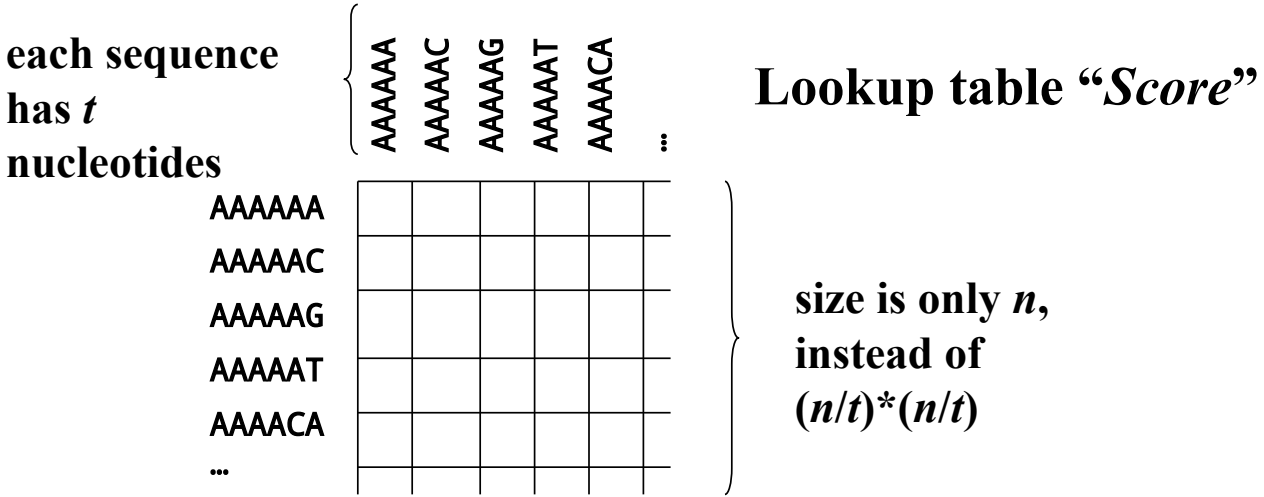
$$O([n/t]*[n/t]*t*t) = O(n^2)$$

- 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 $t = (\log n)/4$. Then $4^t \times 4^t = n$

Look-up Table for Four Russians Technique



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

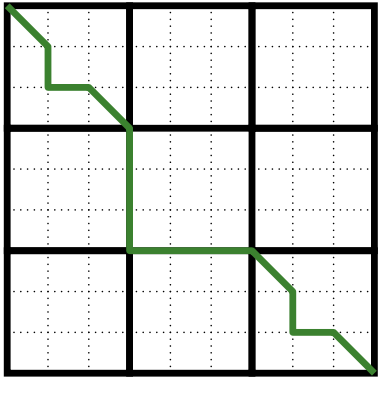
- Since computing the lookup table *Score* of size n takes O(n) time, the running time is mainly limited by the (n/t)*(n/t) accesses to the lookup table
- Each access takes O(log$n$) time
- Overall running time: O([n^2/t^2]*log$n$)
- Since t = log$n$*(/4)*, substitute in:
  - O([n^2/{log$n$}^{2}]*log$n$) $\geq$ O(n^2/log$n$)

So Far…

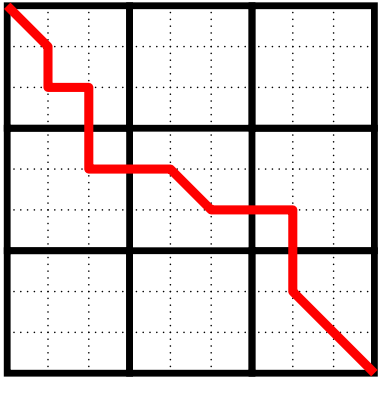
- We can divide up the grid into blocks and run dynamic programming only on the corners of these blocks
- In order to speed up the mini-alignment calculations to under n^2, we create a lookup table of size n, which consists of all scores for all t-nucleotide pairs
- Running time goes from quadratic,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

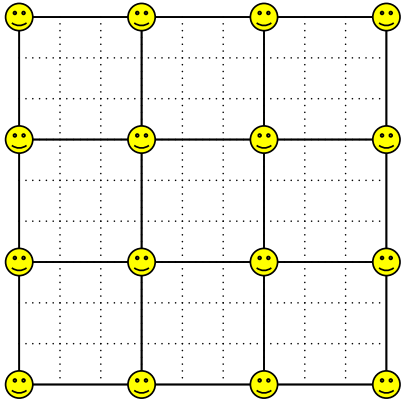


longest common subsequence

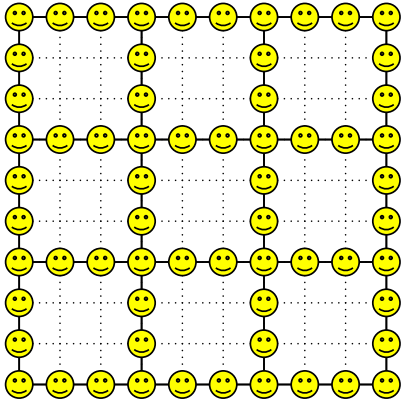
Block Alignment vs. LCS

- 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



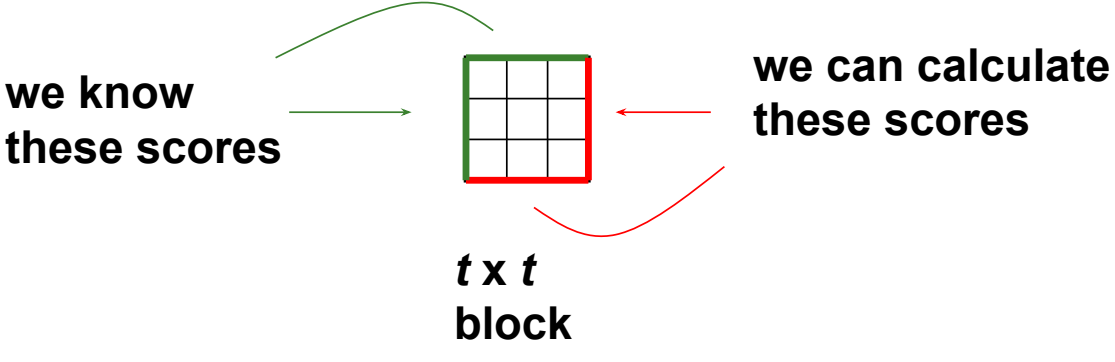
LCS alignment has $O(n^2/t)$ points of interest

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

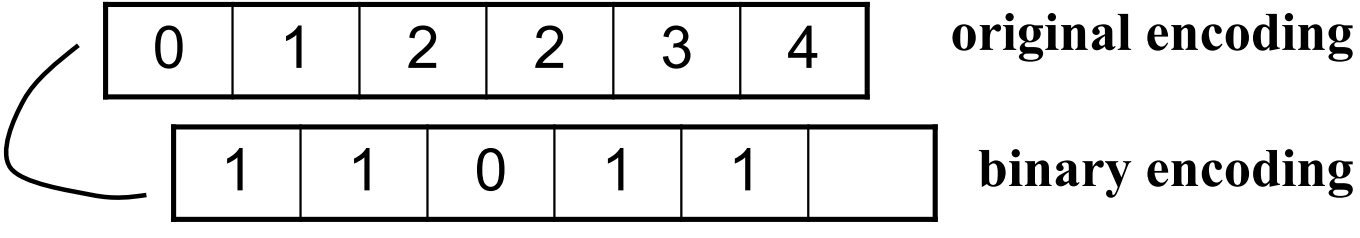
- Build a lookup table for all possible values of the four variables:
  1. all possible scores for the first row $s_{i,*}$
  2. all possible scores for the first column $s_{*,j}$
  3. substring of sequence *u* in this block (4^t possibilities)
  4. substring of sequence *v* in this block (4^t possibilities)
- 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 numbers 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 $x_{l\ldots l'}$, $y_{r\ldots r'}$

Definition:

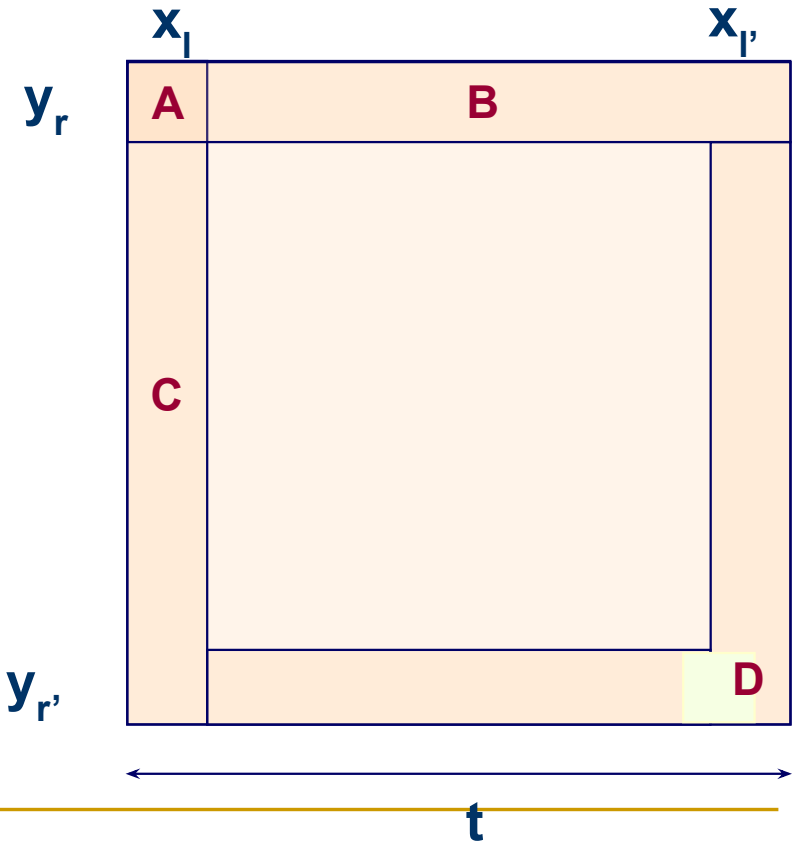
A t-block is a t × t square of the DP matrix

Idea:

Divide matrix in t-blocks,

Precompute t-blocks

Speedup: O(t)



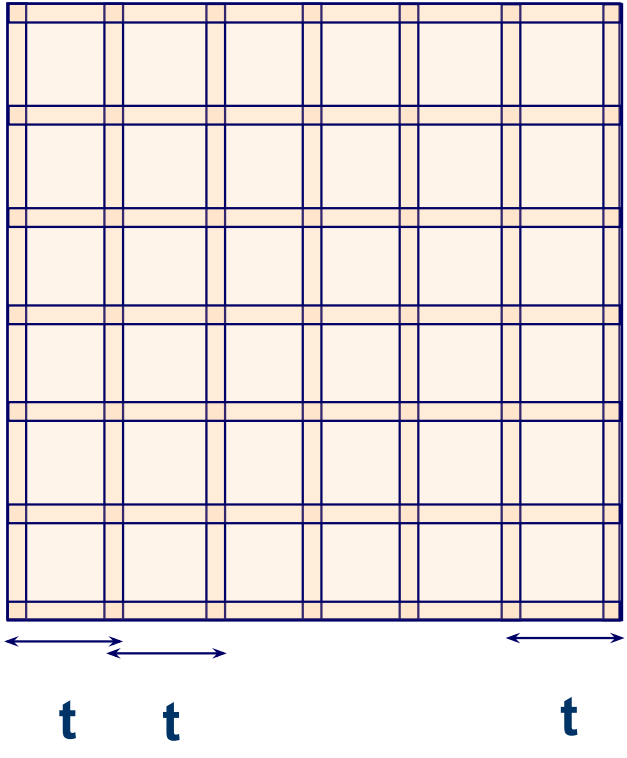
The Four-Russians Algorithm

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



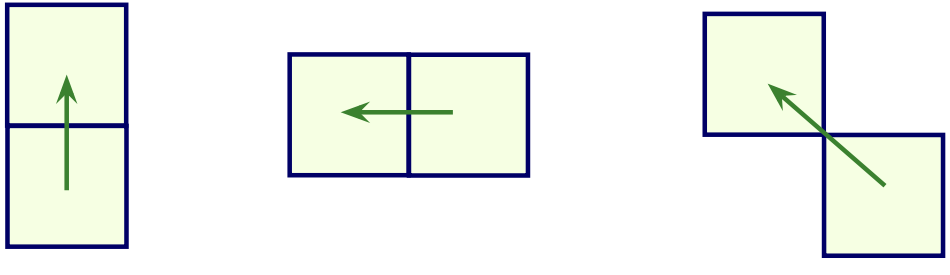
Precomputation

- By definition every cell has a value in [0, …, n]
- There are $(n+1)^t$ possible values for any *t*-length row or column
- If $\sigma = |\sum|$, then there are σ^t possible substrings of length *t*
- Number of distinct computations is $(n+1)^{2t}\ \sigma^{2t}$
- t^2 computations required to evaluate a *t*-block
- Overall: $\Theta((n+1)^{2t}\ \sigma^{2t}t^2) = \Omega(n^2)$

The Four-Russians Algorithm

Another observation:
(Assume match = 0, substitute = 1, delete = 1)

Lemma. Two adjacent cells of F(.,.) differ by at most 1



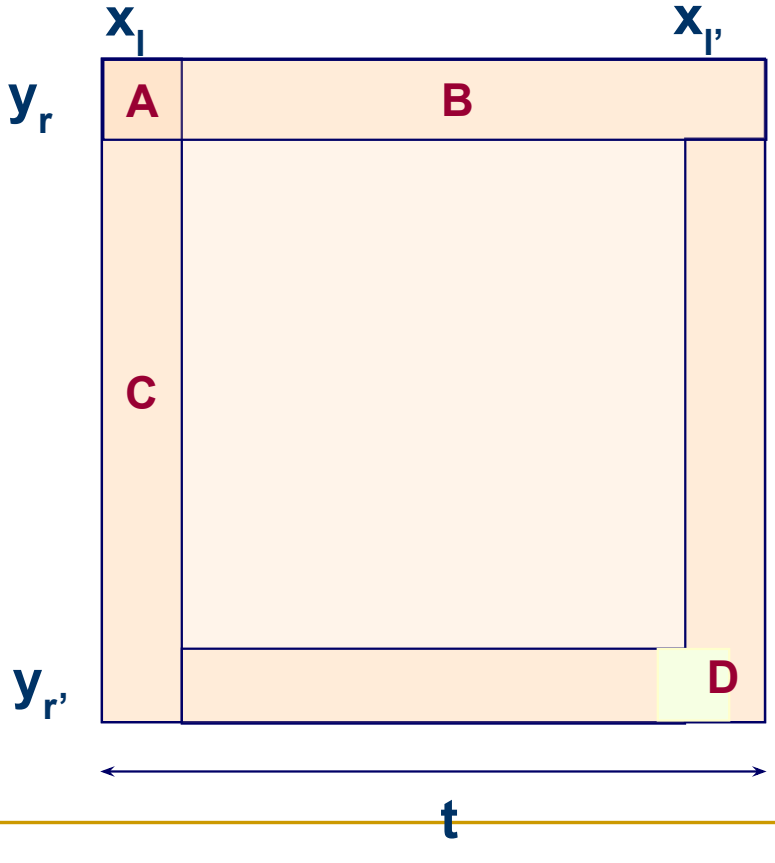
The Four-Russians Algorithm

Definition:

The offset vector is a t-long vector of values from {-1, 0, 1}, where the first entry is 0

If we know the value at A, and the top row, left column offset vectors, and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

Then we can find D



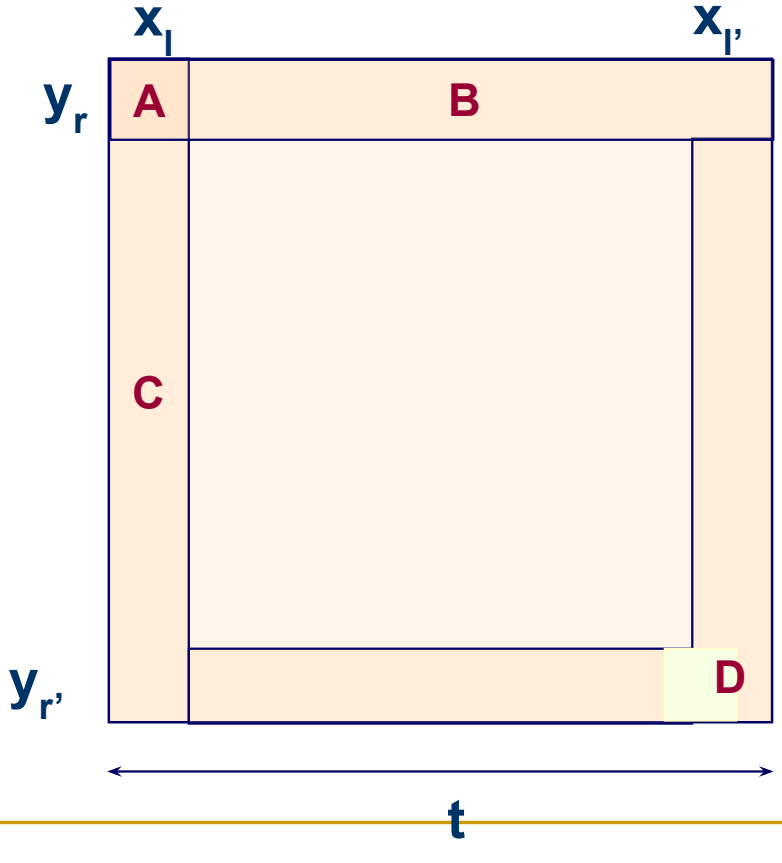
The Four-Russians Algorithm

Definition:

The offset function of a t-block is a function that for any given offset vectors of top row, left column,

and $x_l \ldots\ldots x_{l'}$, $y_r \ldots\ldots y_{r'}$,

produces offset vectors of bottom row, right column



An Example

	----	C	T	T	C	G	A	T	G	A
----	0	0	0	0	0	0	0	0	0	0
T	0	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
T	0	*0*	1	2	2	*2*	2	2	2	*2*
A	0	*0*	1	2	2	*2*	3	3	3	*3*
C	0	*1*	1	2	3	*3*	3	3	3	*3*
G	0	*1*	*1*	*2*	*3*	*4*	*4*	*4*	*4*	*4*
T	0	*1*	2	2	3	*4*	4	5	5	*5*
G	0	*1*	2	2	3	*4*	4	5	6	*6*
C	0	*1*	2	2	3	*4*	4	5	6	*6*
A	0	*1*	*2*	*2*	*3*	*4*	*5*	*5*	*6*	*7*

An Example

	----	C	T	T	C	G	A	T	G	A

T		*0/0*	*1*	*0*	*0*	*1/0*	*0*	*0*	*0*	*1/0*
T		*0*				*1*				*1*
A		*0*				*0*				*1*
C		*1*				*1*				*0*
G		*0/1*	*0*	*1*	*1*	*1/1*	*0*	*0*	*0*	*1/0*
T		*0*				*0*				*1*
G		*0*				*0*				*1*
C		*0*				*0*				*0*
A		*0/1*	*1*	*0*	*1*	*0/1*	*1*	*0*	*1*	*1/1*

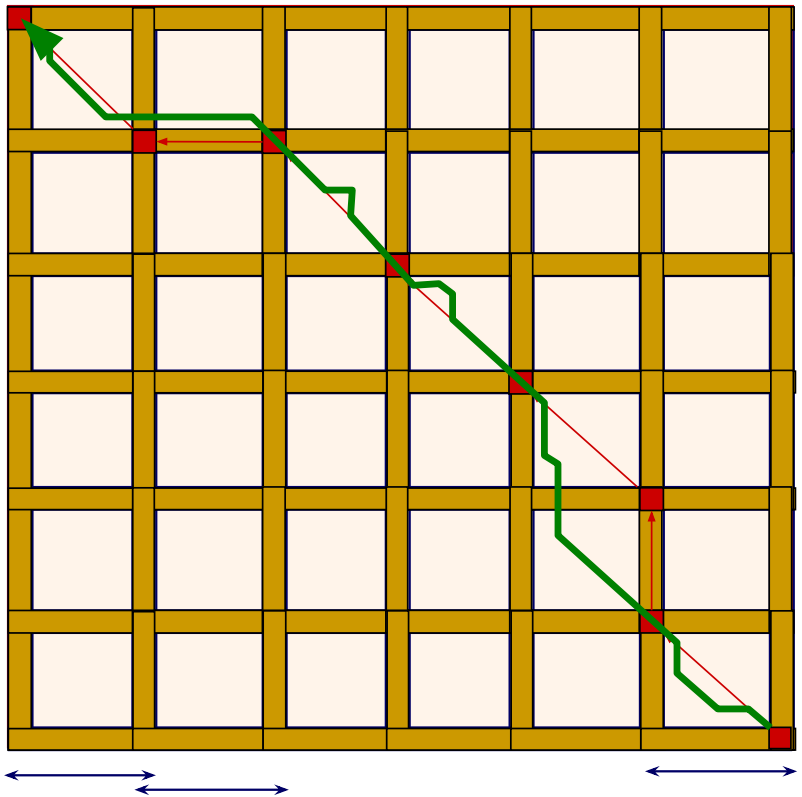
The Four-Russians Algorithm

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



An Example: score maximization

	----	C	T	T	C	G	A	T	G	A
----	0	0	0	0	0	0	0	0	0	0
T	0									
T	0									
A	0									
C	0									
G	0									
T	0									
G	0									
C	0									
A	0									

Match: +1 **Mismatch: 0** **Gap: 0** **t=5**

Precompute

	C	T	T	C	G
T	*0*	*1*	*1*	*1*	*1*
T	*0*	1	2	2	*2*
A	*0*	1	2	2	*2*
C	*1*	1	2	3	*3*
G	*1*	*1*	*2*	*3*	*4*

Align

	C	T	T	C	G
T	*0/0*	*1*	*0*	*0*	*1/0*
T	*0*				*1*
A	*0*				*0*
C	*1*				*1*
G	*0/1*	*0*	*1*	*1*	*1/1*

Encode

Precompute

	G	A	T	G	A
T	*0*	*0*	*1*	*1*	*1*
T	*0*	0	1	1	*1*
A	*0*	1	1	1	*2*
C	*0*	1	1	1	*2*
G	*1*	*1*	*1*	*2*	*2*

Align

	G	A	T	G	A
T	*0/0*	*0*	*1*	*0*	*0/1*
T	*0*				*1*
A	*0*				*0*
C	*0*				*1*
G	*1/1*	*0*	*0*	*1*	*1/1*

Encode

Will not use this one, but will be precomputed

Precompute

	G	A	T	G	A
T	*1/0*	1	1	1	*1*
T	*1*	1	2	2	*2*
A	*0*	2	2	2	*3*
C	*1*	2	2	2	*3*
G	*1/1*	1	2	3	*3*

Align

	G	A	T	G	A
T	*1/0*	*0*	*0*	*0*	*0*
T	*1*				*1*
A	*0*				*1*
C	*1*				*0*
G	*1/1*	*0*	*0*	*0*	*1/0*

Encode

Will use this one

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