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)

Sorting Problem Revisited

- Given: an unsorted array

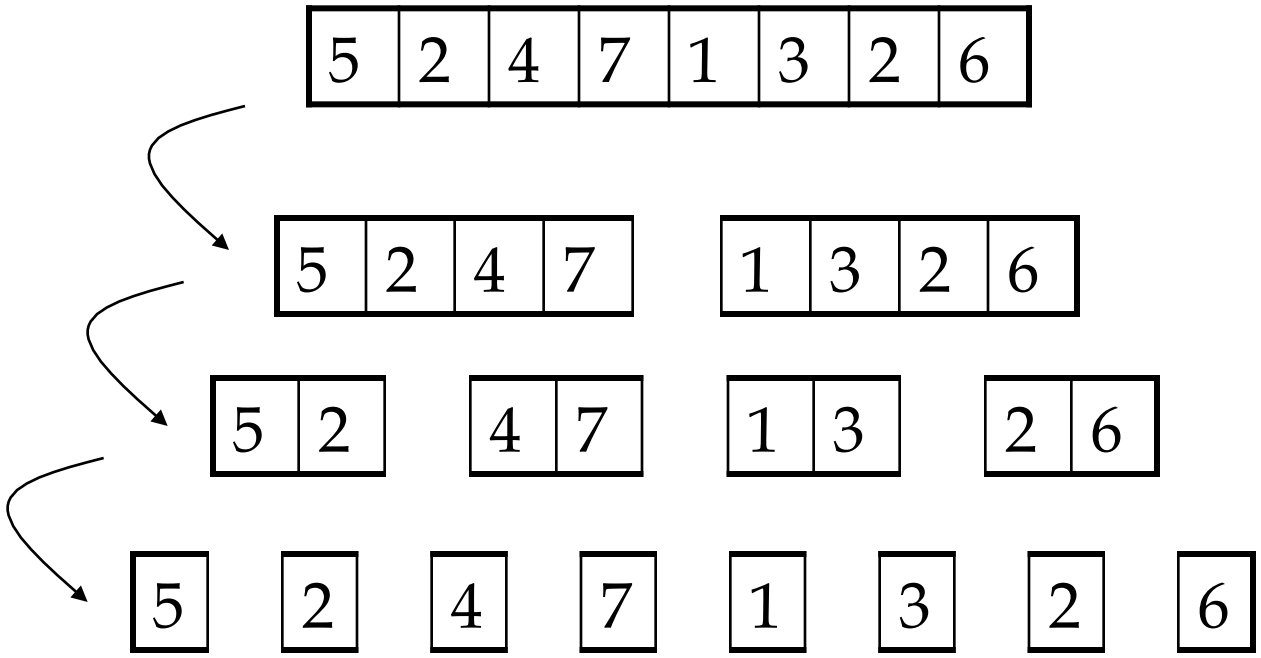
5	2	4	7	1	3	2	6

- Goal: sort it

1	2	2	3	4	5	6	7

Mergesort: Divide Step

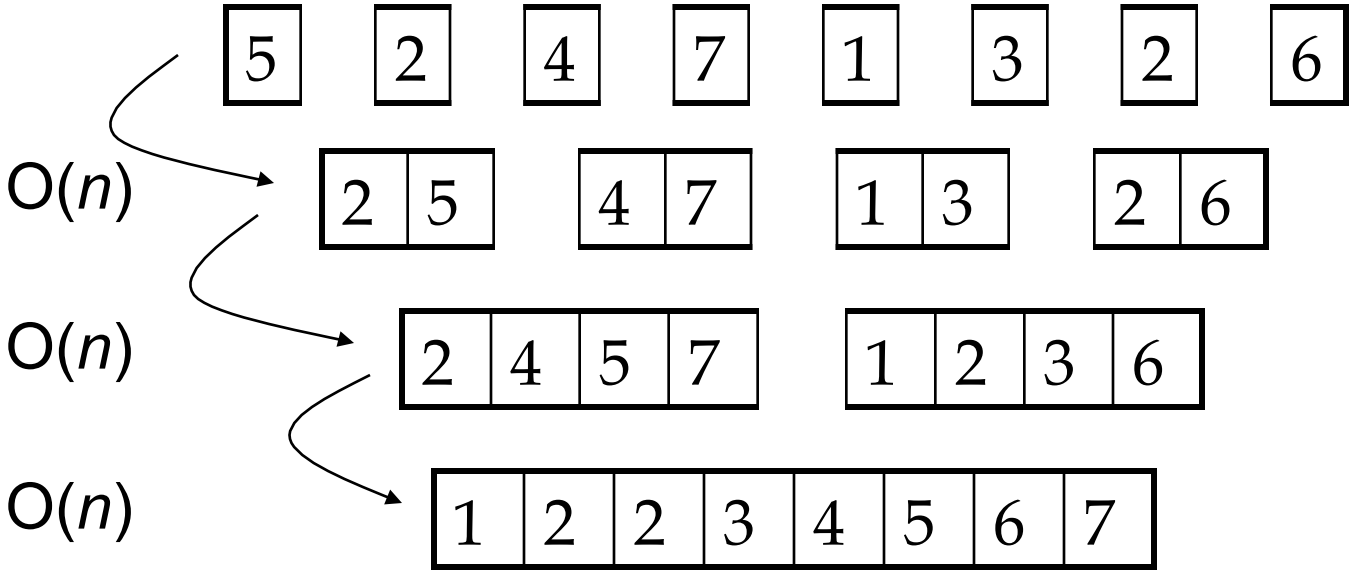
Step 1 – **Divide**



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: Merge

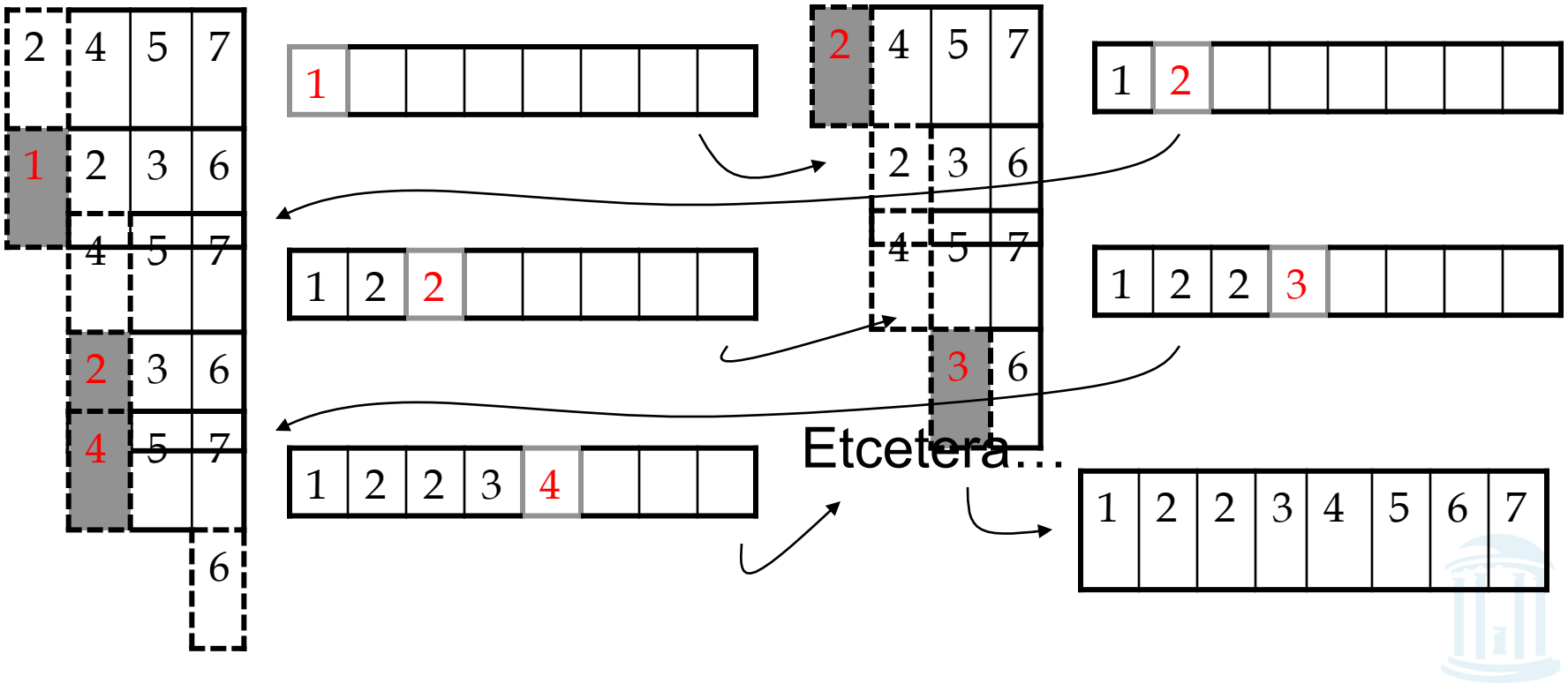
Merge

- 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: Merge

Merge 2 arrays of size 4



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 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

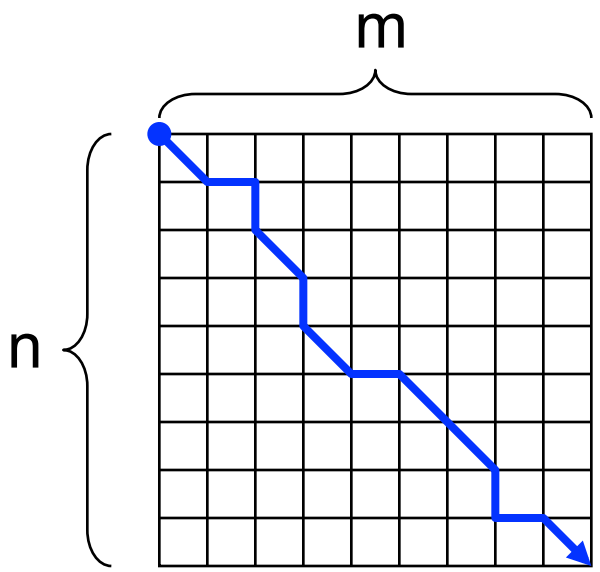
- The problem is simplified to baby steps
  - for the *i*'th merging iteration, the complexity of the problem is *O(n)*
  - number of iterations is *O(log n)*
  - running time: O(*n* log*n*)

Now for a biological problem

Alignments Require Quadratic Memory

Alignment Path

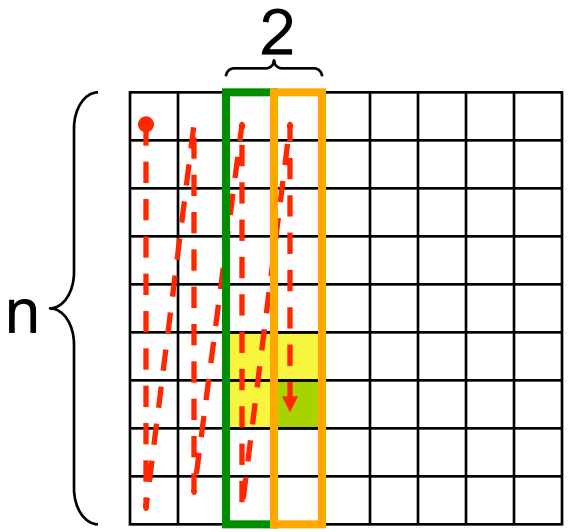
- Space complexity for computing alignment path for sequences of length n and m is $O(nm)$
- We keep a table of all scores and backtracking references in memory to reconstruct the path (backtracking)



Computing Alignment Score with Linear Memory

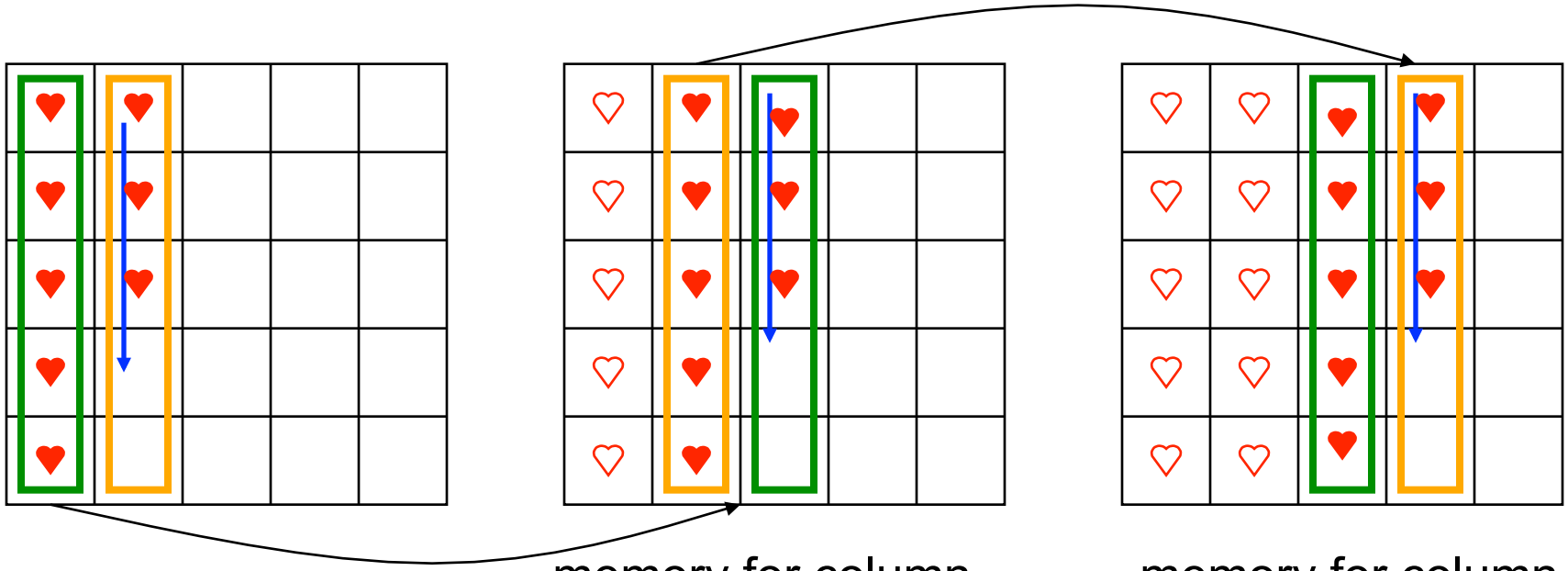
Alignment Score

- 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

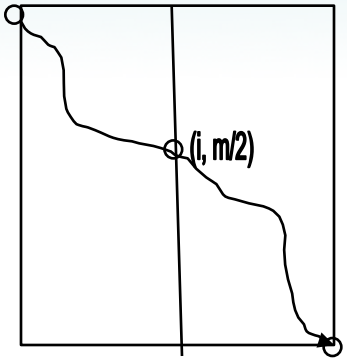


memory for column 1 is used to calculate column 3

memory for column 2 is used to calculate column 4

D&C Sequence Alignment

Find the best scoring path aligning two sequences

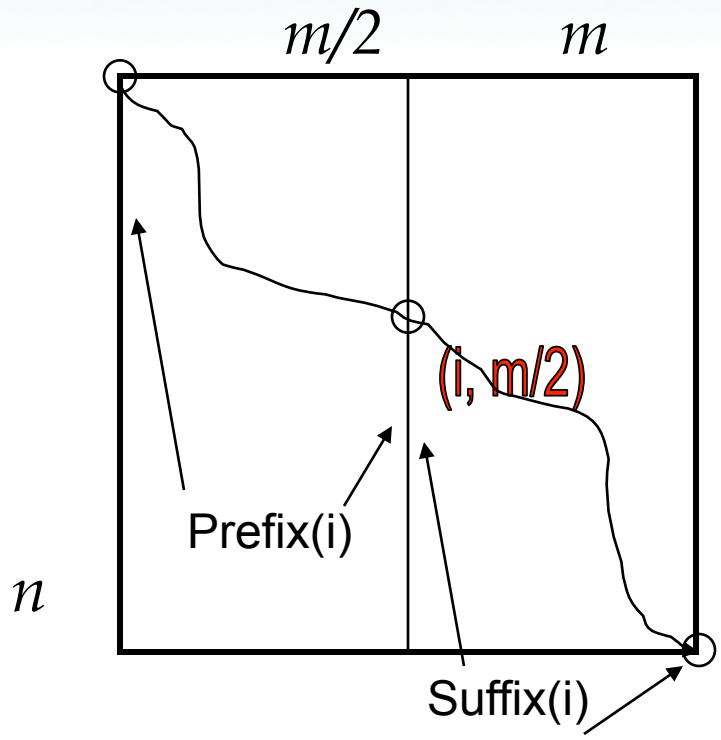


Path(*source, sink*)

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

The only problem left is how to find this "middle vertex"!

Computing the Alignment Path



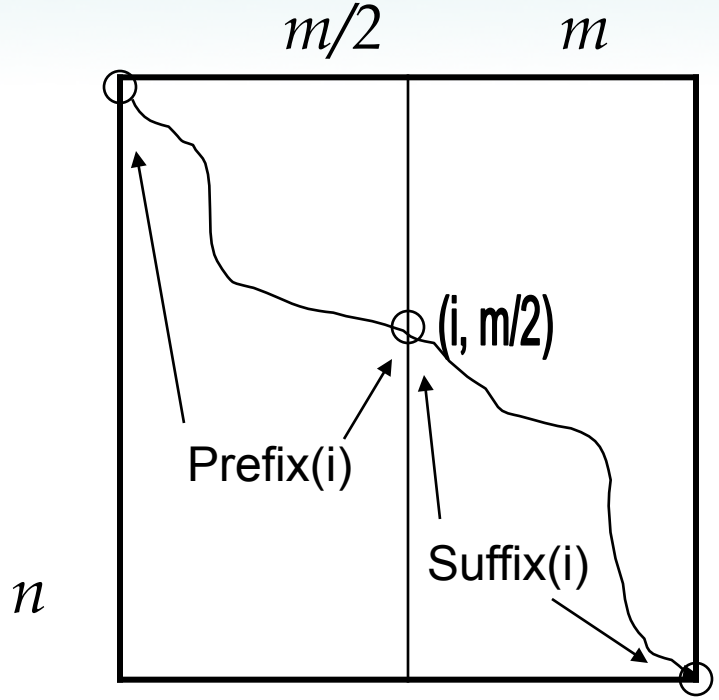
We want to calculate the longest path from (0,0) to (*n*,*m*) that passes through (*i*,*m*/2) where *i* ranges from 0 to *n* and represents the *i*-th row

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 Midline

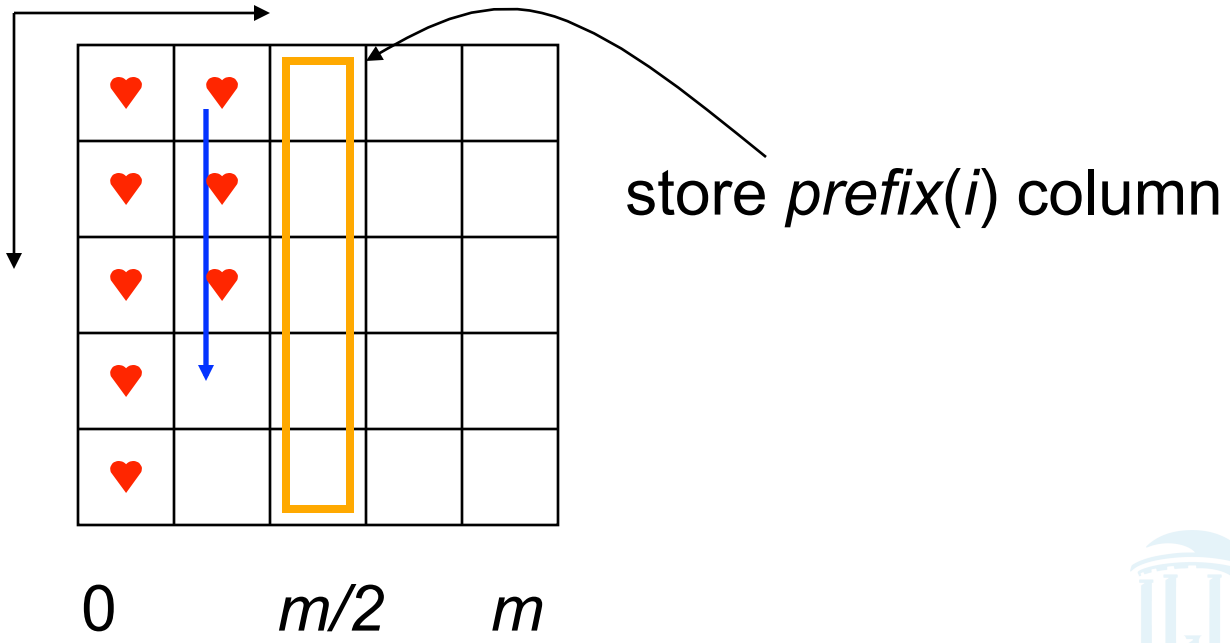


Define (*mid*,*m*/2) as the vertex where the longest path crosses the middle column.

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

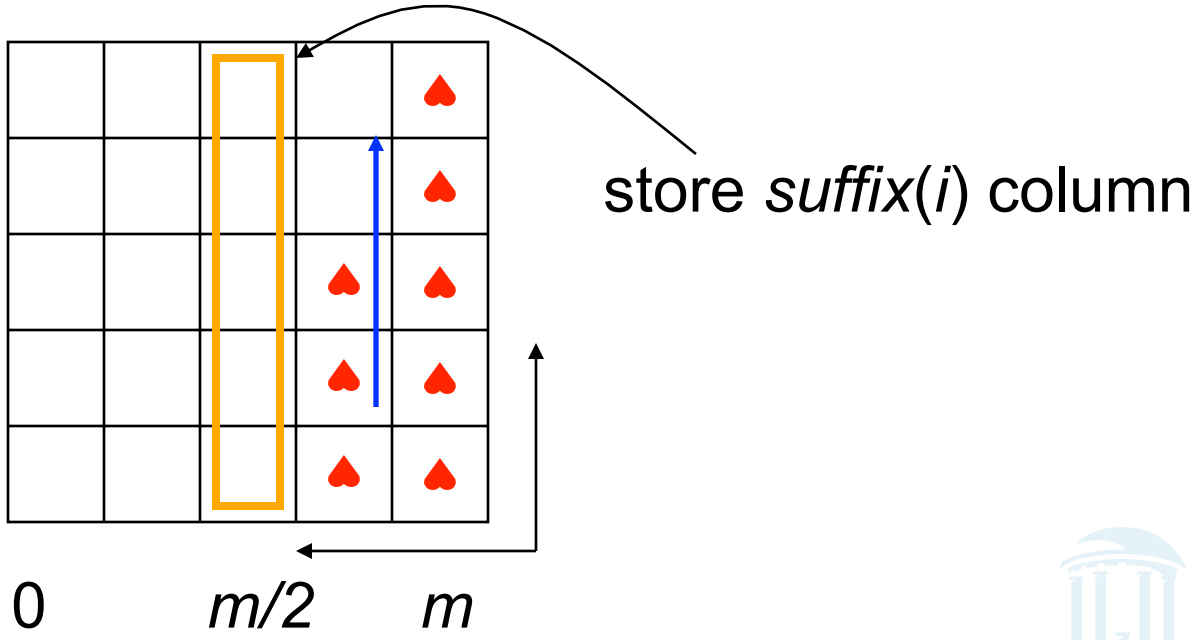
Computing Prefix(i)

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



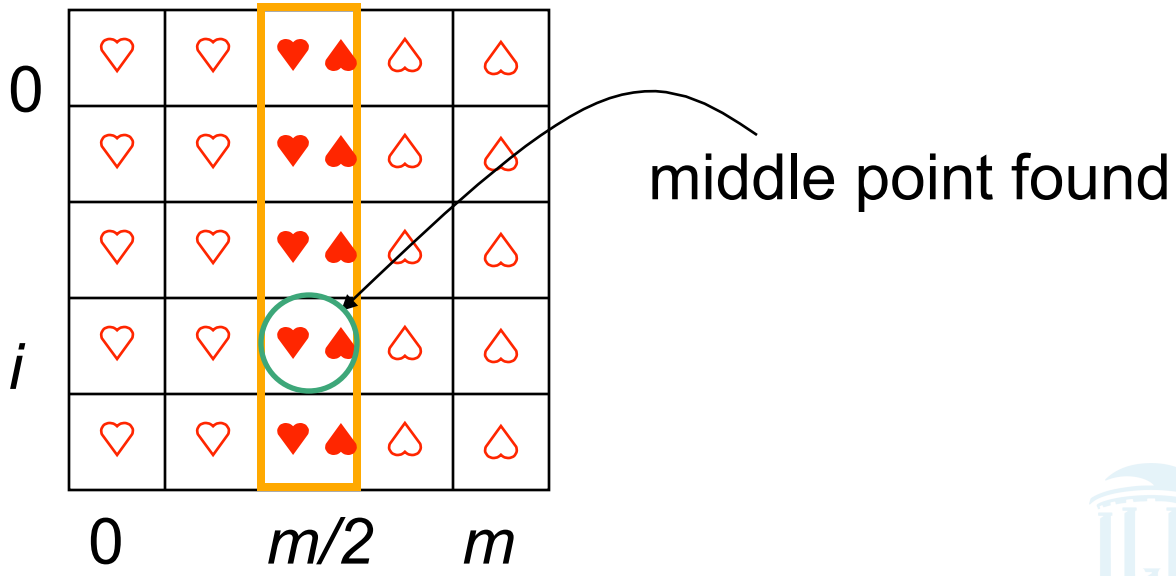
Computing Suffix(*i*)

- *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*) in the right half of the "reversed" matrix

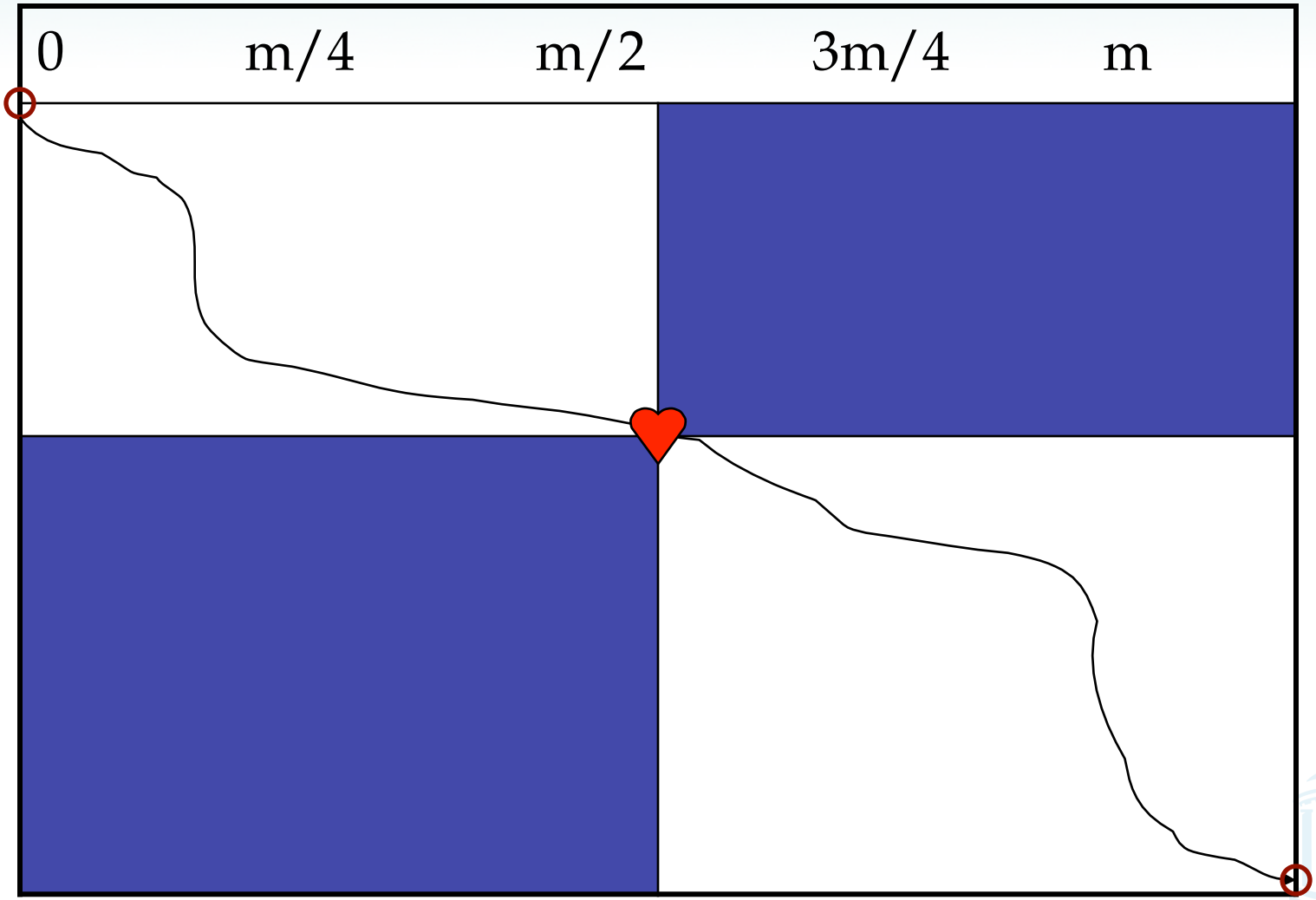


Length(i) = Prefix(i) + Suffix(i)

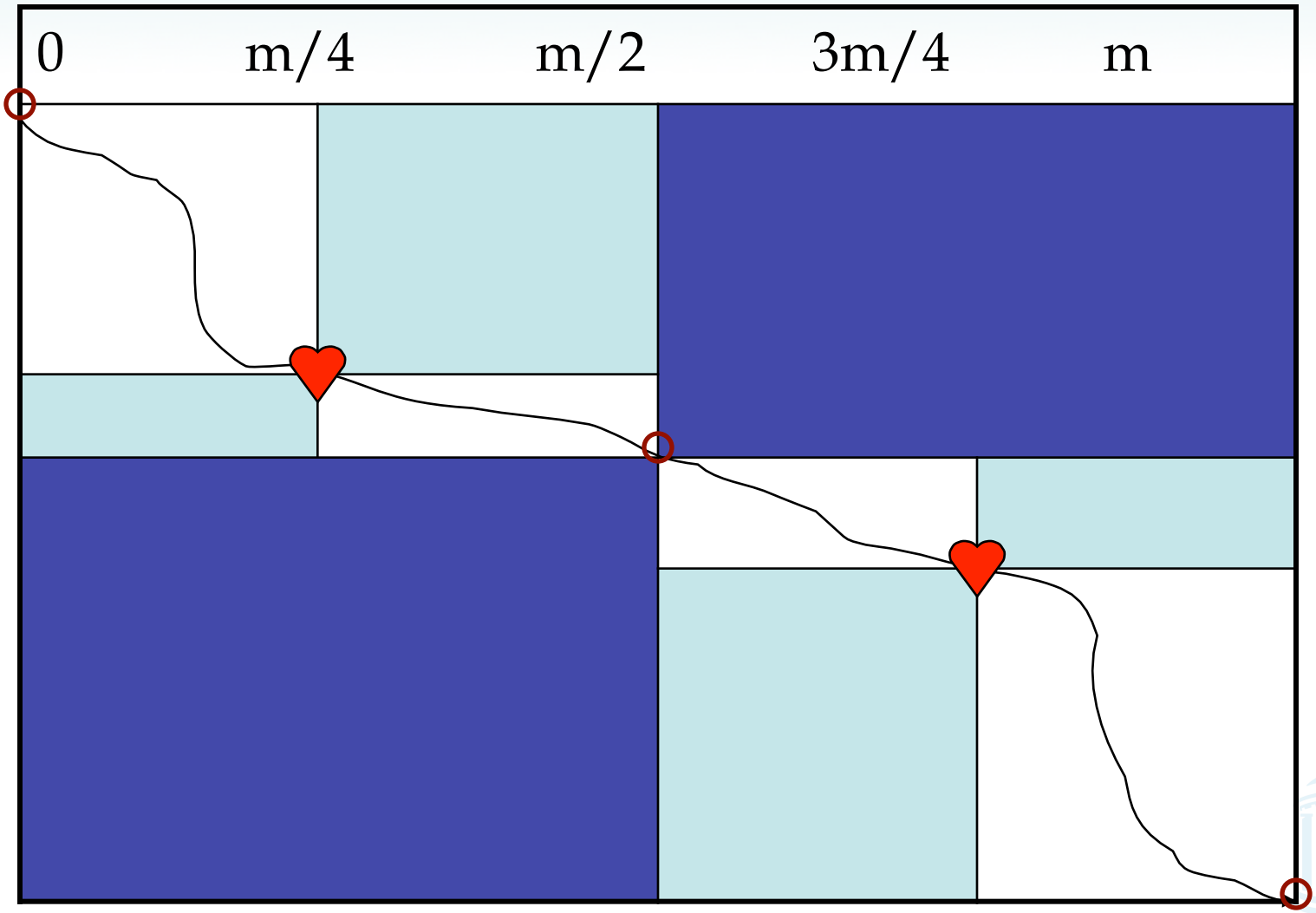
- Add *prefix*(*i*) and *suffix*(*i*) to compute *length(i):*
  - *length*(*i*)=*prefix*(*i*) + *suffix*(*i*)
- You now have a middle vertex of the maximum path (*i*,*m*/2) as maximum of *length(i)*



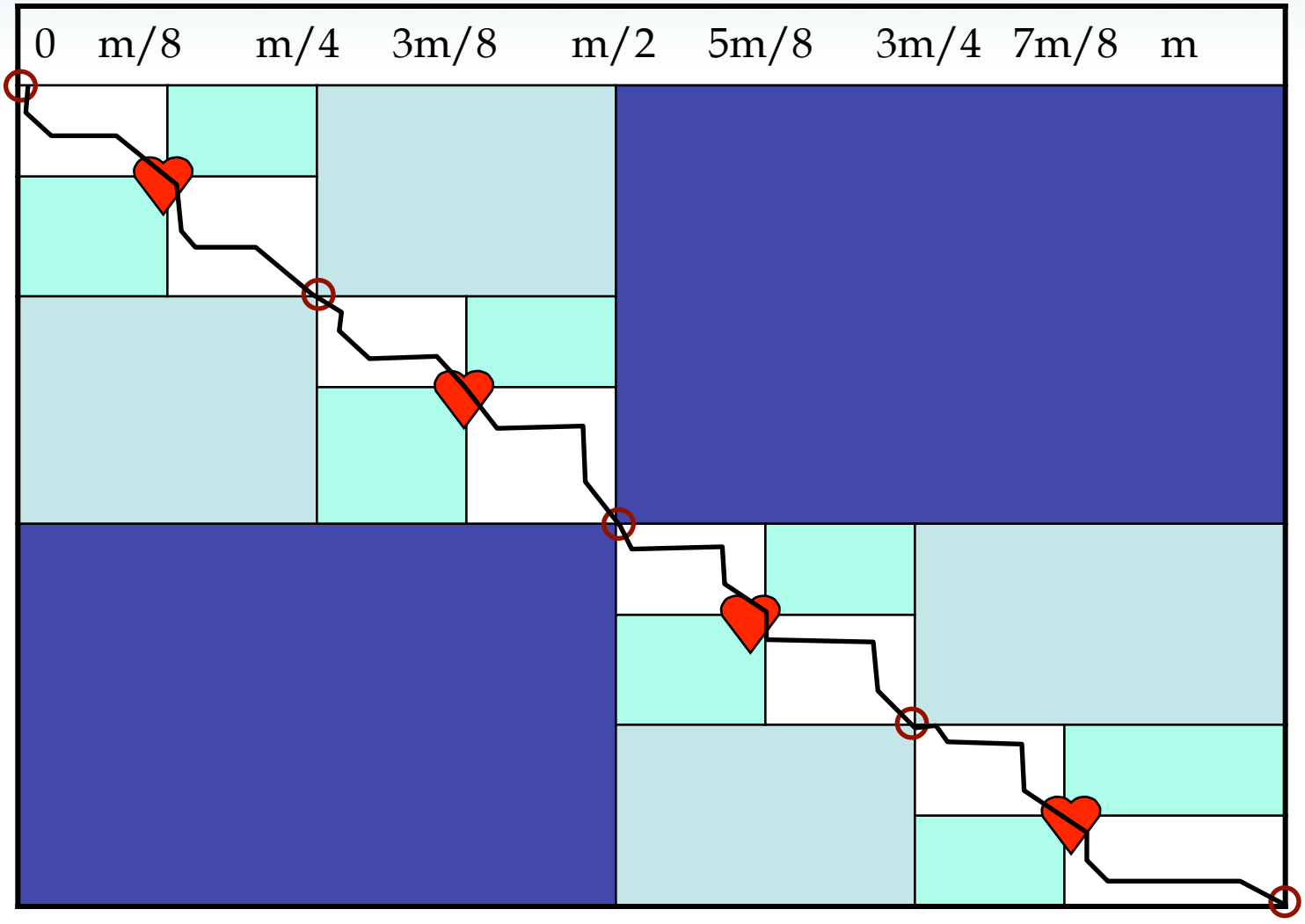
Finding the Middle Point



Finding the Middle Point again



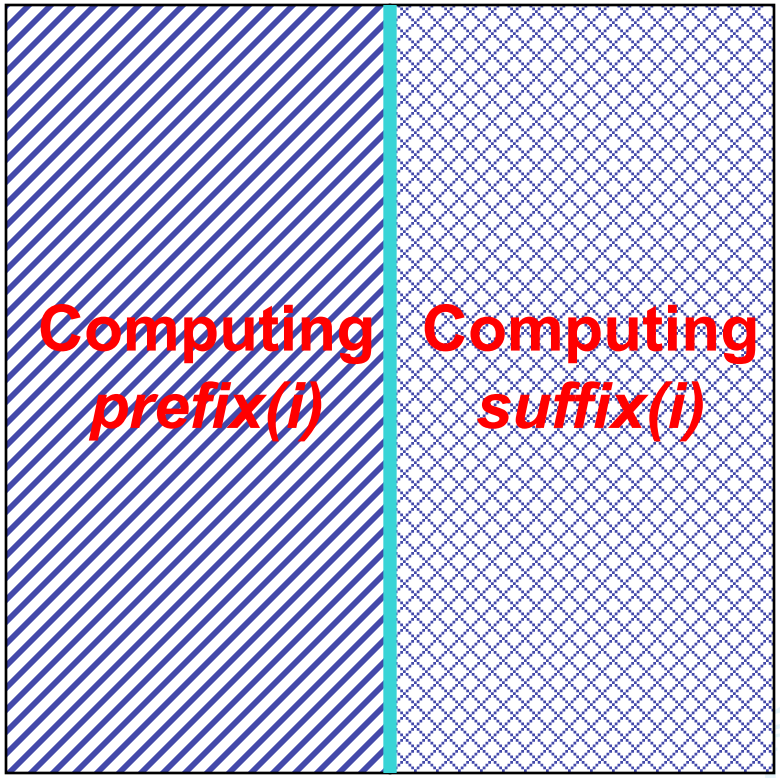
And Again



Time = Area: First Pass

- On first level, the algorithm touches the entire area

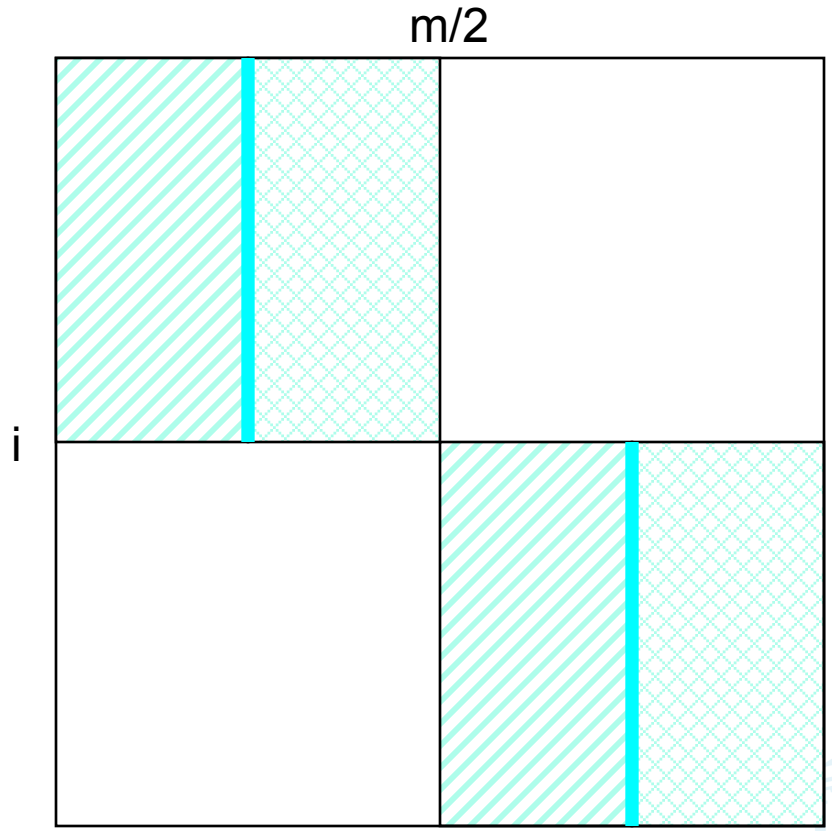
Area = $n*m$



Time = Area: Second Pass

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

Area/2

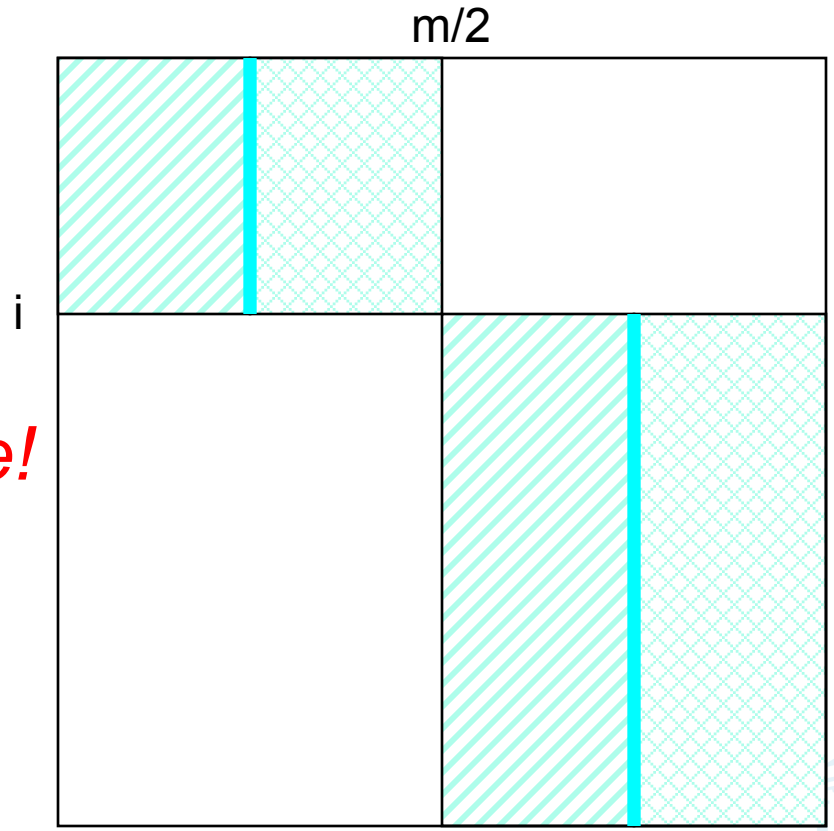


Time = Area: Second Pass

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

Area/2

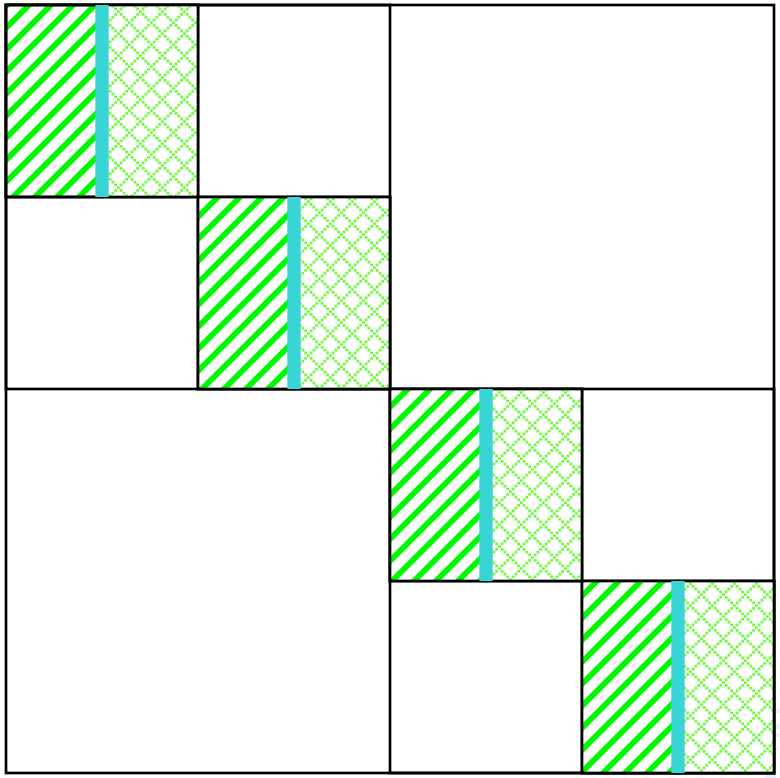
Regardless of i's value!



Time = Area: Third Pass

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

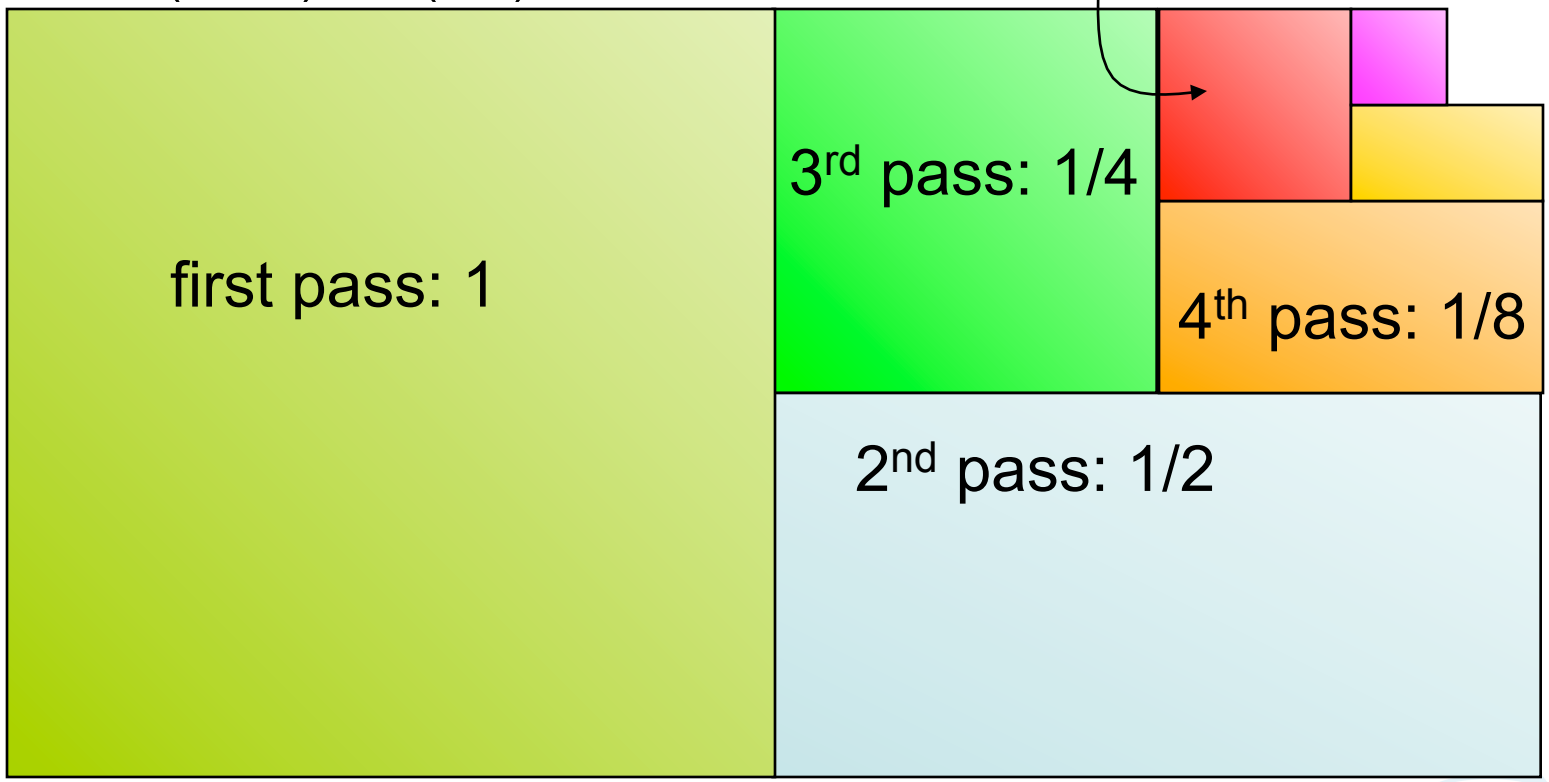
Area/4



Geometric Reduction At Each Iteration

$1 + ½ + ¼ + \ldots + (½)^k \leq 2$

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



- 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 ≈ m
- Yes, using the *Four-Russians Speedup*

Partitioning Sequences into Blocks

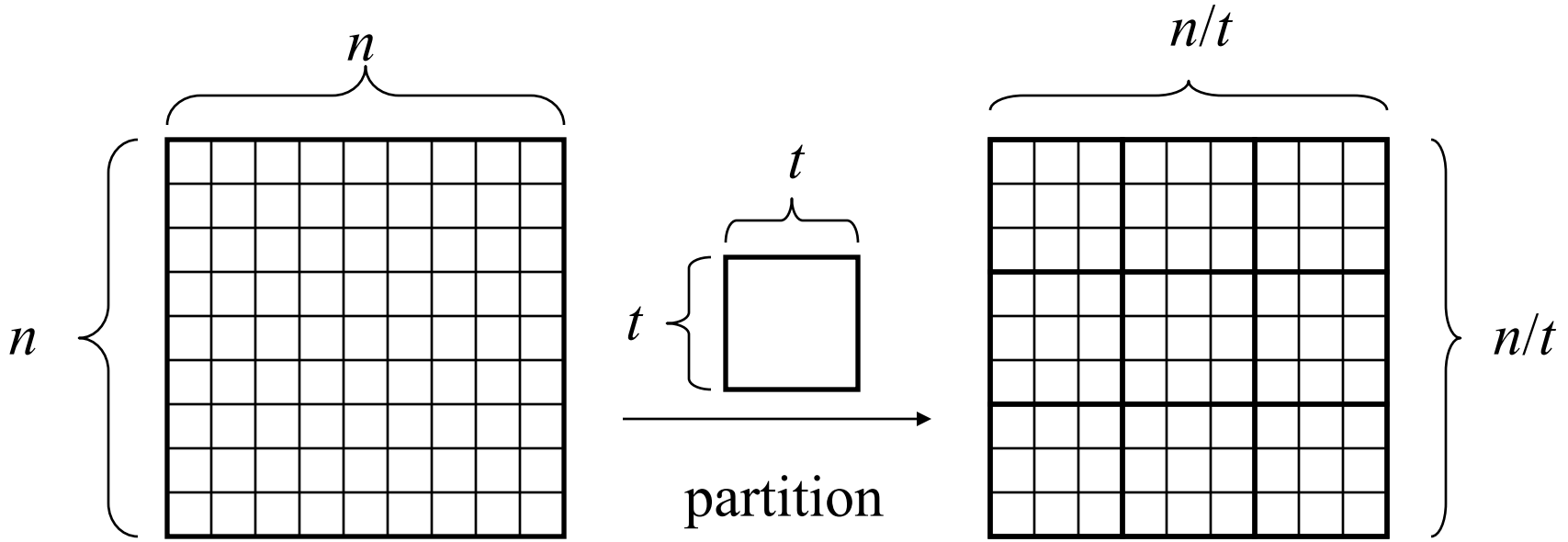
- Partition the n x n grid into blocks of size t x t
- We are comparing two sequences, each of size n, and each sequence is sectioned off into chunks, each of length t
- Sequence $\boldsymbol{u} = u_1 \ldots u_n$ becomes

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

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

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

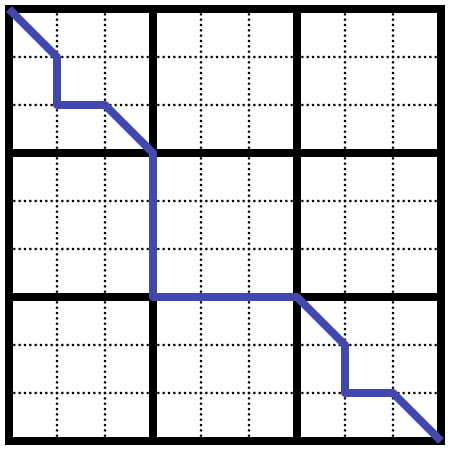
Partitioning Alignment Grid into Blocks



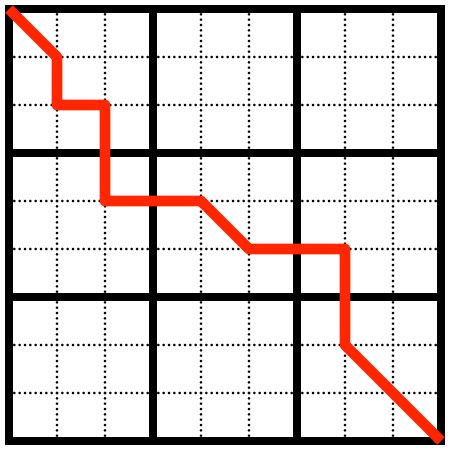
Block Alignment

- **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

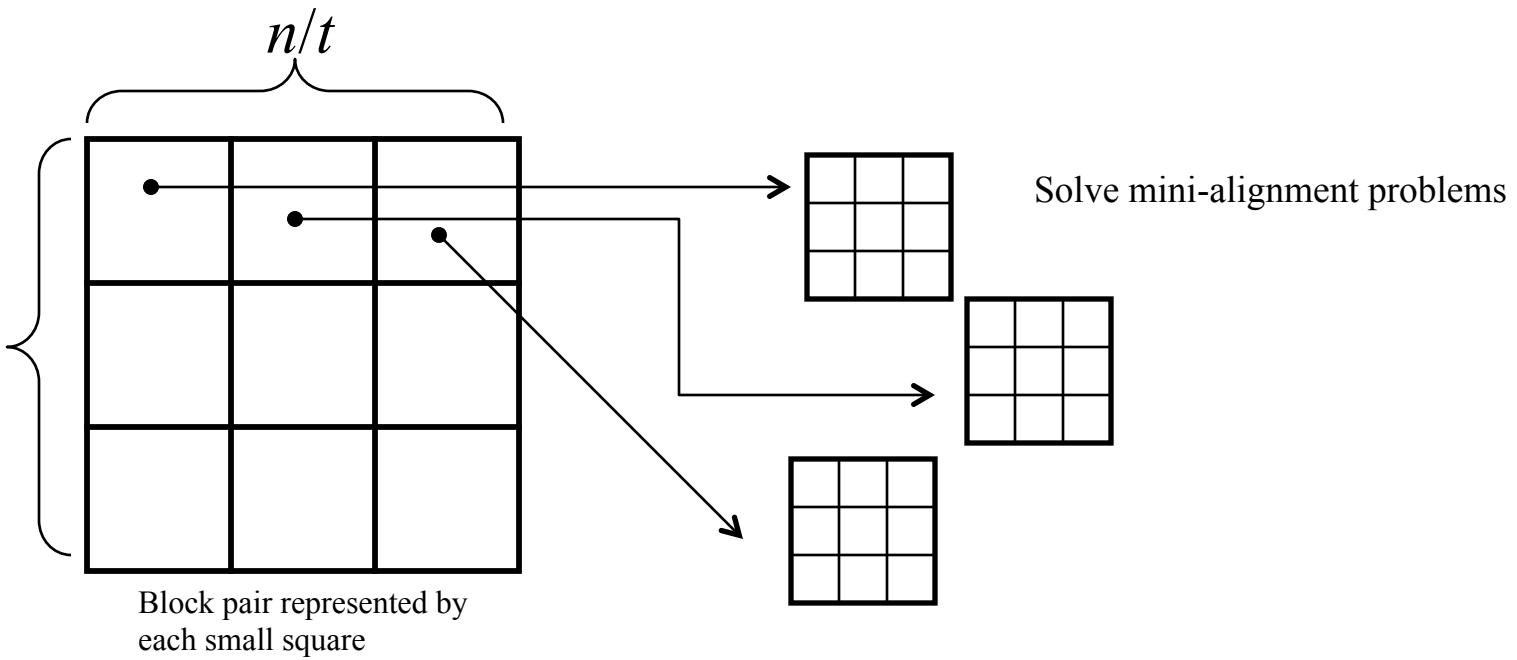
Constructing Alignments within Blocks

- 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, which requires $t \; x \; t = O(t^2)$ effort
- Looks like a wash $O((n/t)^2 \; t^2) = O(n^2)$, but is it?

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

$$O([n/t]*[n/t]*O(\beta_{i,j})) = O(n^2/t^2)$$

- 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

$$O((n^2/t^2)\ t^2) = O(n^2)$$

- 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…
  - Lets assume $n = m = 4000$ and $t = 4$
  - n/t = 1000, so there are 1,000,000 blocks
  - How many possible many 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 $(n/t)*(n/t)$ 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 $4^t \times 4^t = \sqrt[4]{n^2} \times \sqrt[4]{n^2} = n$

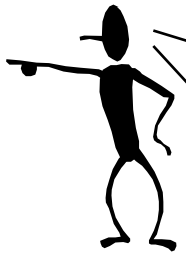
Look-up Table for Four Russians Technique

each sequence has *t* nucleotides

	AAAAAA	AAAAAC	AAAAAG	AAAAAT	AAAACA	⋮
AAAAAA	6	4	4	4	4	
AAAAAC	4	6	4	4	3	
AAAAAG	4	4	6	4	3	
AAAAAT	4	4	4	6	3	
AAAACA	4	3	3	3	6	
…						

Lookup table "*Score*"

size is *n*, which is much smaller than $(n/t)*(n/t)$ → 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

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

Four Russians Speedup Runtime

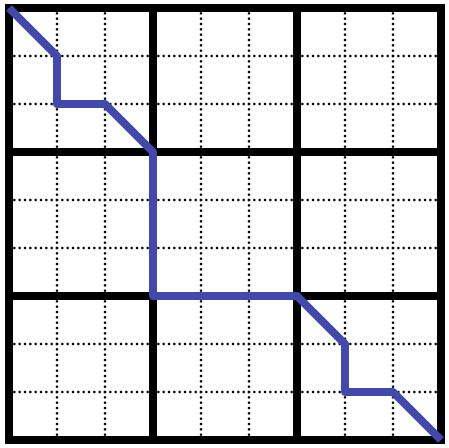
- Since computing the lookup table *Score* of size n takes O(n) time, the running time is dominated by the $(n/t)*(n/t)$ accesses to the lookup table
- Overall running time: O($[n^2/t^2]$)
- Since $t = (\log_2 n)/4$, substitute in:
- O($[n^2/\{\log_2 n\}^2]$) = O($n^2/\log(n \log n)$)

So Far…

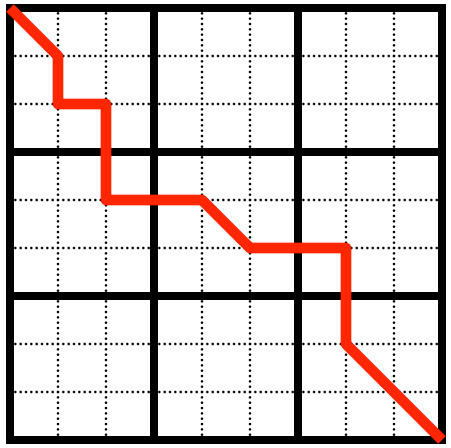
- We can divide up the grid into blocks and run dynamic programming only on the corners of these blocks
- In order to speed up the mini-alignment calculations to under n^2, we create a lookup table of size n, which consists of all scores for all t-nucleotide pairs
- Running time goes from quadratic,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

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

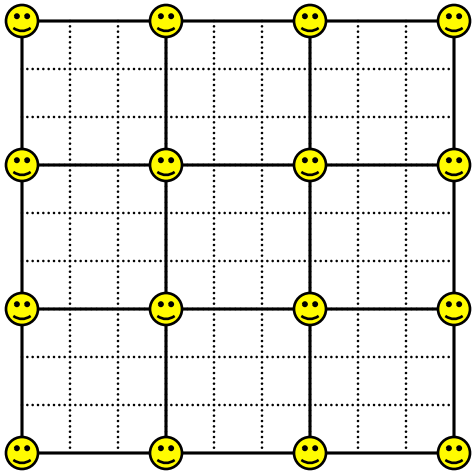


longest common subsequence

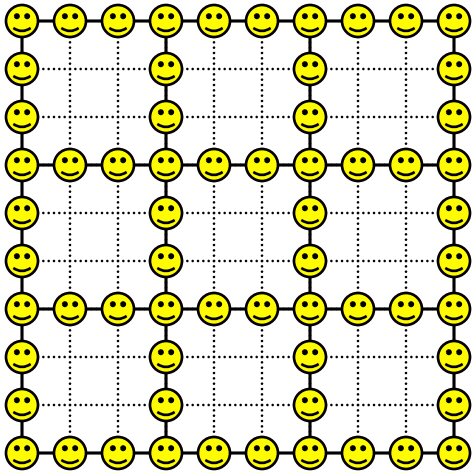
Block Alignment vs. LCS

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



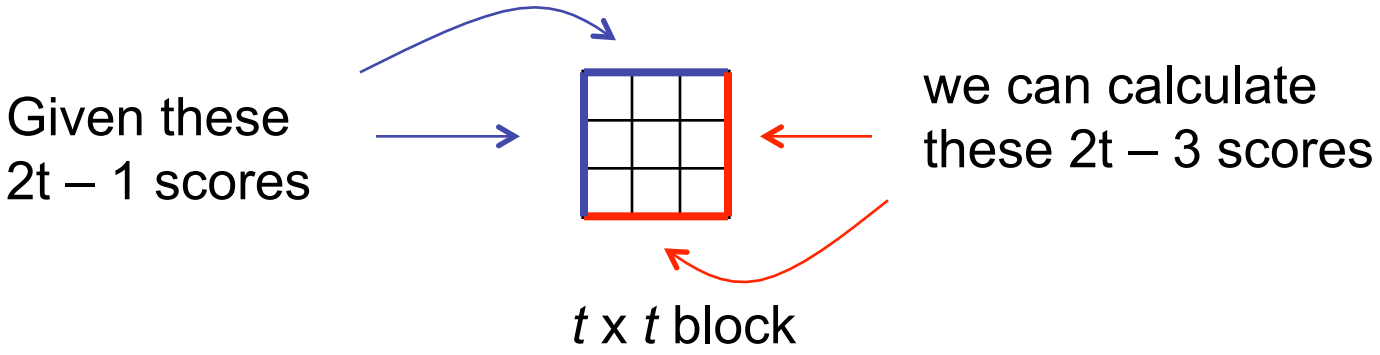
LCS alignment has $O(n^2/t)$ points of interest

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

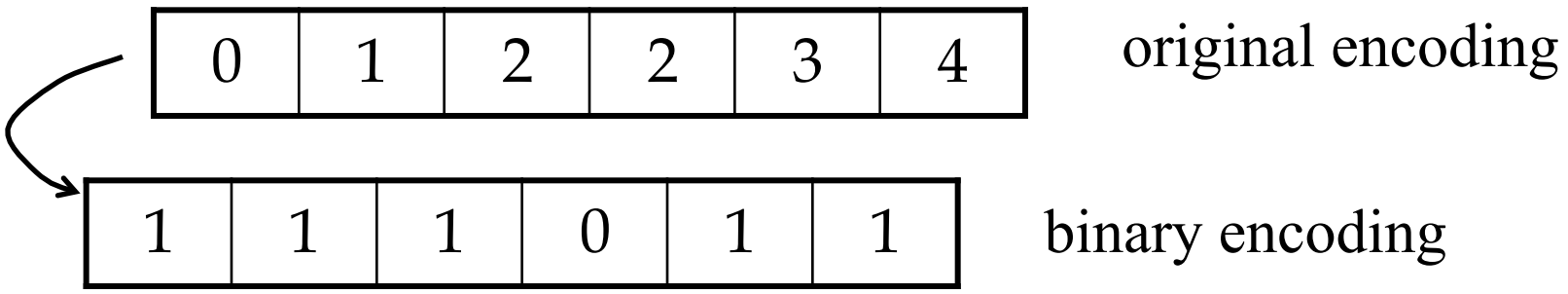
- Build a lookup table for all possible values of the four variables:
  1. all possible scores for the first row $s_{*,j}$
  2. all possible scores for the first column $s_{*,j}$
  3. substring of sequence u in this block (4^t possibilities)
  4. substring of sequence v in this block (4^t possibilities)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

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: $O(n^2/\log(n \log n))$
- Next Time: Graph Algorithms