Divide and Conquer

Bioinformatics Programming - 2016

Computer Engineering, Chiang Mai University

Divide-and-Conquer

- A strategy to solve a large problem that can be built from the solutions of smaller problem instance
- Two phases:
  - Divide – splits a problem instance into smaller problem instances and solve them
  - Conquer – combines the solutions to the smaller problems into a solution to the bigger one
- Often used to improve the efficiency of a polynomial algorithm (quadratic time)
  - From $O(n^2)$ to $O(n \log n)$ or $O(n^2/\log n)$

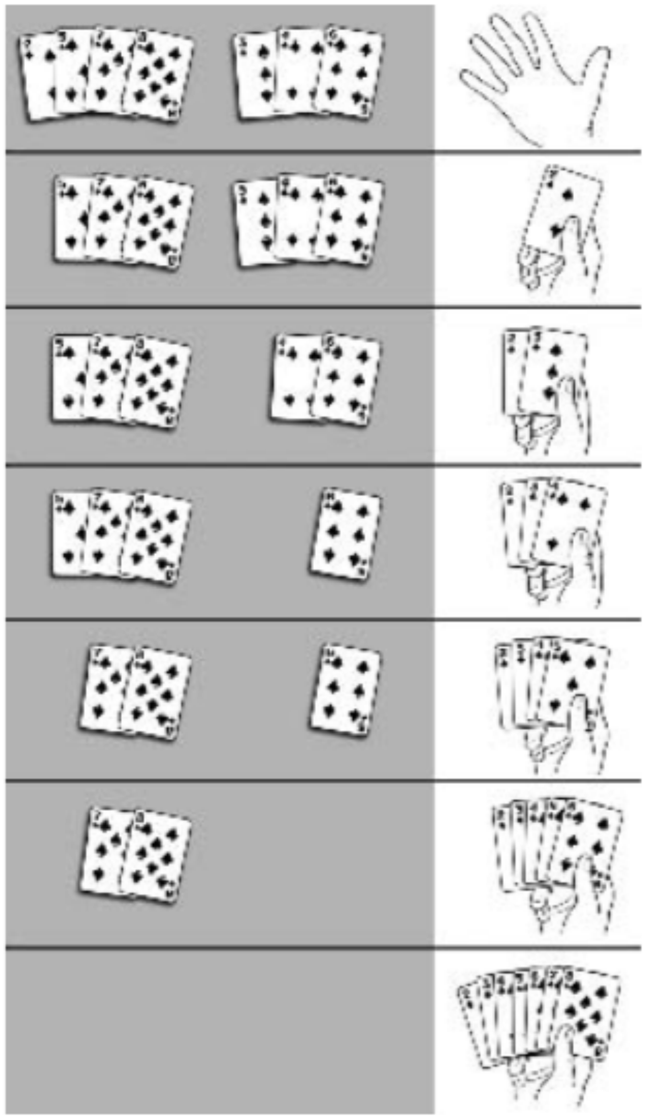
Approach to Sorting

- We have introduced an algorithm to sorting problem before, Selection sort, that required $O(n^2)$
  - Now we want a faster approach
- MERGE algorithm
  - If we have two lists of sorted data of length: n_1, n_2
  - How could we combined them into a single list – Merge
    - Traverse each list simultaneously
    - Pick the smaller element and put it in the output list
  - Time: $O(n_1 + n_2)$

Approach to Sorting

- MERGE Algorithm

```
MERGE(a, b)
1   $n1 \leftarrow$ size of a
2   $n2 \leftarrow$ size of b
3   $a_{n1+1} \leftarrow \infty$
4   $b_{n2+1} \leftarrow \infty$
5   $i \leftarrow 1$
6   $j \leftarrow 1$
7   for $k \leftarrow 1$ to $n1 + n2$
8       if $a_i < b_j$
9           $c_k \leftarrow a_i$
10          $i \leftarrow i + 1$
11      else
12          $c_k \leftarrow b_j$
13          $j \leftarrow j + 1$
14  return c
```

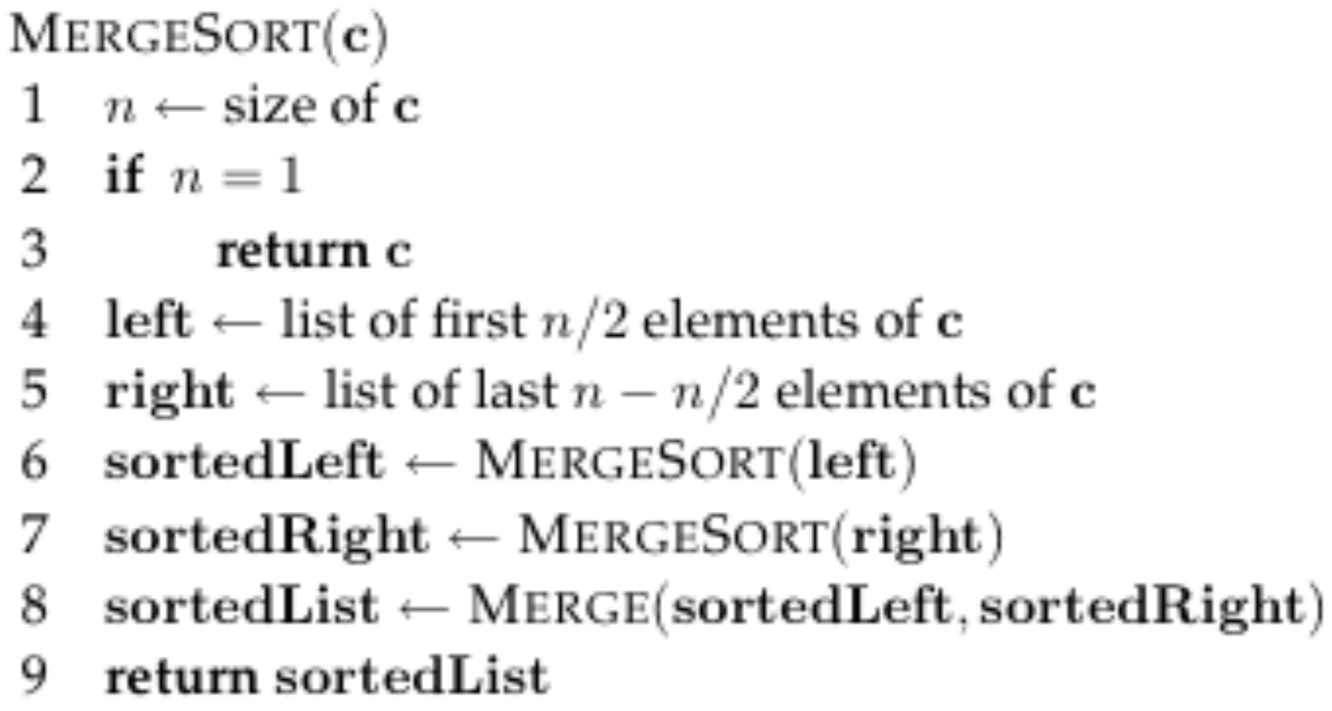


Approach to Sorting

- MERGE Algorithm is easily applied to a list of TWO elements
  - Divide: Break the list into two list of ONE element
  - Both are sorted list (of one element)
  - Conquer: Merge them into a single list
- If we have a list of unsorted FOUR elements
  - Divide: Break it into two list of TWO elements
  - Sort each TWO element list
  - Conquer: Merge the result of sorted lists
- The same idea applies to an arbitrary list - MERGESORT

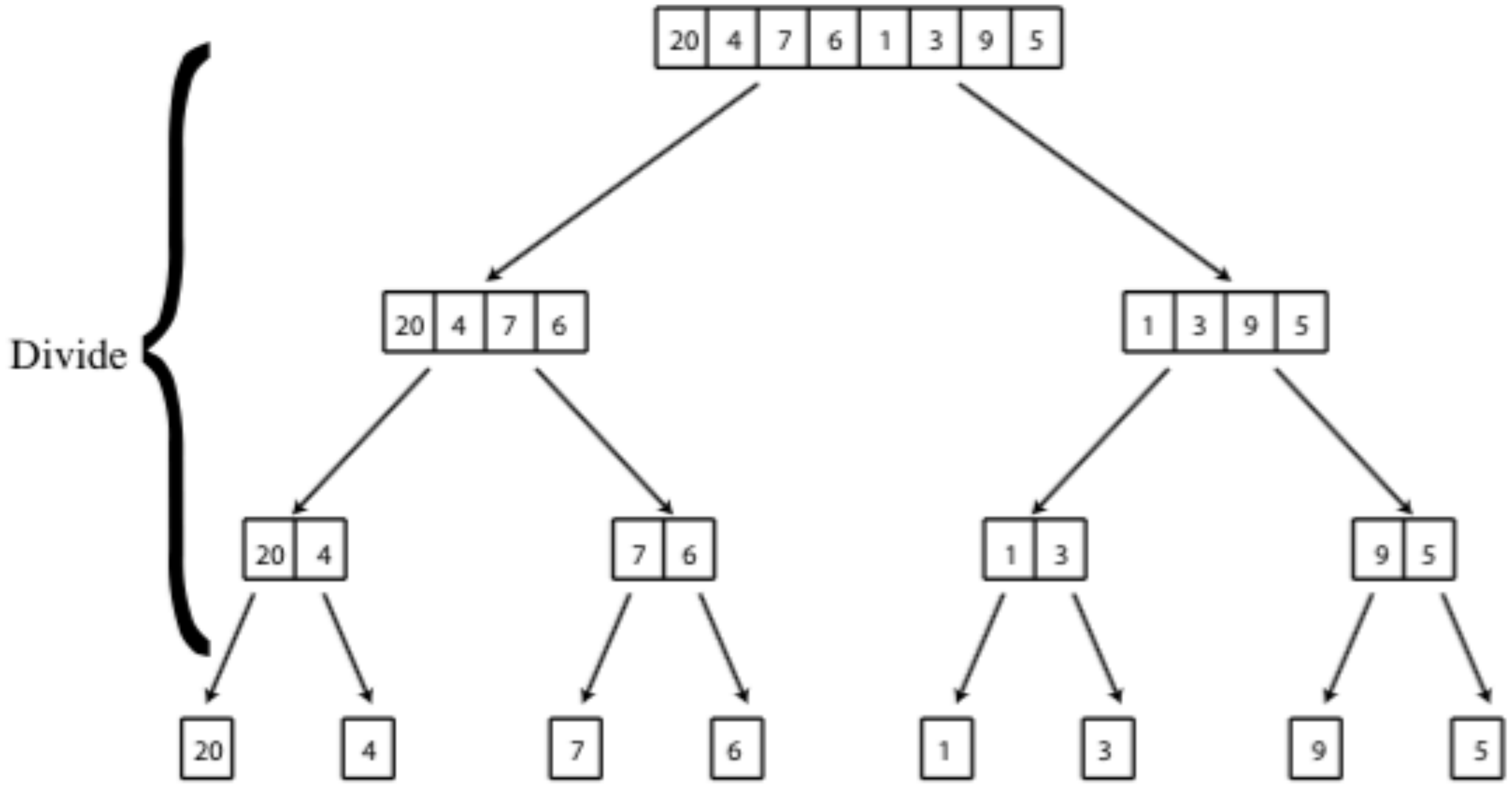
Approach to Sorting

- MERGESORT Algorithm

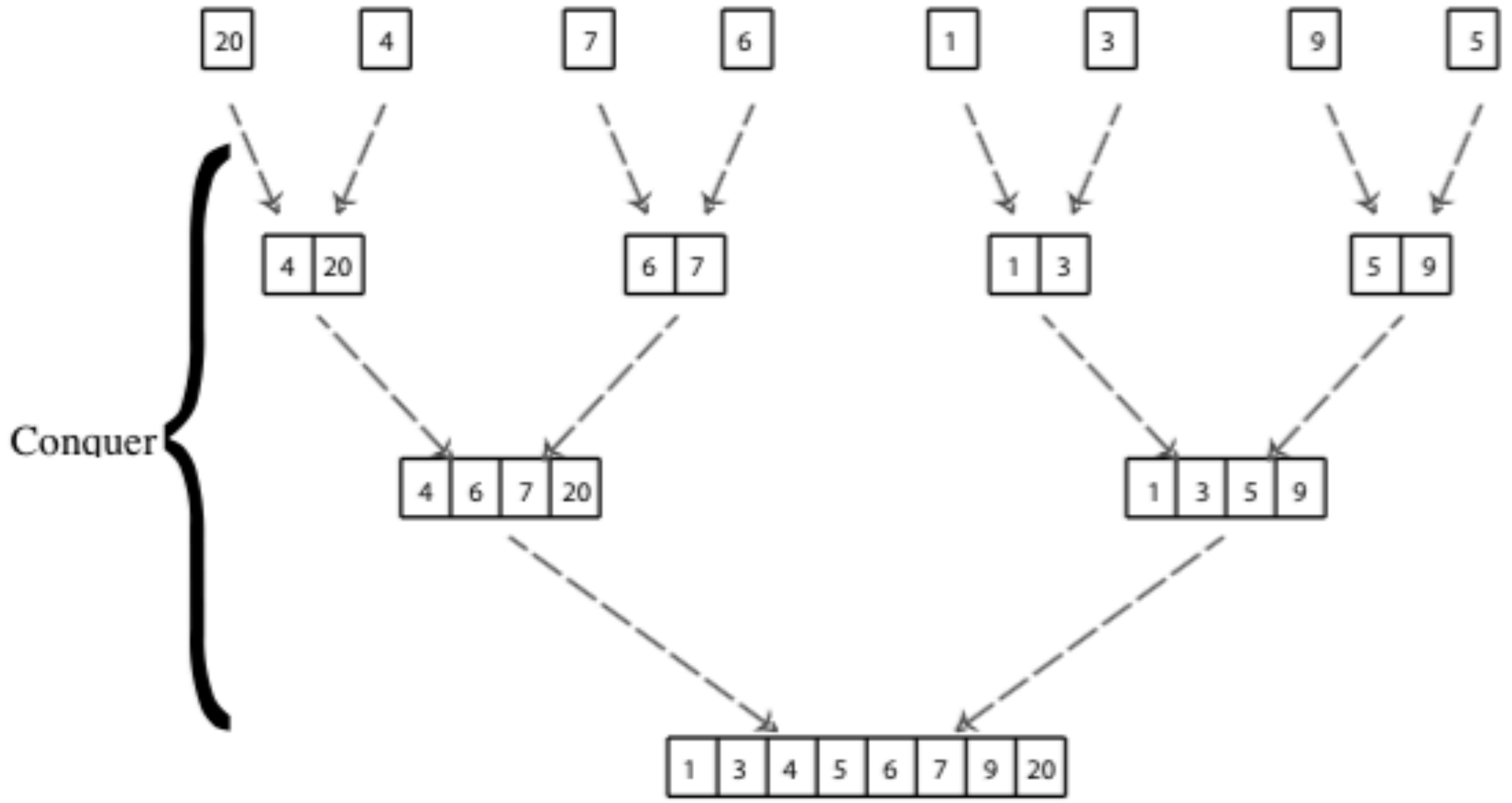


```
MERGESORT(c)
$n \leftarrow$ size of c
if $n = 1$
    return c
left $\leftarrow$ list of first $n/2$ elements of c
right $\leftarrow$ list of last $n - n/2$ elements of c
sortedLeft $\leftarrow$ MERGESORT(left)
sortedRight $\leftarrow$ MERGESORT(right)
sortedList $\leftarrow$ MERGE(sortedLeft, sortedRight)
return sortedList
```

Approach to Sorting



Approach to Sorting



Approach to Sorting

- Time: $T(n)$
  - Two calls to MERGESORT on list of size $n/2$
  - One call to MERGE (on two lists) - $O(n/2+n/2) = O(n)$

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ T(1) &= 1 \end{aligned}$$

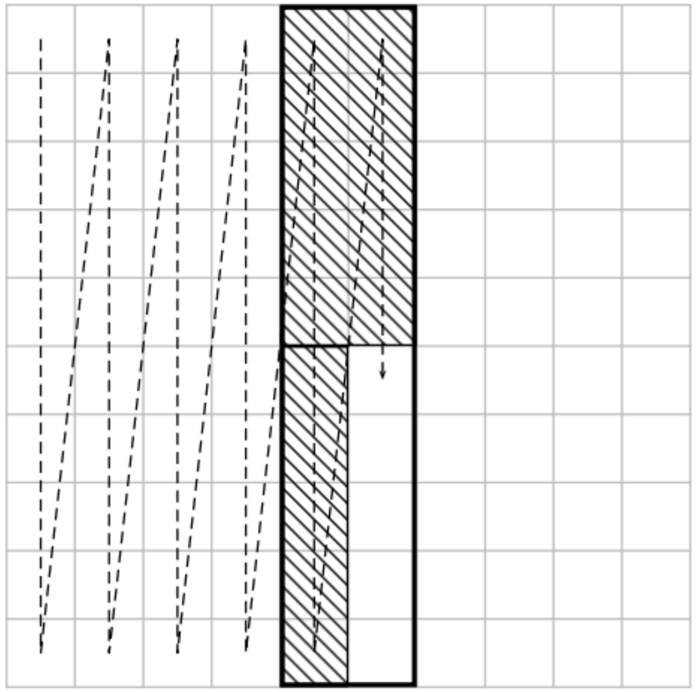
- Overall time complexity:

$$T(n) = O(n \log n)$$

Space-Efficient Sequence Alignment

- Solving sequence alignment problem with limited resource – NOT time but memory
- In 1975, Daniel Hirschberg proposed an algorithm that perform alignment in linear space
  - At a cost of doubling the computational time
- In dynamic programming for aligning sequence of length n and m (using edit graph):
  - Time complexity: proportional to edges – $O(nm)$
  - Space complexity: proportional to #vertices – $O(nm)$

Space-Efficient Sequence Alignment



- Computes only score of alignment:
  - Just the score, NOT the alignment (no backtracking info.) – $S_{i,j}$
  - The space required can be reduced to just twice the #vertices in a single column of edit graph
    - Space complexity - $O(n)$
  - e.g., calculating an alignment score for n x n alignment problem
    - Requires <= $2n$ space

Space-Efficient Sequence Alignment

- Longest path in edit graph connects the source vertex *(0,0)* with the sink *(n,m)*
  - Passes through some unknown middle vertex – *(mid, m/2)*
  - Somewhere in the column *m/2* of the graph
- Definition:
  - *length(i)* – longest path from *(0,0)* to *(n,m)*, passing *(i, m/2)*
    - *length(0)* – passing *(0, m/2)*
    - *length(1)* – passing *(1, m/2)*
    - *length(n)* – passing *(n, m/2)*

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

Space-Efficient Sequence Alignment

- Vertex *(i, m/2)* splits the *length(i)*-long path into subpaths:
  - Prefix subpath: from *(0, 0)* to *(i, m/2)*
    - Length of *prefix(i)*
    - Length of the longest path from *(0, 0)* to *(i, m/2)* $\boxed{s_{i,\frac{m}{2}}}$
  - Suffix subpath: from *(i, m/2)* to *(n, m)*
    - Length of *suffix(i)*
    - Length of the longest path from *(i, m/2)* to *(n, m)*
    - Equal to length of the longest path from *(n, m)* to *(i, m/2)* in the "reversed" edit graph $\boxed{s^{reverse}_{i,\frac{m}{2}}}$

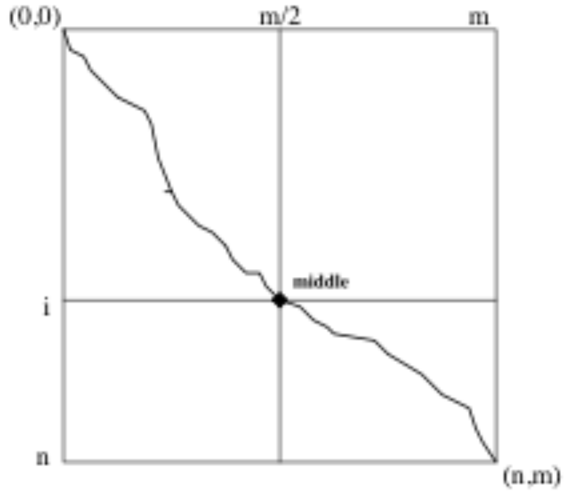
Space-Efficient Sequence Alignment

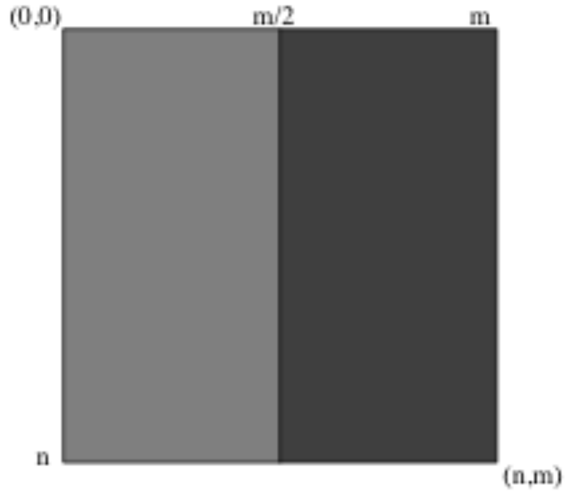
- *length(i) = prefix(i) + suffix(i)*

$$length(i) = prefix(i) + suffix(i) = s_{i,\frac{m}{2}} + s^{reverse}_{i,\frac{m}{2}}$$

  - Computes max *length(i)* gives the longest path and mid value

- Computing all *length(i)* values requires time equal to:
  - Time: Area of the left rectangle + area of the right rectangle
  - Space: O(n)



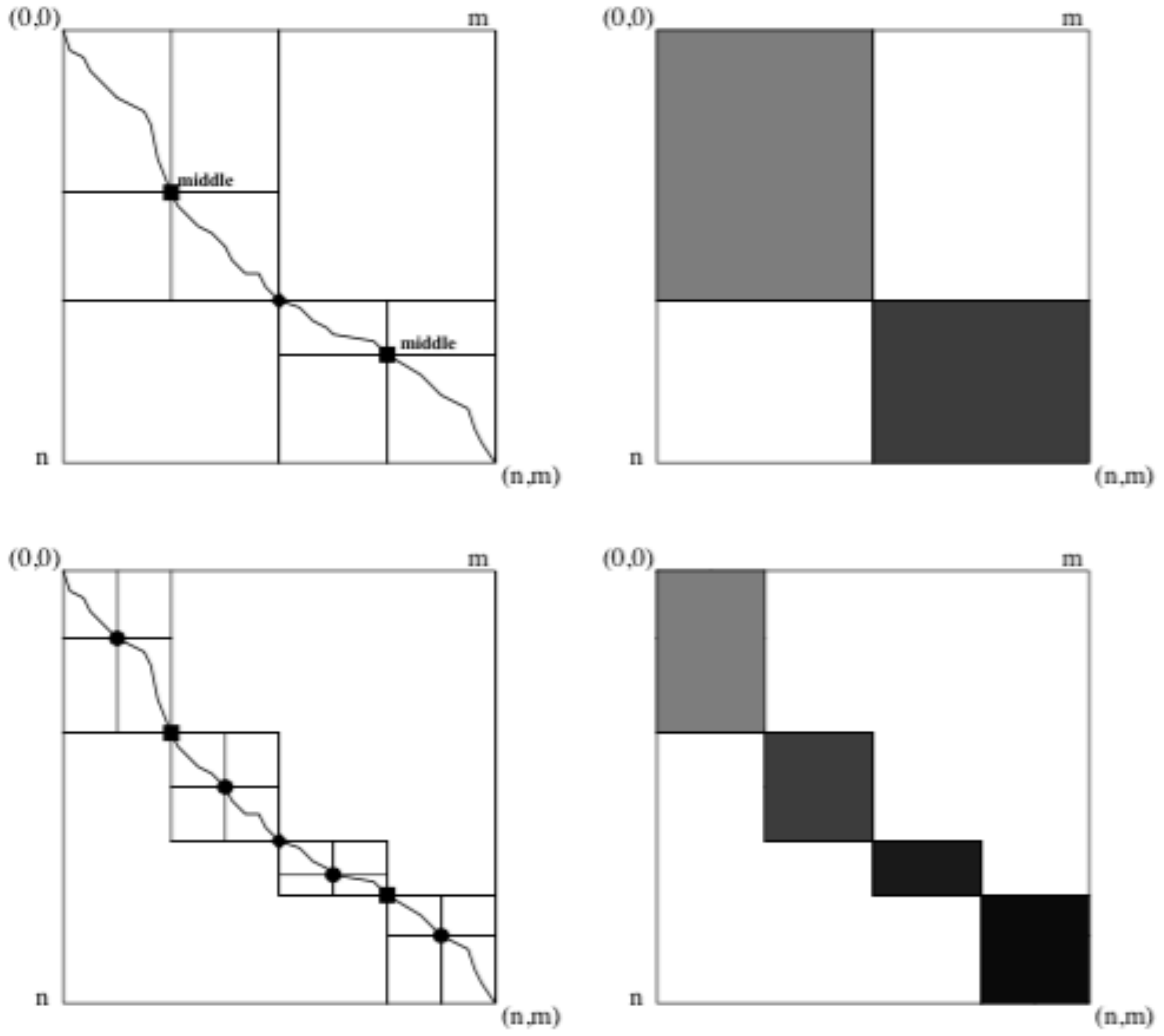


Space-Efficient Sequence Alignment

- After the middle vertex *(mid, m/2)* is found, the problem can be partitioned into two subproblems:
  - Longest path from *(0, 0)* to the *(mid, m/2)*
  - Longest path from *(mid, m/2)* to the *(n, m)*
- We can reapply the middle vertex finding with each smaller rectangles, which is half the size of the original
- Proceeding in this way we will find the middle vertices of all rectangles in time proportion to

$$area + \frac{area}{2} + \frac{area}{4} + \ldots \leq 2 \times area$$

Space-Efficient Sequence Alignment



- Time – $O(nm)$
- Space – $O(n)$

Space-Efficient Sequence Alignment

- PATH Algorithm

```
PATH($source$, $sink$)
if $source$ and $sink$ are in consecutive columns
    output longest path from $source$ to $sink$
else
    $mid \leftarrow$ middle vertex $(i, \frac{m}{2})$ with largest score $length(i)$
    PATH($source$, $mid$)
    PATH($mid$, $sink$)
```

Block Alignment

- Previously in sorting algorithms:
  - SELECTIONSORT – $O(n^2)$
  - MERGESORT – $O(n \log n)$
  - There exists a lower bound for complexity of any sorting algorithms
    - Requires at least $\Omega(n \log n)$ operations
    - Improving the worst-case running time makes no sense
    - Improving the practical running time may be a good idea
- Global alignment problems
  - Dynamic programming for 2 n-nucleotides sequences – $O(n^2)$
  - Any faster alignment algorithm?

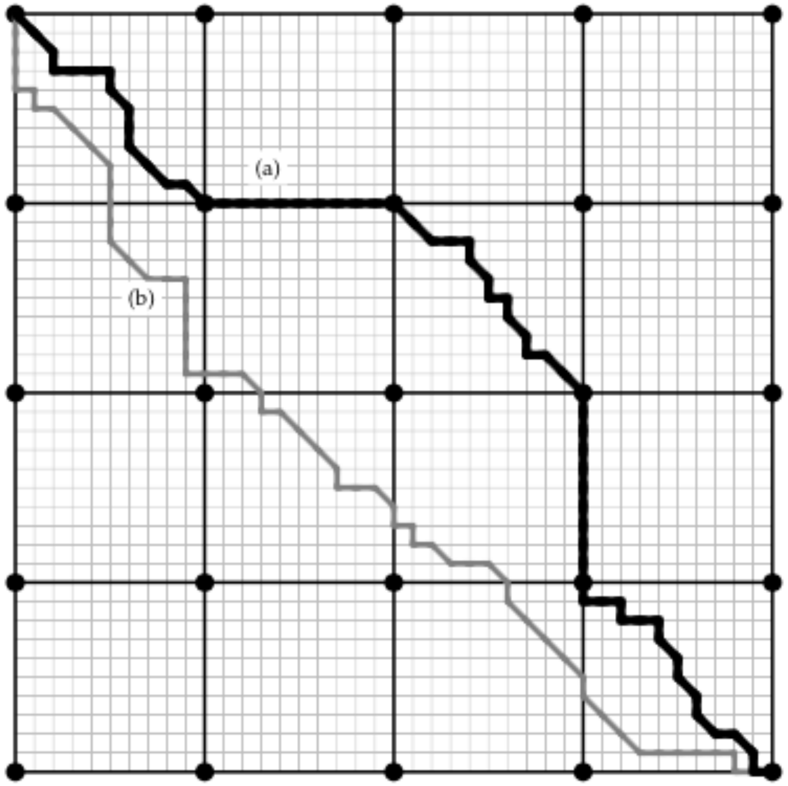
Block Alignment

- Global alignment with $O(n \log n)$ is unknown but there exists a subquadratic $O(n^2/\log n)$
- Block Alignment
  - $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_n$
  - u and v sequences are partitioned into blocks of length t

$$\mathbf{u} = |u_1 \ldots u_t| \; |u_{t+1} \ldots u_{2t}| \; \ldots \; |u_{n-t+1} \ldots u_n|$$
$$\mathbf{v} = |v_1 \ldots v_t| \; |v_{t+1} \ldots v_{2t}| \; \ldots \; |v_{n-t+1} \ldots v_n|$$

  - Every block (i.e. substring) in one sequence is either:
    - Aligned against an entire block in the other sequence,
    - Or is inserted or deleted as a whole

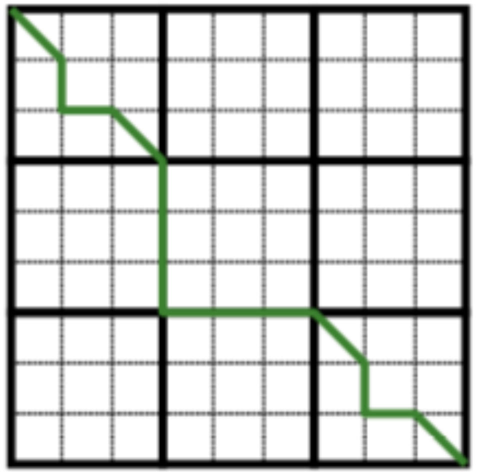
Block Alignment

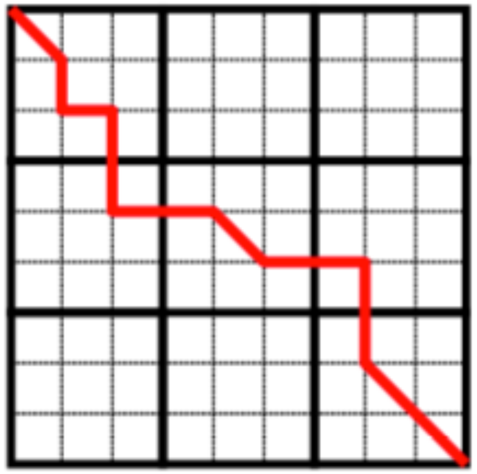


- A path in the edit graph is now called a block path
    - Traverses every $t \times t$ square through its corners
    - Enter and leaves every block at **bold vertices**
- Block alignment problem
    - Find the longest block path through an edit (block) graph

Four-Russians Technique

- Block Alignment vs. LCS
  - Block alignment only cares about the corners of the blocks
  - LCS cares about all points on the edges of the blocks

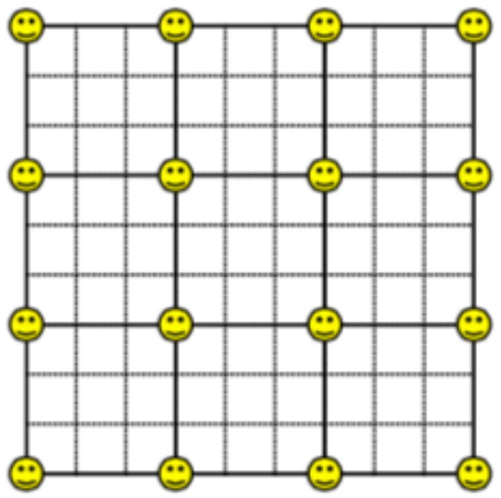

block alignment


longest common subsequence

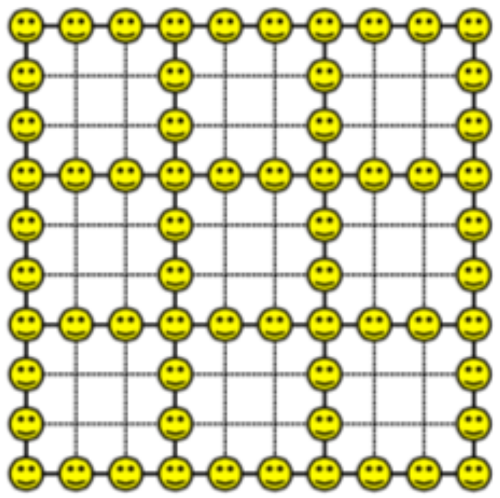
Four-Russians Technique

- How many points of interest?

block alignment



longest common subsequence



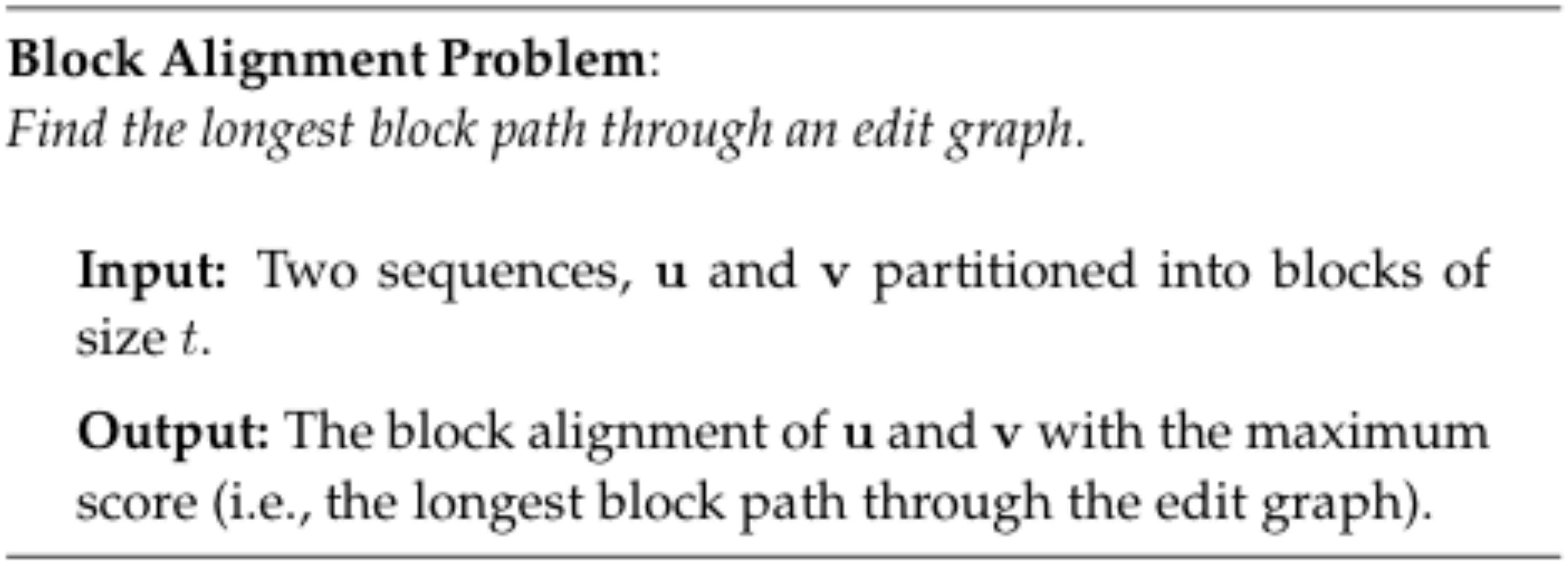
How may blocks?
$(n/t)^*(n/t) = (n^2/t^2)$

How many points of interest? $O(n^2/t)$

n/t rows with n vertices each

n/t columns with n vertices each

Block Alignment



Block Alignment Problem:
Find the longest block path through an edit graph.

Input: Two sequences, **u** and **v** partitioned into blocks of size t.

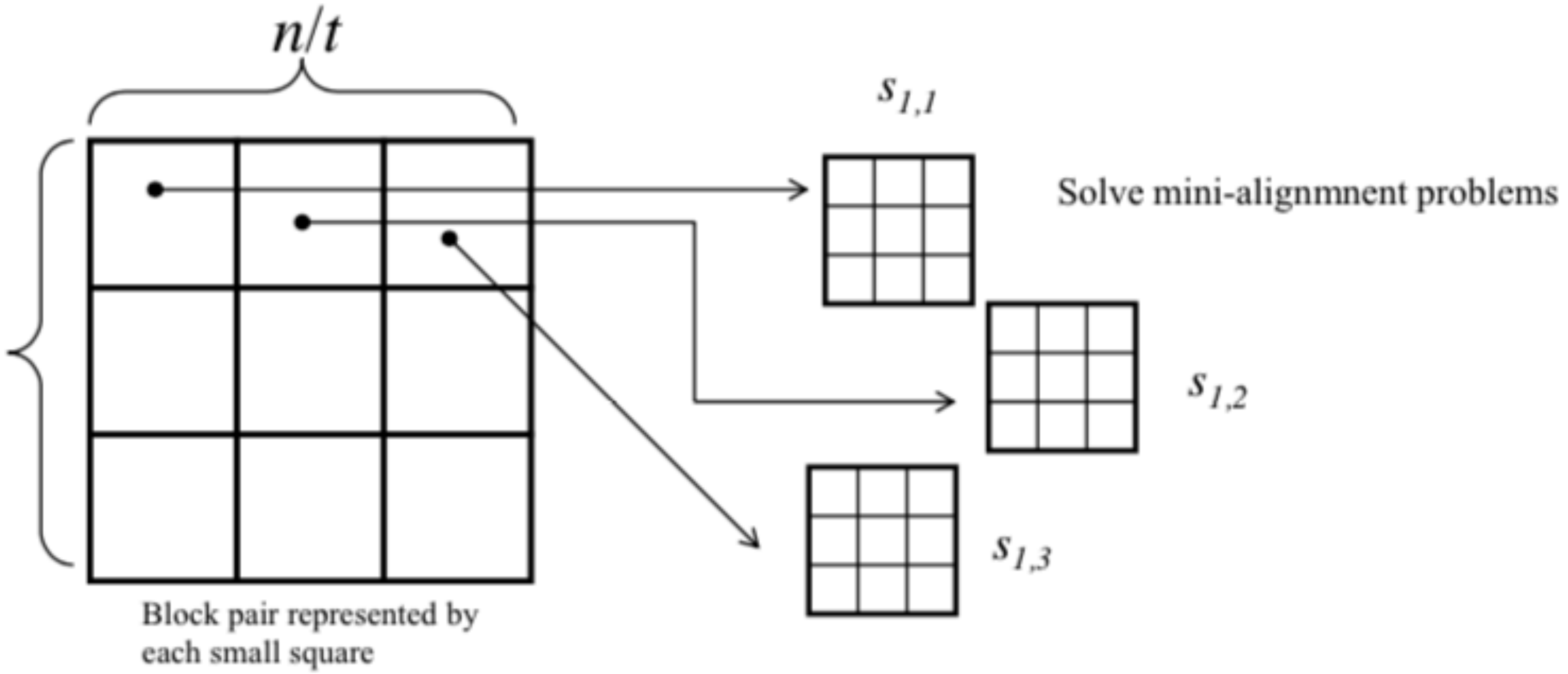
Output: The block alignment of **u** and **v** with the maximum score (i.e., the longest block path through the edit graph).

Block Alignment

- Constructing alignments within Blocks
  - Consider (n/t) x (n/t) pairs of blocks
  - Compute the alignment score $\beta_{i,j}$ for each pair of blocks
    - Substring: $S_1[\, i \ldots (i+t-1) \,]$
    - Substring: $S_2[j \ldots (j+t-1) \,]$
  - For each block pair, solve a minialignment problem of size t x t

Block Alignment

- Step 1: computer the minialignments



n/t
$s_{1,1}$
Solve mini-alignmnent problems
$s_{1,2}$
$s_{1,3}$
Block pair represented by each small square

How many blocks?
$(n/t)*(n/t) = (n^2/t^2)$

Block Alignment

- Step 2: Dynamic Programming
  - The optimal block alignment score , $S_{i,j}$, between the first i blocks of u and the first j blocks of v :

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

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

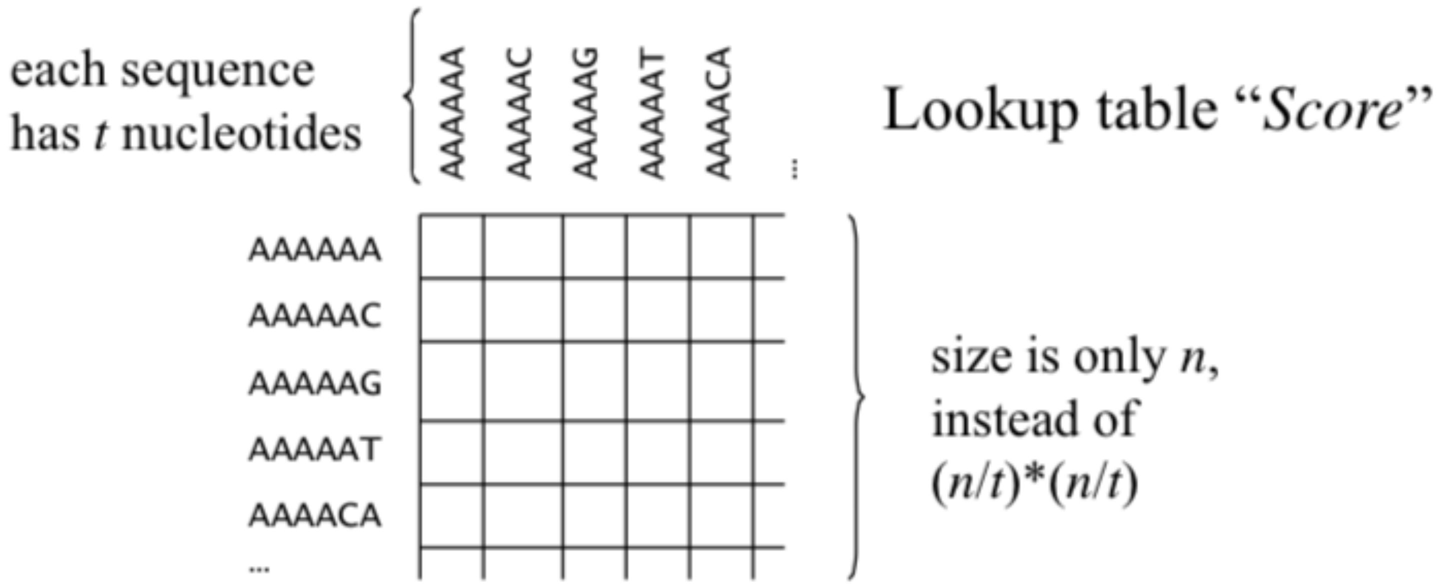
- Block Alignment Runtime
  - Indices i,j range from 0 to n/t
  - Running time of the algorithm – $O(\ n/t\ .\ n/t\ ) = O(\ n^2/t^2\ )$
- Computing all alignment score $\beta_{i,j}$ requires
  - Solving $(n/t).(n/t) = (n^2/t^2)$ mini block alignments
  - Each of the block of size $t \text{ x } t = t^2$
  - All take time – $O(\ n^2/t^2\ .\ t^2\ ) = O(\ n^2\ )$
  - The same as dynamic programming!!!
    - How do we speed this up?

Four-Russians Technique

- The speed reduction of block alignment algorithm is achieved when t is roughly $log\ n$ (base-2 logarithm)
- Basic idea is to precompute parts of the computation involved in filling out the dynamic programming table
- Instead of constructing n/t x n/t minialignments,
  - Constructing 4^t x 4^t minialignments
  - If $t = \frac{\log n}{4}$ then $4^t \times 4^t = n^{\frac{1}{2}} \times n^{\frac{1}{2}} = n$,
    - The $Score$ lookup table has only n entries

Four-Russians Technique

- Lookup Table


each sequence has t nucleotides
AAAAAA
AAAAAC
AAAAAG
AAAAAT
AAAACA
...
Lookup table "*Score*"
size is only n, instead of $(n/t)*(n/t)$

- We construct $4^t \times 4^t = n$ minialignments for all pairs of t-nucleotide strings, and store their alignment scores in a large lookup table
- Computing each entry takes time - $O(\log n \,.\, \log n)$
- Time to compute all entries – $O(n \,.\, (\log n)^2)$

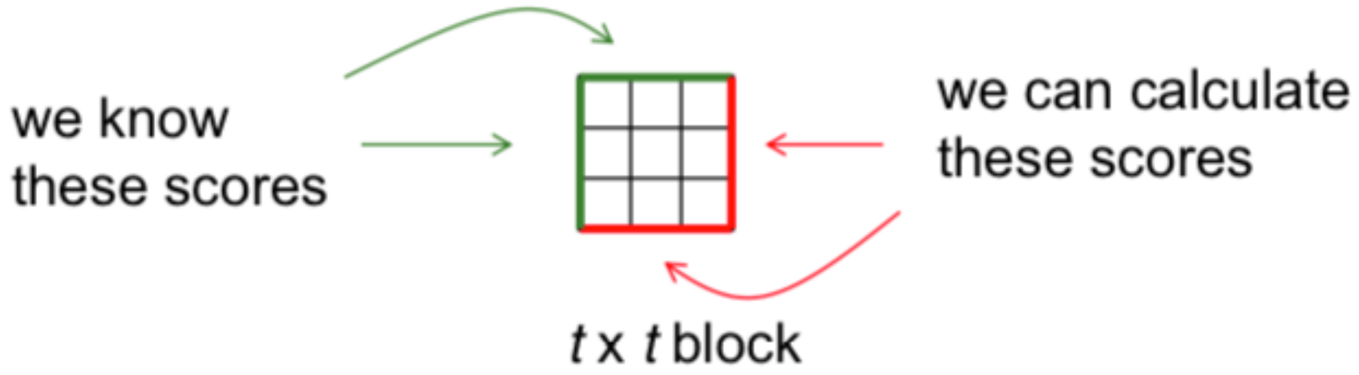
Four-Russians Technique

- The resulting two-dimensional lookup table *Score* of size n is indexed by a pair of t-nucleotide strings :

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

Four-Russians Technique

- With regular dynamic programming, compute the table would take quadratic time – $O(n^2)$
- Applying the "Four-Russians" Tabulation



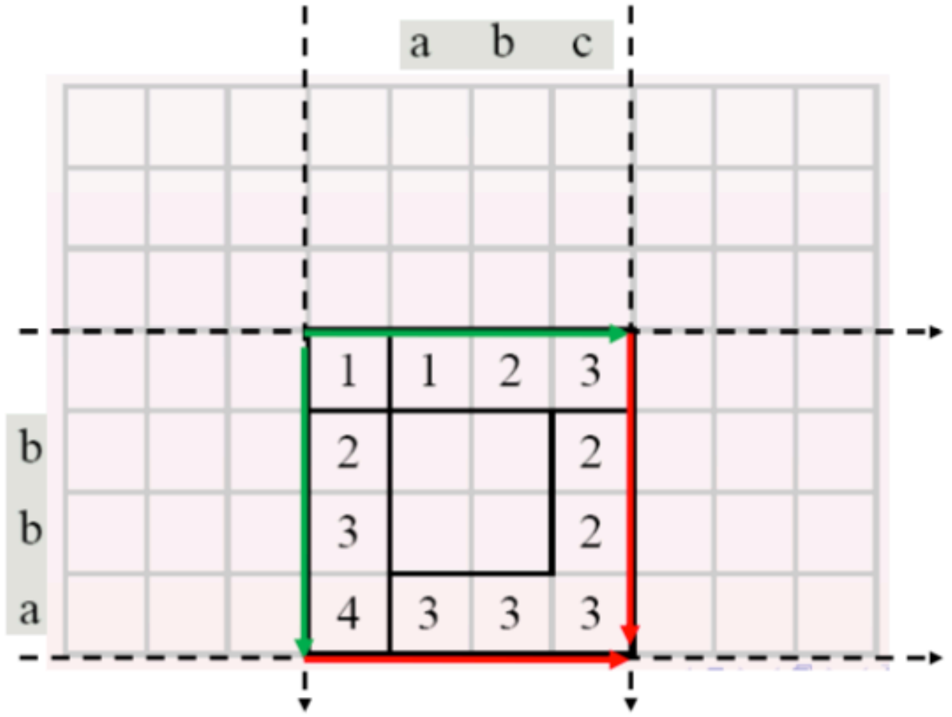
  - Given alignment scores in the first row, $S_{i,*}$, and the first column, $S_{*,j}$, of $t \times t$ block
  - Compute the alignment scores in the last row and column

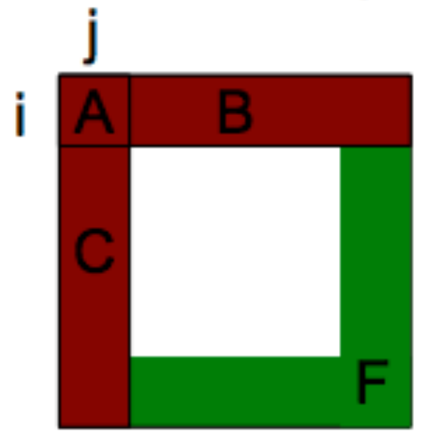
Four-Russians Technique

- We use 5 variables:
  1. Value in the upper left cell
  2. Alignment scores $S_{i,*}$ in the first row
  3. Alignment scores $S_{*,j}$ in the first column
  4. Substring of u in this block (4^t possibilities)
  5. Substring of v in this block (4^t possibilities)
- Build a lookup table for all possible values of variables 2. to 5.
- For each quadruple we store the value of the score for the last row and last column

Four-Russians Technique

- Example:





I = (1,1, 2, 3), (1, 2, 3, 4), abc, bba)

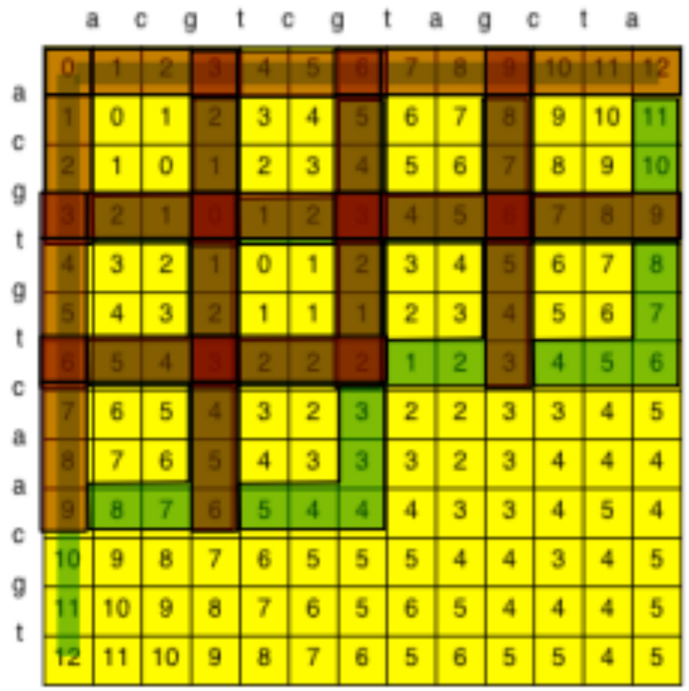
$n^t \quad * \, n^t \quad * \; 4^t \quad * \; 4^t \; = (4n)^{2t}$

This will be a huge table!
we need another trick…

O = (4, 3, 3, 3, 2, 2, 3)

Four-Russians Technique

- Example:



		a	c	g	t	c	g	t	a	g	c	t	a
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	1	0	1	2	3	4	5	6	7	8	9	10	11
c	2	1	0	1	2	3	4	5	6	7	8	9	10
g	3	2	1	0	1	2	3	4	5	6	7	8	9
t	4	3	2	1	0	1	2	3	4	5	6	7	8
g	5	4	3	2	1	1	1	2	3	4	5	6	7
t	6	5	4	3	2	2	2	1	2	3	4	5	6
c	7	6	5	4	3	2	3	2	2	3	3	4	5
a	8	7	6	5	4	3	3	3	2	3	4	4	4
a	9	8	7	6	5	4	4	4	3	3	4	5	4
c	10	9	8	7	6	5	5	5	4	4	3	4	5
g	11	10	9	8	7	6	5	6	5	4	4	4	5
t	12	11	10	9	8	7	6	5	6	5	5	4	5

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return D[n,m]

Note: We (of course) do not allocate the entire D-table, since this would take time $O(n^2)$ by itself. We allocate a “row of blocks”.

Note: here we use distance approach instead of similarity

Four-Russians Technique

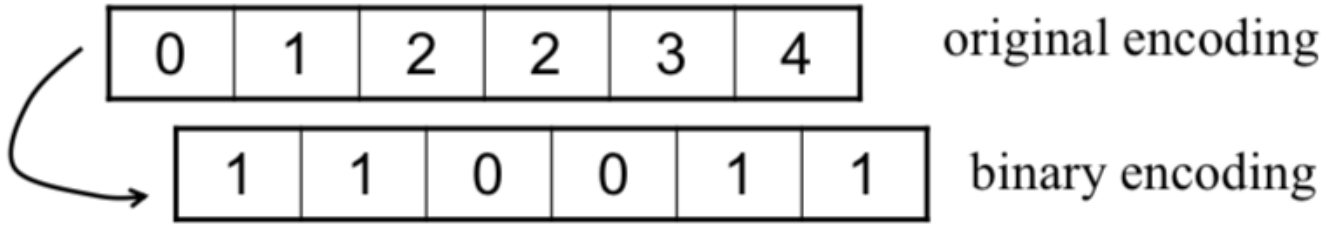
- Overall running time is dominated by the dynamic programming step (accessing to the lookup table)
  - Accessing elements in table = $n/t \times n/t = n^2/t^2$
  - Each access takes time = $O(\log n)$
  - Overall running time - subquadratic

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

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

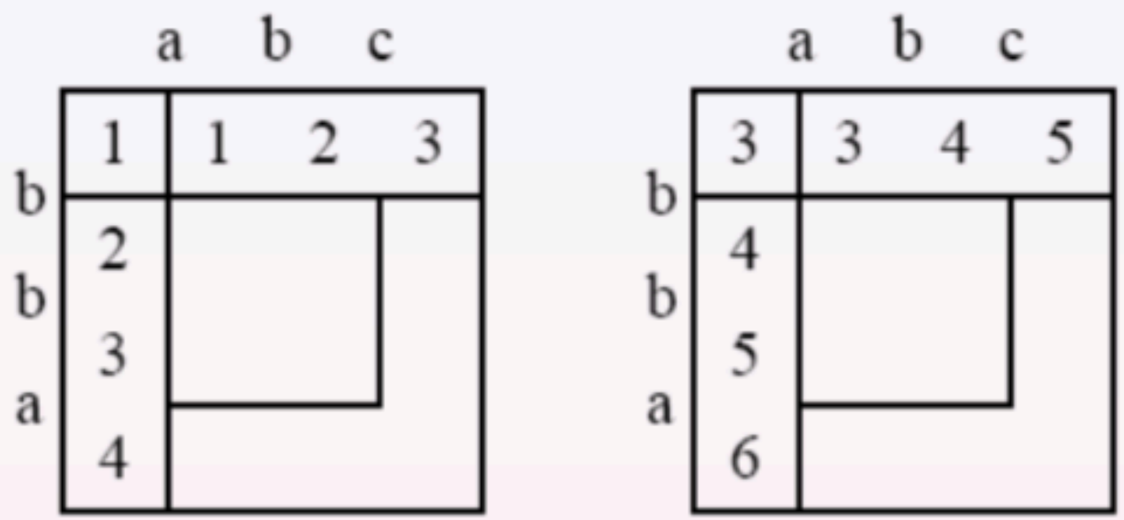
Four-Russians Technique

- A careful analysis of the LCS problem shows that …
  - The possible alignment score in the first row or first column are not so random, i.e., 0, 1, 2, 2, 2, 3, 4
  - Monotonically increasing
  - Adjacent elements cannot differ by more than 1
- We can encode this as a binary vector of differences:



Four-Russians Technique

- Example:

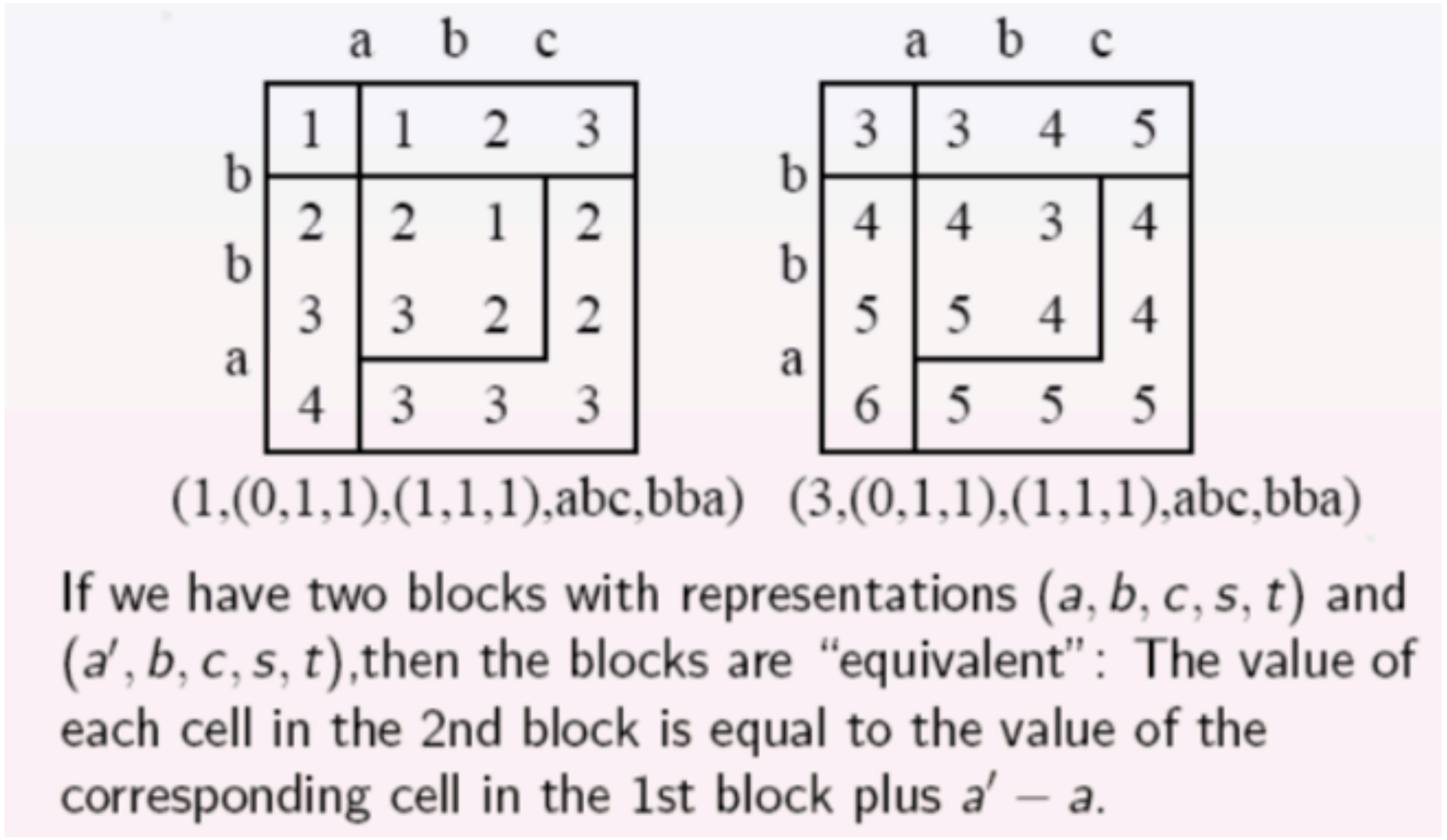


(1,(0,1,1),(1,1,1),abc,bba) (3,(0,1,1),(1,1,1),abc,bba)

If we have two blocks with representations (a, b, c, s, t) and (a', b, c, s, t),then the blocks are "equivalent":

Four-Russians Technique

- Example: precompute only (0, (0,1,1), (1,1,1), abc, bba)



		a	b	c
	1	1	2	3
b	2	2	1	2
b	3	3	2	2
a	4	3	3	3

(1,(0,1,1),(1,1,1),abc,bba)

		a	b	c
	3	3	4	5
b	4	4	3	4
b	5	5	4	4
a	6	5	5	5

(3,(0,1,1),(1,1,1),abc,bba)

If we have two blocks with representations (a, b, c, s, t) and (a', b, c, s, t),then the blocks are "equivalent": The value of each cell in the 2nd block is equal to the value of the corresponding cell in the 1st block plus $a' - a$.

Four-Russians Technique

- Reducing Lookup Table Size

$$(0, (0,1,1), (1,1,1), abc, bba)$$

  - Possibilities = $2^t \,.\, 2^t \,.\, 4^t \,.\, 4^t = 2^{6t}$
  - Computing each entry in the table = t2
  - Total table construction time = $2^{6t} \,.\, t^2$
  - let $t = (\log n)/6$, total table construction time

$$2^{6((\log n)/6)} \,.\, (\log n)^2$$

$$n\,(\log n)^2$$

Four-Russians Technique

- Reducing Lookup Table Size
  - Step 1: Table construction time

$$2^{6((\log n)/6)} . (\log n)^2 = n (\log n)^2$$

  - Step 2: Alignment with dynamic programming

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

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