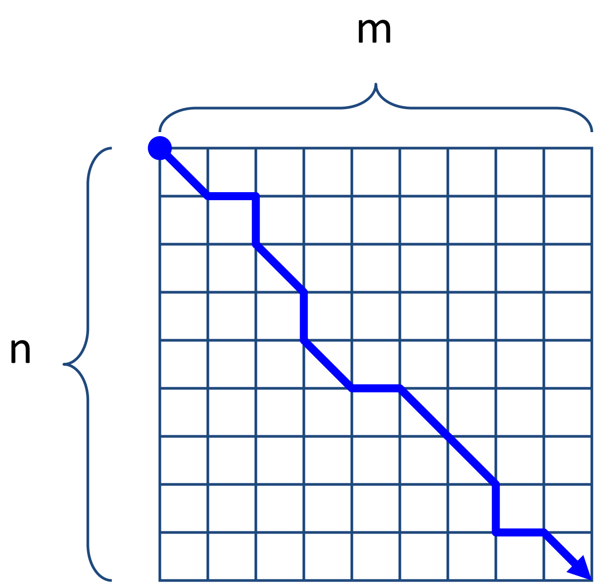
Space Efficient Sequence Alignment

A Divide-and Conquer Algorithm: The limiting resource for computing the dynamic programming computation table is not the running time but the storage or space required to to store the table. We will present a linear space algorithm [Hirschberg,1975] at the expense of doubling the computation time using a divide-and conquer algorithm.

Computing Alignment Path Requires Quadratic Memory

Alignment Path

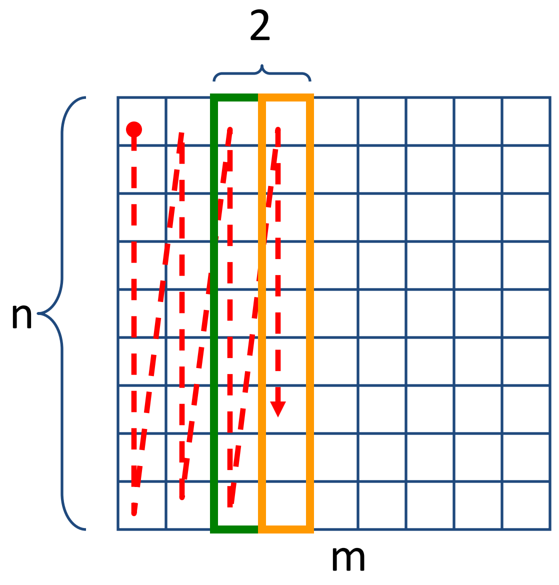
- Space complexity for computing alignment path for sequences of length *n* and *m* is O(*nm*)
- We need to keep all backtracking references in memory to reconstruct the path (backtracking)



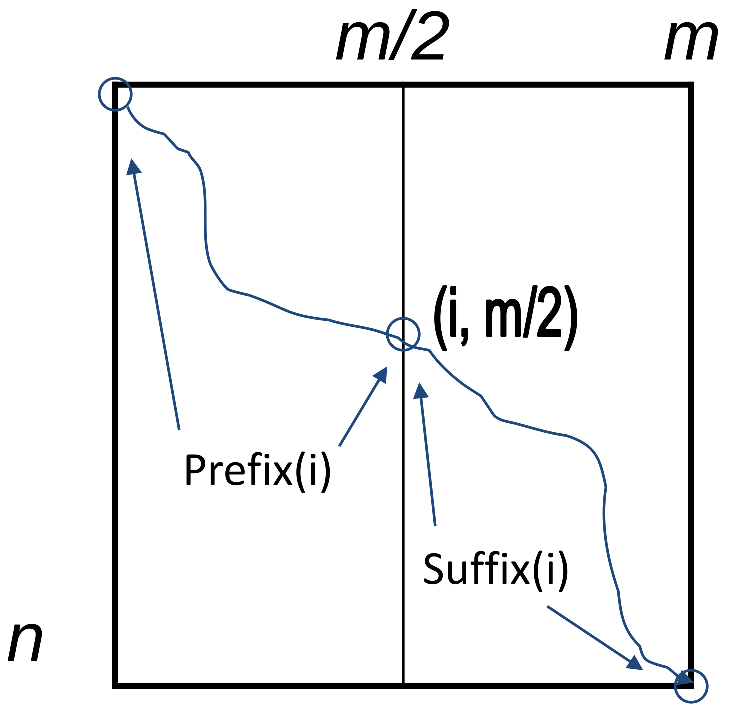
Computing Alignment Score with Linear Memory

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. Alternately, we can do the same row-by-row. But to find longest path in the edit graph, we need to store all the backtracking pointers which takes *O*(*nm*) space.



Crossing the Middle Line



We want to calculate the longest path from (0,0) to (*n*,*m*) that passes through some unknown middle vertex (*mid*,*m*/2). Can we find this middle vertex without knowing the longest path?

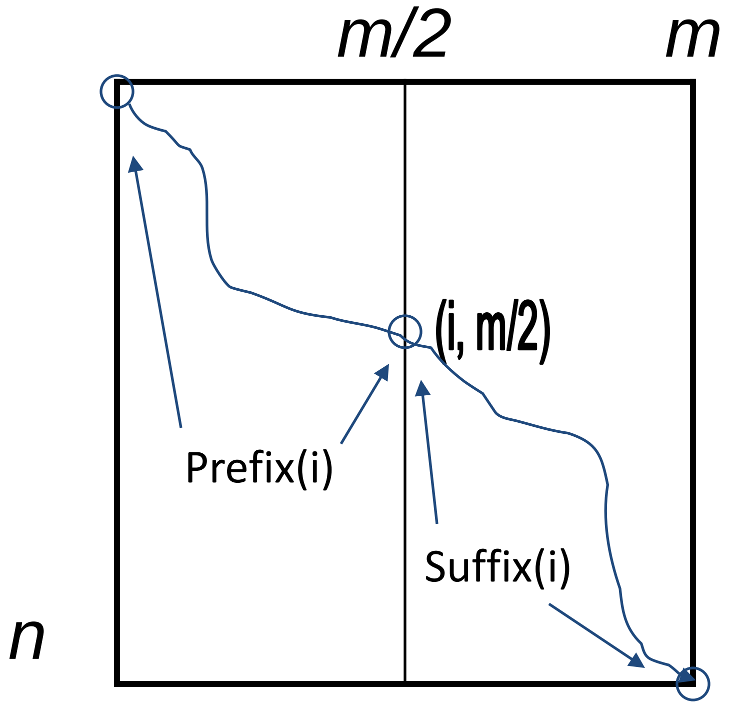
Define *length*(*i*) as the length of the longest path from (0,0) to (*n*,*m*) that passes through vertex (*i*, *m*/2) where *i* ranges from 0 to *n* and represents the *i*-th row

Vertex (*i, m/2*) splits the length(*i*)-long path into a *prefix* and a *suffix* with lengths *prefix*(i) and *suffix(i)* ,respectively. The prefix runs from source (0,0) to (*i, m/2*) and its length is *prefix*(i). The suffix runs from (*i, m/2*) to sink (*n,m*) and its length is length of the longest path from sink (n,m) to (*i, m/2*) computed in the reverse direction in the "reversed edit graph".

We also have

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

Crossing the Middle Line

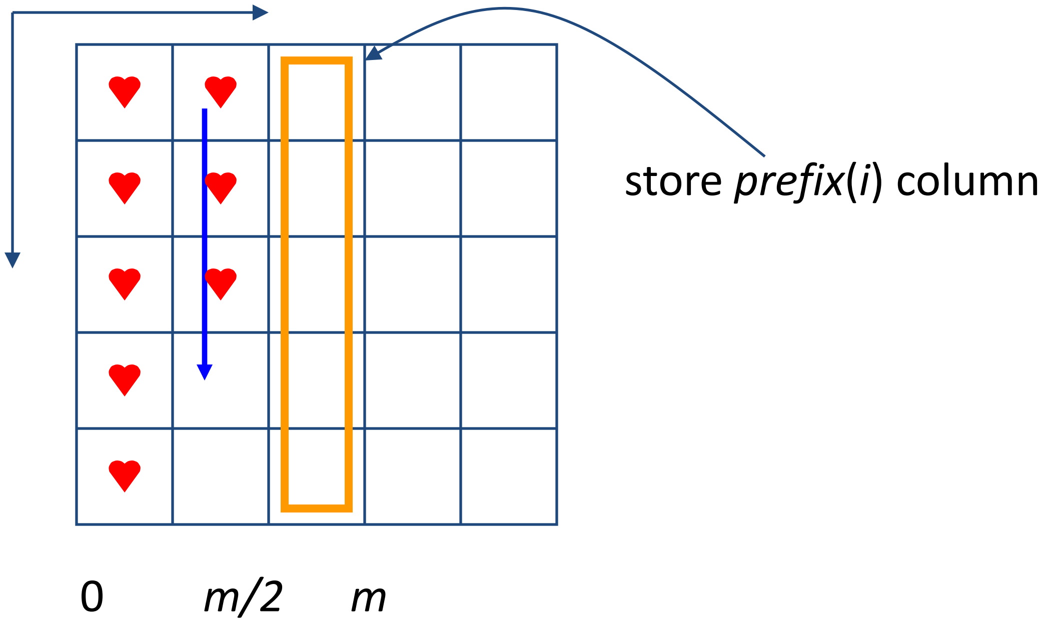


Define (*mid*,*m*/2) as the vertex where the longest path crosses the middle column.

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

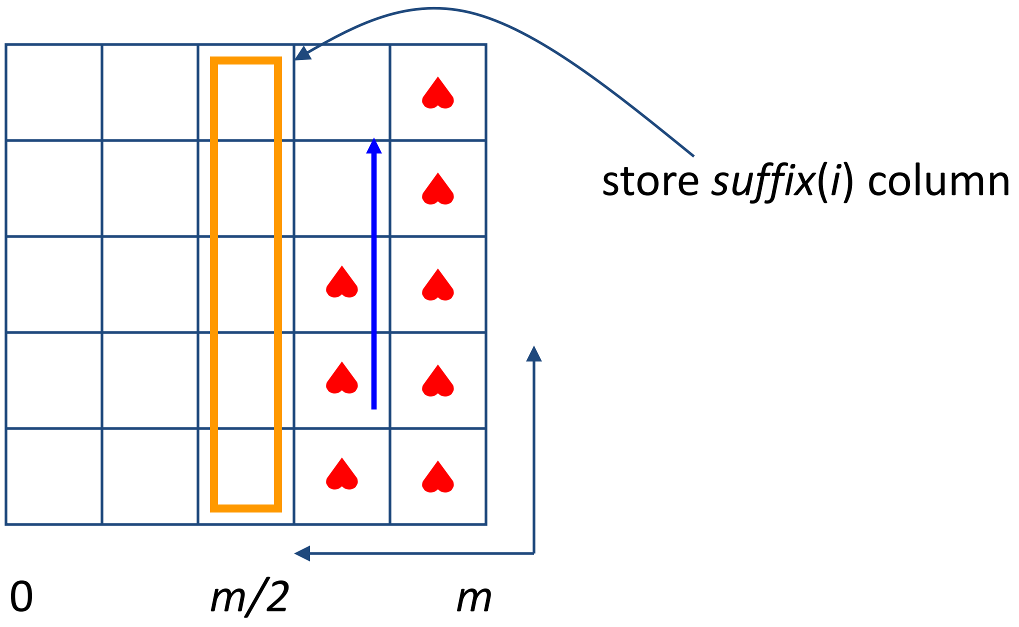
Computing Prefix(*i*)

- *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



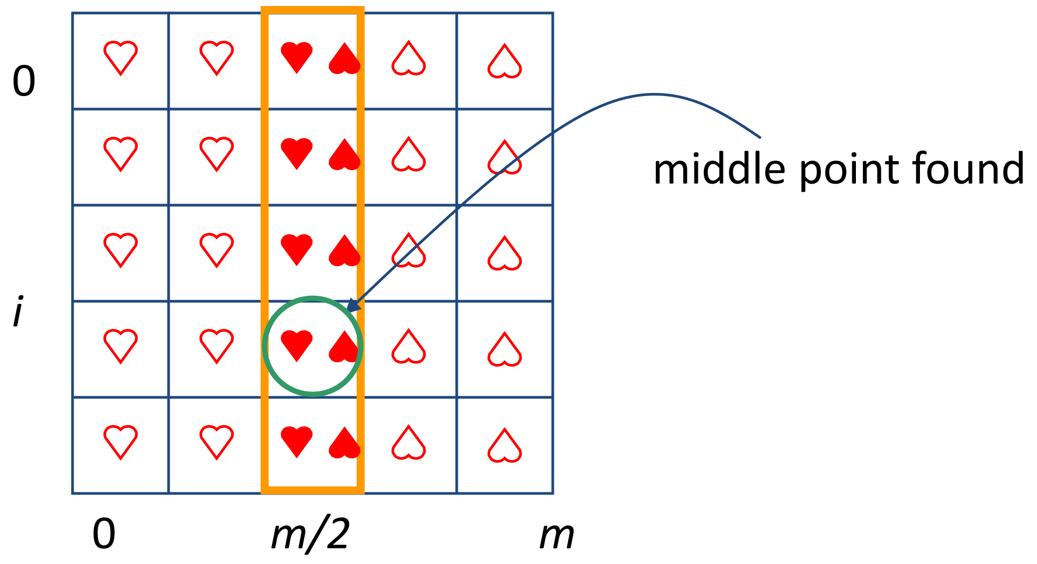
Computing Suffix(*i*)

- *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

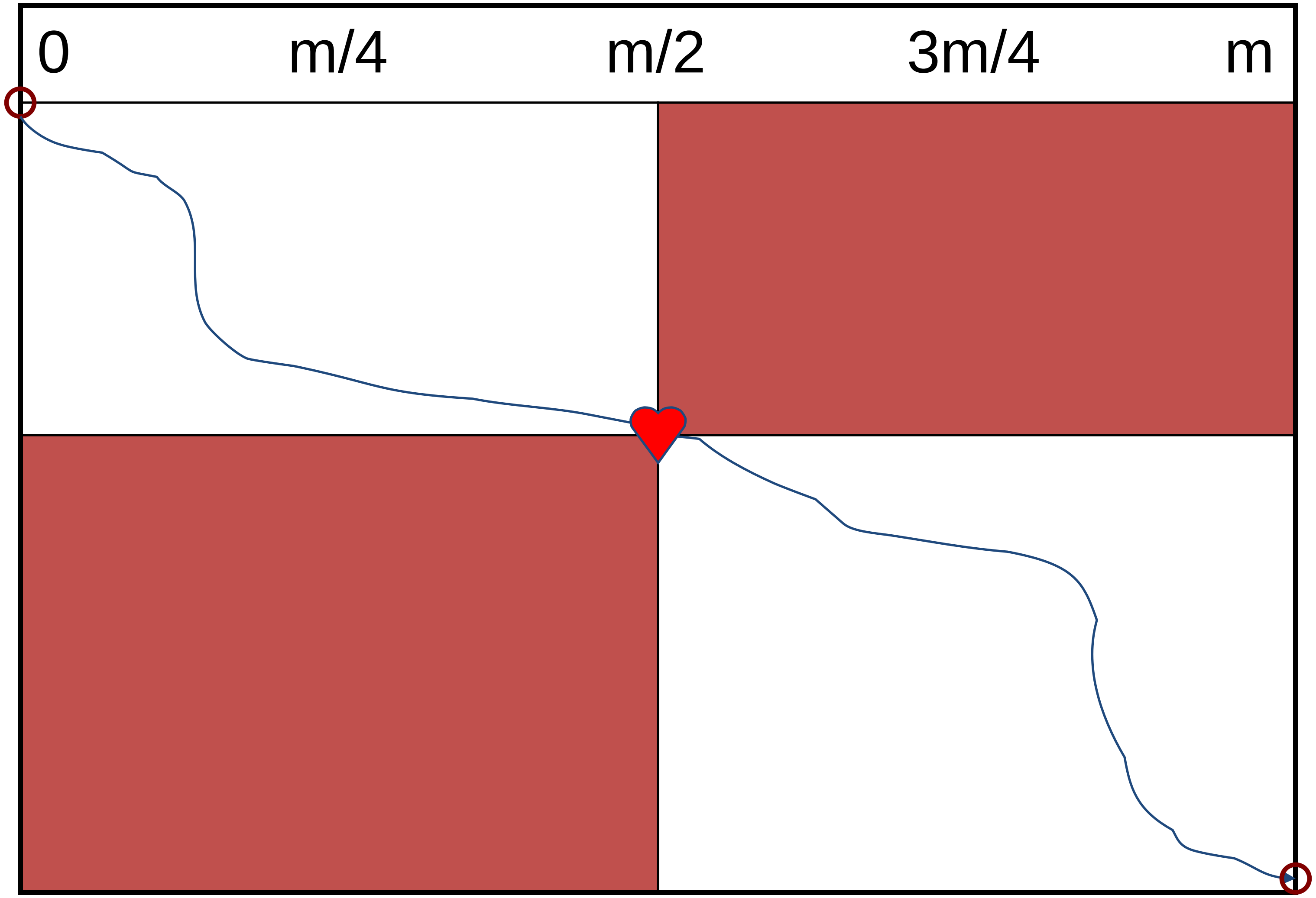


Length(i) = *Prefix*(*i*) + *Suffix*(*i*)

- 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)*

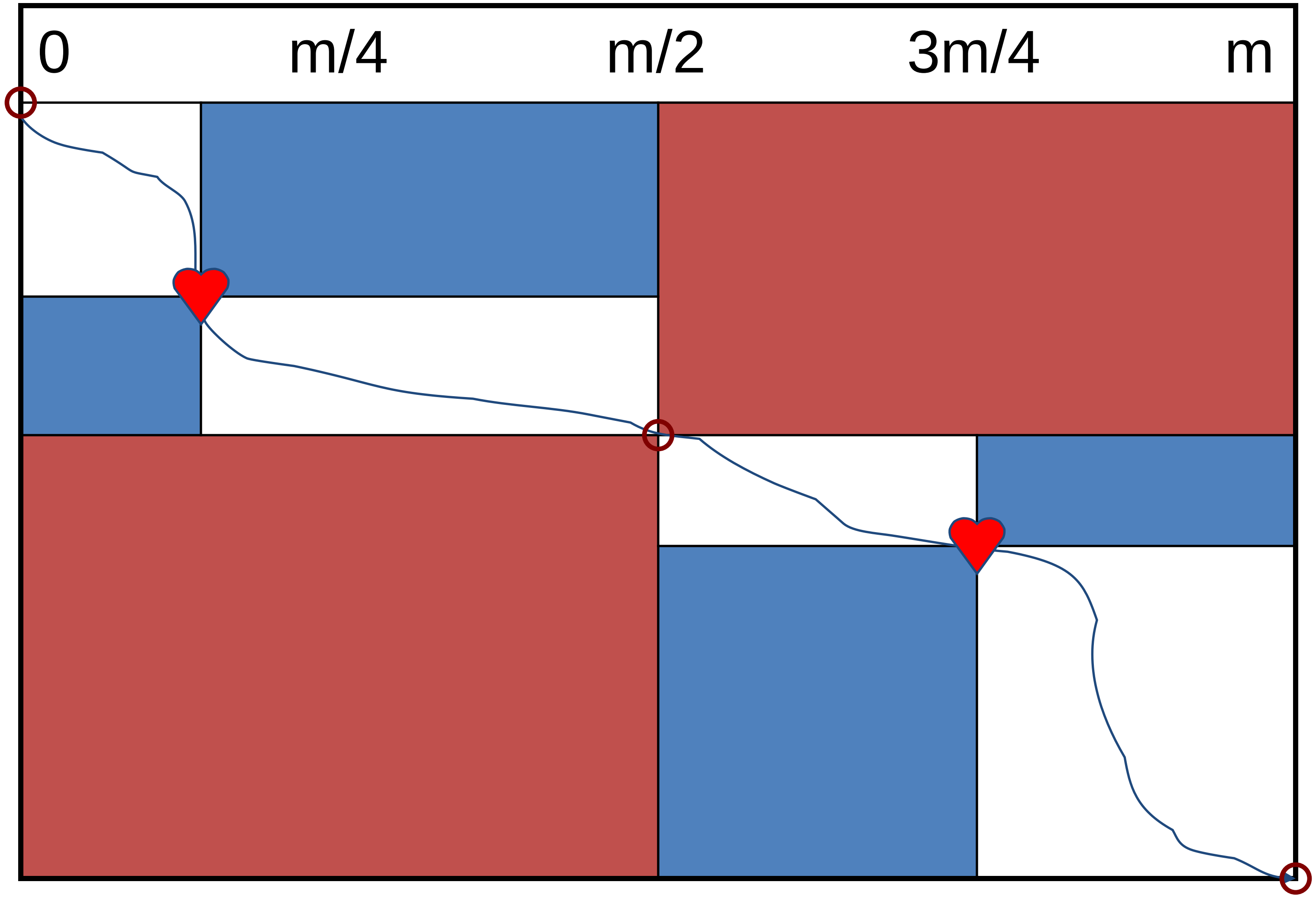


Finding the Middle Point

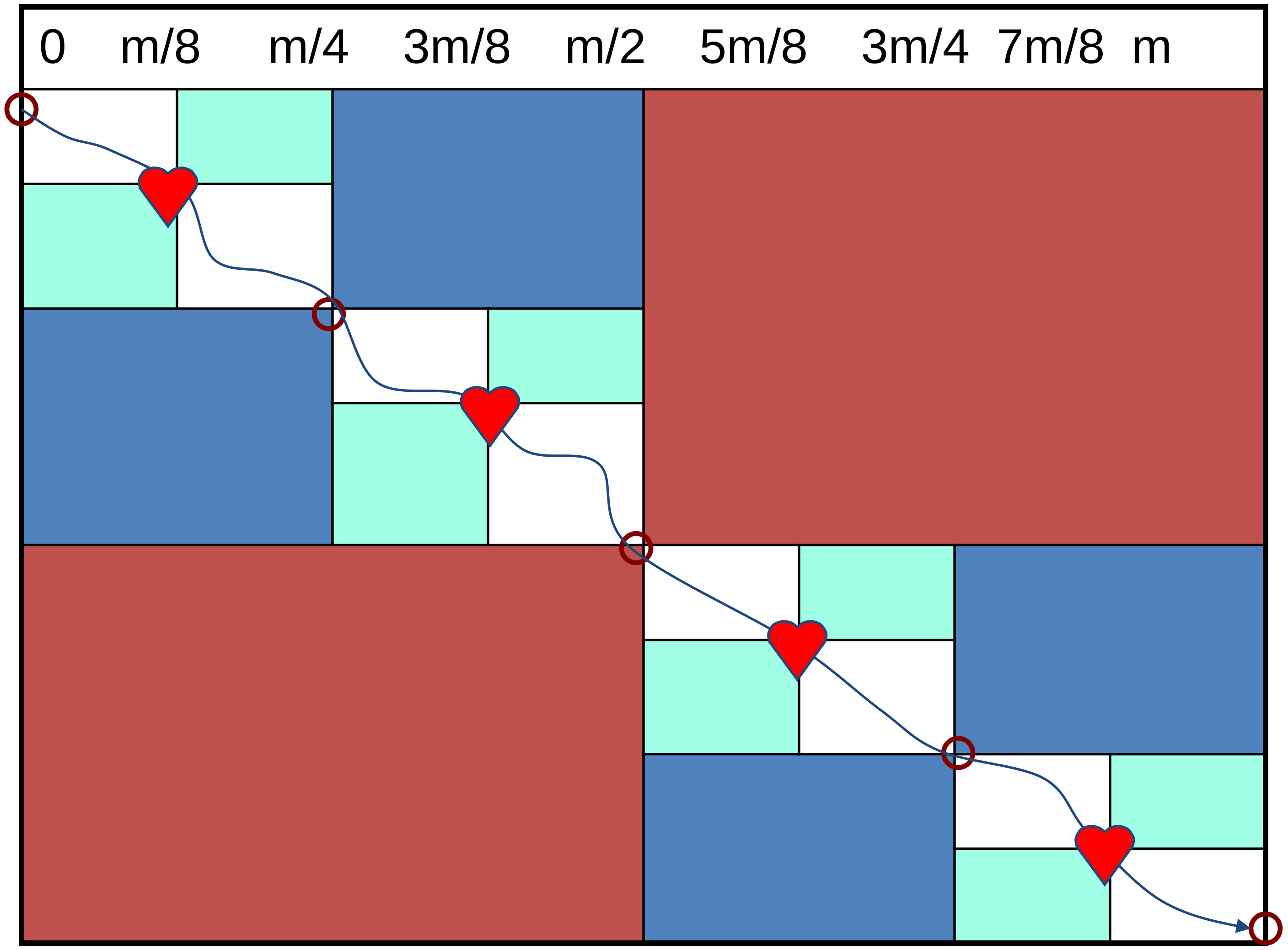


After the middle vertex (*mid, m/2*) is found , the problem of finding the longest path from source to sink can be partitioned into two sub-problems: to find longest path from (0,0) to (*mid, m/2*) and to find longest path from (*mid, m/2*) to (*n,m*). For this, we iterate the partitioning process of finding the "*mid*" points of smaller and smaller rectangles .

Finding the Middle Point again



And Again



The Algorithm

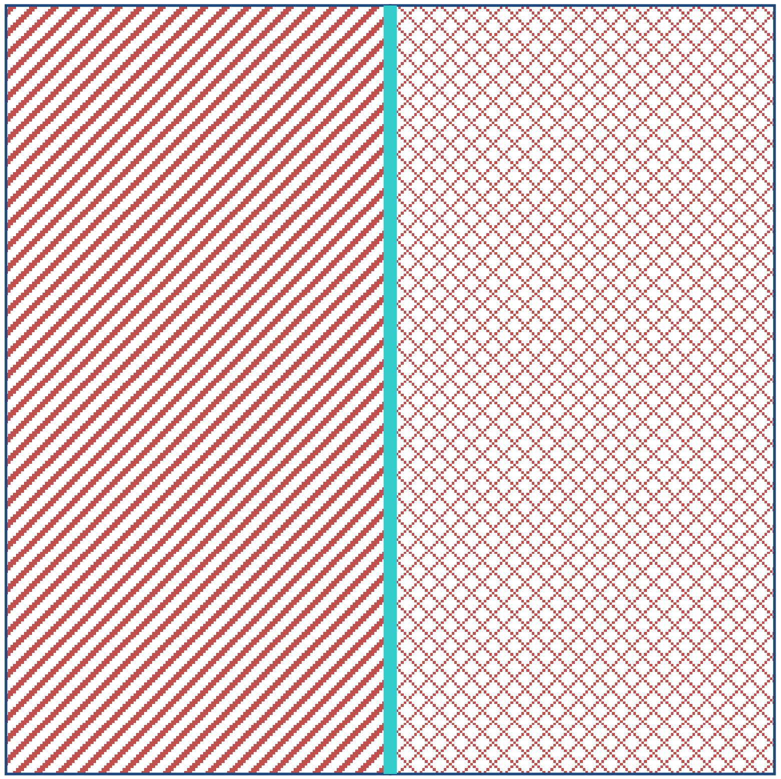
```
Path (source, sink)
If source and sink are in consecutive columns
Output longest path from source to sink
else
  mid <- middle vertex (i , m/2) with largest score length(i)
  Path (source, mid)
  Path (mid, sink)
```

The total storage needed to store the mid points for all the calls for "Path" is $O(n)$

Time = Area: First Pass

- On first pass, the algorithm covers the entire area

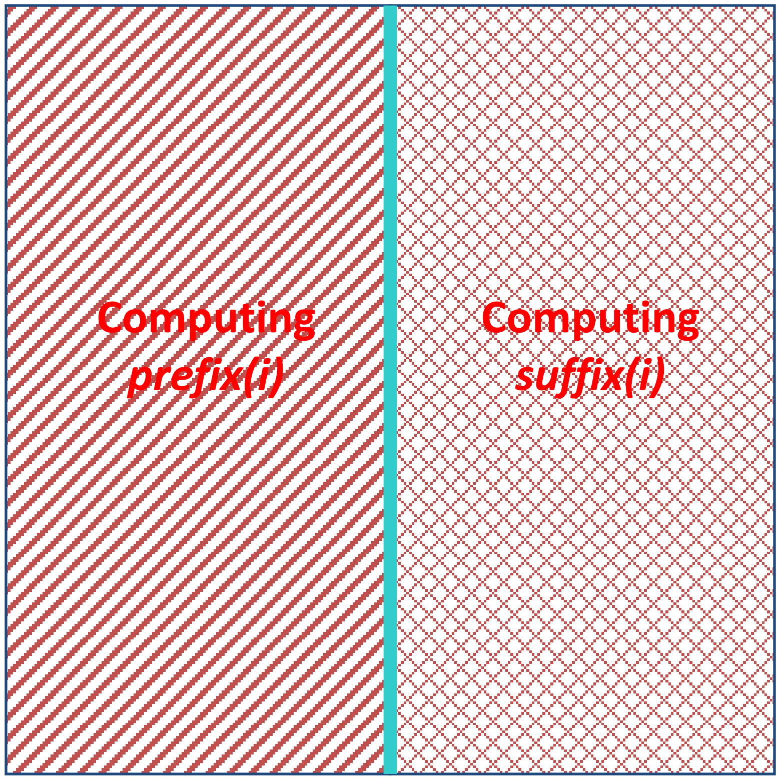
Area $= n{\bullet}m$



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

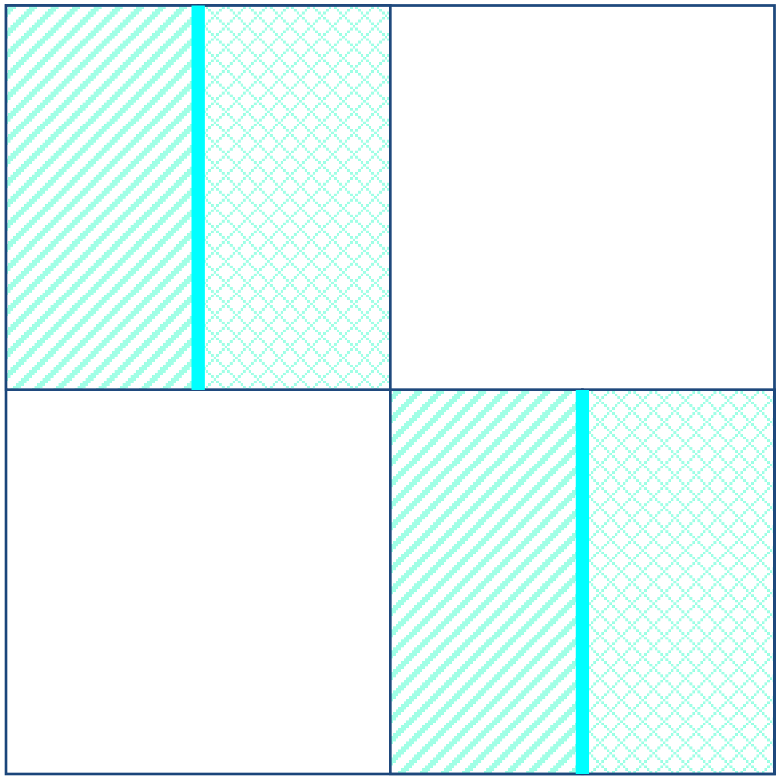
Area $= n \bullet m$



Time = Area: Second Pass

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

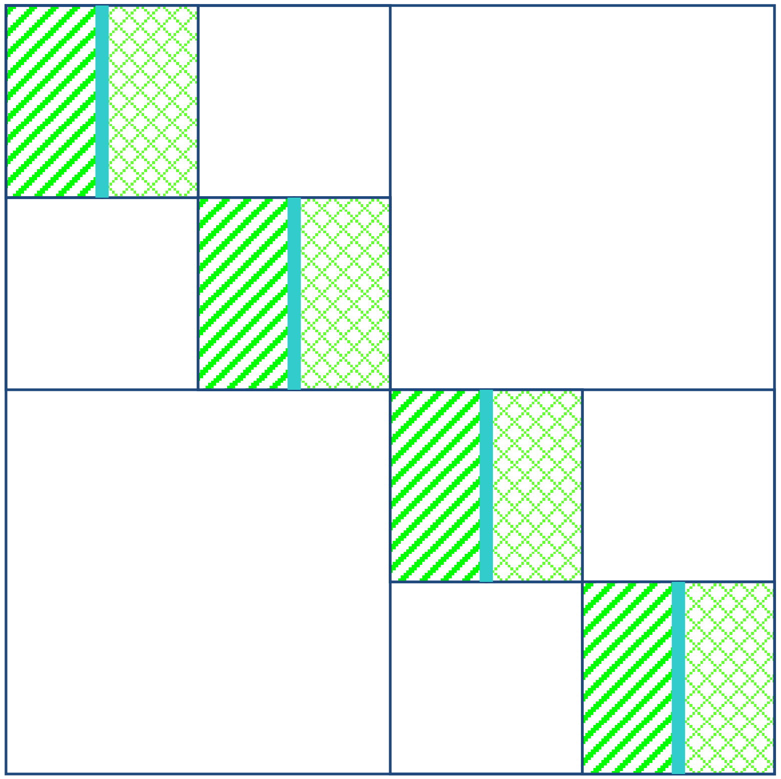
Area/2



Time = Area: Third Pass

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

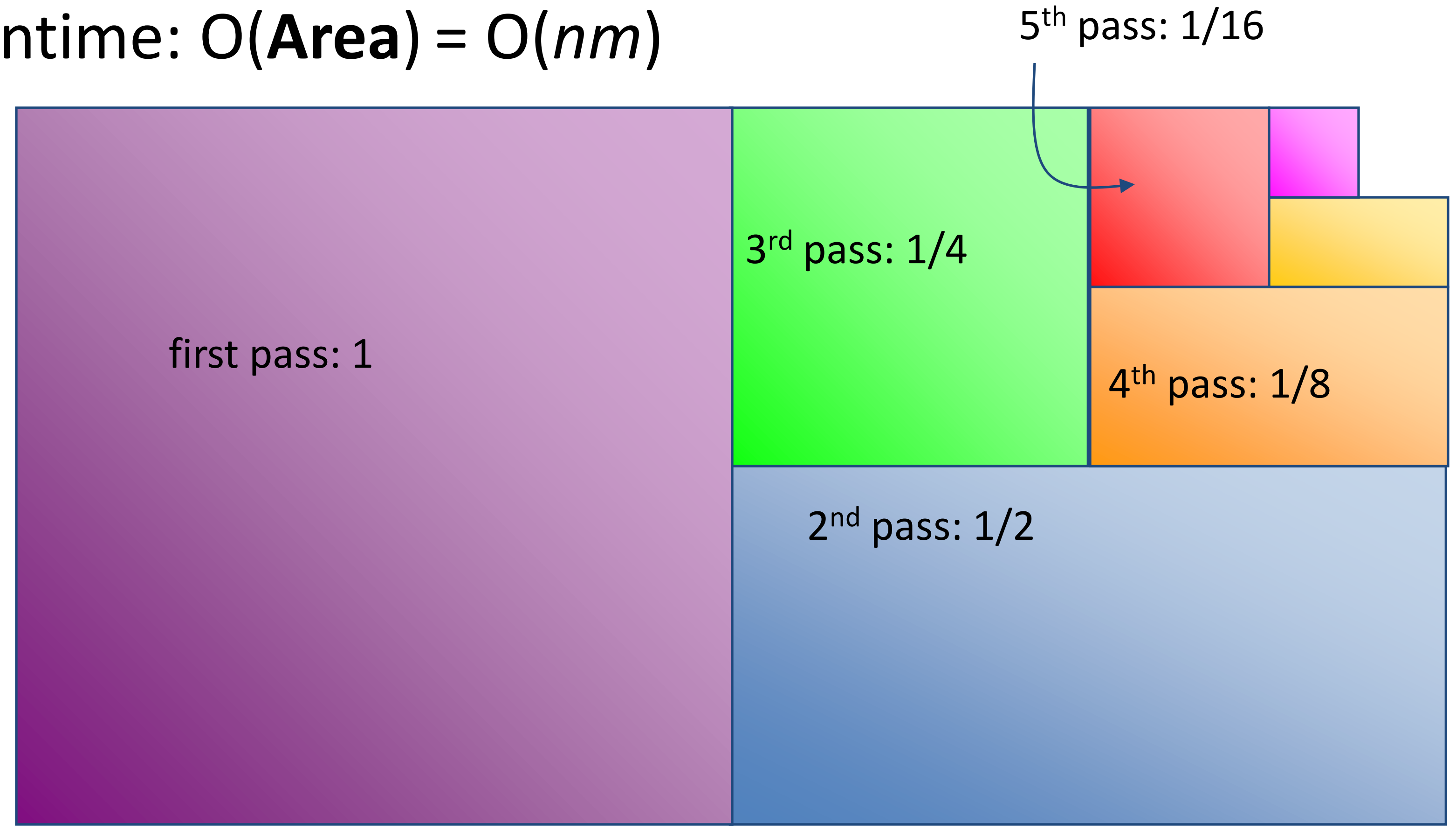
Area$/4$



Geometric Reduction At Each Iteration

$1 + \frac{1}{2} + \frac{1}{4} + \ldots + (\frac{1}{2})^k \leq 2$

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


5th pass: 1/16
first pass: 1
3rd pass: 1/4
4th pass: 1/8
2nd pass: 1/2