Alignment in linear space

Chapter 7 of Jones and Pevzner

Sequence Alignment: Linear Space

- Q. Can we avoid using quadratic space?
- Easy. Optimal value in O(m + n) space and O(mn) time.
  - Compute OPT(i, •) from OPT(i-1, •).
  - No easy way to recover alignment itself.
- Optimal longest common subsequence in O(m + n) space and O(mn) time [Hirschberg (1975)].
  - Clever combination of divide-and-conquer and dynamic programming.
- Application to sequence alignment:
  E.W. Myers and W. Miller. Optimal alignments in linear space. Computer Applications in Biosciences, 4:11-17, 1988.

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

Sorting

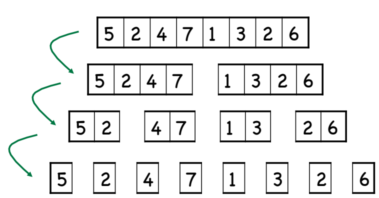
- Given: an unsorted array

5	2	4	7	1	3	2	6

- Goal: sort it

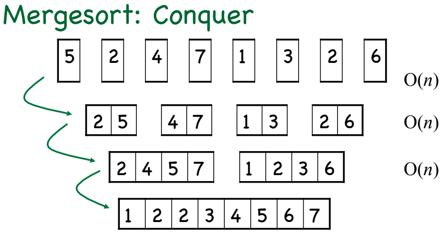
1	2	2	3	4	5	6	7

Mergesort: Divide



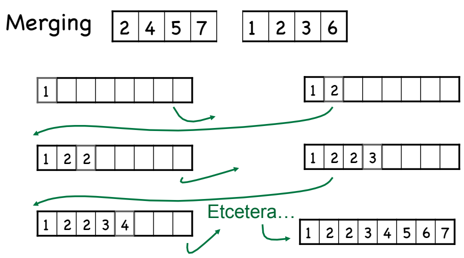
log*(n)* divisions to split an array of size *n* into single element arrays

Mergesort: Conquer



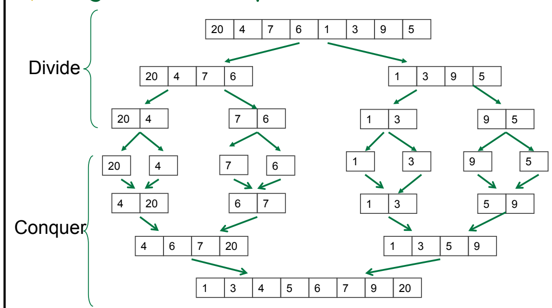
log*(n)* iterations, each iteration takes O*(n)* time
Total time: O*(n log n)*

Mergesort: Merge Step



- 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: Example



MergeSort Algorithm

<u>MergeSort(*c*)</u>

n ← size of array ***c***
if n = 1
  return ***c***
left ← list of first *n*/2 elements of ***c***
right ← list of last *n*–*n*/2 elements of ***c***
sortedLeft ← MergeSort(***left***)
sortedRight ← MergeSort(***right***)
sortedList ← Merge(***sortedLeft***,***sortedRight***)
return sortedList

MergeSort: Running Time

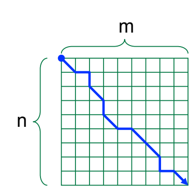
- in the *i*th iteration we do $O(n)$ work
- number of iterations is $O(\log n)$
- running time: $O(n \log n)$

Back to sequence alignment…

The Problem: Computing Alignment Path Requires Quadratic Memory

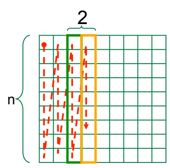
Alignment Path

- 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)

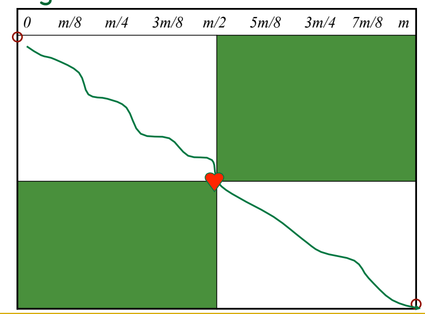


Computing Alignment Score using Linear Memory

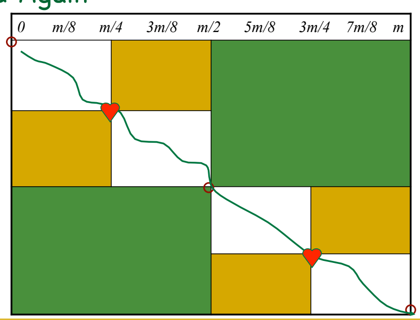
- Space complexity of computing just the score itself is $O(n)$
- Only need the previous column to calculate the current column



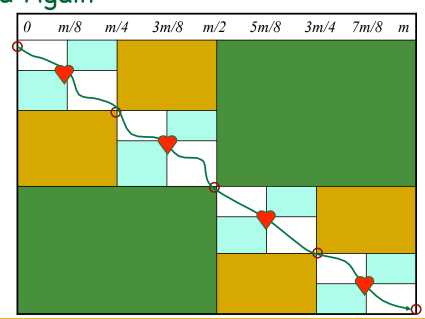
Finding the Middle Point



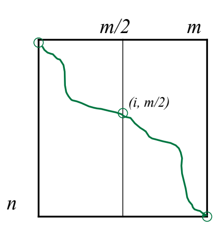
And Again



And Again



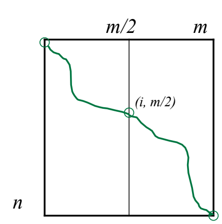
Crossing the Middle Line



Define:

score(i) – the score of the optimal path from (0,0) to (n,m) that passes through (i, m/2)

Crossing the Middle Line



(*mid*,*m*/2): the position where the optimal path crosses the middle column.

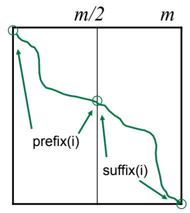
mid = $\text{argmin}_{0 \le i \le n}$ score*(i)*

Crossing the Middle Line

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

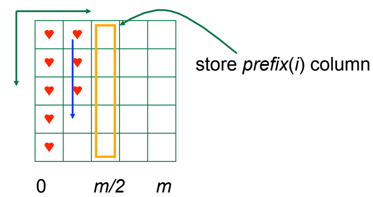
prefix*(i)*: score of the optimal alignment of a length m/2 prefix of y to a prefix of x (takes a path from (0,0) to (*i*,*m*/2))

suffix(i): score of the optimal alignment of a length m/2 suffix of y to a suffix of x (takes a path from (*i*,*m*/2) to *(n,m)*)



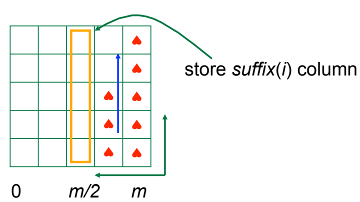
Computing prefix*(i)*

- prefix*(i)*: 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

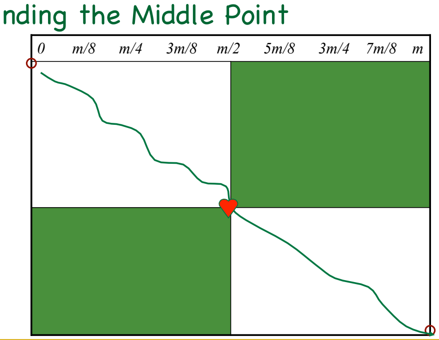


Computing suffix*(i)*

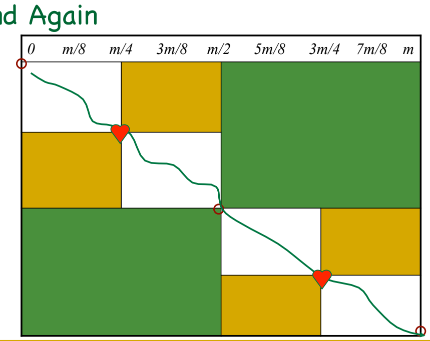
- *suffix(i)*: score of optimal alignment from (*i*,*m*/2) to *(n,m)*
- Can be computed by going in "reverse" from (*n*,*m*) to (*i*,*m*/2)



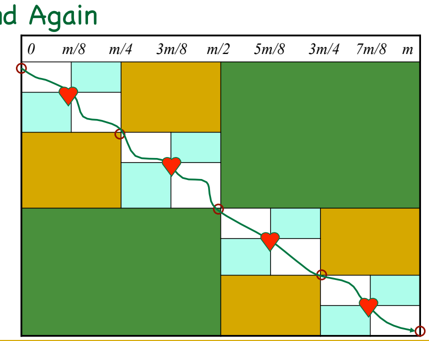
Finding the Middle Point



And Again



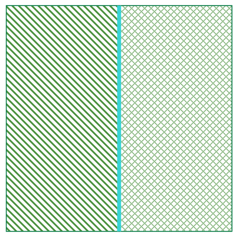
And Again



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

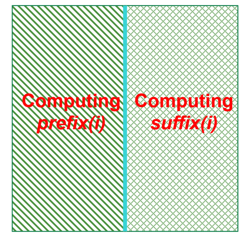
Area = $n \cdot m$



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

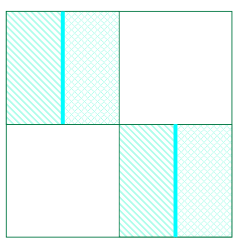
Area = $n \cdot m$



Time = Area: Second Pass

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

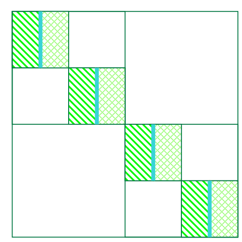
Area/2



Time = Area: Third Pass

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

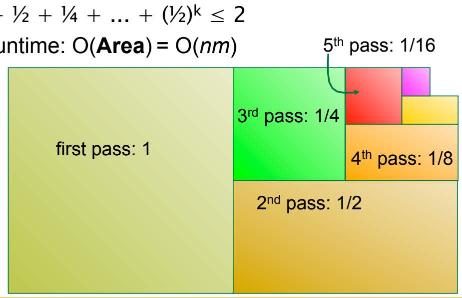
Area/4



Geometric Reduction At Each Iteration

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

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



Run Time Analysis

- Let T(m, n) = max running time of algorithm on strings of length m and n.
  - O(mn) time to compute prefix(•, m/2) and suffix(•, m/2) and find midpoint q.
  - T(q, m/2) + T(n – q, m/2) time for two recursive calls.
  - Choose constant c so that:

$$\begin{aligned} T(m, 2) &\le 2cm \\ T(2, n) &\le 2cn \\ T(m, n) &\le cmn + T(q, m/2) + T(n-q, m/2) \end{aligned}$$

- Claim: T(m, n) <= 2cmn (proof by induction)

Is it Possible to Align Sequences in Subquadratic Time?

- Dynamic programming takes $O(n^2)$ for various alignment methods
- Can we do better?
- Yes: *The Four-Russians Speedup* (works for LCS but not for general sequence alignment problem) $O(n^2/\log n)$