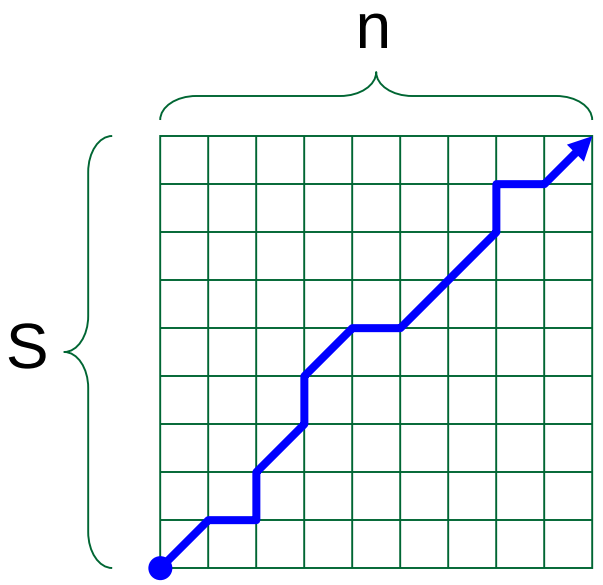
Computing Knapsack Requires Quadratic Memory

Solution Path

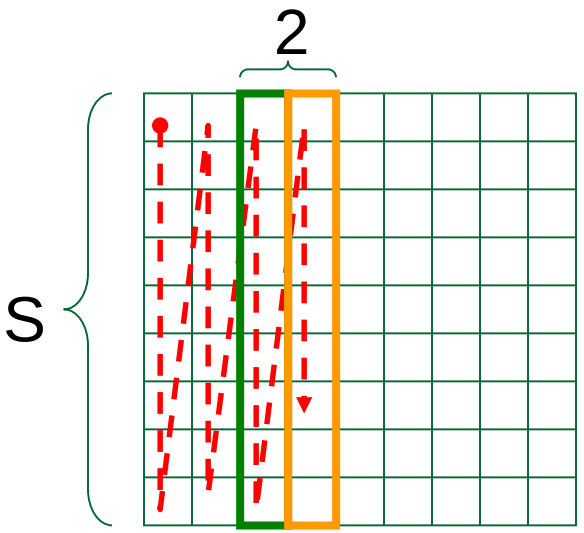
- Space complexity for computing objects used with *n* and size *S* is O(*nS*)
- We need to keep all backtracking references in memory to reconstruct the path (backtracking)



Computing Knapsack Score with Linear Memory

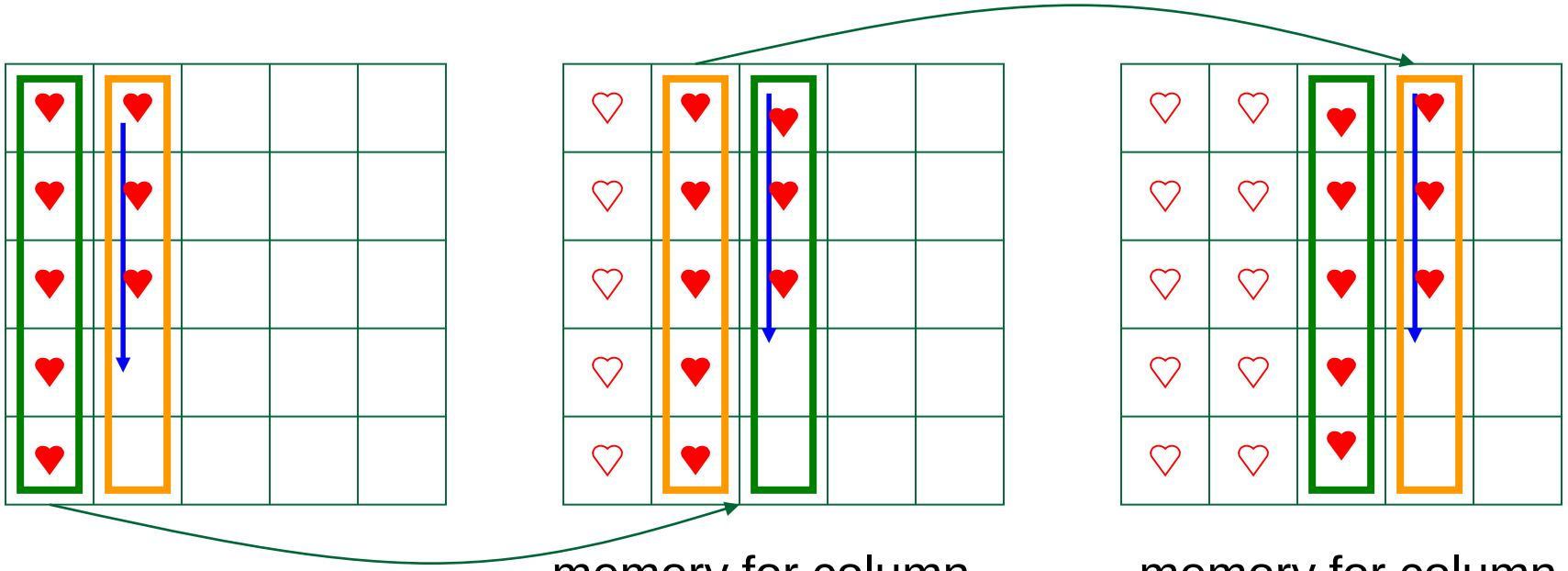
Knapsack Score

- 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



Computing Knapsack Score: Recycling Columns

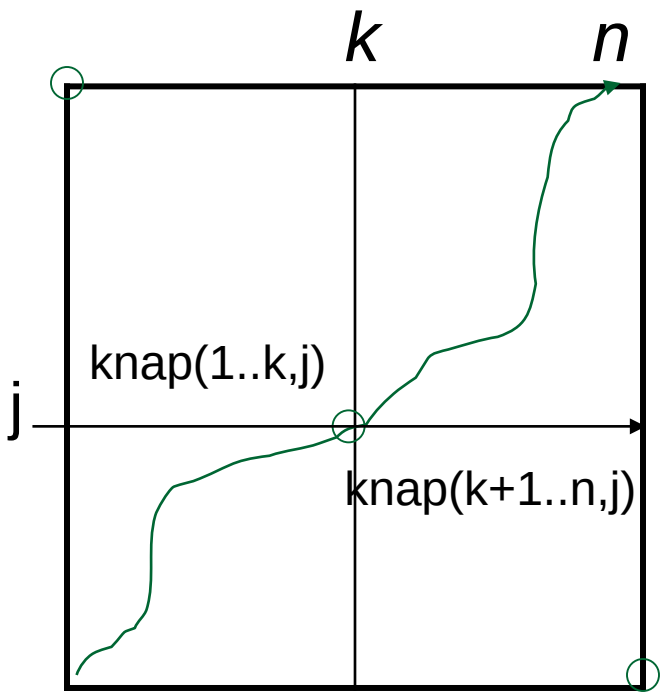
Only two columns of scores are saved at any given time



memory for column 1 is used to calculate column 3

memory for column 2 is used to calculate column 4

Crossing the Middle Line



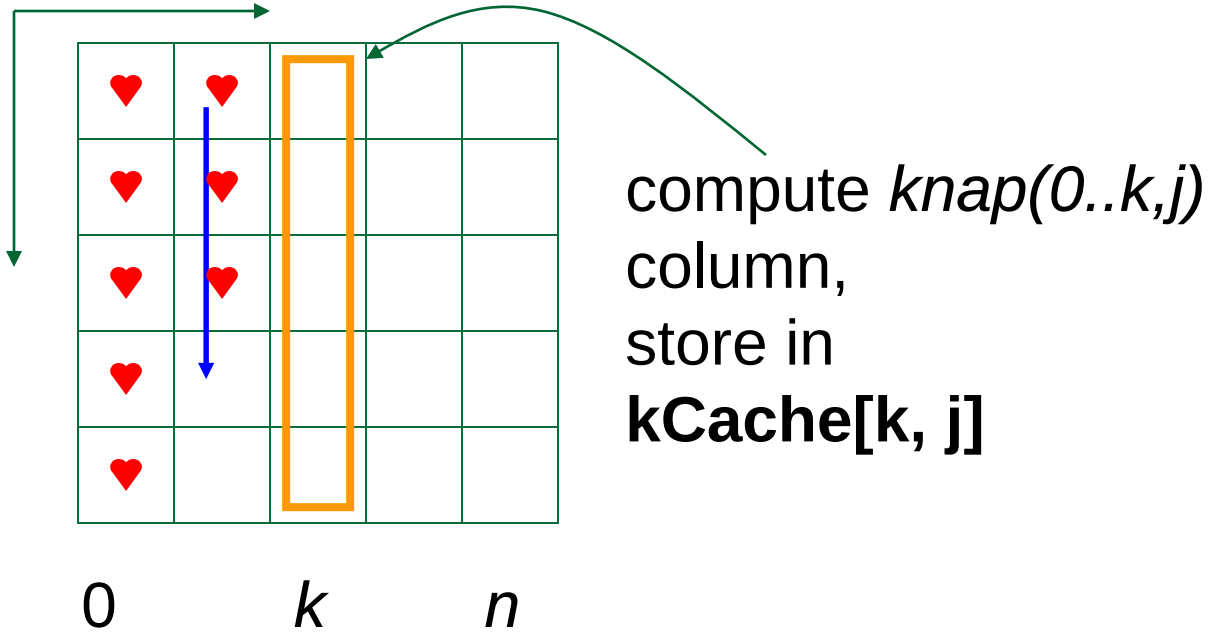
We want to calculate the greatest value paths from (0,0) to (*n*,*S*) that pass through (*k,j*) where *j* 0<=j<=S

Define *bestValue*(j)

as the path with the greatest value from (0,0) to (*n*,*S*) that passes through vertex (*k,j*)

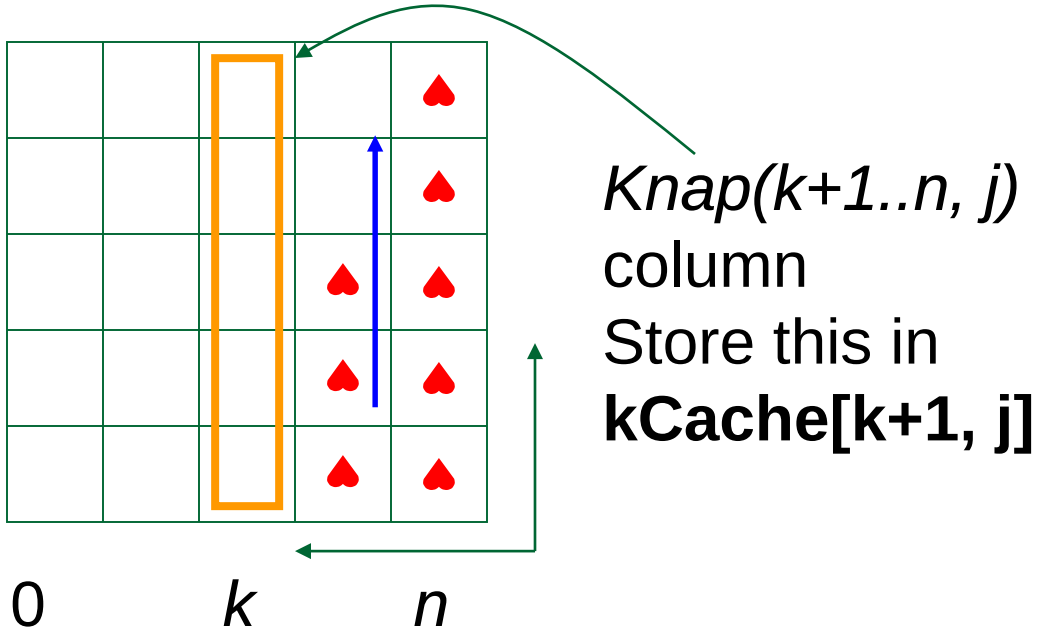
Computing knap(0..k,j)

- *knap(0..k,j)* is the highest value path from (0,0) to (*k,j*) for all 0<=j<=S
- Compute *knap(0..k,j)* by dynamic programming in the left half of the matrix



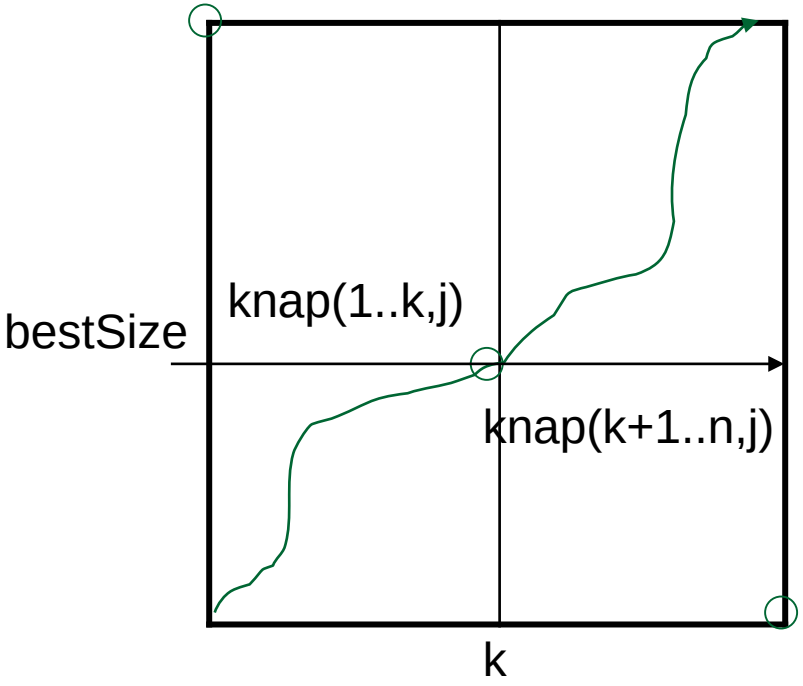
Computing knap(k+1..n,j)

- *knap(k+1..n, j)* is the highest value path from *(n, S) to (k+1, j)* to *for all j, 1<=j<=S*
- Compute *knap(k+1..n, j)* by dynamic programming in the right half of the “reversed” matrix



bestValue(j) = knap(0..k,j) + knap(k+1..n,j)

- For each j add *knap(0..k,j)* to *knap(k+1..n,j)* to get the *bestValue(j)*
- Find the j value where *bestValue(j)* is maximized to find the crossing point
- *bestValue = Max(bestValue(j))*
- *bestSize = argMax(bestValue(j)), 0<=j<=S*

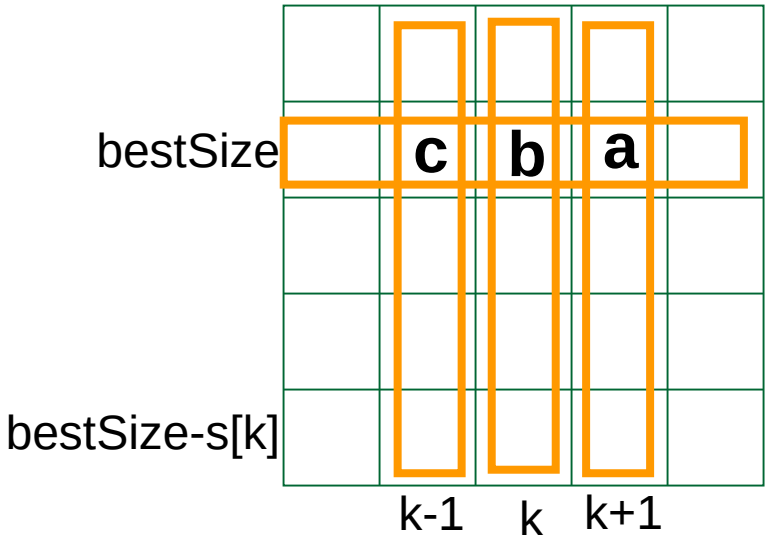


We have found the coordinate of the crossing point of the global solution at index *k: (k, bestSize)*

Note this will not necessary be the middle of S

Now we determine if object k is used for the best solution

- We have bestValue, bestSize and the results of the two DP scans
- Consider a close up of the cached results



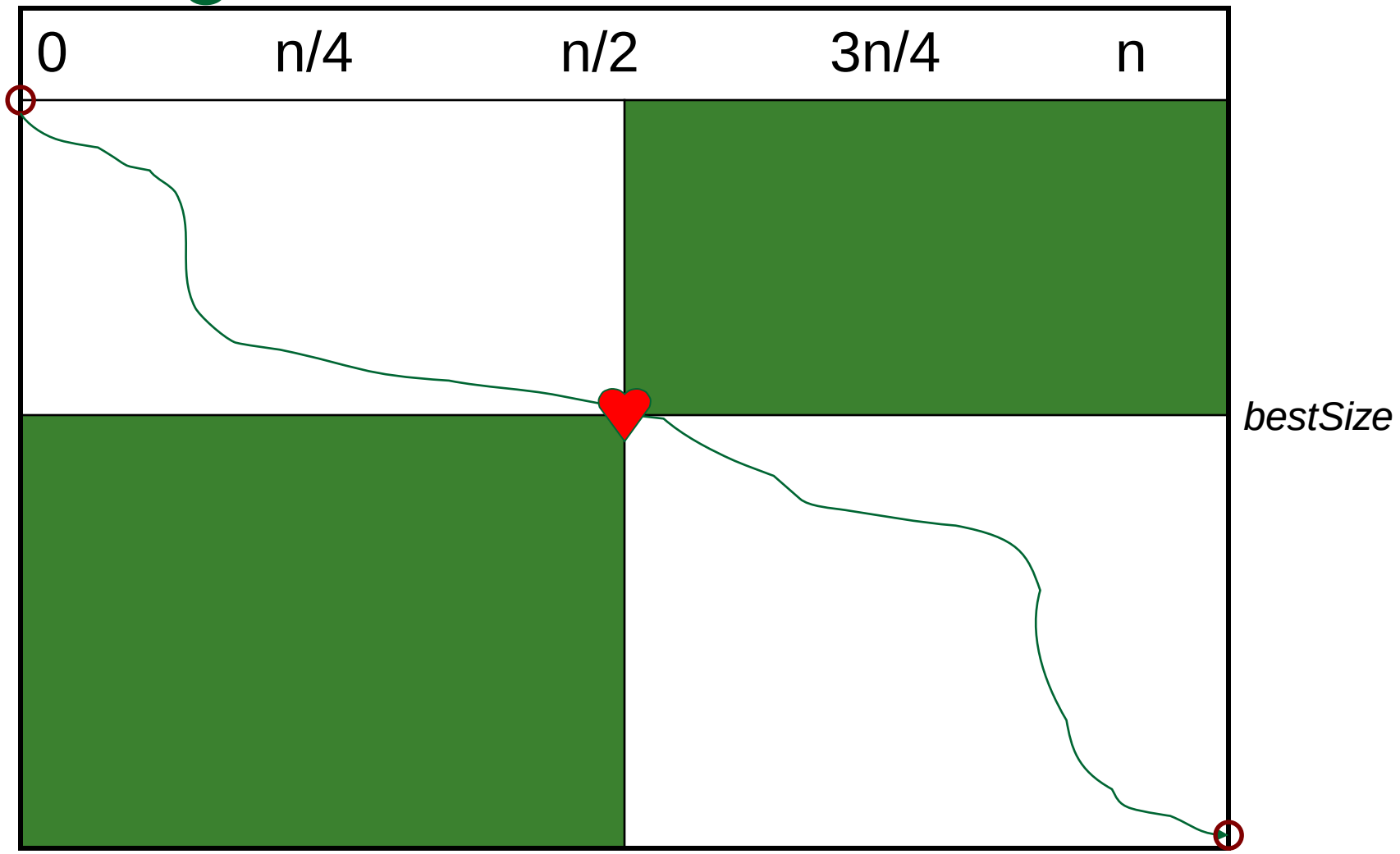
a kCache[k+1,bestSize]

b kCache[k,bestSize]

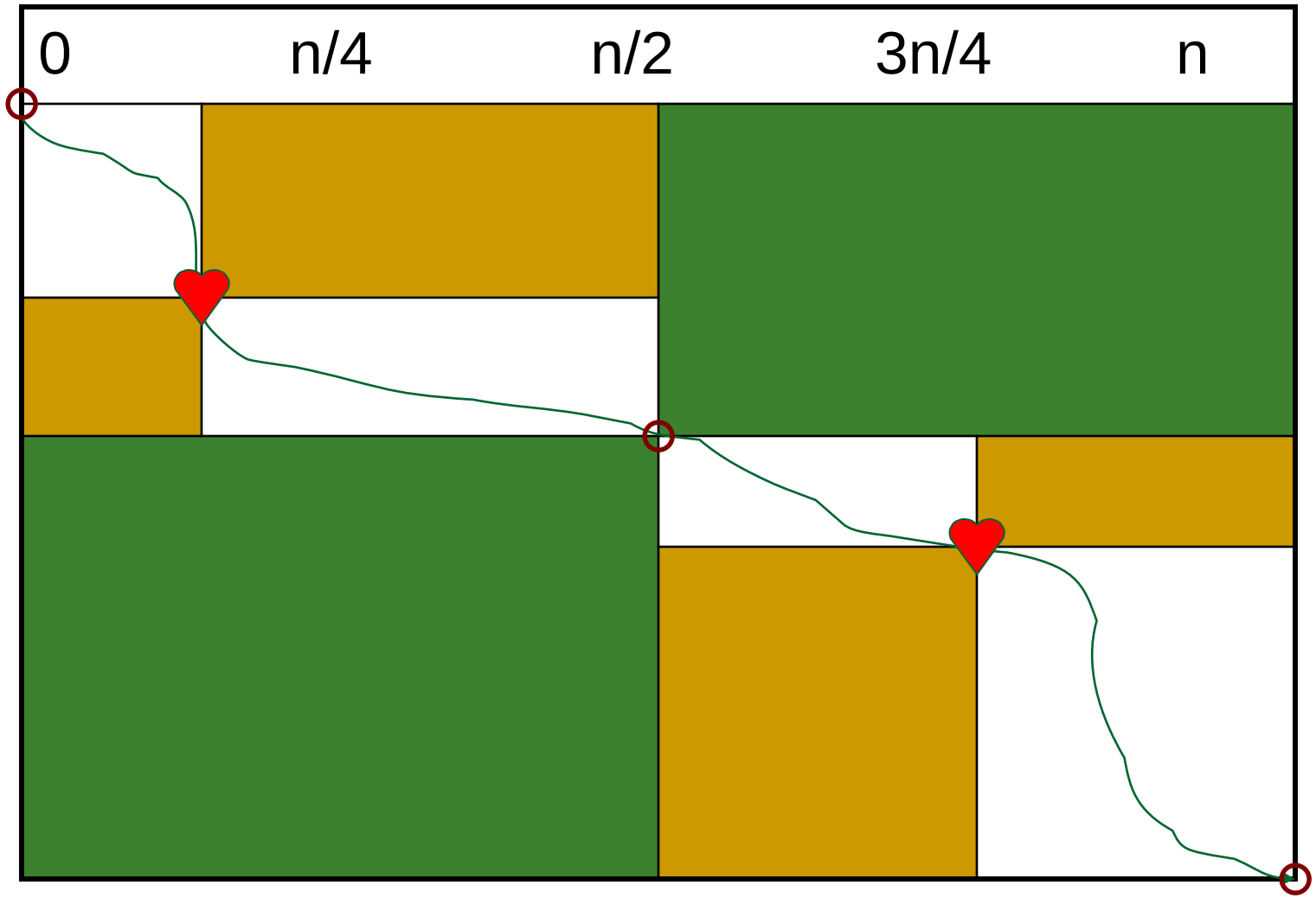
c kCache[k-1,bestSize]

Use object k is true when (**b**!=**c**) is true

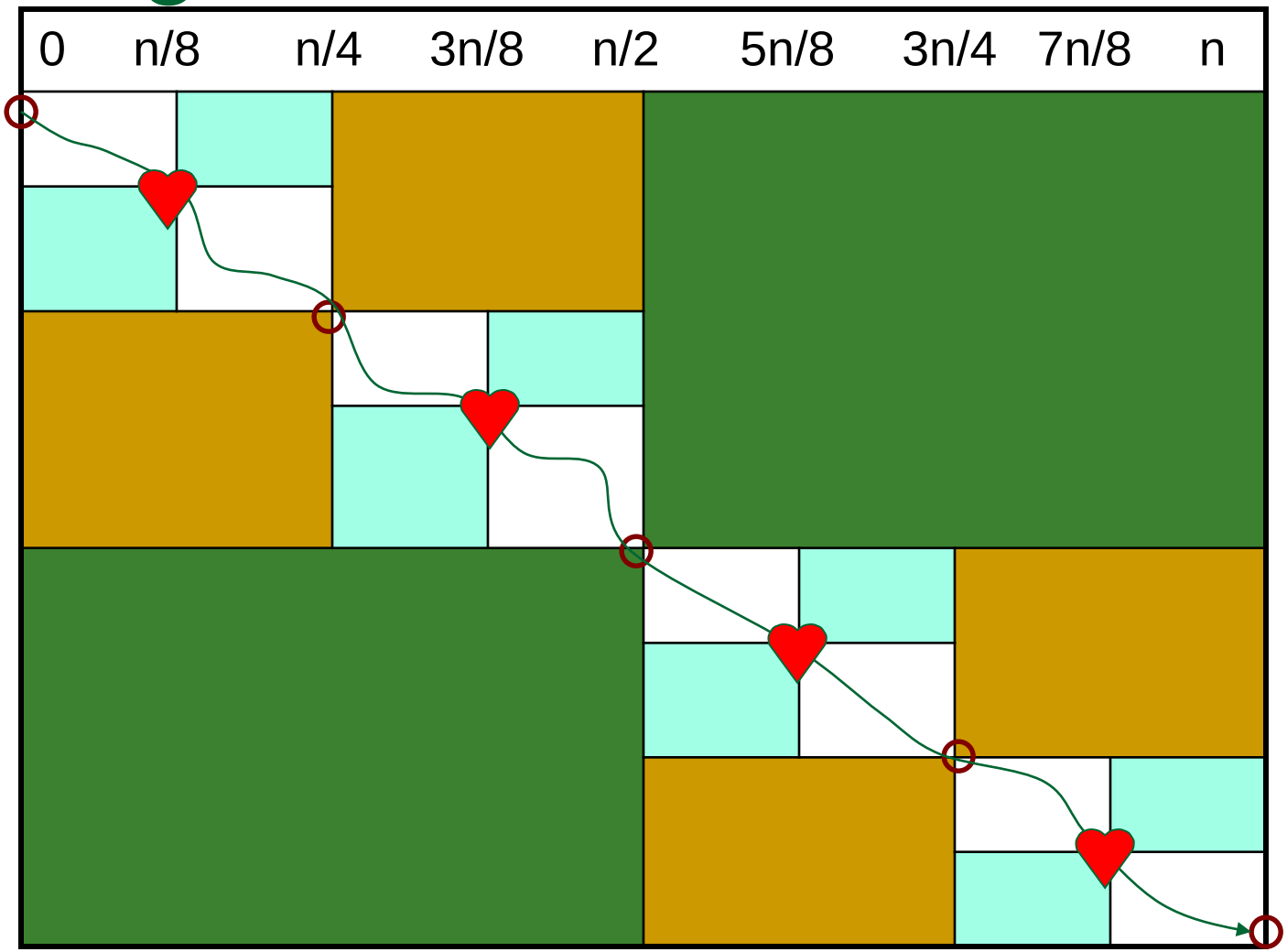
Finding the Middle Point



Finding the Middle Point again



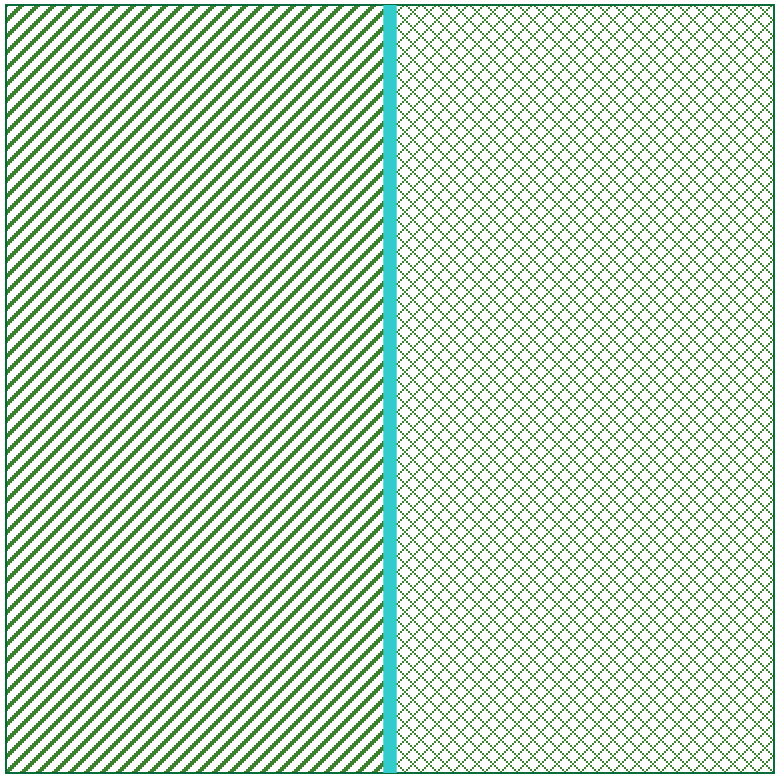
And Again



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

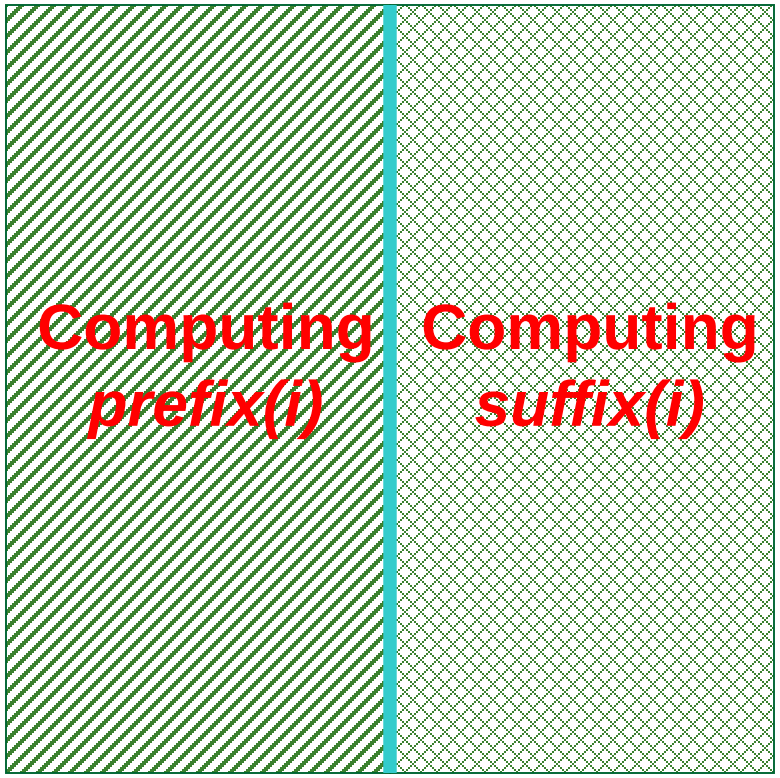
Area = $n \bullet m$



Time = Area: First Pass

- On first pass, the algorithm covers the entire area

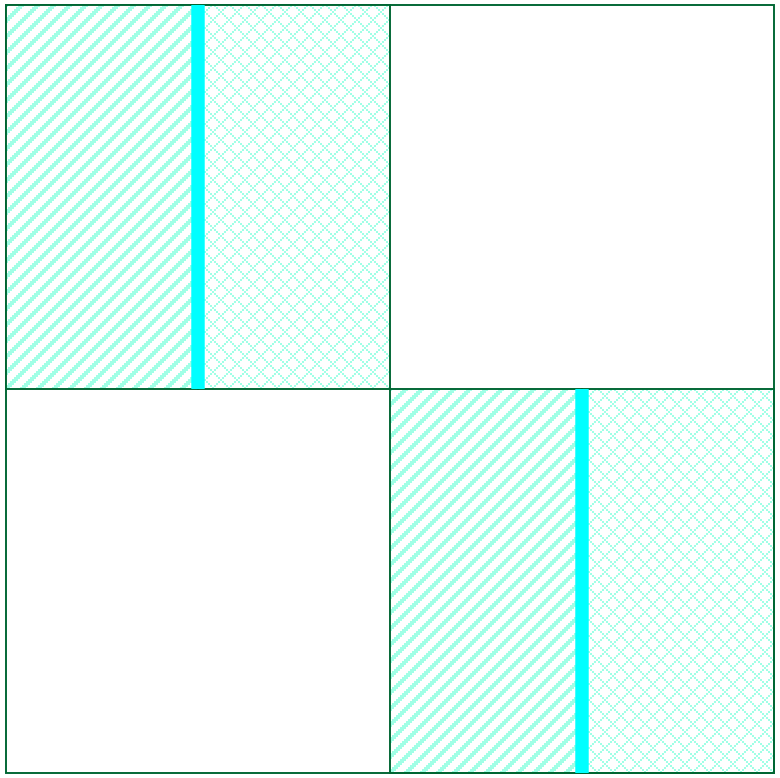
Area = $n \bullet m$



Time = Area: Second Pass

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

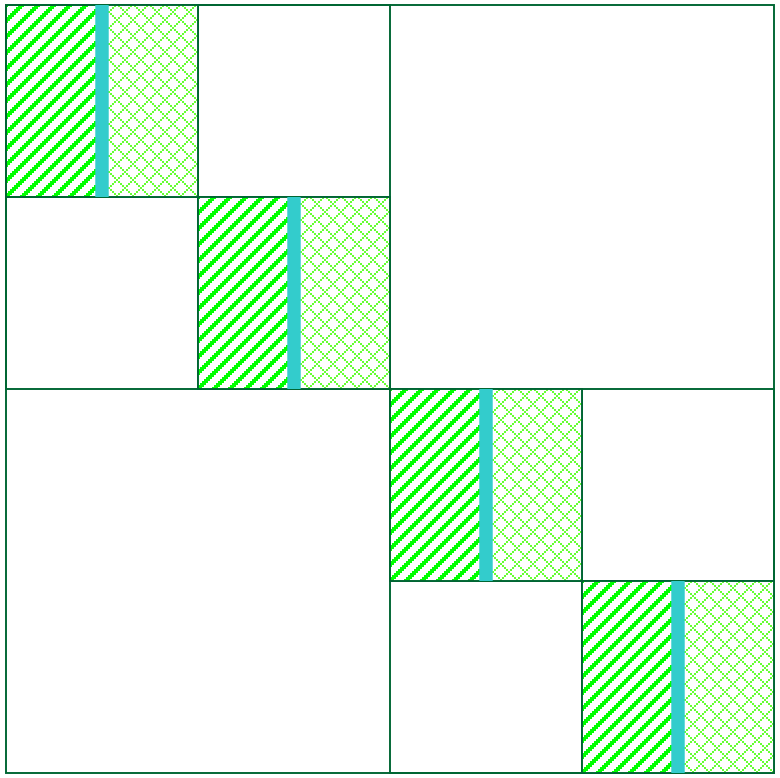
Area/2



Time = Area: Third Pass

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

Area/4



Geometric Reduction At Each Iteration

$1 + ½ + ¼ + \ldots + (½)^k \leq 2$

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