

1 Divide and Conquer

1.1 Idea

A Divide and Conquer algorithm consists of three phases:

1. Divide phase
Split the original problem into smaller instances of same problem (usually $\Theta(1)$ subproblems) of size $\leq \alpha n$ for $\alpha < 1$.
2. Conquer phase
Solve the subproblem recursively.
3. Combining phase
Combine solutions of subproblems to attain the solution to the original problem.

The key to applying this technique is to discover how a solution to the input problem can be composed from (or how to decomposed into) solution of subproblems of the same form. Problem decomposition involves studying the structure of the solution.

1.2 Example

Largest Empty Rectangle

Input: 2-D boolean array $A[1 : m, 1 : n]$ of 0s and 1s.

Output: Find a subarray $A[i : k, j : l]$ with only 0s of largest area.

Exhaustive search(first attempt): exhaustively search for all possible rectangles that contains only 0s and find out the largest. This takes $\Theta(n^3m^3)$ time to do, very inefficient.

A divide and conquer approach:

Cut A vertically into half at the middle column. Then the optimal solution falls into three cases:

- Case-1: strictly in the left half;
- Case-2: strictly in the right half;
- Case-3: spans the middle.

We can recursively solve case-1 and case-2. For case-3, the largest empty rectangle breaks into two pieces:

- Piece-a: left rectangle ends at mid-column and is optimal conditioned on starting and ending at its first and last rows.

- Piece-b: right rectangle starts at mid-column and is optimal conditioned on starting and ending at its first and last rows.

For a given choice of row i and k , we can find the optimal left piece-a (starting at i , ending at k) by taking a min of the column-left-run-length of 0s on rows $i \cdots k$ ending at the mid-column. Similar for the optimal right piece-b.

So we precompute:

$L[i, j] :=$ length of the longest run of 0s in row i that ends at (i, j) .

$R[i, j] :=$ length of the longest run of 0s in row i that starts at (i, j) .

The following pseudo code computes L .

```
Function PrecomputeLeftRuns( $A[1 : m, 1 : n]$ )
  for  $i := 1$  to  $m$ 
     $L[i, 0] := 0$ 
    for  $j := 1$  to  $n$ 
      if  $A[i, j] == 0$ 
         $L[i, j] := 1 + L[i, j - 1]$ 
      else
         $L[i, j] := 0$ 
```

The following pseudo code computes R .

```
Function PrecomputeLeftRuns( $A[1 : m, 1 : n]$ )
  for  $i := 1$  to  $m$ 
    if  $A[i, n] == 0$ 
       $R[i, n] = 1$ 
    else
       $R[i, n] = 0$ 
    for  $j := n - 1$  to  $1$ 
      if  $A[i, j] == 0$ 
         $R[i, j] := 1 + R[i, j + 1]$ 
      else
         $R[i, j] := 0$ 
```

The following pseudo code computes case-3.

```
Function SpanningRect( $A, m, lo, hi, L, R$ )
   $mid := \lfloor \frac{hi+lo}{2} \rfloor$ 
   $S := 0$  // holds the area of the largest empty rectangle spanning the mid column.
  for  $i := 1$  to  $m$  // row start
     $minL = \infty$ 
     $minR = \infty$ 
    for  $k := 1$  to  $m$  // row end
       $minL = \min(minL, L[k, mid])$ 
       $minR = \min(minR, R[k, mid])$ 
       $S = \max(S, (k - i + 1) \times (minR + minL))$ 
  return  $S$ 
```

The final pseudo code is:

```
Function LargestEmptyRect( $A, m, lo, hi, L, R$ )  
  if  $lo < hi$   
     $S := 0$   
     $mid := \lfloor \frac{hi+lo}{2} \rfloor$   
     $S := \max(S, \text{LargestEmptyRect}(A, m, lo, mid, L, R))$   
     $S := \max(S, \text{LargestEmptyRect}(A, m, mid + 1, hi, L, R))$   
     $S := \max(S, \text{SpanningRect}(A, m, lo, hi, L, R))$   
    return  $S$   
  else //  $lo == hi$   
    return the length of longest run of 0s in that column  $A[1 : m, lo]$ 
```