CS 381 – FALL 2019

Week 7.3, Friday, Oct 4

No class on Monday, October 7th (October Break)
No PSOs next week (October Break)
Thursday/Friday PSOs are now ``office hours"
Homework 4 Released: Due October 14th @ 11:59PM (Gradescope)

Basic Matrix Multiplication

- **Inputs:** matrices A (m x n), B (n x r), C (r x k)
- **Observation 2:** Matrix Multiplication is associative
  - i.e. ABC = (AB)C=A(BC)

  - **Option 1:** Compute D=(AB) (m x r) then DC
    - Total Multiplications: mnr + mrk
  - **Option 2:** Compute D =(BC) (n x k) then AD
    - Total Multiplications: nrk + mnk
- Suppose m=100, n=100, r=500 and k=5
  - **Option 1** → 5.25 million integer multiplications
  - **Option 2** → 0.3 million integer multiplications

Given matrices $A_1, A_2, \ldots, A_n$, place parenthesis minimizing the total number of multiplications.

$$((A_1 A_2) A_3)\ (A_4 A_5)$$

1. An optimal solution to matrix chain contains optimal subsolutions.
2. Fill an n×n table **m** where m[i,j] = minimum number of multiplications for generating $A_i \times A_{i+1} \times \ldots \times A_j$

$$\mathbf{m[i,j] = \min \{m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j\}}$$

with $i<j$ and $i \leq k<j$

$$(A_i \times A_{i+1} \times \ldots A_k) \times (A_{k+1} \times \ldots \times A_{j-1} \times A_j)$$

Base Cases: m[i,i]=0

→ **m[i,i+1]= $p_{i-1} \times p_i \times p_{i+1}$**

m[i,j]= min {m[i,k] + m[k+1,j] + $p_{i-1} \times p_k \times p_j$}
with i<j and i≤k<j

The *m*-table computed for $n = 6$ matrices $A_1, A_2, A_3, A_4, A_5, A_6$ and their dimensions 30, 35, 15, 5, 10, 20, 25.

$$((A_1(A_2\,A_3))\,((A_4\,A_5)\,A_6))$$

3. Compute the entries in array m

Need to make sure all values needed have been computed

- Compute values diagonal by diagonal
  - first diagonal has n zeroes
  - second diagonal has n-1 elements, etc
- Last entry computed is m(1,n)
  - It represents the minimum number of multiplications.

MATRIX-CHAIN-ORDER(p)

```
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

$A_i \times A_{i+1} \times \dots A_k \times A_{k+1} \times \dots \times A_{j-1} \times A_j$

4. *Construct an optimal solution*

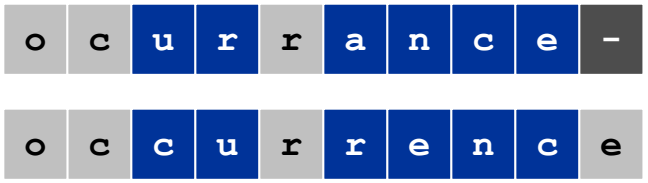
- Keep track of indices inducing the recorded subsolutions
  - Do without increasing time and space (array s in code).
- Final solution is generally created by tracing back
  - s[i,j]=k records the split point giving the optimum solution.

$O(n^3)$ time and $O(n^2)$ space

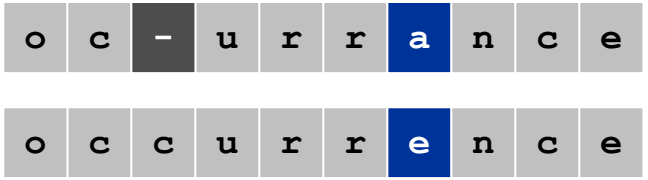
Note: there exists a more efficient (not DP) algorithm

6.6 Sequence Alignment

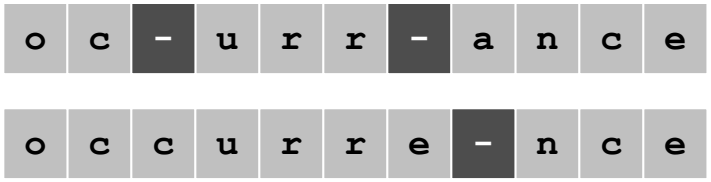
String Similarity



6 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

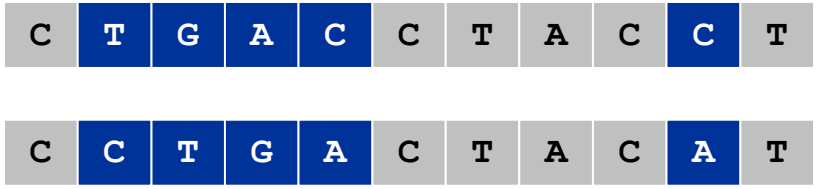
How similar are two strings?

- **ocurrance**
- **occurrence**

Edit Distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

$2\delta + \alpha_{CA}$

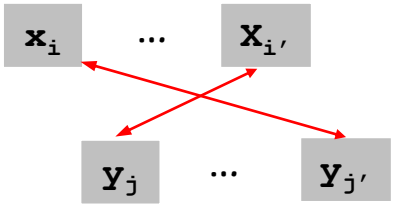
Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

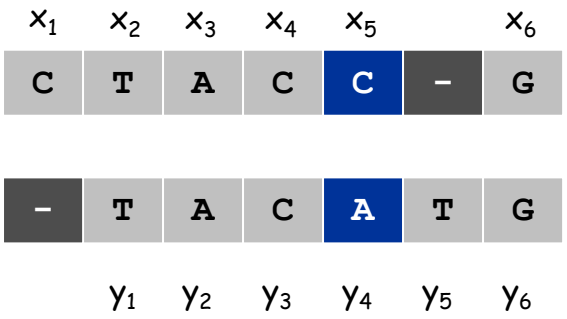


Def. The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.

Sol: $M = x_2$-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.



Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_j$.

- Case 1: OPT matches x_i-y_j.
  - pay mismatch for x_i-y_j + min cost of aligning two strings $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
  - pay gap for x_i and min cost of aligning $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_j$
- Case 2b: OPT leaves y_j unmatched.
  - pay gap for y_j and min cost of aligning $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100{,}000$.

10 billions ops OK, but 10GB array?

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.

- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Sequence Alignment: Linear Space

Recall. $OPT(i, j)$ = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Def. $OPT_{rev}(i, j)$ = min cost of aligning strings $x_{i+1}\, x_{i+2} \ldots x_n$ and $y_{j+1}\, y_{j+2} \ldots y_n$.

Symmetry: Can develop recurrence for $OPT_{rev}(i, j)$

- Computing $OPT(i, j)$ requires $OPT(i,j-1)$ and $OPT(i-1,j)$
- Computing $OPT_{rev}(i, j)$ requires $OPT_{rev}(i+1, j)$ and $OPT_{rev}(i, j+1)$

Observation: For any $j < m$ we have

$$OPT(m, n) = \min_{0 \le i \le m} \{OPT(i, j) + OPT_{rev}(i, j)\}$$

Cost to aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$

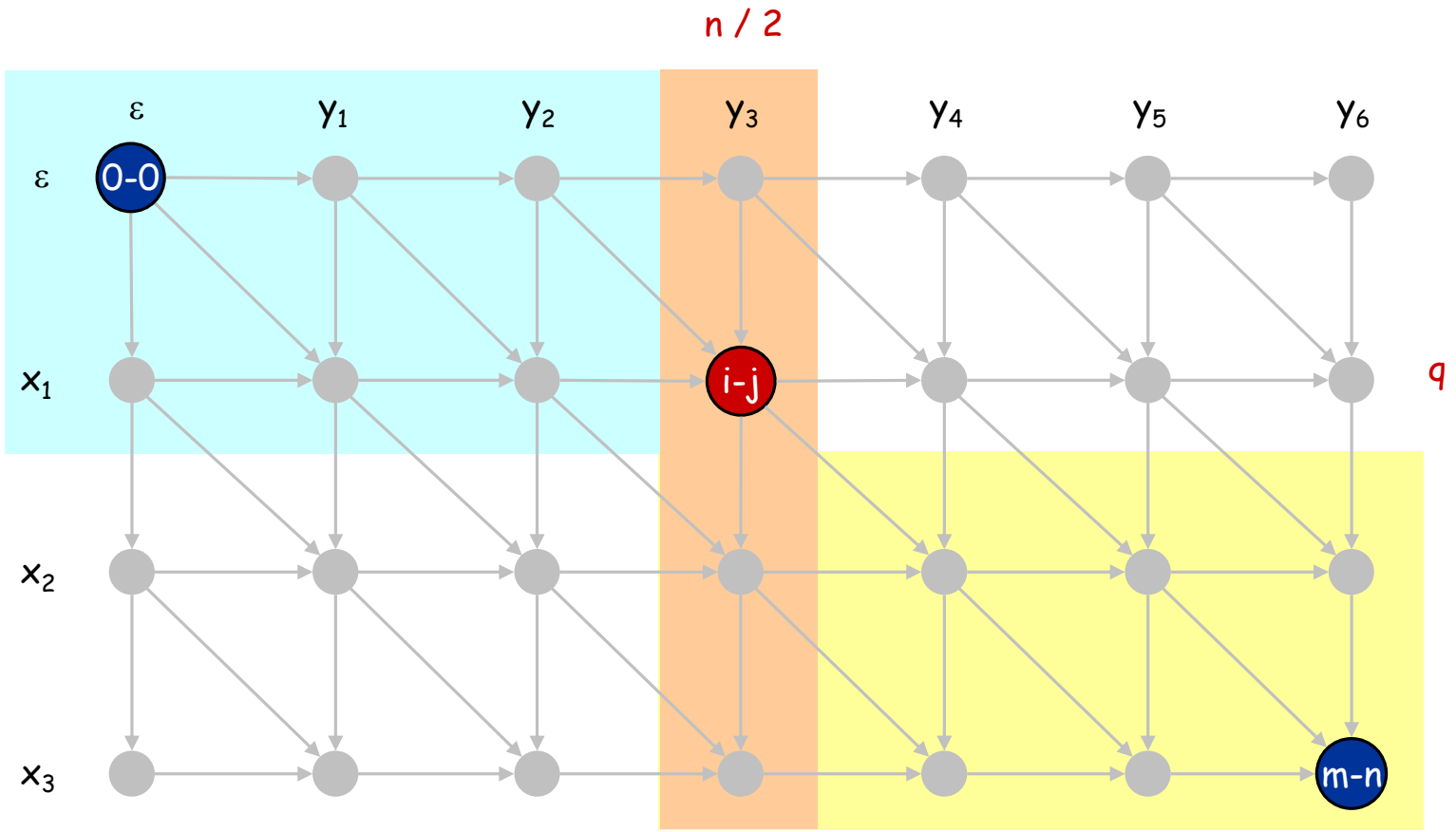
Cost to align remainder $x_{i+1}\, x_{i+2} \ldots x_n$ and $y_{j+1}\, y_{j+2} \ldots y_n$

Sequence Alignment: Linear Space

Divide: find index q that minimizes $OPT(q, n/2) + OPT_{rev}(q, n/2)$ using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.

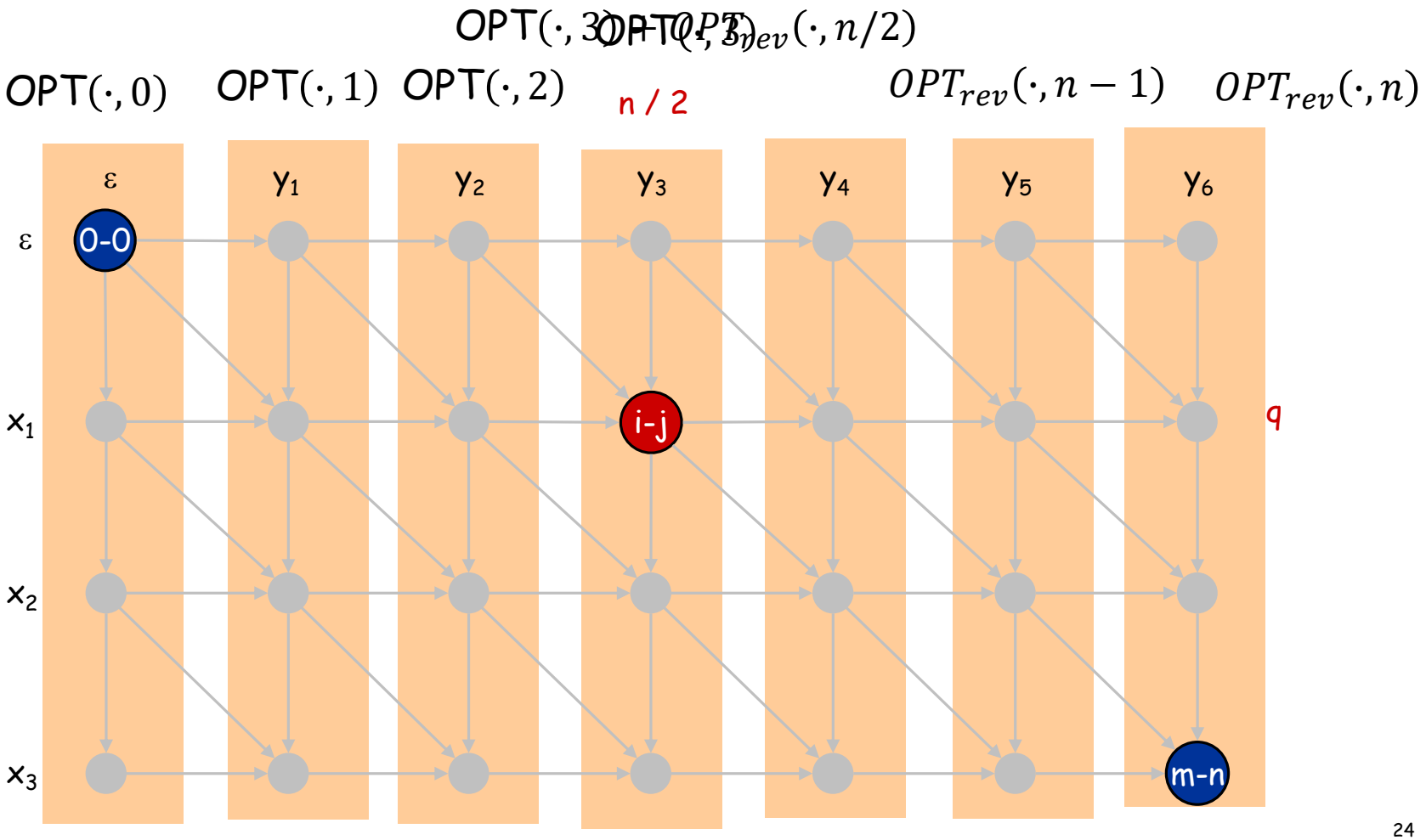


Sequence Alignment: Linear Space

Divide: find index q that minimizes OPT (q, n/2) + OPT_{rev} (q, n/2) using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



Sequence Alignment: Linear Space

Recall. OPT(i, j) = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Def. $OPT_{rev}(i, j)$ = min cost of aligning strings $x_{i+1}\, x_{i+2} \ldots x_n$ and $y_{j+1}\, y_{j+2} \ldots y_n$.

Symmetry: Can develop recurrence for $OPT_{rev}(i, j)$

- Computing OPT(i, j) requires OPT(i,j-1) and OPT(i-1,j)
- Computing $OPT_{rev}(i, j)$ requires $OPT_{rev}(i+1, j)$ and $OPT_{rev}(i, j+1)$

Observation: For any j< m we have

$$OPT(m, n) = \min_{0 \le i \le m} \{OPT(i, j) + OPT_{rev}(i, j)\}$$

Divide: Set j = n/2 and compute rows $OPT(\cdot, j)$ and $OPT_{rev}(\cdot\, , j)$

- Find q minimizing $OPT(q, j) + OPT_{rev}(q, j)$, free memory and recurse
- Time: O(mn)
- Space: O(m+n)
  - Why? Only require row $OPT(\cdot, j-1)$ to compute next row $OPT(\cdot, j)$

Sequence Alignment: Linear Space

Recall. OPT(i, j) = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Def. $OPT_{rev}(i, j)$ = min cost of aligning strings $x_{i+1}\, x_{i+2} \ldots x_n$ and $y_{j+1}\, y_{j+2} \ldots y_n$.

Symmetry: Can develop recurrence for $OPT_{rev}(i, j)$

- Computing OPT(i, j) requires OPT(i,j-1) and OPT(i-1,j)
- Computing $OPT_{rev}(i, j)$ requires $OPT_{rev}(i+1, j)$ and $OPT_{rev}(i, j+1)$

Observation: For any j< m we have

$$OPT(m, n) = \min_{0 \le i \le m} \{OPT(i, j) + OPT_{rev}(i, j)\}$$

Conquer/Merge: Set j = n/2 and compute rows $OPT(\cdot, j)$ and $OPT_{rev}(\cdot\, , j)$

- Find q minimizing $OPT(q, j) + OPT_{rev}(q, j)$, free memory and recurse
- **Sub-problem 1:** Align $x_1\, x_2 \ldots x_q$ and $y_1\, y_2 \ldots y_{n/2}$
- **Sub-problem 2:** Align $x_{i+1}\, x_{i+2} \ldots x_n$ and $y_{n/2+1}\, y_{n/2+2} \ldots y_n$

Sequence Alignment: Running Time Analysis Warmup

Theorem. Let T(m, n) = max running time of algorithm on strings of length at most m and n. T(m, n) = O(mn log n).

$$T(m,n) \leq 2T(m, n/2) + O(mn) \Rightarrow T(m,n) = O(mn \log n)$$

Remark. Analysis is not tight because two sub-problems are of size (q, n/2) and (m - q, n/2). In next slide, we save log n factor.

Sequence Alignment: Running Time Analysis

Theorem. Let T(m, n) = max running time of algorithm on strings of length m and n. T(m, n) = O(mn).

Pf. (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- Choose constant c so that:

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

- Base cases: m = 2 or n = 2.
- Inductive hypothesis: T(m, n) ≤ 2cmn.

$$\begin{aligned} T(m,n) &\leq T(q,n/2) + T(m-q,n/2) + cmn \\ &\leq 2cqn/2 + 2c(m-q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$