DIVIDE AND CONQUER PARADIGM

ANGEL NOÉ MARTÍNEZ GONZÁLEZ

June 26, 2015

THE PARADIGM I

Solves a problem **recursively** applying at each level of the recursion

- **DIVIDE** the problem into smaller sub-problems.
- **CONQUER** via recursive calls.
- **COMBINE** solutions of sub-problems into one for the original problem.

Also consider a **base case** when the sub-problems become small enough.

THE PARADIGM II

MERGE SORT: RETAKEN

MERGE(A, p, q, r)

MERGE-SORT(A, p, r)

1: $B = 1^{st}$ part of array.
2: $C = 2^{nd}$ part of array.
3: $i = 1, j = 1$
1: **if** $p < r$ **then**
2:    $q = \lfloor (p + r)/2 \rfloor$
4: **for** $k = 1$ to n **do**
3:    MERGE-SORT(A, p, q)
5:    **if** $B[i] < C[j]$ **then**
4:    MERGE-SORT(A, q, r)
6:      $A[k] = B[i + +]$
5:    MERGE(A, p, q, r)
7:    **else**
6: **end if**
8:      $A[k] = C[j + +]$
9:    **end if**
10: **end for**

THE PARADIGM III

INSIGHTS

- Sub-problems sizes can be any! e.g. $1/2$, $1/3$, etc.
- Base case is often too naive.
- Generally, in the third step relies the good performance.
- Trying to make third step simple is the best way to tackle the problem.

RECURRENCES I

A RECURRENCE

- ► provide a natural way to describe the running time of divide-and-conquer algorithms.
- ► describes $T(n)$ in terms of the running time of recursive calls.
- ► consider a base case for which

$$T(n) \leq a$$

- ► consider the larger imputs as

$$T(n) \leq f(n)$$

RECURRENCES II

COMPUTING RECURRENCES: MERGE SORT

- Base case: sort two numbers. Swap positions in the worst case.

$$T(1) = 1$$

- Larger inputs: two sub-problems of size $n/2$ plus one call to merge with size n, i.e.

$$T(n) = 2T(n/2) + \Theta(n)$$

RECURRENCES III

leading to

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

help us to get the running time

$$T(n) = \Theta(n \log_2 n)$$

and for the properties of asymptotic notation $T(n) = O(n \log n)$

EXAMPLE: COUNTING INVERSIONS I

MOTIVATION

- List of ranked objects $A = [a_1, a_2, a_3, ..., a_n]$ and a reference list B.
- We want to figure out how similar these lists are.
- Application Collaborative Filtering.

Let B be the sorted list of A, then what we seek is the number of inversions in list A, i.e. the number of pairs (i, j) such that
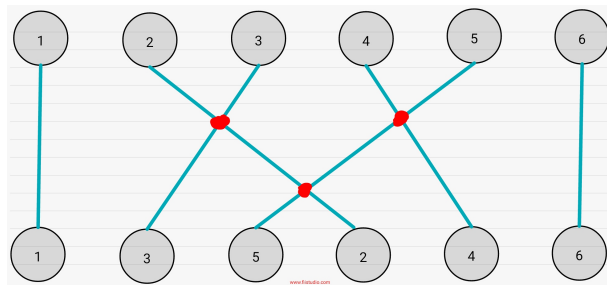
$$i < j \text{ and } A[i] > A[j]$$

EXAMPLE: COUNTING INVERSIONS II

EXAMPLE: $A = [1, 3, 5, 2, 4, 6]$

Inversions:

$(3, 2), (5, 2), (5, 4)$

In general the upper bound of number of inversions is $n(n-1)/2$.

EXAMPLE: COUNTING INVERSIONS III

A brute force algorithm

BRUTE-FORCE(A, n)

1: $count = 0$
2: **for** $i = 1, ..., n$ **do**
3:    **for** $j = i + 1, ..., n$ **do**
4:       **if** $A[i] > A[j]$ **then**
5:          $count = count + 1$
6:       **end if**
7:    **end for**
8: **end for**
9: **return** count

It is correct but

$$T(n) = O(n^2)$$

We can do better!

EXAMPLE: COUNTING INVERSIONS IV

The key idea is to use divide and conquer. Let be (i, j) an inversion with $i < j$

1. Let's call it a left inversion if $i, j \leq n/2$.
2. Let's call it a right inversion if $i, j > n/2$.
3. Let's call it a split inversion if $i \leq n/2 < j$

compute 1 and 2 recursively and implement other routine for 3.

EXAMPLE: COUNTING INVERSIONS V

COUNT-INVERSIONS(A, n)

1: **if** $n = 1$ **then**
2:     **return** 0
3: **end if**
4: x=COUNT-INVERSIONS(first half of A, $n/2$)
5: y=COUNT-INVERSIONS(second half of A, $n/2$)
6: z=COUNT-SPLIT-INVERSIONS(A, $n/2$)
7: **return** $x + y + z$

The goal is to implement efficiently COUNT-SPLIT-INVERSIONS for good performance.

EXAMPLE: COUNTING INVERSIONS VI

The key idea to implement COUNT-SPLIT-INVERSIONS is to consider sorting, more precisely the merge sub-routine of merge-sort

$$\text{MERGE}(A, p, q, r)$$

- ▶ B = left part of array.
- ▶ C = second part of array.
- ▶ A = output array.

1: $i = 1, j = 1$
2: **for** $k = 1$ to n **do**
3:    **if** $B[i] < C[j]$ **then**
4:       $A[k] = B[i + +]$
5:    **else**
6:       $A[k] = C[j + +]$
7:    **end if**
8: **end for**

EXAMPLE: COUNTING INVERSIONS VII
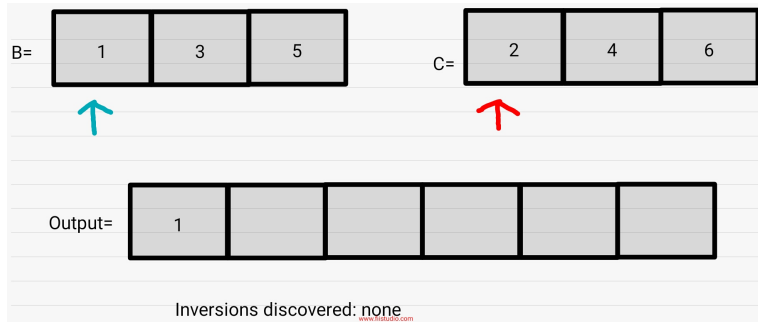
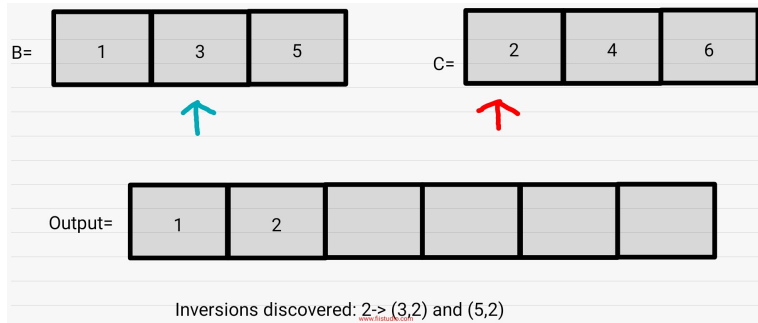Now suppose *A* has no split inversions, then following properties holds

- ▸ All elements in *B* are less than the elements in *C*.
- ▸ All elements in *B* are copied back to *A* before the elements in *C*.

The second half has the clue to discover the inversions!
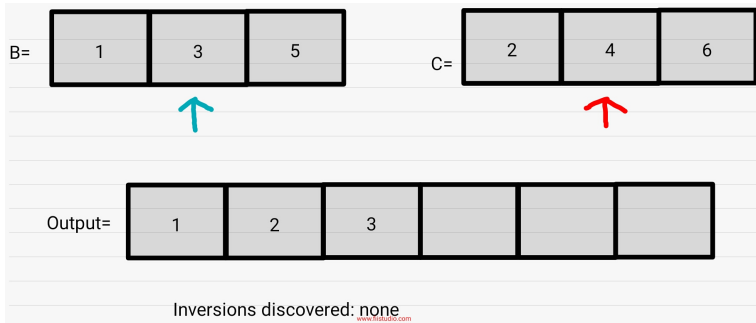
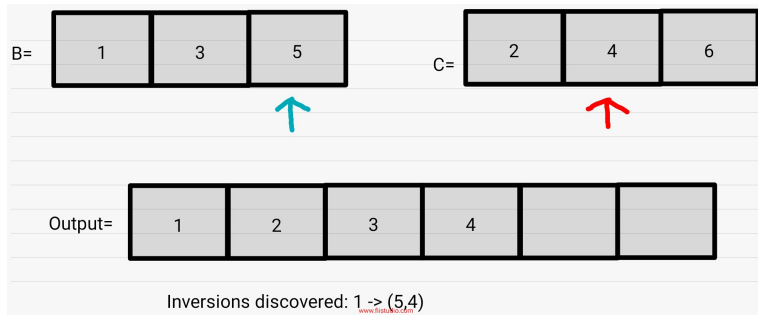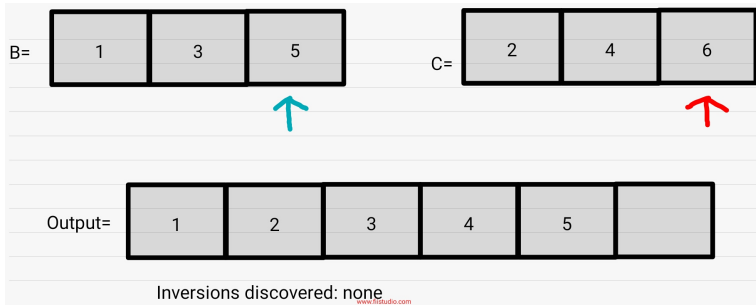# EXAMPLE: COUNTING INVERSIONS VIII

EXAMPLE: COUNTING INVERSIONS IX

Inversions discovered: 2-> (3,2) and (5,2)

EXAMPLE: COUNTING INVERSIONS X

B= | 1 | 3 | 5 | C= | 2 | 4 | 6 |

Output= | 1 | 2 | 3 | | | |

Inversions discovered: none

EXAMPLE: COUNTING INVERSIONS XI

Inversions discovered: 1 -> (5,4)

EXAMPLE: COUNTING INVERSIONS XII

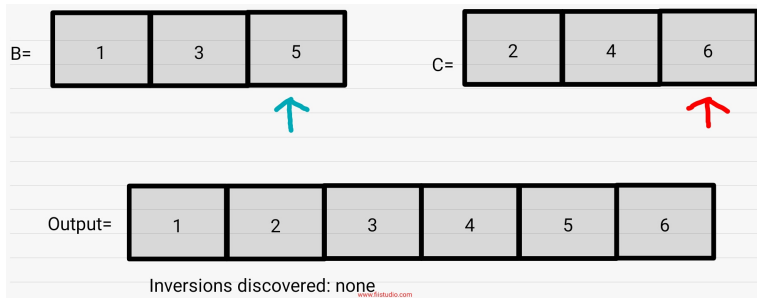# EXAMPLE: COUNTING INVERSIONS XIII

EXAMPLE: COUNTING INVERSIONS XIV

The split inversions involving an element $y_i \in C$ is the number of remaining positions from i in B when it is copied to the output array D.

Inversions discovered: 2-> (3,2) and (5,2)

EXAMPLE: COUNTING INVERSIONS XV

THE ALGORITHM

COUNT-INVERSIONS(A, n)

1: **if** $n = 1$ **then**
2:     **return** 0
3: **end if**
4: B, x=COUNT-INVERSIONS(first half of A, $n/2$)
5: C, y=COUNT-INVERSIONS(second half of$A$, $n/2$)
6: D, z=COUNT-SPLIT-INVERSIONS(B, C, $n/2$)
7: **return** $x + y + z$, D

EXAMPLE: COUNTING INVERSIONS XVI

COUNT-SPLIT-INVERSIONS(B, C, n)

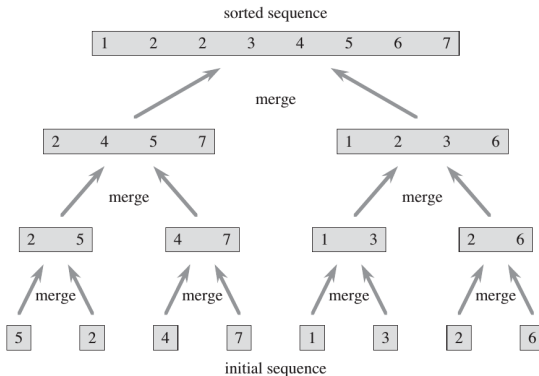1: $i = 1, j = 1, inv = 0$
2: **for** $k = 1$ to n **do**
3:     **if** $B[i] < C[j]$ **then**
4:         $D[k] = B[i + +]$
5:     **else**
6:         $inv+ = n/2 - i$
7:         $D[k] = C[j + +]$
8:     **end if**
9: **end for**
10: **return** inv, D

The recurrence is

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

What is the running time?

SOLVING RECURRENCES : RECURSION TREE

At each level $j = 0, 1, ..., log_2(n)$, there are 2^j subproblems of size $n/2^j$.

SOLVING RECURRENCES : THE MASTER METHOD I

- ▶ Also known as the Master Theorem.
- ▶ Black box for solving recurrences.
- ▶ Take as input few parameters to get the solution.
- ▶ Assumption all sub-problems have the same size.
- ▶ We will only consider the method that yields upper bounds, i.e. $O()$.

SOLVING RECURRENCES : THE MASTER METHOD II

RECURRENCE FORMAT

- Base case: $T(n) \leq \alpha$ for sufficiently small n.
- For larger n

$$T(n) \leq aT(n/b) + O(n^d)$$

where

a is the number of recursive calls (≥ 1).

b is the factor by which the input size shrinks (> 1).

d exponent in summing time of combining step (≥ 0).

and $a, b, c \perp n$

SOLVING RECURRENCES : THE MASTER METHOD III

THE METHOD

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (1)} \\ O(n^d) & \text{if } a < b^d \text{ (2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (3)} \end{cases}$$

SOLVING RECURRENCES : THE MASTER METHOD IV

COUNTING INVERSIONS SOLUTION
The parameters

- Two recursive calls to COUNT-INVERSIONS: $a = 2$.
- Divide the array into two sub-problems: $b = 2$.
- COUNT-SPLIT-INVERSIONS runs in linear time $d = 1$.
- $a = b^d? : 2 = 2^1$.

Falls into case # 1

$$T(n) = O(n \log n)$$

SOLVING RECURRENCES : THE MASTER METHOD V

MORE: BINARY SEARCH

BINARY-SEARCH(A, a, i, p, r)

1: **if** $p < r$ **then**
2:    $q = \lfloor (p+r)/2 \rfloor$
3:    **if** $A[q] == a$ **then**
4:      $i = q$
5:    **end if**
6:    **if** $A[q] > a$ **then**
7:      BINARY-
        SEARCH($A, a, i, p, q-1$)
8:    **else**
9:      BINARY-
        SEARCH($A, a, i, q+1, r$)
10:   **end if**
11: **end if**

The parameters

- $a = 1$.

- $b = 2$.

- $d = 0$, no combining step.

Falls into case # 1

$$T(n) = O(\log n)$$

SOLVING RECURRENCES : THE MASTER METHOD VI

MORE: STRASSEN ALGORITHM FOR MATRICES MULTIPLICATION

The recurrence is

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 7T(n/2) + O(n^2) & \text{if } n > 1 \end{cases}$$

The parameters: $a = 7$, $b = 2$, $d = 2$. $a > b^d$, i.e. $7 > 4$, falls into case # 3

$$T(n) = O(n^{log_2 7}) = O(n^{2.81})$$

and beats the naive $O(n^3)$.