CS 341: Algorithms

Lecture 3: Divide and conquer

Éric Schost

based on lecture notes by many other CS341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2024

The framework

To solve a problem in size n:

Divide

- break the input into **smaller** problems
- ideally few such problems, all of size n/b for some constant b

Conquer

- solve these subproblems recursively

Recombine

- deduce the solution of the main problem from the subproblems

When should you use this?

- original problem nicely decomposable (not much overlap in the subproblems)
- combining solutions is not too costly
- subproblems are not overly unbalanced

Polynomial and matrix multiplication

Multiplying polynomials

Goal: given $\boldsymbol{F = f_0 + \cdots + f_{n-1}x^{n-1}}$ and $\boldsymbol{G = g_0 + \cdots + g_{n-1}x^{n-1}}$, compute

$$H = FG = \boldsymbol{f_0 g_0 + (f_0 g_1 + f_1 g_0)x + \cdots + f_{n-1} g_{n-1} x^{2n-2}}$$

Remark: assume all f_i and g_i fit in one word. Then, input and output size $\boldsymbol{\Theta(n)}$, easy algorithm in $\boldsymbol{\Theta(n^2)}$.

```
1.    for i = 0, ..., n − 1 do
2.        for j = 0, ..., n − 1 do
3.            h_{i+j} = h_{i+j} + f_i g_j
```

Divide and conquer

Idea: write $\boldsymbol{F = F_0 + F_1 x^{n/2}}$, $\boldsymbol{G = G_0 + G_1 x^{n/2}}$. Then

$$H = F_0 G_0 + (F_0 G_1 + F_1 G_0) x^{n/2} + F_1 G_1 x^n$$

Divide and conquer

Idea: write $\boldsymbol{F = F_0 + F_1 x^{n/2}}$, $\boldsymbol{G = G_0 + G_1 x^{n/2}}$. Then

$$H = F_0G_0 + (F_0G_1 + F_1G_0)x^{n/2} + F_1G_1x^n$$

Analysis:

- **4** recursive calls in size $n/2$
- $\boldsymbol{\Theta(n)}$ additions to compute $F_0G_1 + F_1G_0$
- multiplications by $x^{n/2}$ and x^n are free
- $\boldsymbol{\Theta(n)}$ additions to handle overlaps

(Sloppy) recurrence: $T(n) = 4T(n/2) + cn$

- $a = 4$, $b = 2$, $y = 1$ so $\boldsymbol{T(n) \in \Theta(n^2)}$

Not better than the naive algorithm. We do the same operations.

Divide and conquer

Idea: write $\boldsymbol{F = F_0 + F_1 x^{n/2}}$, $\boldsymbol{G = G_0 + G_1 x^{n/2}}$. Then

$$H = F_0 G_0 + (F_0 G_1 + F_1 G_0) x^{n/2} + F_1 G_1 x^n$$

Analysis:

- **4** recursive calls in size $n/2$
- $\boldsymbol{\Theta(n)}$ additions to compute $F_0 G_1 + F_1 G_0$
- multiplications by $x^{n/2}$ and x^n are free
- $\boldsymbol{\Theta(n)}$ additions to handle overlaps

(Sloppy) recurrence: $T(n) = 4T(n/2) + cn$

- $a = 4$, $b = 2$, $y = 1$ so $\boldsymbol{T(n) \in \Theta(n^2)}$

Not better than the naive algorithm. We do the same operations.

Exercise

Use **one** multiplication of polynomials to get $F_0 G_1 + F_1 G_0$, starting from F_0, F_1, G_0, G_1, $F_0 G_0$, $F_1 G_1$

Karatsuba's algorithm

Idea: use the identity

$$(F_0+F_1x^{n/2})(G_0+G_1x^{n/2}) = \boldsymbol{F_0G_0}+((\boldsymbol{F_0} + \boldsymbol{F_1})(\boldsymbol{G_0} + \boldsymbol{G_1})-\boldsymbol{F_0G_0}-\boldsymbol{F_1G_1})x^{n/2}+\boldsymbol{F_1G_1}x^n$$

Karatsuba's algorithm

Idea: use the identity

$$(F_0+F_1x^{n/2})(G_0+G_1x^{n/2}) = \boldsymbol{F_0G_0}+(\boldsymbol{(F_0+F_1)(G_0+G_1)}-\boldsymbol{F_0G_0}-\boldsymbol{F_1G_1})x^{n/2}+\boldsymbol{F_1G_1}x^n$$

Analysis:

- **3** recursive calls in size $n/2$
- $\boldsymbol{\Theta(n)}$ additions to compute F_0+F_1 and G_0+G_1
- multiplications by $x^{n/2}$ and x^n are free
- $\boldsymbol{\Theta(n)}$ additions and subtractions to combine the results

Recurrence: $T(n) = 3T(n/2) + cn$

- $a=3$, $b=2$, $y=1$ so $\boldsymbol{T(n) \in \Theta(n^{\log_2 3})}$ $\qquad \log_2 3 = 1.58\ldots$

Karatsuba's algorithm

Idea: use the identity

$$(F_0+F_1x^{n/2})(G_0+G_1x^{n/2}) = \boldsymbol{F_0G_0}+((\boldsymbol{F_0+F_1})(\boldsymbol{G_0+G_1})-\boldsymbol{F_0G_0}-\boldsymbol{F_1G_1})x^{n/2}+\boldsymbol{F_1G_1}x^n$$

Analysis:

- **3** recursive calls in size $n/2$
- $\boldsymbol{\Theta(n)}$ additions to compute F_0+F_1 and G_0+G_1
- multiplications by $x^{n/2}$ and x^n are free
- $\boldsymbol{\Theta(n)}$ additions and subtractions to combine the results

Recurrence: $T(n) = 3T(n/2) + cn$

- $a = 3$, $b = 2$, $y = 1$ so $\boldsymbol{T(n) \in \Theta(n^{\log_2 3})}$ $\qquad \log_2 3 = 1.58\ldots$

Remark: key idea = a formula for degree-1 polymomials that does **3** multiplications

Toom-Cook and FFT

Took-Cook:

- a family of algorithms based on similar expressions as Karatsuba
- for $k \geq 2$, $\boldsymbol{2k-1}$ recursive calls in size $\boldsymbol{n/k}$
- so $\boldsymbol{T(n) \in \Theta(n^{\log_k(2k-1)})}$
- gets as close to exponent 1 as we want (but very slowly)

FFT:

- if we use complex coefficients, FFT can be used to multiply polynomials
- FFT follows the same recurrence as merge sort, $\boldsymbol{T(n) = 2T(n/2) + cn}$
- so we can multiply polynomials in $\boldsymbol{\Theta(n \log(n))}$ ops over $\mathbb{C}$

Multiplying matrices

Goal: given $\boldsymbol{A = [a_{i,j}]_{1\le i,j\le n}}$ and $\boldsymbol{B = [b_{j,k}]_{1\le j,k\le n}}$ compute $C = AB$

Remark: input and output size $\Theta(n^2)$, easy algorithm in $\Theta(n^3)$

1. **for** $i = 1, \dots, n$ **do**
2. **for** $j = 1, \dots, n$ **do**
3. **for** $k = 1, \dots, n$ **do**
4. $c_{i,k} = c_{i,k} + a_{i,j} b_{j,k}$

Divide and conquer

Setup: write

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

with all $A_{i,k}, B_{i,j}$ of size $n/2 \times n/2$. Then

$$C = \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix}$$

Naively: **8** recursive calls in size $\boldsymbol{n/2}$ + $\boldsymbol{\Theta(n^2)}$ additions $\implies$ $\boldsymbol{T(n) \in \Theta(n^3)}$

Goal: find a better formula for 2×2 matrices

Strassen's algorithm

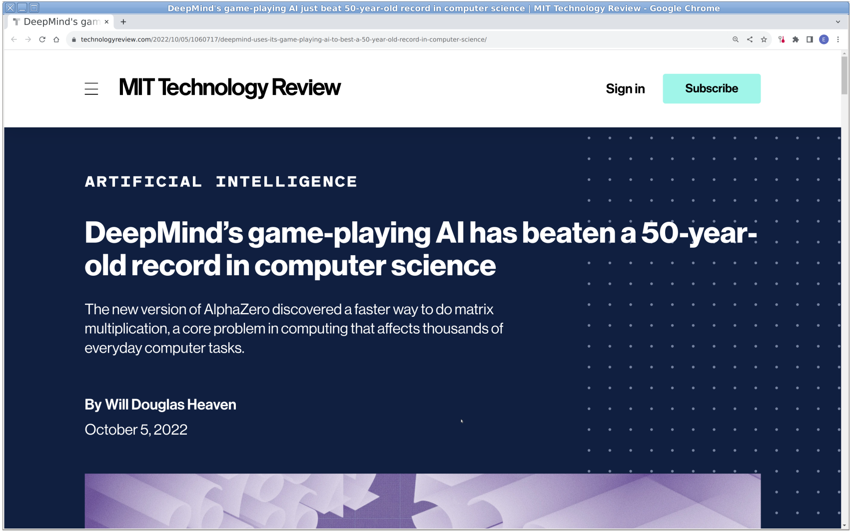
Compute

$$\left|\begin{array}{lcl} Q_1 &=& (A_{1,1}-A_{1,2})B_{2,2} \\ Q_2 &=& (A_{2,1}-A_{2,2})B_{1,1} \\ Q_3 &=& A_{2,2}(B_{1,1}+B_{2,1}) \\ Q_4 &=& A_{1,1}(B_{1,2}+B_{2,2}) \\ Q_5 &=& (A_{1,1}+A_{2,2})(B_{2,2}-B_{1,1}) \\ Q_6 &=& (A_{1,1}+A_{2,1})(B_{1,1}+B_{1,2}) \\ Q_7 &=& (A_{1,2}+A_{2,2})(B_{2,1}+B_{2,2}) \end{array}\right. \quad \text{and} \quad \left|\begin{array}{lcl} C_{1,1} &=& Q_1-Q_3-Q_5+Q_7 \\ C_{1,2} &=& Q_4-Q_1 \\ C_{2,1} &=& Q_2+Q_3 \\ C_{2,2} &=& -Q_2-Q_4+Q_5+Q_6 \end{array}\right.$$

Analysis: $\mathbf{7}$ recursive calls in size $\boldsymbol{n/2}$ + $\boldsymbol{\Theta(n^2)}$ additions $\implies$ $\boldsymbol{T(n) \in \Theta(n^{\log_2(7)})}$

$\log_2(7) = 2.80\ldots$

Faster algorithms: AI to the rescue



DeepMind's game-playing AI just beat 50-year-old record in computer science | MIT Technology Review - Google Chrome

DeepMind's gam ×

technologyreview.com/2022/10/05/1060717/deepmind-uses-its-game-playing-ai-to-best-a-50-year-old-record-in-computer-science/

MIT Technology Review

Sign in

Subscribe

ARTIFICIAL INTELLIGENCE

DeepMind's game-playing AI has beaten a 50-year-old record in computer science

The new version of AlphaZero discovered a faster way to do matrix multiplication, a core problem in computing that affects thousands of everyday computer tasks.

By Will Douglas Heaven

October 5, 2022

Beyond Strassen

Direct generalization

- an algorithm that does $\boldsymbol{k}$ multiplications for matrices of size $\boldsymbol{\ell}$ gives
  $\boldsymbol{T(n) \in \Theta(n^{\log_\ell(k)})}$ (we always have $k > \ell^2$, so no log)

Beyond Strassen

Direct generalization

- an algorithm that does $\boldsymbol{k}$ multiplications for matrices of size $\boldsymbol{\ell}$ gives $\boldsymbol{T(n) \in \Theta(n^{\log_\ell(k)})}$ (we always have $k > \ell^2$, so no log)

A challenge: find best k for small values of ℓ

- SAT solving, gradient descent, ...
- AlphaTensor found some better values, but none beats Strassen (except for matrices over $\{0, 1\}$, with operations modulo 2)

Beyond Strassen

Direct generalization

- an algorithm that does $\boldsymbol{k}$ multiplications for matrices of size $\boldsymbol{\ell}$ gives
  $\boldsymbol{T(n) \in \Theta(n^{\log_\ell(k)})}$ (we always have $k > \ell^2$, so no log)

A challenge: find best k for small values of ℓ

- SAT solving, gradient descent, ...
- AlphaTensor found some better values, but none beats Strassen (except for matrices over $\{0, 1\}$, with operations modulo 2)

Best exponent to date (using more than just divide and conquer)

- $\boldsymbol{O(n^{2.37188})}$, improves from previous record $\boldsymbol{O(n^{2.37286})}$
- galactic algorithms

Counting inversions

Counting inversions

Goal: given an unsorted array $A[1..n]$, find the number of **inversions** in it.

Def: (i, j) is an inversion if $i < j$ and $A[i] > A[j]$

Example: with $A = [1, 5, 2, 6, 3, 8, 7, 4]$, we get

$$(2, 3), (2, 5), (2, 8), (4, 5), (4, 8), (6, 7), (6, 8), (7, 8)$$

Remark 1. we show the **indices** where inversions occur

Counting inversions

Goal: given an unsorted array $A[1..n]$, find the number of **inversions** in it.

Def: (i, j) is an inversion if $i < j$ and $A[i] > A[j]$

Example: with $A = [1, 5, 2, 6, 3, 8, 7, 4]$, we get

$$(2, 3), (2, 5), (2, 8), (4, 5), (4, 8), (6, 7), (6, 8), (7, 8)$$

Remark 1. we show the **indices** where inversions occur

Remark 2. easy algorithm (two nested loops) in $\boldsymbol{\Theta(n^2)}$

Remark 3. to do better than n^2, we cannot **list** all inversions

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, \mathbf{5}, \mathbf{2}, 6, 3, 8, 7, 4]$

- $\boldsymbol{c_\ell = 1}$ $(2, 3)$
- $c_r = 3$ $(6, 7), (6, 8), (7, 8)$
- $c_t = 4$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, 5, 2, 6, 3, \mathbf{8}, \mathbf{7}, 4]$

- $c_\ell = 1$ $(2, 3)$
- $\boldsymbol{c_r = 3}$ $(6, 7), (6, 8), (7, 8)$
- $c_t = 4$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, 5, 2, 6, 3, \mathbf{8}, 7, \mathbf{4}]$

- $c_\ell = 1$ $(2, 3)$
- $\boldsymbol{c_r = 3}$ $(6, 7), (6, 8), (7, 8)$
- $c_t = 4$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, 5, 2, 6, 3, 8, \mathbf{7}, \mathbf{4}]$

- $c_\ell = 1$ $(2, 3)$
- $\boldsymbol{c_r = 3}$ $(6, 7), (6, 8), (7, 8)$
- $c_t = 4$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \le n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, \mathbf{5}, 2, 6, \mathbf{3}, 8, 7, 4]$

- $c_\ell = 1$ $(2, 3)$
- $c_r = 3$ $(6, 7), (6, 8), (7, 8)$
- $\boldsymbol{c_t = 4}$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, \mathbf{5}, 2, 6, 3, 8, 7, \mathbf{4}]$

- $c_\ell = 1$ $(2,3)$
- $c_r = 3$ $(6,7), (6,8), (7,8)$
- $\boldsymbol{c_t = 4}$ $(2,5), (2,8), (4,5), (4,8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, 5, 2, \mathbf{6}, \mathbf{3}, 8, 7, 4]$

- $c_\ell = 1$ $(2, 3)$
- $c_r = 3$ $(6, 7), (6, 8), (7, 8)$
- $\boldsymbol{c_t = 4}$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Toward a divide and conquer algorithm

Idea

- c_ℓ = number of inversions in $A[1..n/2]$
- c_r = number of inversions in $A[n/2+1..n]$
- c_t = number of **transverse** inversions with $i \leq n/2$ and $j > n/2$
- return $c_\ell + c_r + c_t$

Example: with $A = [1, 5, 2, \mathbf{6}, 3, 8, 7, \mathbf{4}]$

- $c_\ell = 1$ $(2, 3)$
- $c_r = 3$ $(6, 7), (6, 8), (7, 8)$
- $\boldsymbol{c_t = 4}$ $(2, 5), (2, 8), (4, 5), (4, 8)$

c_ℓ and c_r done recursively. What about c_t?

Transverse inversions

Goal: how many pairs (i, j) with $i \leq n/2$, $j > n/2$, $A[i] > A[j]$?

Example: with $A = [1, 5, 2, 6, 3, 8, 7, 4]$, we get

$$c_t = \#i\text{'s greater than } 3 + \#i\text{'s greater than } 8 + \#i\text{'s greater than } 7 + \#i\text{'s greater than } 4$$

or

$$c_t = \#j\text{'s less than } 1 + \#j\text{'s less than } 5 + \#j\text{'s less than } 2 + \#j\text{'s less than } 6$$

Transverse inversions

Goal: how many pairs (i, j) with $\boldsymbol{i \leq n/2}$, $\boldsymbol{j > n/2}$, $\boldsymbol{A[i] > A[j]}$?

Example: with $A = [\mathbf{1}, \mathbf{5}, \mathbf{2}, \mathbf{6}, \mathbf{3}, \mathbf{8}, \mathbf{7}, \mathbf{4}]$, we get

$$c_t = \#i\text{'s greater than } 3 + \#i\text{'s greater than } 8 + \#i\text{'s greater than } 7 + \#i\text{'s greater than } 4$$

or

$$c_t = \#j\text{'s less than } 1 + \#j\text{'s less than } 5 + \#j\text{'s less than } 2 + \#j\text{'s less than } 6$$

Observation: this number does not change if both sides are **sorted**, so assume that left and right are sorted after the recursive calls.

Example: With the same input, we get

$$[\mathbf{1}, \mathbf{2}, \mathbf{5}, \mathbf{6}, \mathbf{3}, \mathbf{4}, \mathbf{7}, \mathbf{8}]$$

$$c_t = \#j\text{'s less than } 1 + \#j\text{'s less than } 2 + \#j\text{'s less than } 5 + \#j\text{'s less than } 6$$

Option 1

Algorithm: binary-search all left elements in the right subarray. Then mergesort.

Option 1

Algorithm: binary-search all left elements in the right subarray. Then mergesort.

- this is $O(\log(n))$ per i, so total $\boldsymbol{O(n\log(n))}$
- after that, another $\boldsymbol{\Theta(n\log(n))}$ for sorting
- recurrence: $\boldsymbol{T(n) = 2T(n/2) + cn\log(n)}$
- gives $\boldsymbol{T(n) \in \Theta(n\log^2(n))}$

Option 1

Algorithm: binary-search all left elements in the right subarray. Then mergesort.

- this is $O(\log(n))$ per i, so total $\boldsymbol{O(n\log(n))}$
- after that, another $\boldsymbol{\Theta(n\log(n))}$ for sorting
- recurrence: $\boldsymbol{T(n) = 2T(n/2) + cn\log(n)}$
- gives $\boldsymbol{T(n) \in \Theta(n\log^2(n))}$

Proof:

$$\begin{aligned} T(n) &= 2T(n/2) + n\log(n) \\ &= 4T(n/4) + n\log(n/2) + n\log(n) \\ &= \cdots = n(\log(n) + \log(n/2) + \cdots + \log(2)) \\ &\leq n\log^2(n) \end{aligned}$$

Exercise

Prove $T(n) \in \Omega(n\log^2(n))$

Option 2: enhance mergesort

Observation: if left and right side are sorted, no need to sort everything, just **merge**

Goal: find c_t during merge.

Merge($A[1..n]$) (both halves of A assumed sorted)

copy A into a new array S
$i = 1$; $j = n/2 + 1$;
for $(k \leftarrow 1; k \leq n; k{+}{+})$ **do**
  if $(i > n/2)$ $A[k] \leftarrow S[j{+}{+}]$
  else if $(j > n)$ $\boldsymbol{A[k] \leftarrow S[i{+}{+}]}$
  else if $(S[i] < S[j])$ $\boldsymbol{A[k] \leftarrow S[i{+}{+}]}$
  else $A[k] \leftarrow S[j{+}{+}]$

When we insert $S[i]$ back in A, need to count how many j's have been processed already

EnhancedMerge$(A[1..n])$ (both halves of A assumed sorted)

1. copy A into a new array S; $c = 0$
2. $i = 1$; $j = n/2 + 1$;
3. **for** $(k \leftarrow 1;\ k \leq n;\ k{+}{+})$ **do**
4. **if** $(i > n/2)$ $A[k] \leftarrow S[j{+}{+}]$
5. **else if** $(j > n)$ $A[k] \leftarrow S[i{+}{+}]$; $\boldsymbol{c = c + n/2}$
6. **else if** $(S[i] < S[j])$ $A[k] \leftarrow S[i{+}{+}]$; $\boldsymbol{c = c + j - (n/2 + 1)}$
7. **else** $A[k] \leftarrow S[j{+}{+}]$

EnhancedMerge($A[1..n]$) (both halves of A assumed sorted)

```
copy A into a new array S;  c = 0
i = 1; j = n/2 + 1;
for (k ← 1; k ≤ n; k++) do
    if (i > n/2) A[k] ← S[j++]
    else if (j > n) A[k] ← S[i++]; c = c + n/2
    else if (S[i] < S[j]) A[k] ← S[i++]; c = c + j − (n/2 + 1)
    else A[k] ← S[j++]
```

Example: with $[\mathbf{1, 2, 5, 6}, \mathbf{3, 4, 7, 8}]$

- when we insert 1 back into A, $j = 5$ so $c = c + 0$
- when we insert 2 back into A, $j = 5$ so $c = c + 0$
- when we insert 5 back into A, $j = 7$ so $c = c + 2$
- when we insert 6 back into A, $j = 7$ so $c = c + 2$

Enhanced merge is still $\boldsymbol{\Theta(n)}$ so total remains $\boldsymbol{\Theta(n \log(n))}$.