CSE 331: Algorithms & Complexity

"Multiplication"

Prof. Charlie Anne Carlson (She/Her)

Lecture 25

Friday Dec 25th, 2025

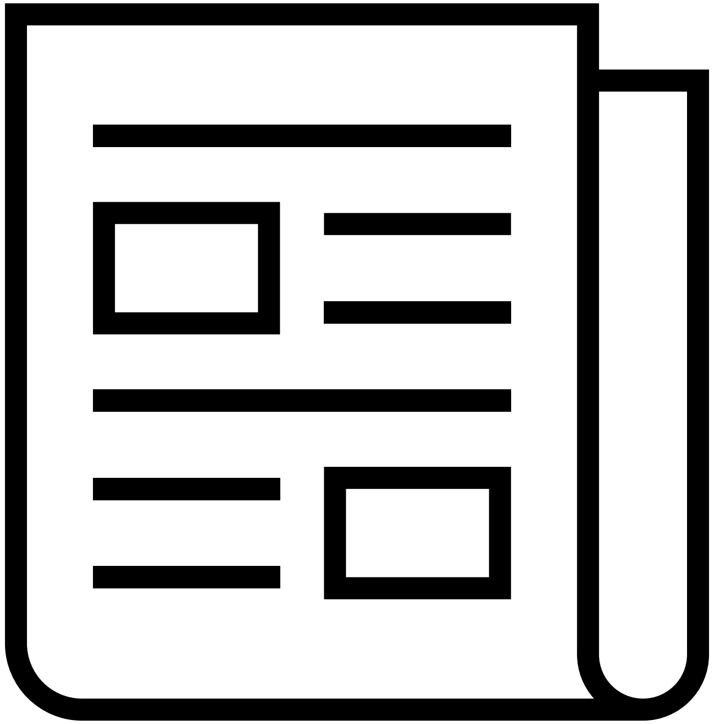
University at Buffalo

Schedule

1. Course Updates
2. Counting Inversions
3. Multiplication

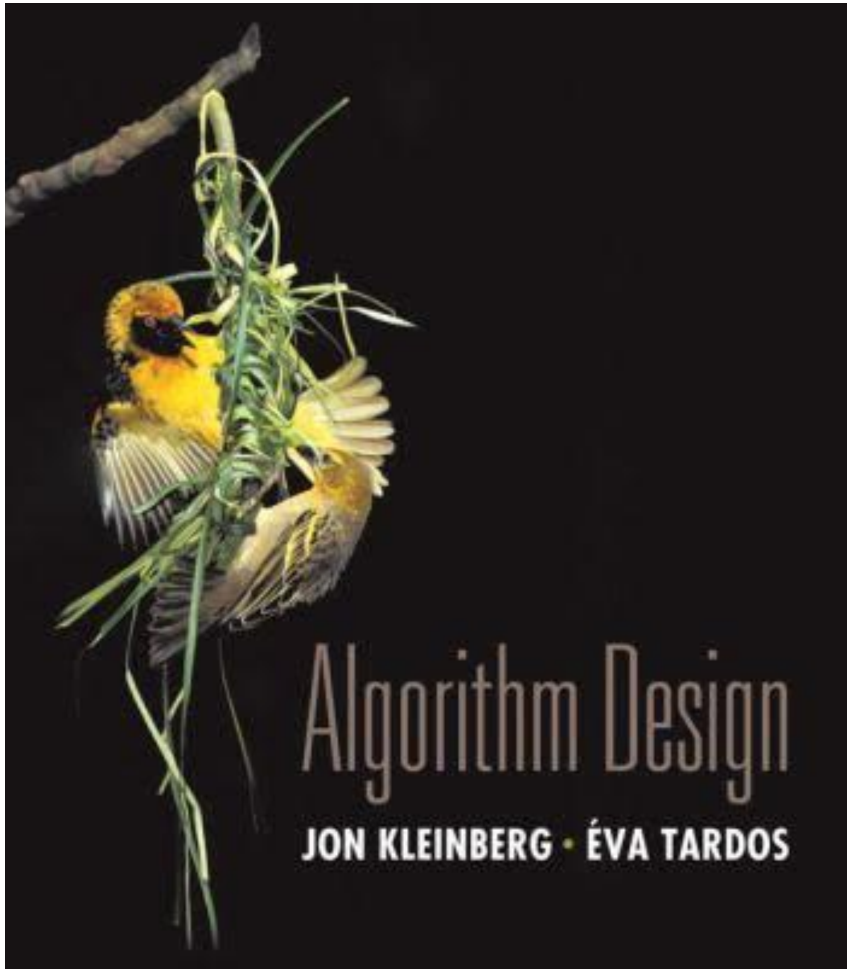
Course Updates

- HW 6 Out
- Group Project
  - Code 1 & 2 Due ?
  - Reflections 1 & 2 Due ?



Reading

- You should have read:
  - Started 5.5
  - Started 5.4
- Before Next Class:
  - Finished KT 5.5
  - Finished KT 5.4
  - Read Unraveling the mystery behind the identity



Divide & Conquer Algorithm

- We will use the logic of the previous lecture to make a Merge-and-Count(A,B) algorithm that will merge two sorted lists and count the number of "spanning" inversions.

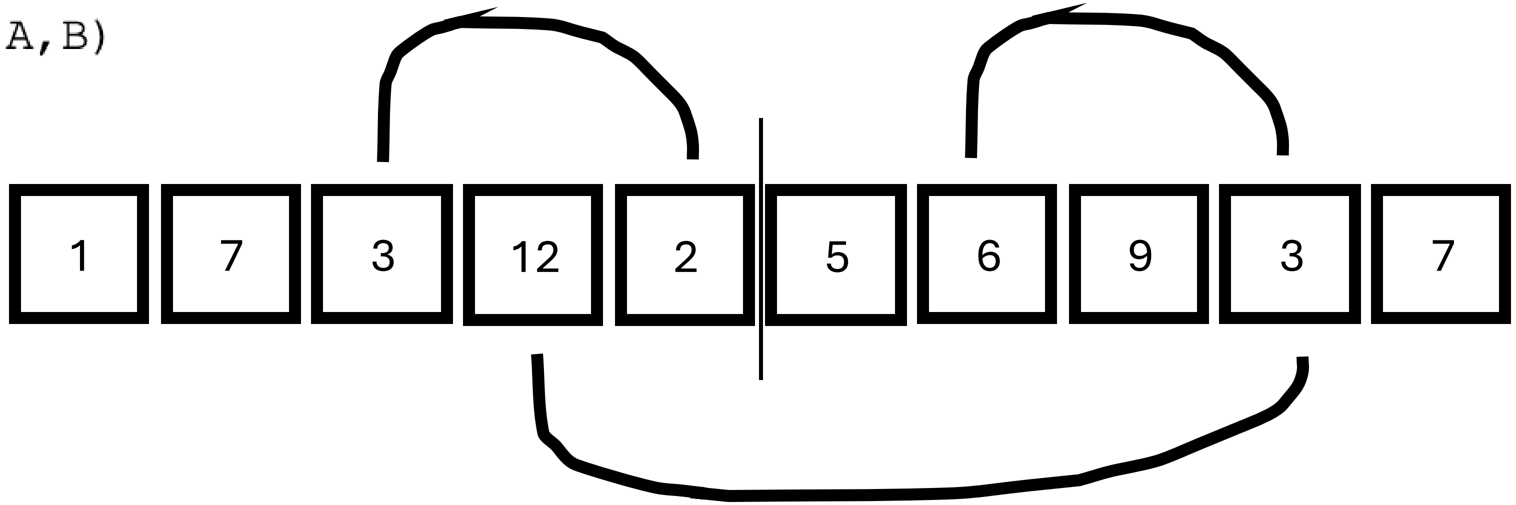
- We will now make a new algorithm called Sort-and-Count(L) that will take a list and return the list sorted and return the number of inversions before being sorted.

Sort-and-Count

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

When is each type of inversion counted?

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```



Sort-and-Count Runtime?

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

Sort-and-Count Runtime?

- Observations:
  - Takes O(n) time to divide.
  - Takes 2T(n/2) time to do recursive calls.
  - Takes O(n) time to merge.
  - Takes O(1) time to do base case.

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

Sort-and-Count Runtime

- We have the same recurrence we had for mergesort and if we solve it using the methods from before, we get the same runtime of O(nlog(n)).

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

Sort-and-Count Runtime?

- **Question**: What would you change to get the list of all inversions?
- **Question**: How would this change the runtime?

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

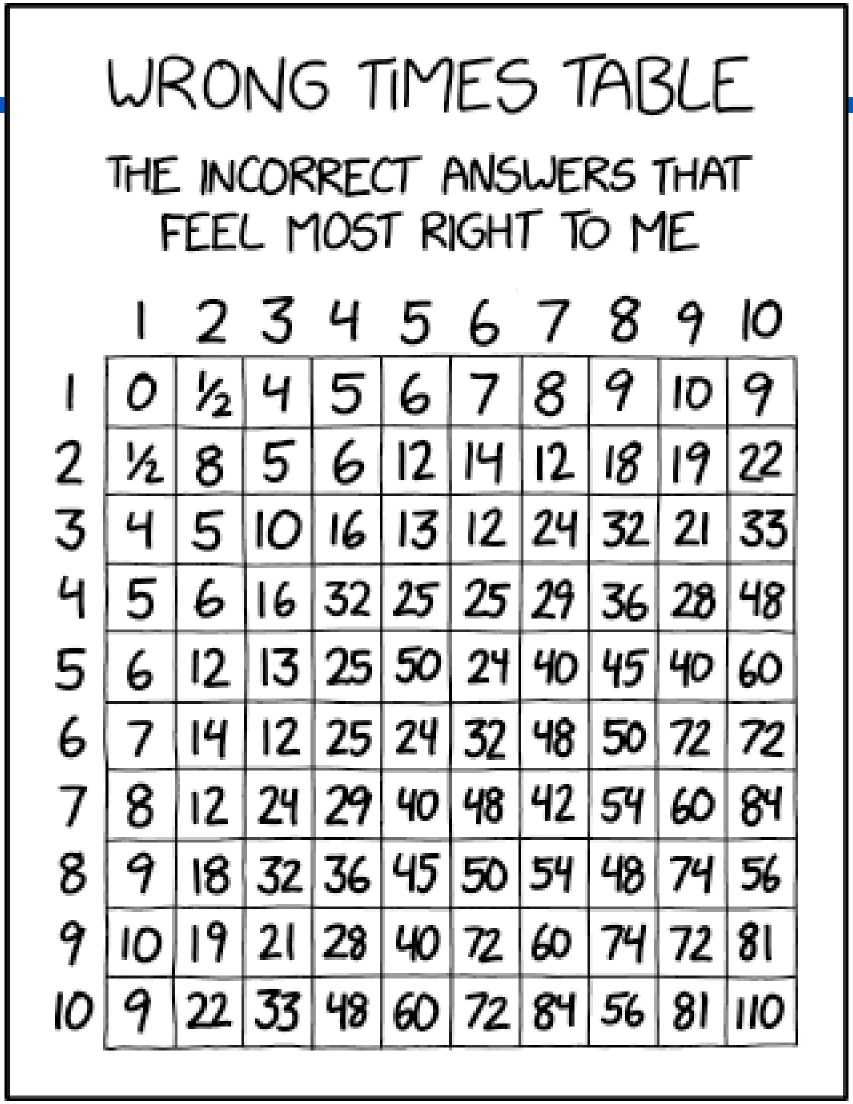
Sort-and-Count Runtime?

- **Answer**: You'd want to change your Sort-and-Count to return list of inversions.
- **Answer**: This would take longer because we do have to list all pairs in some cases.

```
Input: list L of length n
 If the list has on element:
  there are no inversions
 Else:
  Divide the list into two halves:
   A contains first ⌈n/2⌉ elements
   B contains second ⌊n/2⌋ elements
  (r,A) = Sort-and-Count(A)
  (q,B) = Sort-and-Count(B)
 (k,L) = Merge-and-Count(A,B)
 Return (r+q+k,L)
```

Multiplication

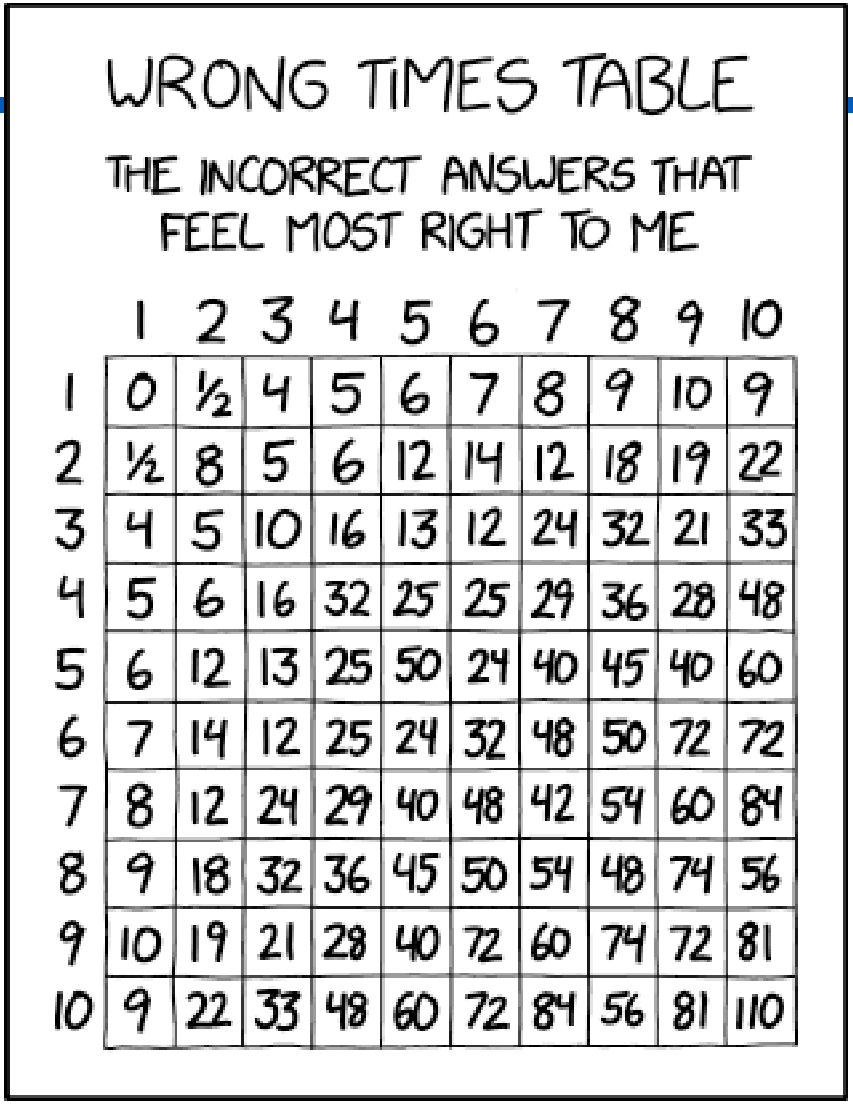
- Input: Given two numbers a and b in binary
  - $a = (a_1, a_2, \dots, a_n)$
  - $b = (b_1, b_2, \dots, b_n)$
- Goal: Compute $c = a \times b$



https://xkcd.com/2313/

Multiplication

- Input: Given two numbers a and b in binary
  - $a = (a_1, a_2, \dots, a_n)$
  - $b = (b_1, b_2, \dots, b_n)$
- Goal: Compute $c = a \; x \; b$



	1	2	3	4	5	6	7	8	9	10
1	0	½	4	5	6	7	8	9	10	9
2	½	8	5	6	12	14	12	18	19	22
3	4	5	10	16	13	12	24	32	21	33
4	5	6	16	32	25	25	29	36	28	48
5	6	12	13	25	50	24	40	45	40	60
6	7	14	12	25	24	32	48	50	72	72
7	8	12	24	29	40	48	42	54	60	84
8	9	18	32	36	45	50	54	48	74	56
9	10	19	21	28	40	72	60	74	72	81
10	9	22	33	48	60	72	84	56	81	110

https://xkcd.com/2313/

Grade School Algorithm

- Compute a "partial product" for each digit of a by b.
- Add up all partial products.
  - Don't forget how to add!

- Question: What is the runtime of this algorithm for two n bit numbers?

```
       1100
  x    1101
  ---------
       1100
      0000
     1100
  + 1100
  ---------
   10011100
```

Grade School Algorithm

- Compute a “partial product” for each digit of a by b.
- Add up all partial products.
  - Don’t forget how to add!

- Answer: It is an O(n^2) algorithm!

```
      1100
  x   1101
  --------
      1100
     0000
    1100
 + 1100
 ---------
  10011100
```

Divide and Conquer Algorithm

- We will rewrite a and b into their high and low bit components.
  - $m = \lfloor n/2 \rfloor$
  - $a = a^{\mathrm{H}} \cdot 2^{\mathrm{m}} + a^{\mathrm{L}}$
    - $a^{\mathrm{H}} = \lfloor a/2^{\mathrm{m}} \rfloor$
    - $a^{\mathrm{L}} = a \bmod 2^{m}$
  - $\mathrm{b} = \mathrm{b}^{\mathrm{H}} \cdot 2^{\mathrm{m}} + \mathrm{b}^{\mathrm{L}}$
    - $\mathrm{b}^{\mathrm{H}} = \lfloor \mathrm{b}/2^{\mathrm{m}} \rfloor$
    - $\mathrm{b}^{\mathrm{L}} = \mathrm{b} \bmod 2^{m}$

E.g.:

a^{H} (1000), a^{L} (1101)

a = 10001101

b = 11100001

b^{H} (1110), b^{L} (0001)

Divide and Conquer Algorithm

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

Divide and Conquer Algorithm

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- There are 4 subproblems of size ~n/2
- There are two shifts by O(n)
- There is two sums of O(n) bit bumbers

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- There are 4 subproblems of size ~n/2
- There are two shifts by O(n)
- There is two sums of O(n) bit bumbers

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- There are 4 subproblems of size ~n/2 **<- 4T(n/2) time**
- There are two shifts by O(n) **<- O(n) time**
- There is two sums of O(n) bit bumbers **<- O(n) time**

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- $T(n) \leq 4T(n/2) + cn$ when n big
- $T(1) \leq c$

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- $\mathrm{T(n)} \leq 4\mathrm{T(n/2)} + \mathrm{cn}$ when n big
- $T(1) \leq c$
- **This is not that good, T(n) is O(n^2)**

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- $\mathrm{T(n)} \leq 4\mathrm{T(n/2)} + \mathrm{cn}$ when n big
- $T(1) \leq c$
- **This is not that good, T(n) is O(n^2)**

What is the runtime, T(n)?

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- We know from Section 5.2 that if instead of 4 recursive calls, we did only 3, we could get a much better running time.
  - We would get $T(n) \in O(n^{1.59})$

Reducing Calls

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- **Key Observation:**

$$(a^H + a^L) \cdot (b^H + b^L) = a^H \cdot b^H + a^H \cdot b^L + a^L \cdot b^H + a^L \cdot b^L$$

Reducing Calls

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- **Key Observation:**

$$(a^H + a^L) \cdot (b^H + b^L) = a^H \cdot b^H + a^H \cdot b^L + a^L \cdot b^H + a^L \cdot b^L$$

Reducing Calls

- We can now write:

$$a \cdot b = (a^H \cdot 2^m + a^L)(b^H \cdot 2^m + b^L)$$
$$= a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

- **Key Observation:**

$$(a^H + a^L) \cdot (b^H + b^L) = a^H \cdot b^H + a^H \cdot b^L + a^L \cdot b^H + a^L \cdot b^L$$

- We can compute $a^H \cdot b^H$ and $a^L \cdot b^L$ and then use all these values to compute $a^H \cdot b^L + a^L \cdot b^H$!

Reducing Calls

- Instead of compute all of these coefficients with a call

$$a^H \cdot b^H \cdot 2^{(2m)} + (a^H \cdot b^L + a^L \cdot b^H) \cdot 2^m + a^L \cdot b^L$$

  we can compute $(a^H + a^L) \cdot (b^H + b^L)$, $a^H \cdot b^H$ and $a^L \cdot b^L$ with three calls and then do O(n) work to combine (subtraction, addition, shifts) them together to get all the coefficients!

$$(a^H + a^L) \cdot (b^H + b^L) = a^H \cdot b^H + a^H \cdot b^L + a^L \cdot b^H + a^L \cdot b^L$$

Recursive Algorithm

Recursive-Multiply(x,y):

Write $x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

Compute $x_1 + x_0$ and $y_1 + y_0$

p = Recursive-Multiply($x_1 + x_0$, $y_1 + y_0$)

x_1y_1 = Recursive-Multiply(x_1, y_1)

x_0y_0 = Recursive-Multiply(x_0, y_0)

Return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$

Runtime

Recursive-Multiply(x,y):

  Write $x = x_1 \cdot 2^{n/2} + x_0$

    $y = y_1 \cdot 2^{n/2} + y_0$

  Compute $x_1 + x_0$ and $y_1 + y_0$

  p = Recursive-Multiply($x_1 + x_0$, $y_1 + y_0$)

  x_1y_1 = Recursive-Multiply(x_1, y_1)

  x_0y_0 = Recursive-Multiply(x_0, y_0)

  Return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$

- In the non recursive case, we do three calls of size n/2.
  - Hence, $\mathrm{T(n)} \leq 3\mathrm{T(n/2)} + \mathrm{cn}$ when n is big.
  - Thus, $T(n) \in O\left(n^{\log_2(3)}\right)$ <- **See K.T. 5.2**

Runtime

```
Recursive-Multiply(x,y):
  Write x = x_1 · 2^{n/2} + x_0
        y = y_1 · 2^{n/2} + y_0
  Compute x_1 + x_0 and y_1 + y_0
  p = Recursive-Multiply(x_1 + x_0, y_1 + y_0)
  x_1y_1 = Recursive-Multiply(x_1, y_1)
  x_0y_0 = Recursive-Multiply(x_0, y_0)
  Return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · 2^{n/2} + x_0y_0
```

- In the non recursive case, we do three calls of size n/2.
  - Hence, $\mathrm{T(n)} \leq 3\mathrm{T(n/2)} + \mathrm{cn}$ when n is big.
  - Thus, $T(n) \in O\left(n^{\log_2(3)}\right)$ <- **See K.T. 5.2**

Want to know more?

De-Mystifying the Integer Multiplication Algorithm

In class, we saw an $O\left(n^{\log_2 3}\right)$ time algorithm to multiply two n bit numbers that used an identity that seemed to be plucked out of thin air. In this note, we will try and de-mystify how one might come about thinking of this identity in the first place.

The setup

We first recall the problem that we are trying to solve:

Multiplying Integers

Given two n bit numbers $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$, output their product $c = a \times b$.