Lecture 10: Multiplication

COSC 311 *Algorithms*, Fall 2022

Announcements

1. Homework 2 Finalized Today (1 additional question)
2. No reading/lecture ticket for Monday
3. Thoughts on Reading

Overview

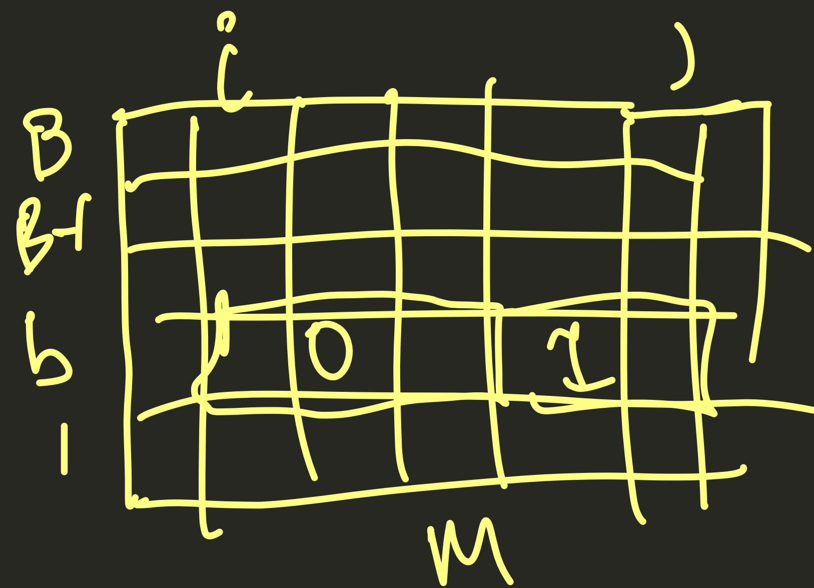
1. Recap of Binary Radix Sort
2. Binary Arithmetic
3. Multiplication via Divide and Conquer

Last Time

Binary Radix Sort

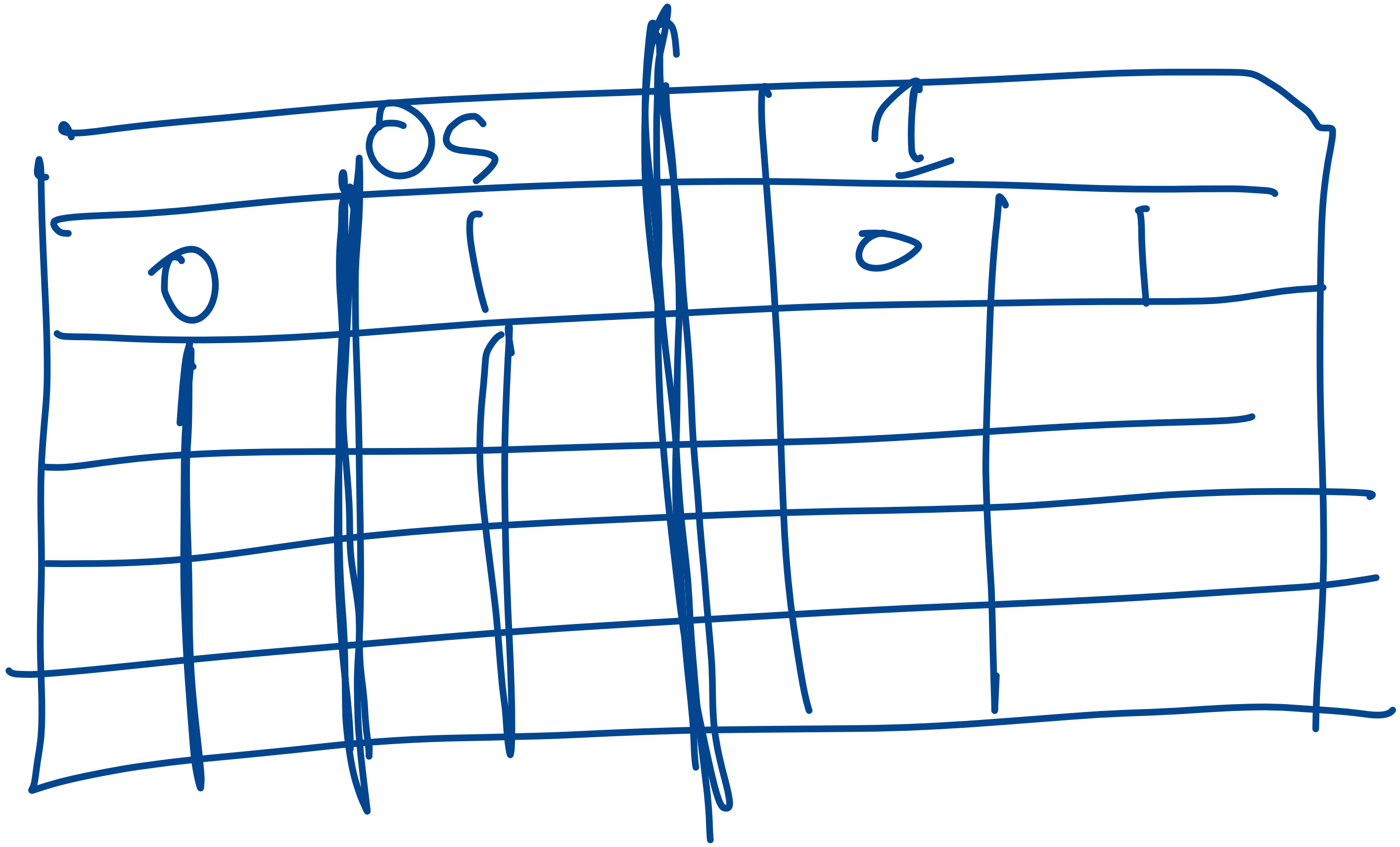
```
RadixSort(a, B):                  # B is number of bits
  RadixSort(a, 1, size(a)+1, B)

RadixSort(a, i, j, b):
  if j - i <= 1 then
    return
  endif
  m <- BitSplit(a, i, j, b)
  RadixSort(a, i, m, b-1)
  RadixSort(a, m, j, b-1)
```

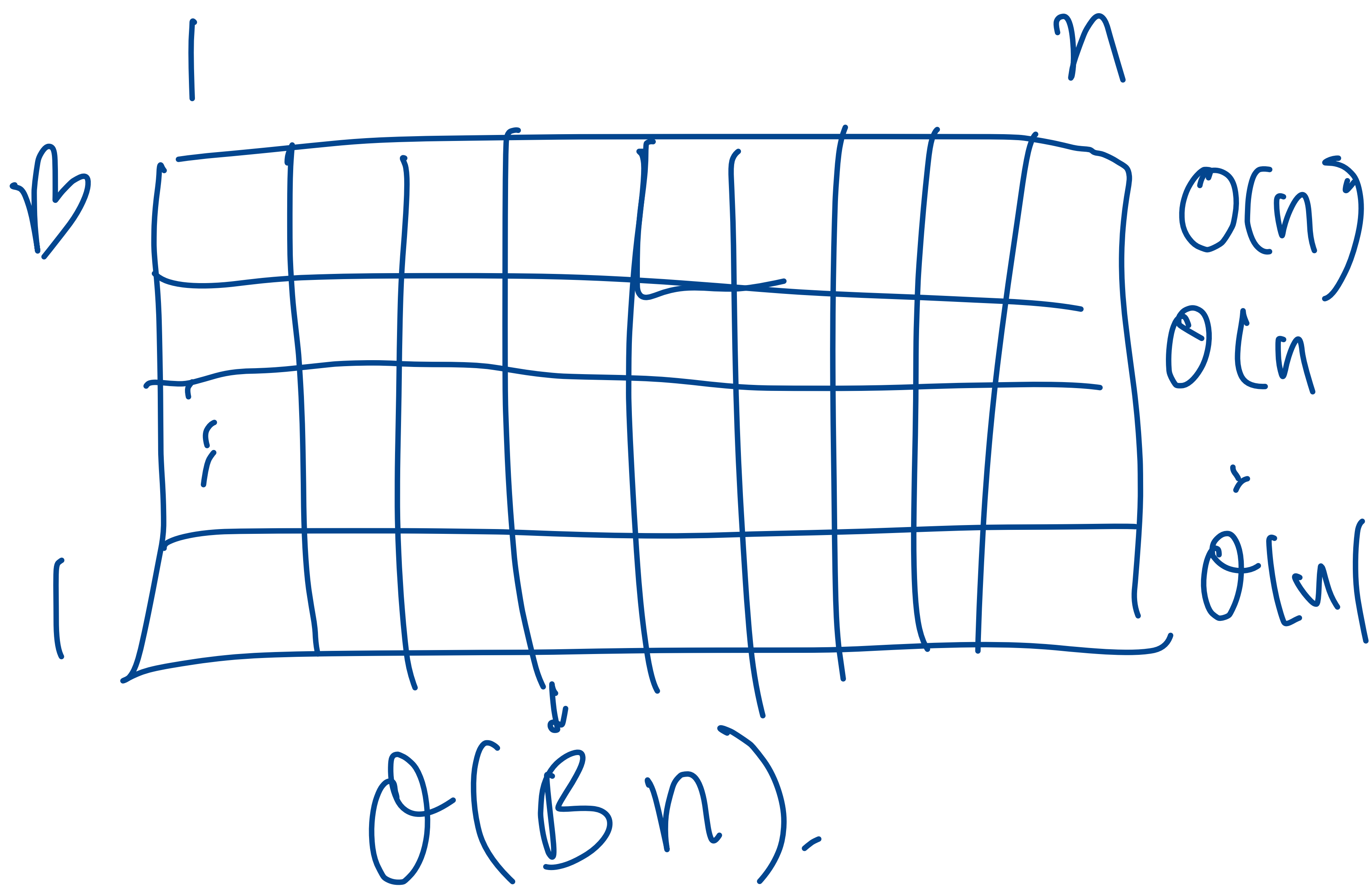


Illustration

Why Does RadixSort Work?



What is RadixSort Running Time?



Conclusion

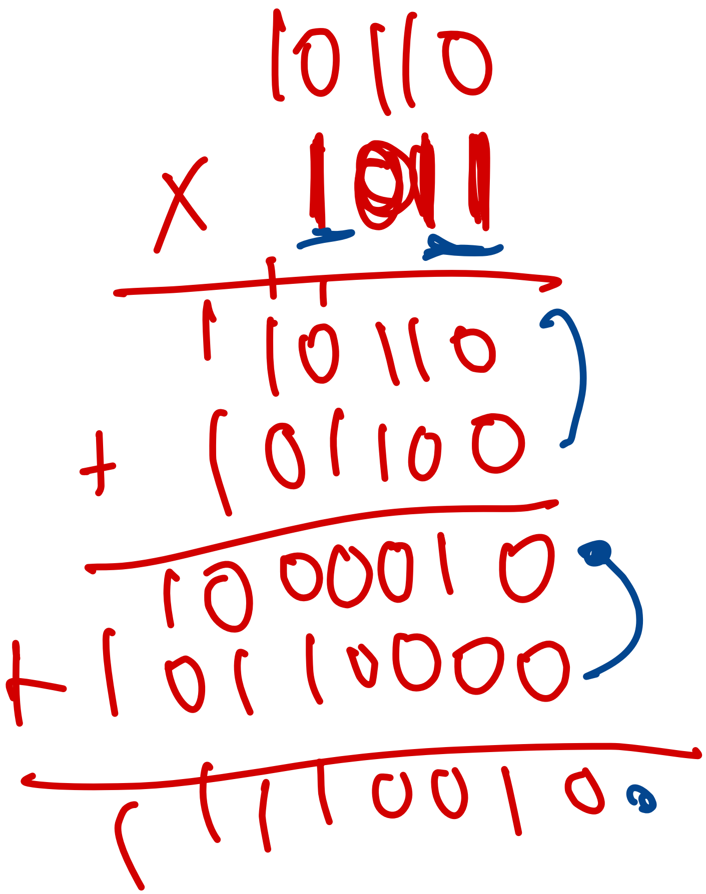
Lower Bound. Any algorithm that sorts all permutations of size n using only compare and swap operations requires $\Omega(n \log n)$ comparisons.

Caveat. If values are all represented with B bits, then RadixSort sorts n using $O(Bn)$ bit-wise comparisons.

Binary Arithmetic

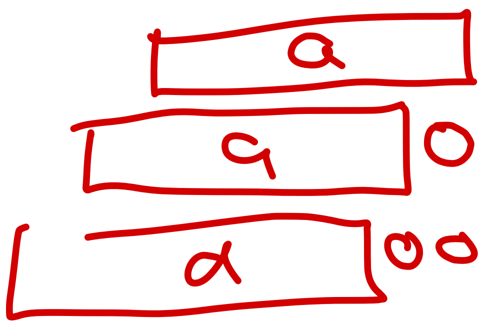
Multiplication in Binary

Example. Compute $10110 * 1011$.



Multiplication Procedure

```
Multiply(a, b):
product <- 0
shifted <- a   # copy of a we will shift
for i = 1 to size b do
  if b[i] = 1 do
    product <- Add(product, shifted)
  endif
  shifted << 1
endfor
```



Question

```
Multiply(a, b):
product <- 0
shifted <- a     # copy of a we will shift
for i = 1 to size b do
  if b[i] = 1 do
    product <- Add(product, shifted)
  endif
  shifted << 1
endfor
```

If a and b are represented with n bits, what is the running time of Multiply(a, b)?

a, b w/ $n = \log N$ bits

$O(n)$

iter. n

$O(n^2)$?

$n \cdot O(n) = O(n^2)$.

$a, b \leq N \leq 2^n - 1$

$a = a_n a_{n-1} \cdots a_1$

$\leq 2^{n-1} + 2^{n-2} + \cdots + 2^0$

$= 2^n - 1$

Another Question

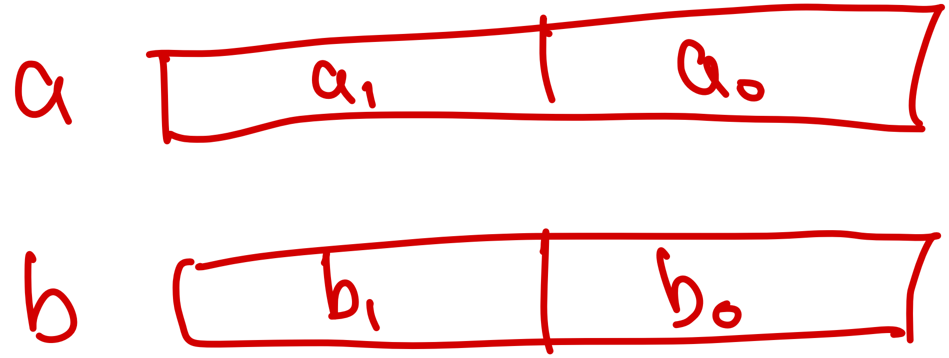
Why did we previously assume arithmetic takes $O(1)$ time?

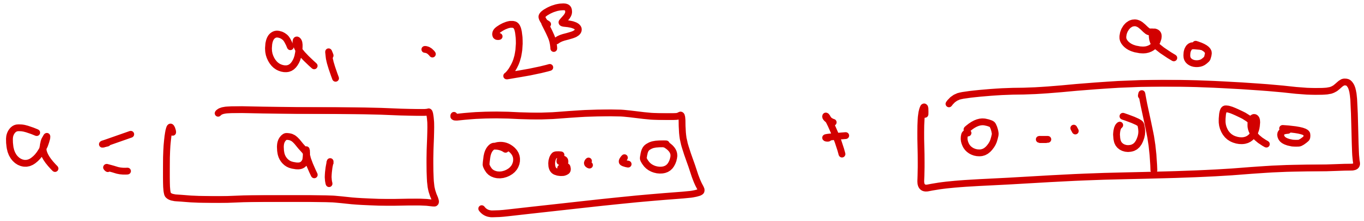
int in Java uses 32 bits

Multiplication via Divide and Conquer

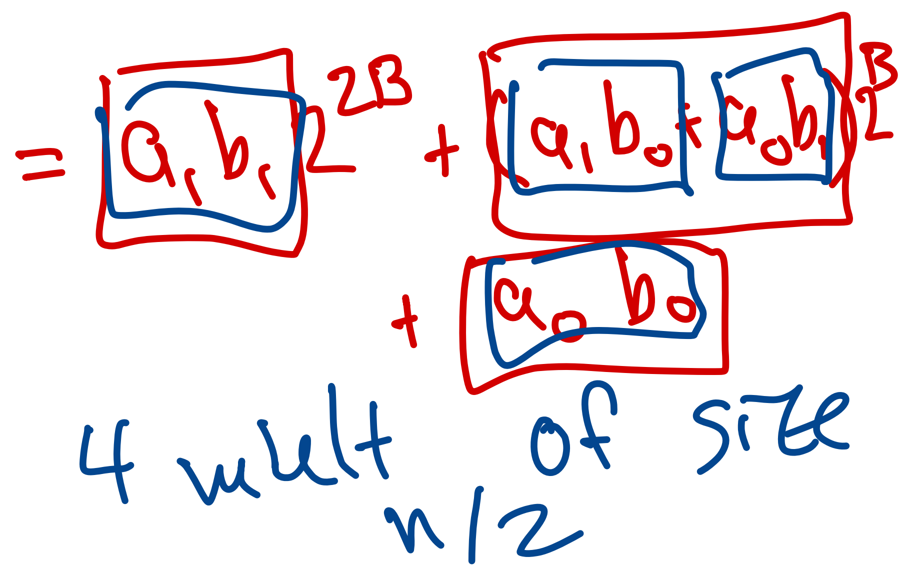
Idea. Break numbers up into parts

- Assume a and b are both represented with $n = 2B$ bits, n power of 2
- Write:
  - $a = a_1 a_0 = a_1 2^B + a_0$
  - $b = b_1 b_0 = b_1 2^B + b_0$





- Rewrite multiplication
  - $ab = (a_1 2^B + a_0)(b_1 2^B + b_0)$



mult size n

Does This Help?

$$ab = (a_1 2^B + a_0)(b_1 2^B + b_0) = a_1 b_1 2^{2B} + (a_1 b_0 + a_0 b_1) 2^B +$$

- Replaced 1 product with n bit numbers with 4 products of $n/2$ bits

Not yet...

A Magic Trick

$$ab = a_1 b_1 2^{2B} + (a_1 b_0 + a_0 b_1) 2^B + a_0 b_0$$

- Define:

$$c_2 = a_1 b_1 \quad c_1 = a_1 b_0 + a_0 b_1 \quad c_0 = a_0 b_0$$

- Now: $ab = c_2 2^{2B} + c_1 2^B + c_0$
- Consider:
  $c^* = (a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_1 b_0 + a_0 b_1 + a_0 b_0$

Question. How do c_0, c_1, c_2 relate to c^*?

$c^* = c_2 + c_1 + c_0$

$\Rightarrow c_1 = c^* - c_2 - c_0$

$\frac{3}{4} n^2$

n^2 ops for g.s. mult

$3 \cdot (\frac{n}{2})^2 + O(n)$

mult of size $n/2$

Counting Products

- Standard multiplication:
  - $c_2 = a_1 b_1$
  - $c_1 = a_1 b_0 + a_0 b_1$
  - $c_0 = a_0 b_0$
- Tricky multiplication
  - $c_2 = a_1 b_1$
  - $c_0 = a_0 b_0$
  - $c^* = (a_1 + a_0)(b_1 + b_0)$
  - $c_1 = c^* - c_2 - c_0$

Progress?

By using c^* to compute ab:

- $c_2 = a_1 b_1$
- $c_0 = a_0 b_0$
- $c^* = (a_1 + a_0)(b_1 + b_0)$

Compute

- $ab = c_2 2^{2B} + (c^* - c_2 - c_0)2^B + c_0$

Computing ab uses:

- 3 multiplications of size $n/2$
- $O(1)$ additions/subtractions/shifts of size $O(n)$

Karatsuba Multiplication

```
KMult(a, b):
  n <- size(a) (= size(b))
  if n = 1 then return a*b
      a = a1 a0
      b = b1 b0
      c2 <- KMult(a1, b1)
      c0 <- KMult(a0, b0)
      c <- KMult(a1 + a0, b1 + b0)
      return (c2 << n) + ((c - c2 - c0) << (n/2)) + c0
```

Karatsuba Recursion Tree

Efficiency of Karatsuba

At depth k:

- 3^k calls to `KMult`
- size of each call is $n/2^k$
- depth of recursion is $\log n$

Total running time:

- $O(n) + \frac{3}{2}O(n) + \left(\frac{3}{2}\right)^2 O(n) + \cdots + \left(\frac{3}{2}\right)^{\log n} O(n)$

Can show:

- This expression is $O(3^{\log n})$

Simplify:

Final Running Time

Result. The running time of Karatsuba multiplication is $O(n^{\log 3}) \approx O(n^{1.58})$

- when n is reasonbly large, $n^{1.58} \ll n^2$
- E.g., $1,000^2 = 1,000,000$ vs $1,000^{1.58} \approx 55,000$

Next Time

- More Divide and Conquer
- Solving General Recurrences