Algorithm Design & Analysis (CS3383)[1]

Unit 1: Divide and Conquer Introduction

Rasoul Shahsavarifar

January 14, 2019

[1]Thanks to Dr. Ptricia Evans and Dr. David Bremner at UNB, Dr. Erik Demaine at MIT for sharing the teaching stuffs

Outline[2]

Divide and Conquer

- Intro
- Merge Sort
- Recursion Tree for recurrences
- Integer Multiplication

[2] Reading:

- Main textbook (DPV), Divide and conquer algorithms, Chapter 2 mainly 2.1 to 2.5
- Algorithms, Cormen, Chapter 4 (4.2, 4.3, 4.4, 4.5)

unit prereqs

- mergesort
- geometric series (CLRS A.5)

Contents

Divide and Conquer
- Intro
- Merge Sort
- Recursion Tree for recurrences
- Integer Multiplication

Structure of divide and conquer

```
function SOLVE(P)
    if |P| is small then
        SolveDirectly(P)
    else
        P_1 ... P_k = Partition(P)
        for i = 1 ... k do
            S_i = Solve(P_i)
        end for
        Combine(S_1 ... S_k)
    end if
end function
```

- Where is the actual work?

Structure of divide and conquer

function $\textsc{Solve}$(P)
    if $|P|$ is small **then**
        SolveDirectly(P)
    else
        $P_1 \ldots P_k = \text{Partition}(P)$
        for $i = 1 \ldots k$ **do**
            $S_i = \text{Solve}(P_i)$
        end for
        Combine($S_1 \ldots S_k$)
    end if
end function

- Where is the actual work?
- How many subproblems?

Structure of divide and conquer

function SOLVE(P)
    if $|P|$ is small **then**
        SolveDirectly(P)
    else
        $P_1 \ldots P_k = \text{Partition}(P)$
        for $i = 1 \ldots k$ **do**
            $S_i = \text{Solve}(P_i)$
        end for
        Combine($S_1 \ldots S_k$)
    end if
end function

- Where is the actual work?
- How many subproblems?
- How big are the subproblems?

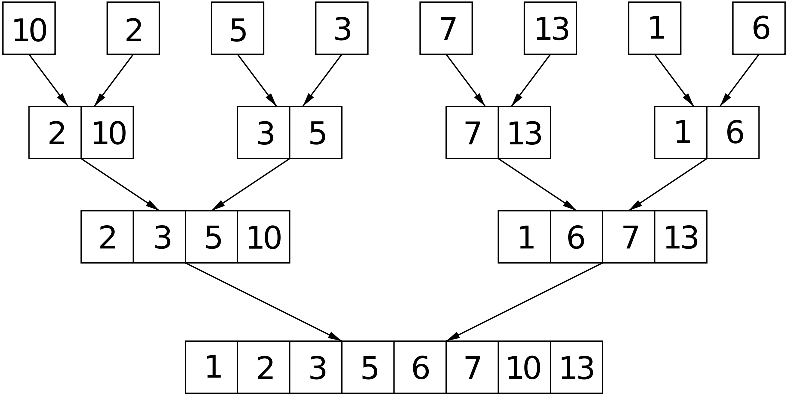
Contents

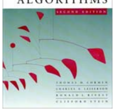
Divide and Conquer
- Intro
- Merge Sort
- Recursion Tree for recurrences
- Integer Multiplication

Merge Sort

Input:

10	2	5	3	7	13	1	6



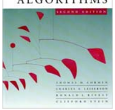


Merge sort

MERGE-SORT $A[1\,..\,n]$

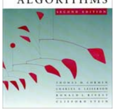
1. If $n = 1$, done.
2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**



Merging two sorted arrays

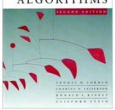
20	12
13	11
7	9
2	1



Merging two sorted arrays

20	12
13	11
7	9
2	1

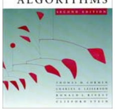
1



Merging two sorted arrays

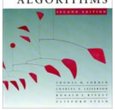
20	12	‖	20	12
13	11	‖	13	11
7	9	‖	7	9
2	1	‖	2	

1



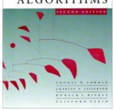
Merging two sorted arrays

20	12	‖	20	12
13	11	‖	13	11
7	9	‖	7	9
2	1	‖	2	
1		‖	2	



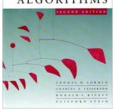
Merging two sorted arrays

20	12	20	12	20	12
13	11	13	11	13	11
7	9	7	9	7	9
2	1	2			
	1		2		



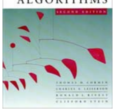
Merging two sorted arrays

20	12	‖	20	12	‖	20	12
13	11	‖	13	11	‖	13	11
7	9	‖	7	9	‖	7	9
2	1	‖	2		‖		
1			2			7	

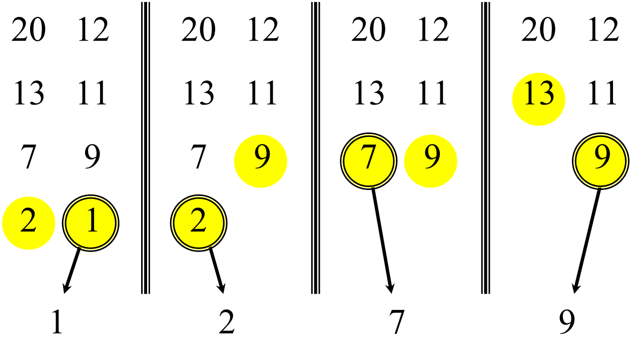


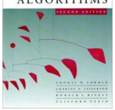
Merging two sorted arrays

20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11
7	9	7	9	7	9		9
2	1	2					
1		2		7			

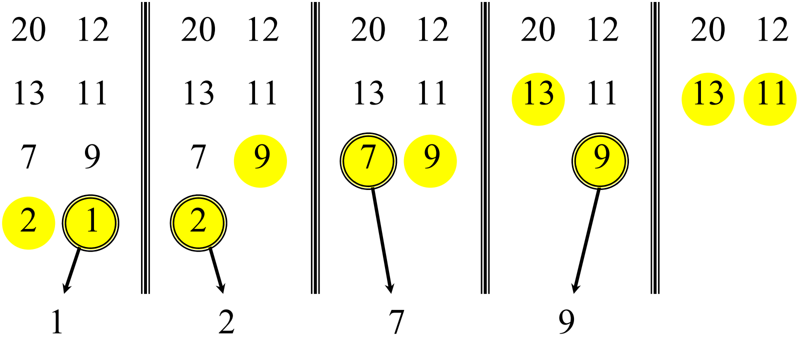


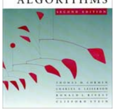
Merging two sorted arrays



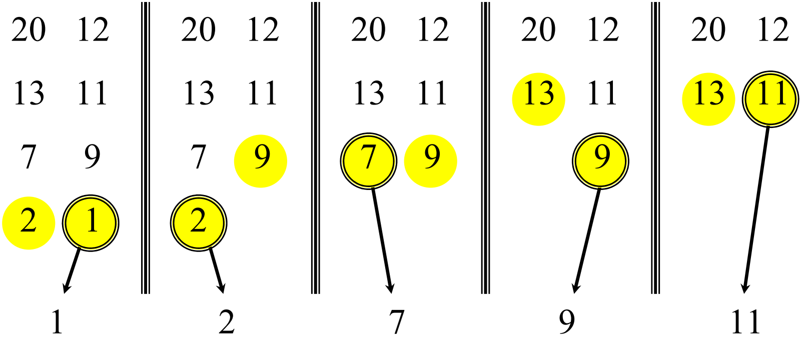


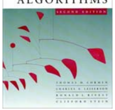
Merging two sorted arrays



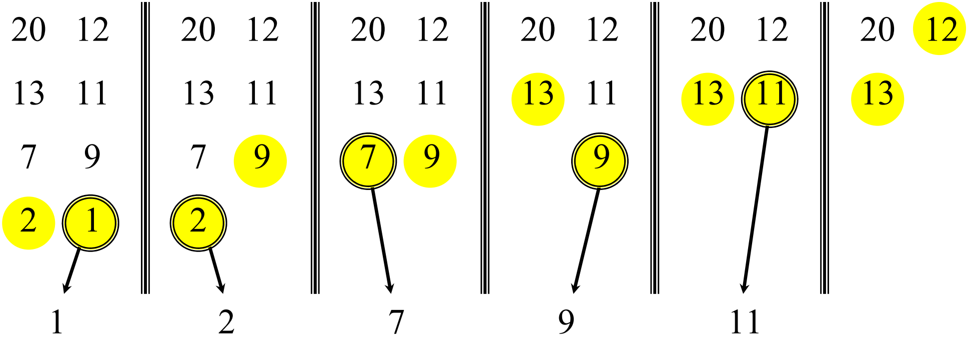


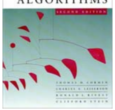
Merging two sorted arrays



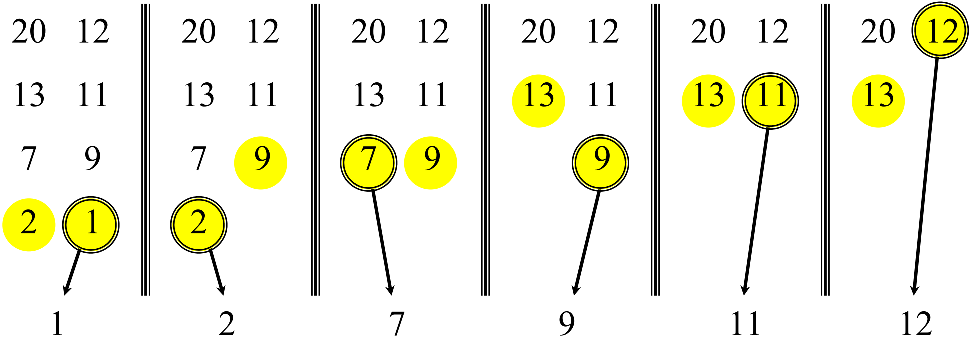


Merging two sorted arrays

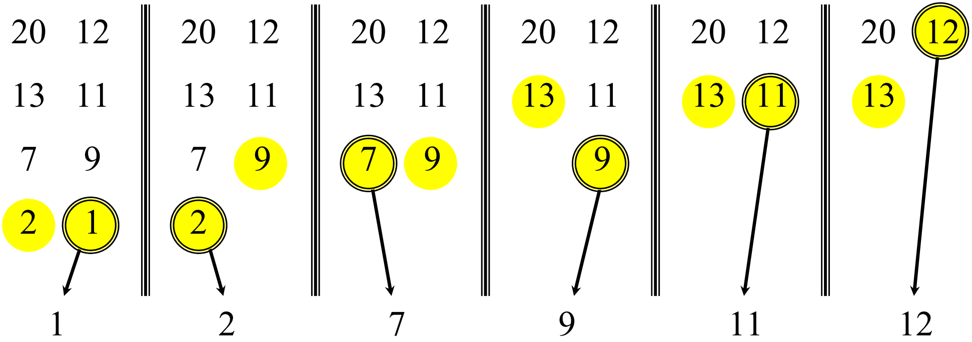




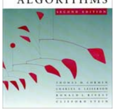
Merging two sorted arrays



Merging two sorted arrays



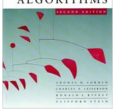
Time = $\Theta(n)$ to merge a total of n elements (linear time).



Analyzing merge sort

$T(n)$	MERGE-SORT $A[1 \,.\,.\, n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

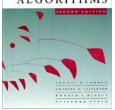
Sloppiness: Should be $T(\,\lceil n/2 \rceil) + T(\,\lfloor n/2 \rfloor\,)$, but it turns out not to matter asymptotically.



Recurrence for merge sort

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

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- We will see several ways starting with "Rec. Tree" to find a good upper bound on $T(n)$.



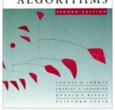
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

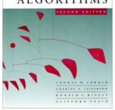


Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$cn$$

$$T(n/2) \qquad\qquad T(n/2)$$



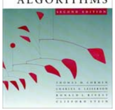
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$cn$$

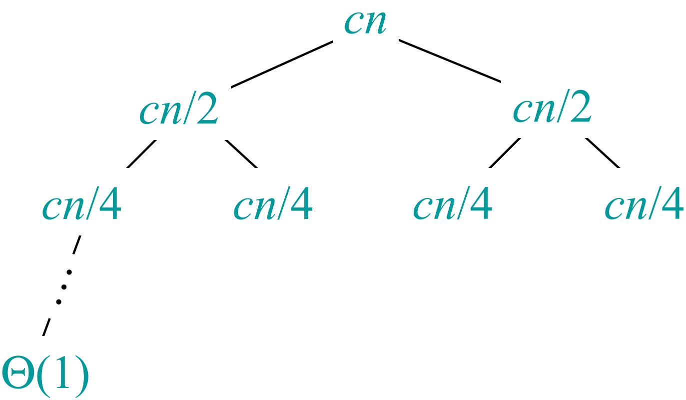
$$cn/2 \qquad\qquad cn/2$$

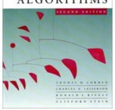
$$T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4)$$



Recursion tree

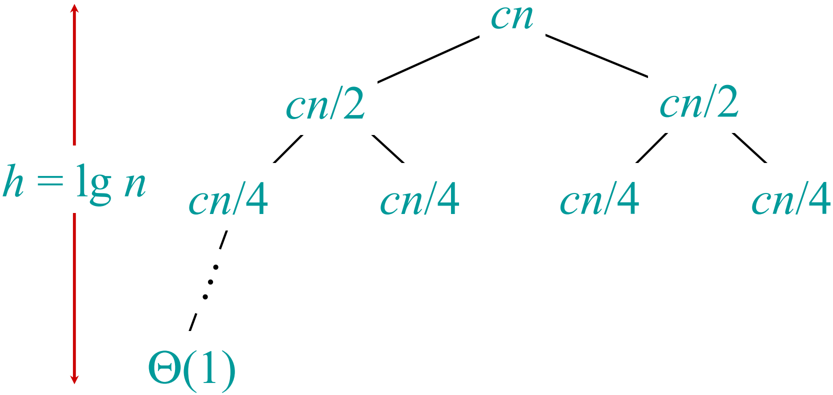
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

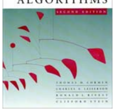




Recursion tree

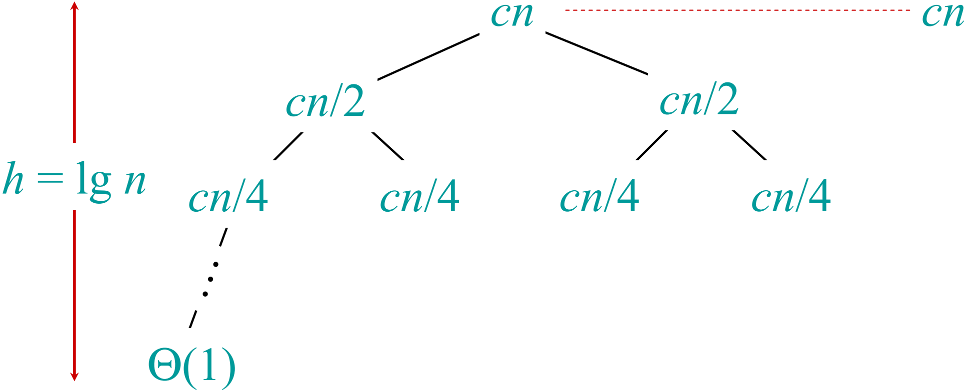
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

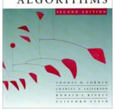




Recursion tree

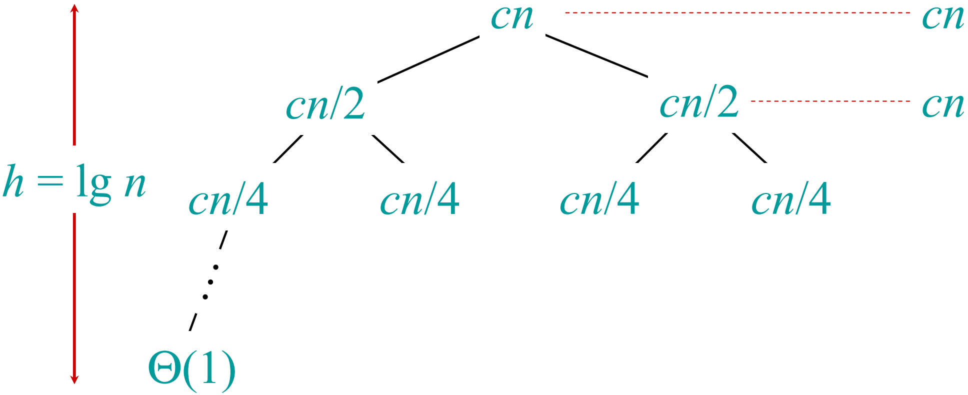
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

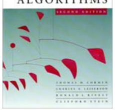




Recursion tree

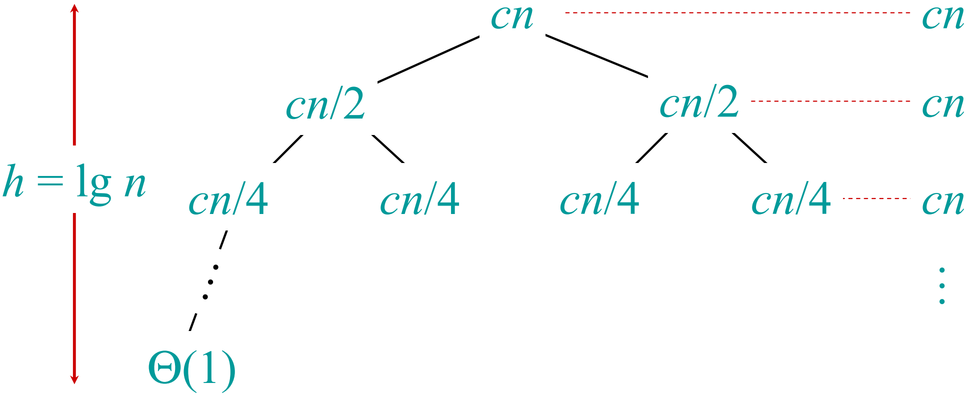
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

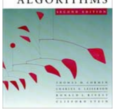




Recursion tree

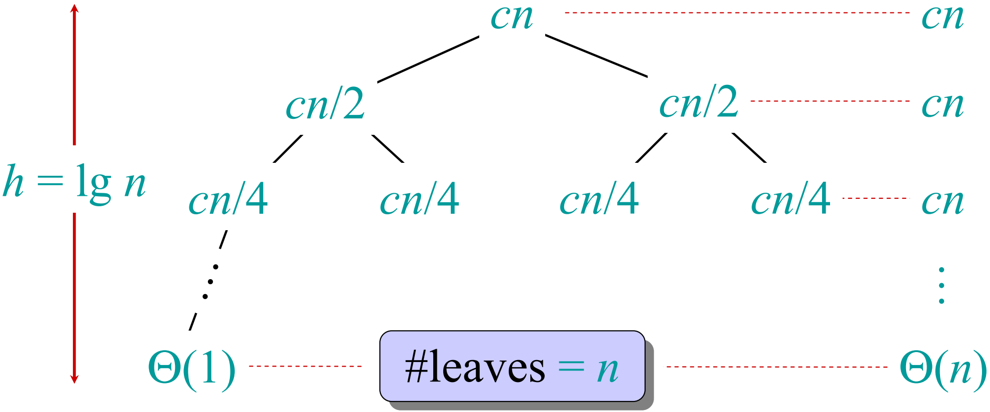
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

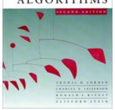




Recursion tree

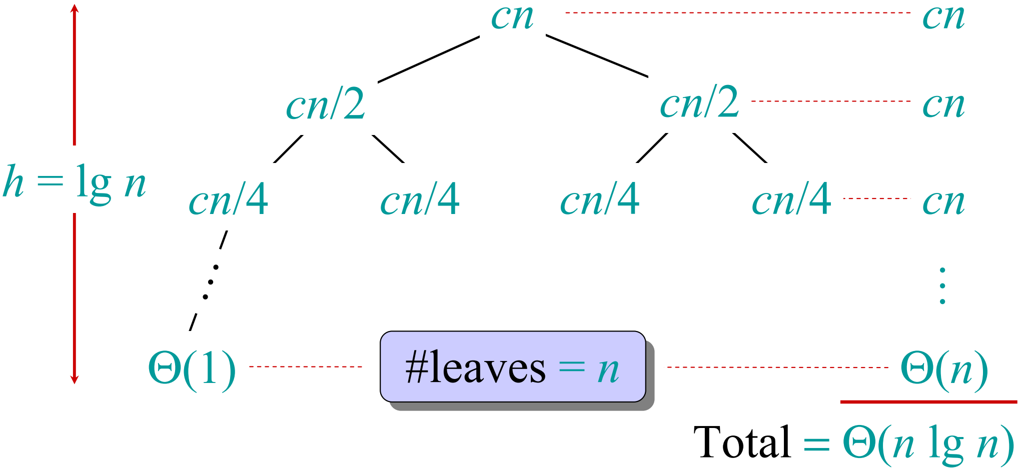
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.





Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Contents

Divide and Conquer
- Intro
- Merge Sort
- Recursion Tree for recurrences
- Integer Multiplication

Example

$$T(n) = 2T(3n/8) + n^2$$

- board

Contents

Divide and Conquer
- Intro
- Merge Sort
- Recursion Tree for recurrences
- Integer Multiplication

Integer Multiplication

Consider the problem of multiplying together two arbitrarily large numbers.

Integer Multiplication

Consider the problem of multiplying together two arbitrarily large numbers.

Input: positive integers x and y, each n bits long
Output: positive integer z where $z = x \cdot y$

Integer Multiplication

Consider the problem of multiplying together two arbitrarily large numbers.

Input: positive integers x and y, each n bits long
Output: positive integer z where $z = x \cdot y$

A straightforward approach using base-2 arithmetic, akin to how we multiply by hand, takes $\Theta(n^2)$ time.

Integer Multiplication

Consider the problem of multiplying together two arbitrarily large numbers.

Input: positive integers x and y, each n bits long
Output: positive integer z where $z = x \cdot y$

A straightforward approach using base-2 arithmetic, akin to how we multiply by hand, takes $\Theta(n^2)$ time.

Could we do better if we used results from subinstances?

Integer Multiplication

A Divide and Conquer approach can be considered to be a very large scale version of multiplication, only using base $2^{\lfloor \frac{n}{2} \rfloor}$ instead of a constant base.

Integer Multiplication

A Divide and Conquer approach can be considered to be a very large scale version of multiplication, only using base $2^{\lfloor \frac{n}{2} \rfloor}$ instead of a constant base.

For simplicity we assume that n is a power of 2, so $\frac{n}{2}$ will always be integer.

Integer Multiplication

A Divide and Conquer approach can be considered to be a very large scale version of multiplication, only using base $2^{\lfloor \frac{n}{2} \rfloor}$ instead of a constant base.

For simplicity we assume that n is a power of 2, so $\frac{n}{2}$ will always be integer.

So we split the bitstring for each of x and y in half, generating x_L, x_R, y_L, y_R such that

$$x = 2^{\frac{n}{2}} \cdot x_L + x_R$$
$$y = 2^{\frac{n}{2}} \cdot y_L + y_R\,.$$

Using x_L, x_R, y_L, y_R we can now express our multiplication of the n-bit integers as *four* multiplications of $\frac{n}{2}$-bit integers:

$$\begin{aligned} x \cdot y &= (2^{\frac{n}{2}} \cdot x_L + x_R) \cdot (2^{\frac{n}{2}} \cdot y_L + y_R) \\ &= 2^n \cdot x_L y_L + 2^{\frac{n}{2}} \cdot (x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Using x_L, x_R, y_L, y_R we can now express our multiplication of the n-bit integers as *four* multiplications of $\frac{n}{2}$-bit integers:

$$\begin{aligned} x \cdot y &= (2^{\frac{n}{2}} \cdot x_L + x_R) \cdot (2^{\frac{n}{2}} \cdot y_L + y_R) \\ &= 2^n \cdot x_L y_L + 2^{\frac{n}{2}} \cdot (x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Computing this with four half-size multiplications gives us a time recurrence of

$$T(n) \ = \ 4T\left(\frac{n}{2}\right) + cn$$

Computing this with four half-size multiplications gives us a time recurrence of

$$T(n) \;=\; 4T\left(\frac{n}{2}\right) + cn$$

since the operations to split the numbers and put the multiplication results together all take time linear in the number of bits:

- the numbers are split by bitshifting

Computing this with four half-size multiplications gives us a time recurrence of

$$T(n) \;=\; 4T\left(\frac{n}{2}\right) + cn$$

since the operations to split the numbers and put the multiplication results together all take time linear in the number of bits:

- the numbers are split by bitshifting
- combining the recursion results takes three addition operations and two bitshifts, all linear

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

$$T(n) = \sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} \cdot 4^i$$

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

$$T(n) = \sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} \cdot 4^i$$

$$= \ cn \cdot \sum_{i=0}^{\log_2 n} \frac{4^i}{2^i} \ = \ cn \cdot \sum_{i=0}^{\log_2 n} 2^i$$

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

$$T(n) = \sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} \cdot 4^i$$

$$= \; cn \cdot \sum_{i=0}^{\log_2 n} \frac{4^i}{2^i} \; = \; cn \cdot \sum_{i=0}^{\log_2 n} 2^i$$

A geometric series

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

$$T(n) = \sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} \cdot 4^i$$

$$= \ cn \cdot \sum_{i=0}^{\log_2 n} \frac{4^i}{2^i} \ = \ cn \cdot \sum_{i=0}^{\log_2 n} 2^i$$

A geometric series

$$= \ cn \cdot \frac{2^{\log_2 n+1} - 1}{2-1} \ = \ 2cn \cdot 2^{\log_2 n} - cn$$

To solve $T(n) = 4T(\frac{n}{2}) + cn$, we can use recursion tree analysis. Each instantiation makes four calls, each on half the size, and takes linear time otherwise, so:

$$T(n) = \sum_{i=0}^{\log_2 n} c \cdot \frac{n}{2^i} \cdot 4^i$$

$$= \; cn \cdot \sum_{i=0}^{\log_2 n} \frac{4^i}{2^i} \; = \; cn \cdot \sum_{i=0}^{\log_2 n} 2^i$$

A geometric series

$$= \; cn \cdot \frac{2^{\log_2 n+1} - 1}{2 - 1} \; = \; 2cn \cdot 2^{\log_2 n} - cn$$

$$= \; 2cn \cdot n - cn \; = \; 2cn^2 - cn \in \Theta(n^2)$$

Gauss's Method

Consider a different way of computing $(x_L y_R + x_R y_L)$, the coefficient of $2^{\frac{n}{2}}$.

Gauss's Method

Consider a different way of computing $(x_L y_R + x_R y_L)$, the coefficient of $2^{\frac{n}{2}}$.

We are already computing $x_L y_L$ and $x_R y_R$

Gauss's Method

Consider a different way of computing $(x_L y_R + x_R y_L)$, the coefficient of $2^{\frac{n}{2}}$.

We are already computing $x_L y_L$ and $x_R y_R$

Considering the binomial product

$$(x_L + x_R)(y_L + y_R) = x_L y_L + x_L y_R + x_R y_L + y_R x_R$$

we get that

$$x_L y_R + x_R y_L \ = \ (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

Gauss's Method

Consider a different way of computing $(x_L y_R + x_R y_L)$, the coefficient of $2^{\frac{n}{2}}$.

We are already computing $x_L y_L$ and $x_R y_R$

Considering the binomial product

$$(x_L + x_R)(y_L + y_R) = x_L y_L + x_L y_R + x_R y_L + y_R x_R$$

we get that

$$x_L y_R + x_R y_L \ = \ (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

This might be better because we already compute $x_L y_L$ and $x_R y_R$

So the recursive algorithm is:

- first compute x_L, x_R, y_L, y_R and $x_L + x_R, y_L + y_R$ in linear time

So the recursive algorithm is:

- first compute x_L, x_R, y_L, y_R and $x_L + x_R, y_L + y_R$ in linear time
- then calculate $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ recursively

So the recursive algorithm is:

- first compute x_L, x_R, y_L, y_R and $x_L + x_R, y_L + y_R$ in linear time
- then calculate $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ recursively
- and assemble the results in linear time

So the recursive algorithm is:

- first compute x_L, x_R, y_L, y_R and $x_L + x_R, y_L + y_R$ in linear time
- then calculate $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ recursively
- and assemble the results in linear time

Using this approach, we make *three* recursive calls, each of size $\frac{n}{2}$, yielding the time recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

So the recursive algorithm is:

- first compute x_L, x_R, y_L, y_R and $x_L + x_R, y_L + y_R$ in linear time
- then calculate $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ recursively
- and assemble the results in linear time

Using this approach, we make *three* recursive calls, each of size $\frac{n}{2}$, yielding the time recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

Except that's not quite right. What we actually have is

$$T(n) = 2T\left(\frac{n}{2}\right) + T(\frac{n}{2} + 1) + O(n)$$

Solving the recurrence (board)

$$T(n) = 2T\left(\frac{n}{2}\right)+T(\frac{n}{2}+1)+O(n)$$

- Does the $+1$ make any difference? Probably not, but how to be sure?