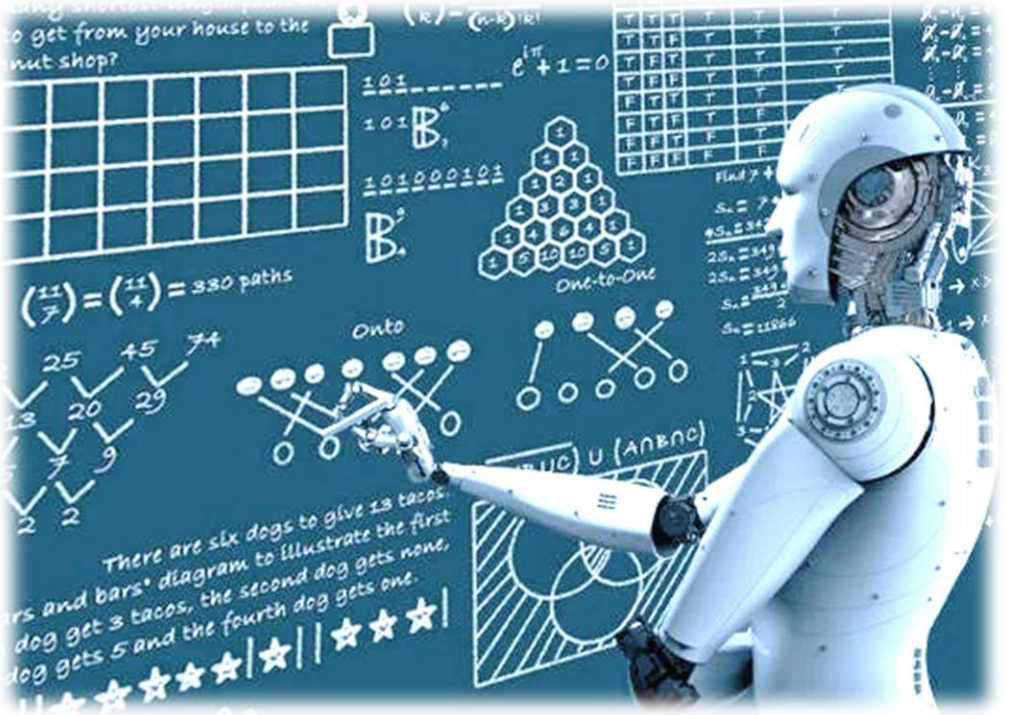
ALGORITHMS PART I

Sid Chi-Kin Chau

[Lecture 5]

Why are Algorithms Important?

- Major functions of computers
  - Problem solving
  - Data processing
  - Computing, etc.
- Programming is more than correctness
  - We have to write “efficient” code to solve problems with *fast running time* and *small computational resources*
- Algorithms are systematic ways of solving specific problems
  - Example: How to sort, search, and process data



Recap from Data Structures

- Data structures are optimized for efficient data operations
  - Fast search, insertion, deletion, enumeration
- Consider searching an item with a given key
  - **Linked list**
    - Search sequentially in the list
    - Take at most *n* steps
  - **Binary search tree**
    - Search recursively in one of the sub-trees
    - Faster on average, still take at most *n* steps in an unbalanced tree
  - **Red-black tree**
    - Search recursively in one of the sub-trees in always balanced tree
    - Take at most log(*n*) steps

Goals of This Lecture

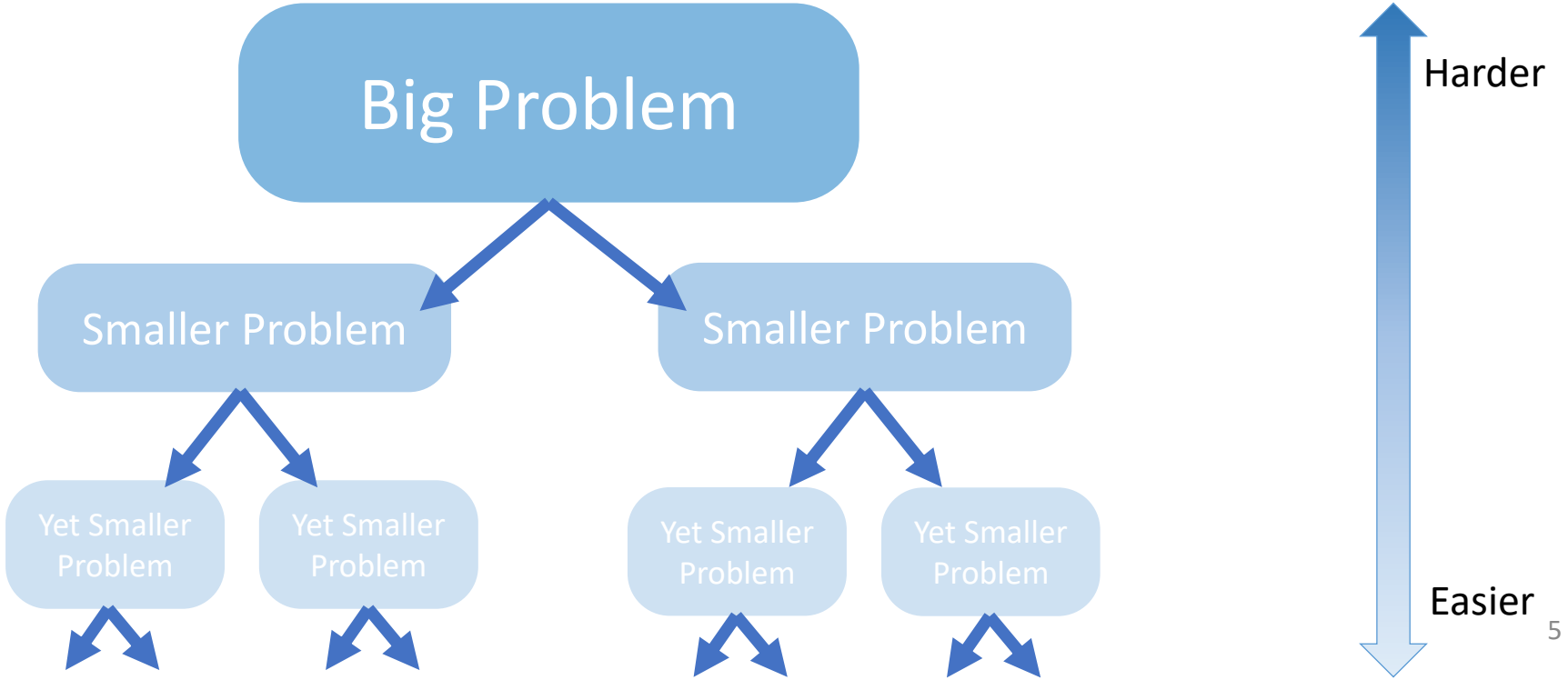
- Learn a general pattern for efficient algorithms
  - "Divide and conquer"
    - Reduce to simpler sub-problems and solve sub-problems recursively
    - Intuitive approach
    - Often quite efficient
  - Applications
    - Binary search
    - Tower of Hanoi
    - Merge sort
    - Karatsuba multiplication
- Something else to learn
  - How to compare the efficiency of different algorithms

Divide and Conquer Paradigm

- "Nothing is particularly hard if you divide it into small jobs"

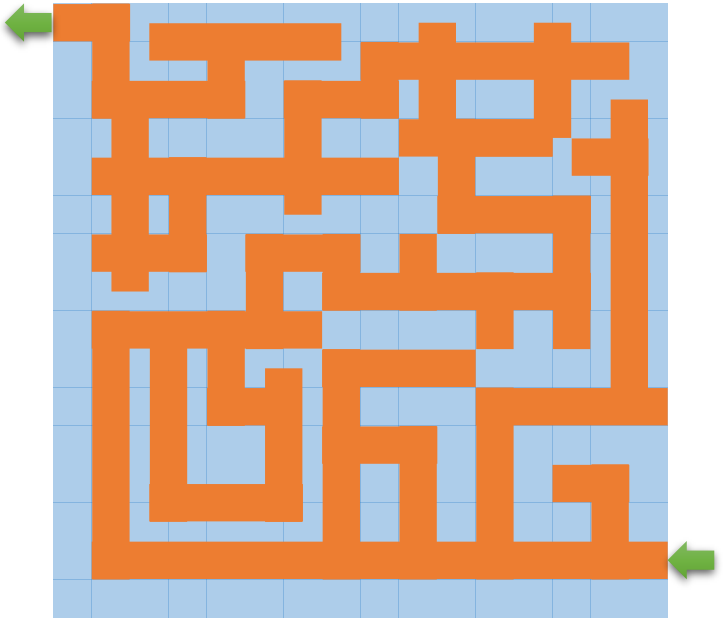
  Henry Ford

- Idea: break up a problem into smaller (easier) sub-problems

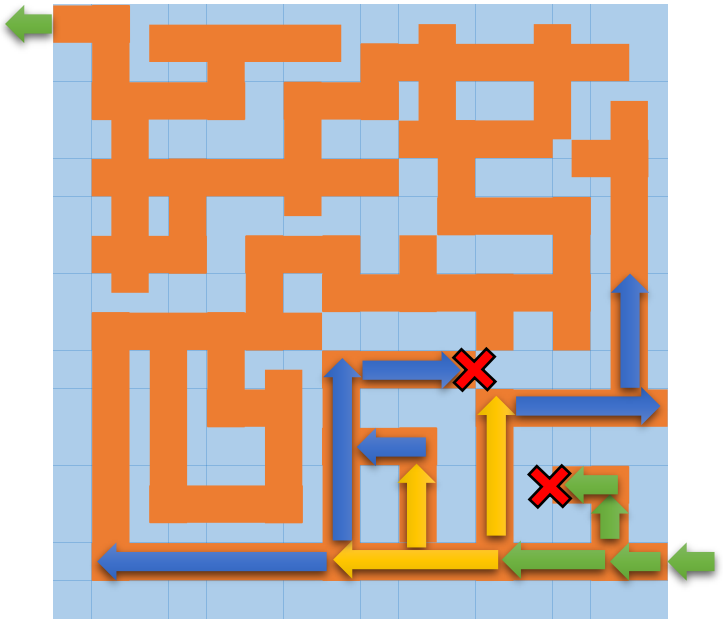


General Search Algorithms

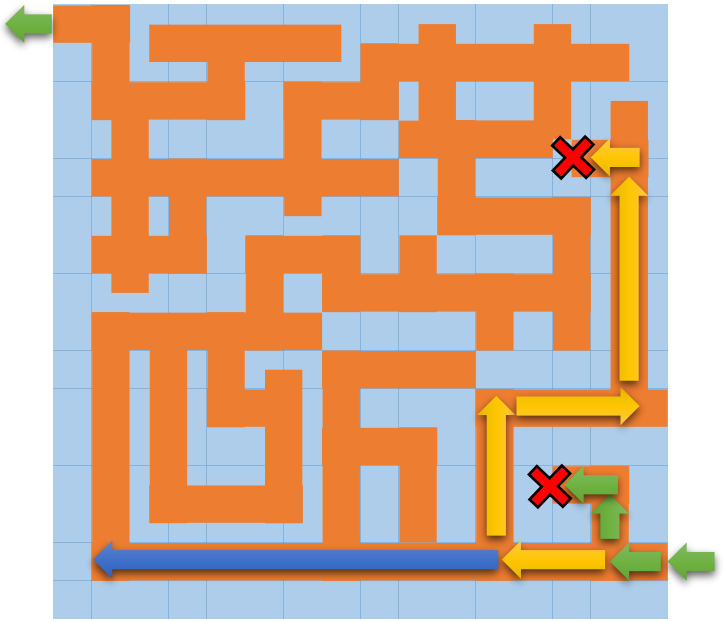
- Search something in an **unstructured** space
- Specifically, search in a Maze
  - Each step gives a result
    - Found or not
  - Options of next action
    - Walk North, South, East, West
  - Able to backtrack
- **Breath-first search**
  - Explore all possible options recursively at each step
- **Depth-first search**
  - Explore only one option at each step until hitting a dead-end, then backtrack and explore another option



- Let *n* be the steps of the longest path
- **Breath-first search**
  - Explore all possible options recursively at each step; Take at most *n* steps
- **Depth-first search**
  - Explore only one option at each step until hitting a dead-end, then backtrack and explore another option; Take at most *n* steps

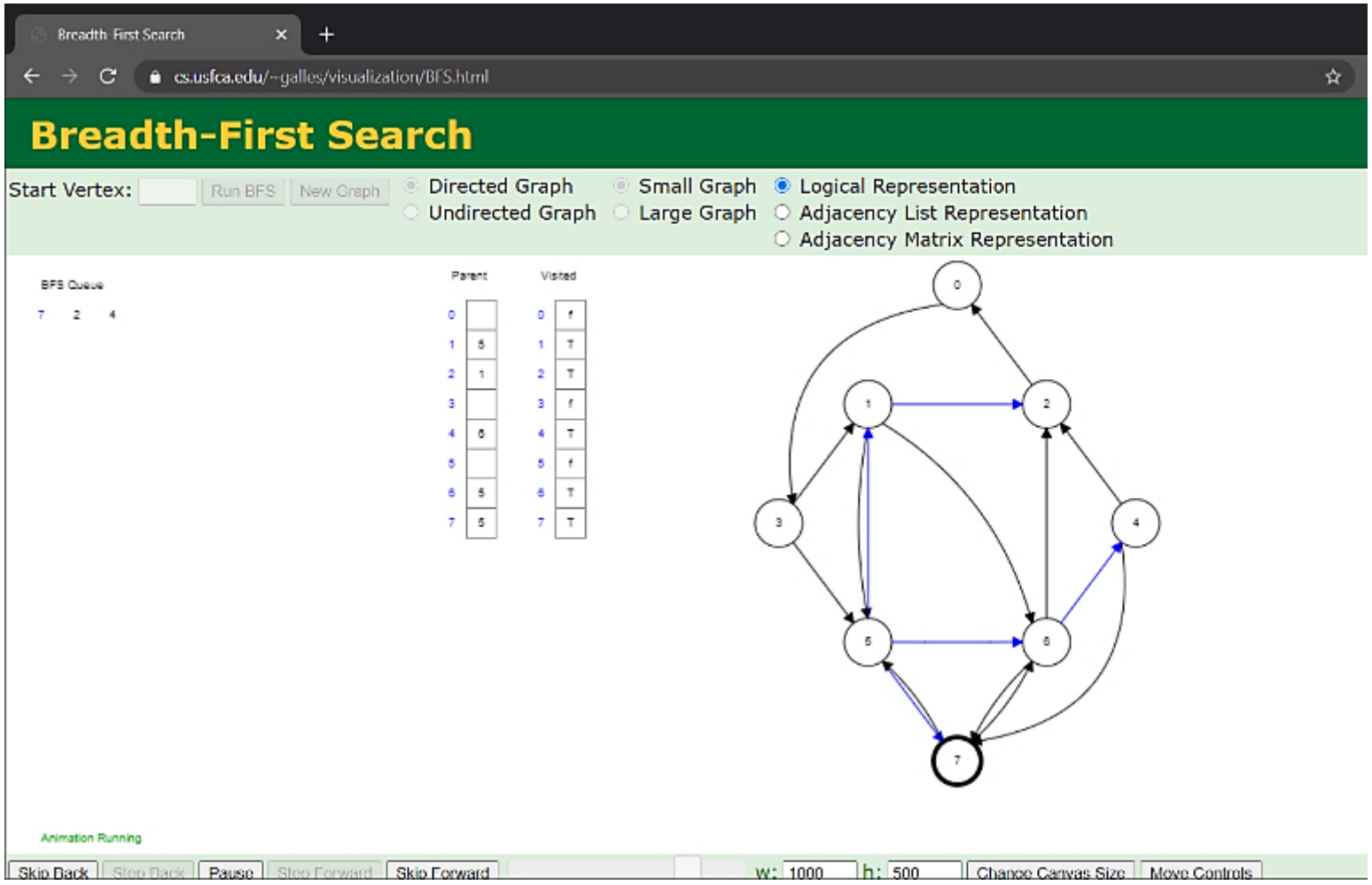


Breath-first search



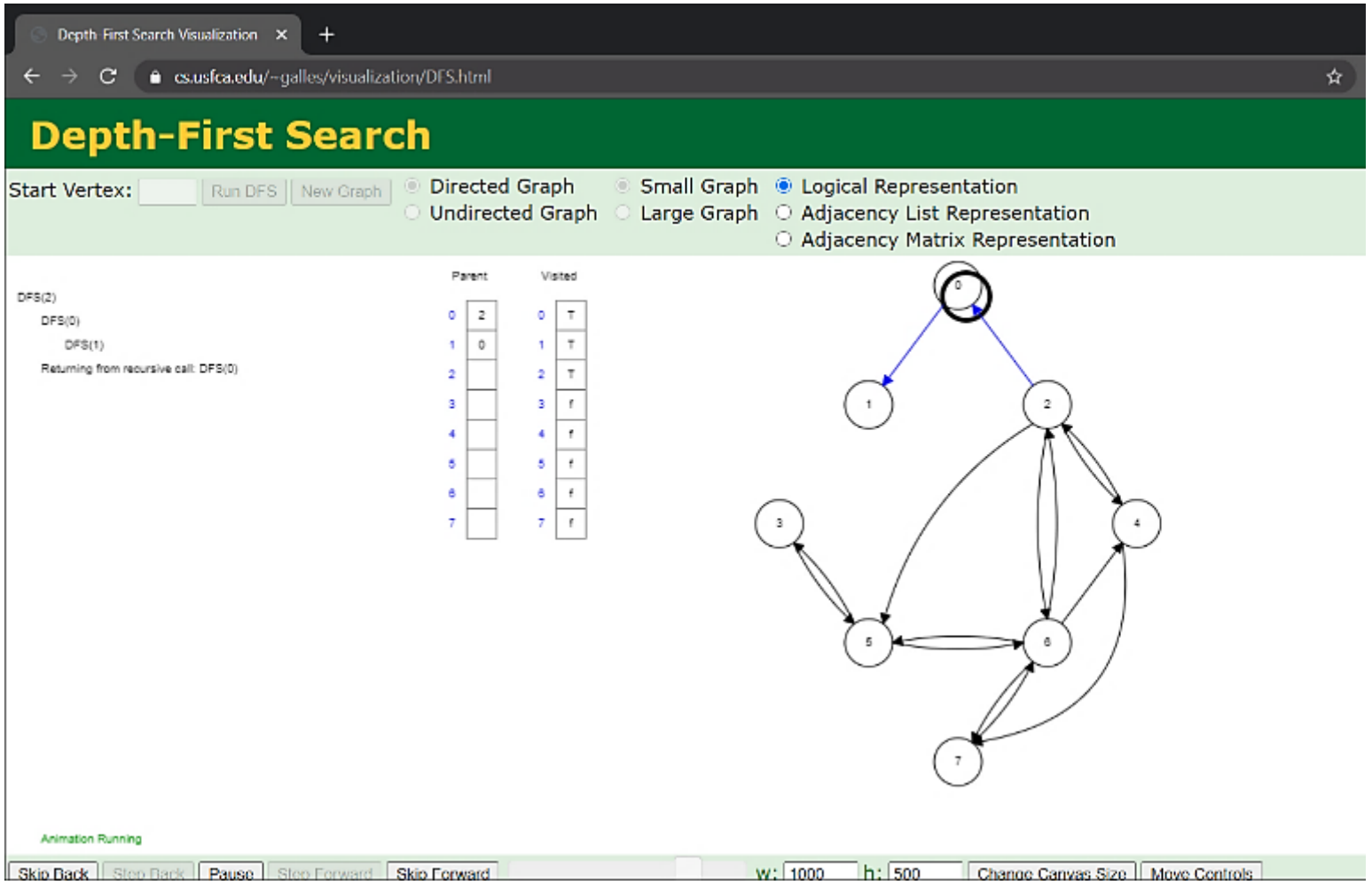
Depth-first search

Demo



https://www.cs.usfca.edu/~galles/visualization/BFS.html

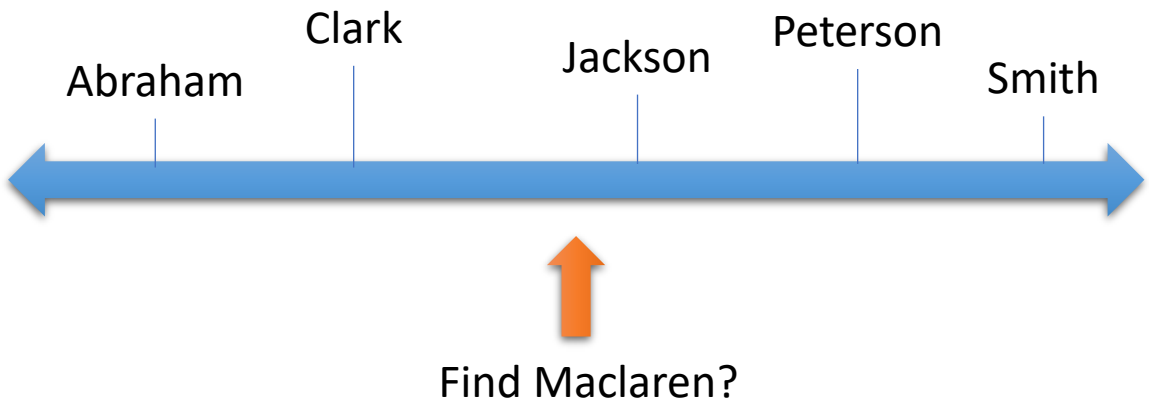
Demo



https://www.cs.usfca.edu/~galles/visualization/DFS.html

Search in "Structured" Space

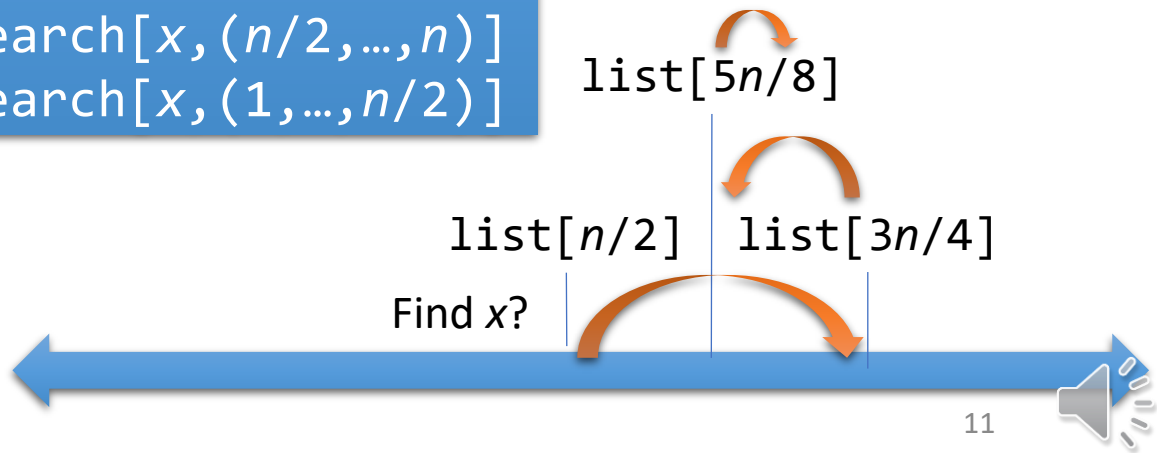
- Consider an ordered list according to certain IDs
  - Examples
    - Phone book (ID: name)
    - Dictionary (ID: vocabulary)
    - Database (ID: record ID)
  - Search an item in a linearly ordered list
    - Find a person record in a database



Binary Search

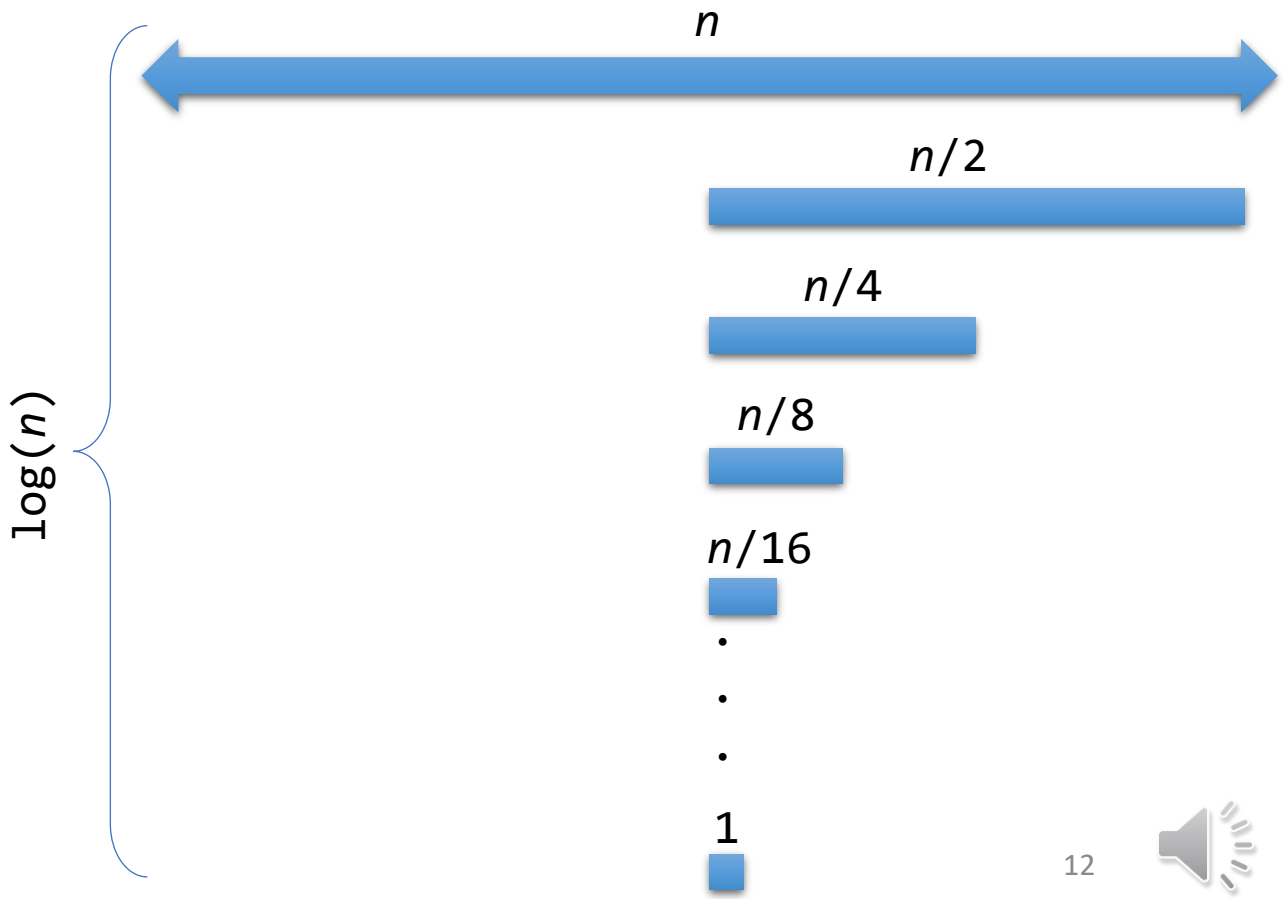
- Let the *n*-th item in the list be `list[n]`

```
BinarySearch[x,(1,…,n)]
//Search x in a list with list[1]<…<list[n]

Start at list[n/2]     //The middle item in the list
Is x = list[n/2]?
  If Yes, Then return list[n/2]
  If No, Then
     If x > list[n/2], return BinarySearch[x,(n/2,…,n)]
     If x < list[n/2], return BinarySearch[x,(1,…,n/2)]
```



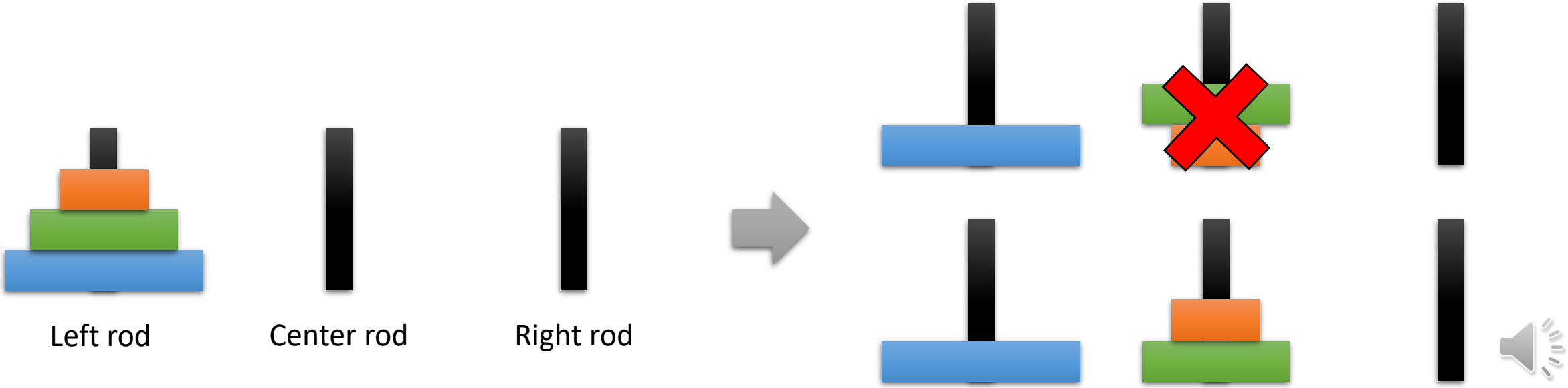
Binary Search

- Let the *n*-th item in the list be `list[n]`
- Binary search reduces the search range iteratively
- Divide the search range by half at each step
  - Until only 1 item in the range
- It takes at most log(*n*) steps
  - $2^{\log(n)} = n$, where log is of base 2
- Binary search is only possible for structured search space
  - Example: linearly ordered list



Tower of Hanoi

- Consists of three rods and *n* disks of various sizes
- *Goal*: Move all disks from one rod to another rod
- *Rule*: Only smaller disks can be moved on top of a bigger disk
- How to accomplish the goal? How many steps with *n* disks?

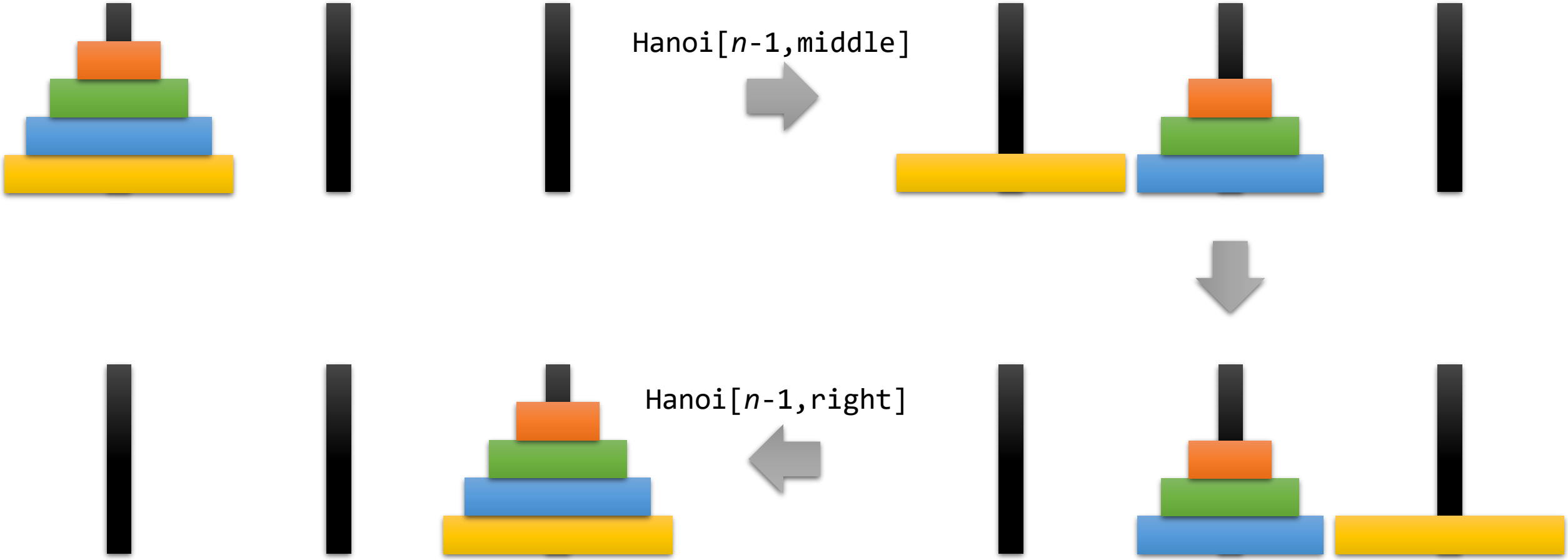


Tower of Hanoi

- Let Hanoi[*n*,right] be the algorithm that moves the smallest *n* disks to the right rod
  - Hanoi[*n*-1,right] solves the subproblem that moves the smallest *n*-1 disks

```
Hanoi[n, r]
//Move the smallest n disks to rod r

If n > 1 Then
  Call Hanoi[n-1,middle] //Move the smallest n-1 disks to the middle rod
  Move the remaining n-th disk to the right rod
  Call Hanoi[n-1, r]  //Move the smallest n-1 disks to the right rod
Else
  Move the smallest disk to rod r
```

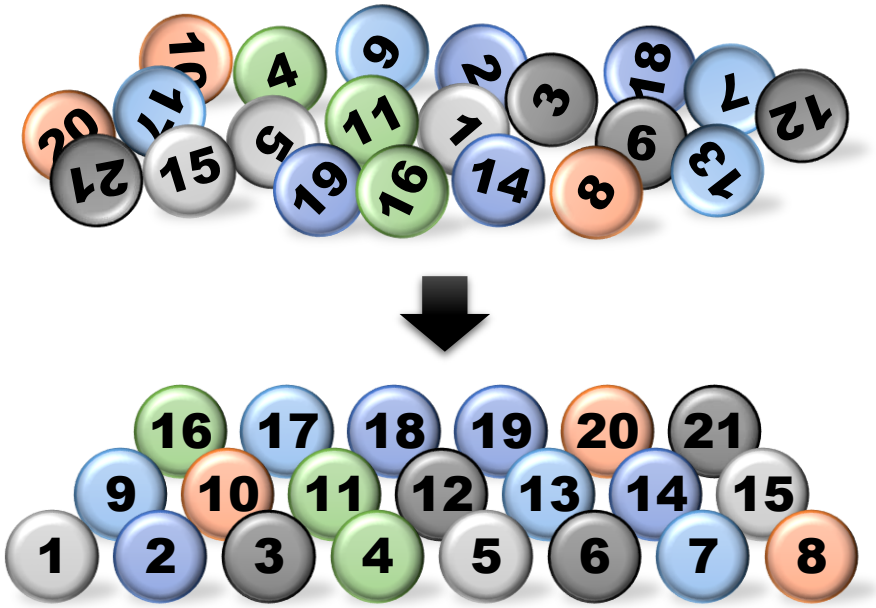
Hanoi[*n*,right]



- How many steps are needed for Hanoi[*n*,right]?
  - At most 2^n steps

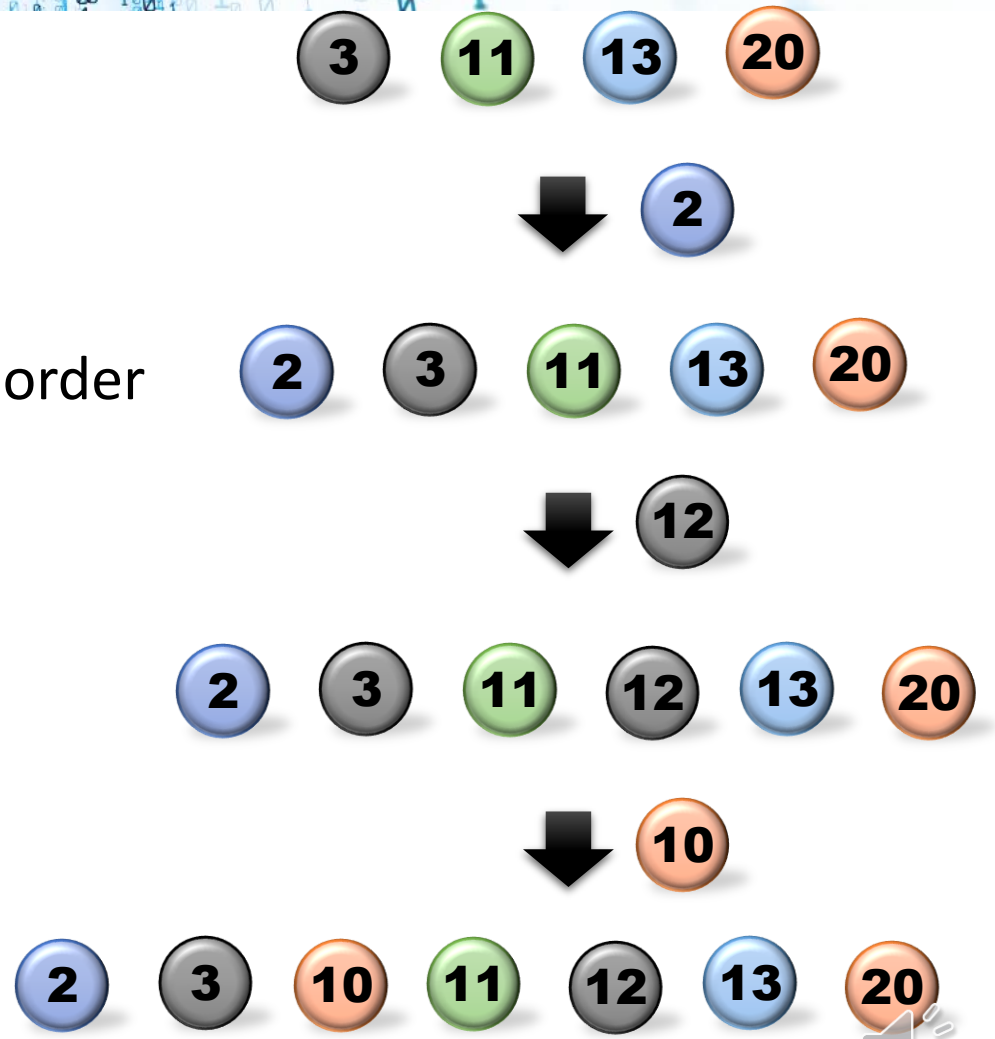
How to Sort

- Sorting is one of most important operations
  - Find the smallest, largest, *k*-th ordered, median
  - Produces structured data spaces (for searching)
- Need efficient systematic methods for sorting
- Key Questions:
  - How do we sort efficiently?
  - What is the minimum running time for sorting?
  - How much memory resource does it need?



Insertion Sort

- Insertion sort:
  - Create an empty list
  - Insert an item one by one in the list
  - The inserted item must be placed in the sorted order in the list
- What is the running time
  - The *k*-th inserted item will have at most (*k*-1) comparisons with the existing items in the list
  - Hence, the total number of comparison is
    - $T(n) = 1 + 2 + \ldots + n = n(n-1)/2$

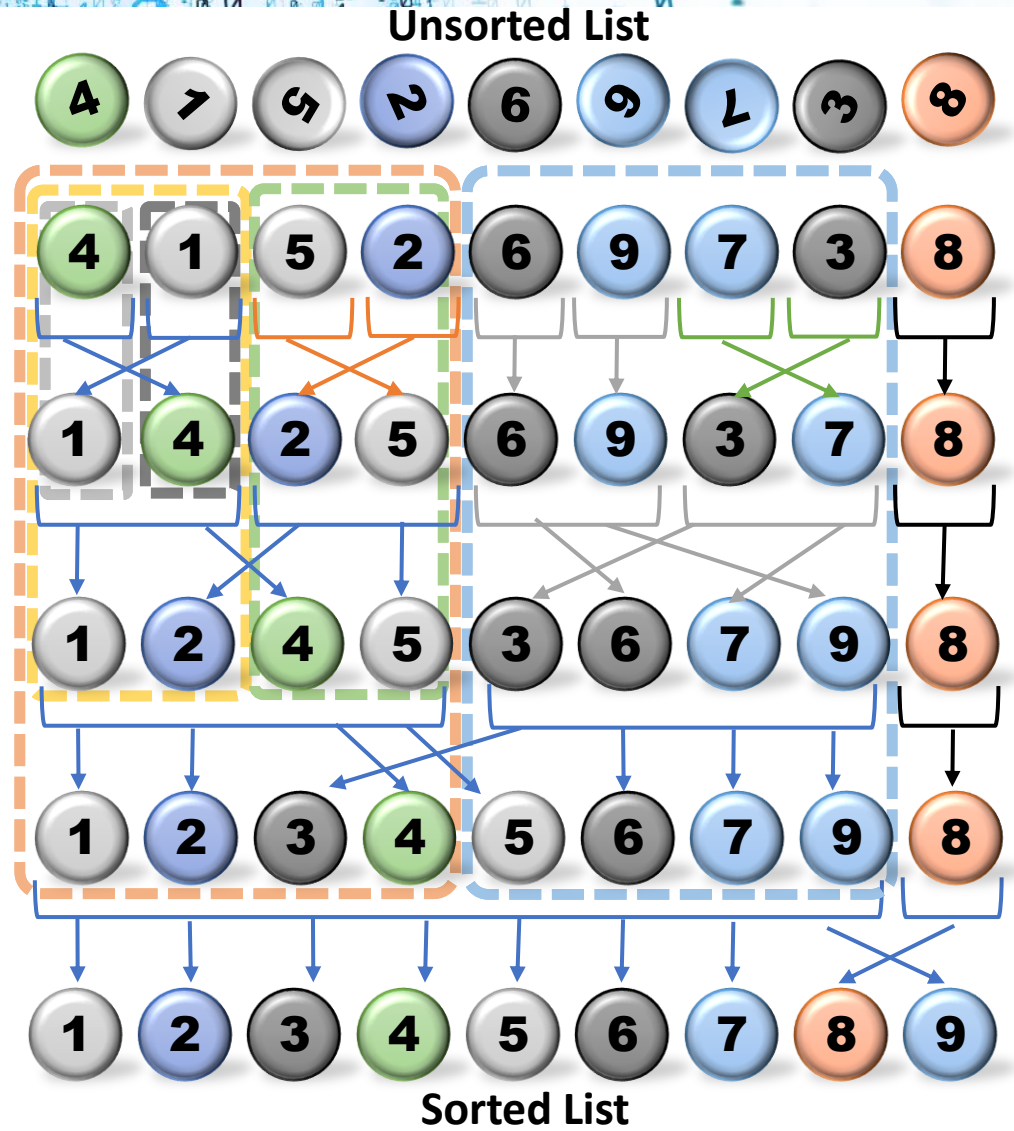


Merge Sort

- Sort a list of *n* items
  - Divide into two sublists of *n*/2 items
  - Sort each sublist of *n*/2 items
    - Divide into two sublists of *n*/4 items
    - Sort each sublist of *n*/4 items
      - Divide into two sublists of *n*/8 items
      - Sort each sublist of *n*/8 items
        - ……
          - Divide into two sublists of 1 item
          - Merge two sublists of 1 item
        - ……
      - Merge two *n*/8-item sublists
    - Merge two *n*/4-item sublists
  - Merge two *n*/2-item sublists

Merge Sort

- Sort a list of *n* items
  - Divide into two sublists of *n*/2 items
  - Sort each sublist of *n*/2 items
    - Divide into two sublists of *n*/4 items
    - Sort each sublist of *n*/4 items
      - Divide into two sublists of *n*/8 items
      - Sort each sublist of *n*/8 items
        - ……
          - Divide into two sublists of 1 item
          - Merge two sublists of 1 item
        - ……
      - Merge two *n*/8-item sublists
    - Merge two *n*/4-item sublists
  - Merge two *n*/2-item sublists



```
MergeSort[1,…,n]
//Sort a list of list[1],…,list[n]

//Sort each sublist
list1 ← MergeSort[1,…,n/2]
list2 ← MergeSort[n/2+1,…,n]

Return Merge[list1, list2]
```

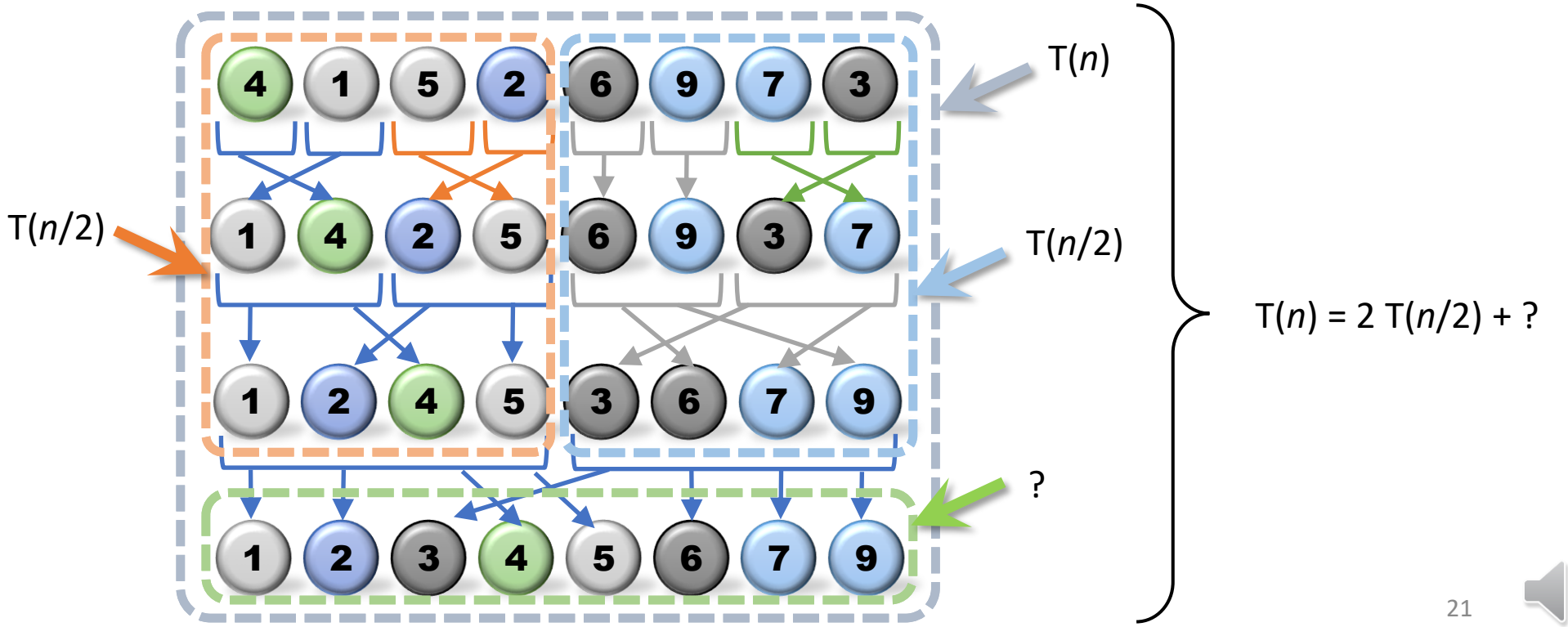
- Assumption
  - *n* is even at each step
  - Simple modification for odd *n*

```
Merge[list1, list2]
//Merge two sorted lists list1 and list2

i ← 1, j ← 1
Do
  If list1[i] < list2[j] Then
     Insert list1[i] into newlist
     i ← i+1
  Else
     Insert list2[j] into newlist
     j ← j+1
While i ≤ Length[list1] and j ≤ Length[list2]

If j > Length[list2] Then
  Insert the rest of list1 into newlist
Else
  Insert the rest of list2 into newlist

Return newlist
```


Running Time

- Comparison

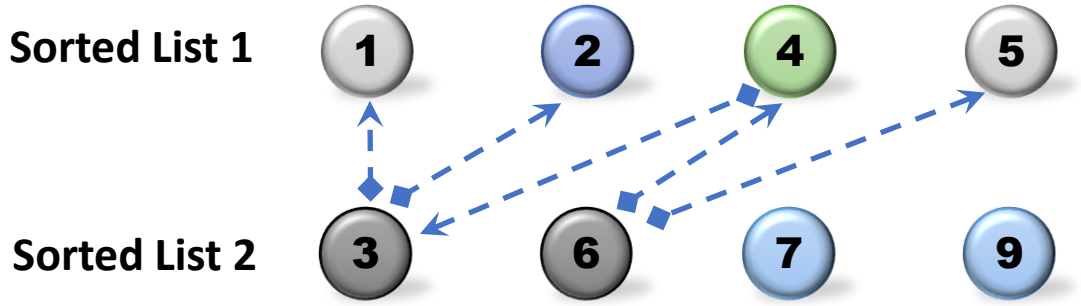
T(n)	Running time for sorting a list of n items
T(n/2)	Running time for sorting a list of n/2 items
?	Running time for merging two lists of n/2 sorted items



Running Time

- *How to merge two sorted lists into a sorted list*?
  1. Compare the smallest unmerged elements in the lists
  2. Insert the smaller element into the new list
  3. Repeat until all elements are merged

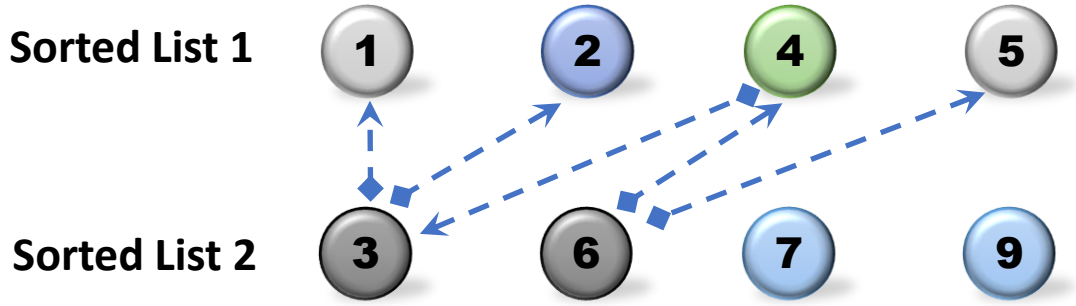
T(n)	Running time for sorting a list of n items
T(n/2)	Running time for sorting a list of n/2 items
?	Running time for merging two lists of n/2 sorted items



Running Time

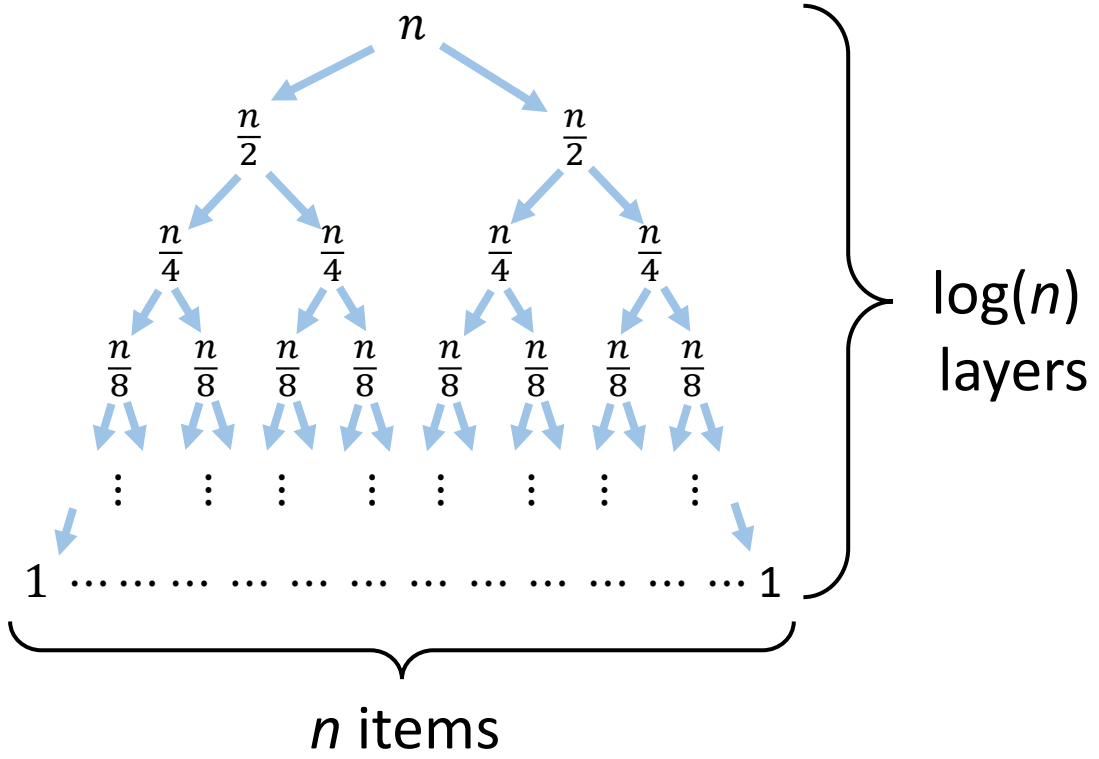
- *What is the running time for merging two sorted lists*?
  1. Running time is proportional to the numbers of comparisons and insertions
  2. Number of insertions = *n*, Number of comparisons = *n*/2
  3. Total running time is proportional to *n*

T(*n*)	Running time for sorting a list of *n* items
T(*n*/2)	Running time for sorting a list of *n*/2 items
n	Running time for merging two lists of *n*/2 sorted items



Proof of Running Time

$$T(n) = 2\,T(n/2) + n$$
$$= 4\,T(n/4) + 2(n/2) + n$$
$$= 4\,T(n/4) + 2n$$
$$\ldots$$
$$= n\,T(1) + \log(n)\,n$$
$$\leq C\,n\log(n)$$



$T(n)$	Running time for sorting a list of n items
$T(n/2)$	Running time for sorting a list of $n/2$ items
n	Running time for merging two lists of $n/2$ sorted items

Comparison of Running Time

	Worst-case Running Time
Merge Sort	$C\,n\log(n)$
Quick Sort	$C\,n^2$
Bubble Sort	$C\,n^2$
Heap Sort	$C\,n\log(n)$
Shell Sort	$C\,n\log^2(n)$
Insertion Sort	$C\,n^2$

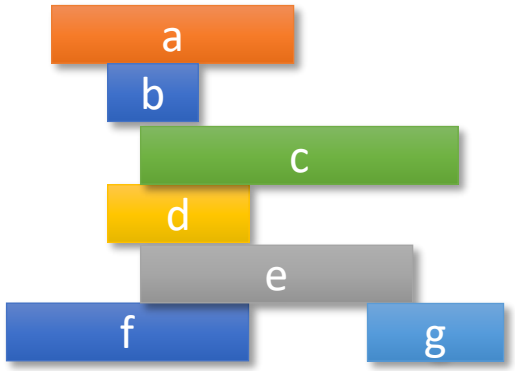
Some Reflections

- Why Divide-and-Conquer works**:**
  1. **Recurrent sub-structures**:
     - Possible to break into smaller sub-problems
     - E.g., sorting *n* items reduces to sorting *n*/2 items
  2. **Compact memory size**:
     - Need small memory to store results of sub-problems
     - E.g., Need only *n*-array to store *n* items
  3. **Efficient merging**:
     - Merging sub-problems is efficient
     - E.g., Merge 2 sub-lists of *n*/2 sorted items takes at most *n* steps

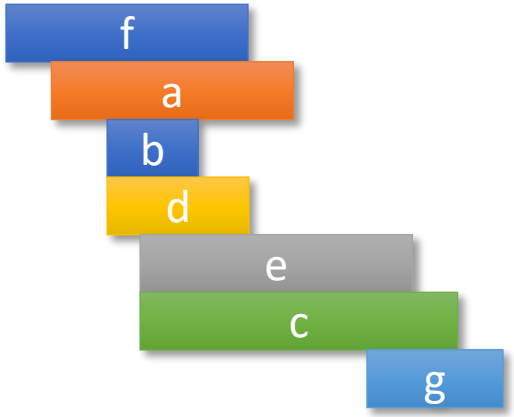
How About : Merge Sort for Intervals?

- **Intervals**
  - Earlier starting times always appear first
  - For the same starting times, earlier ending times appear first
  - *How do we sort a list of intervals*?

Unsorted Intervals



Sorted Intervals



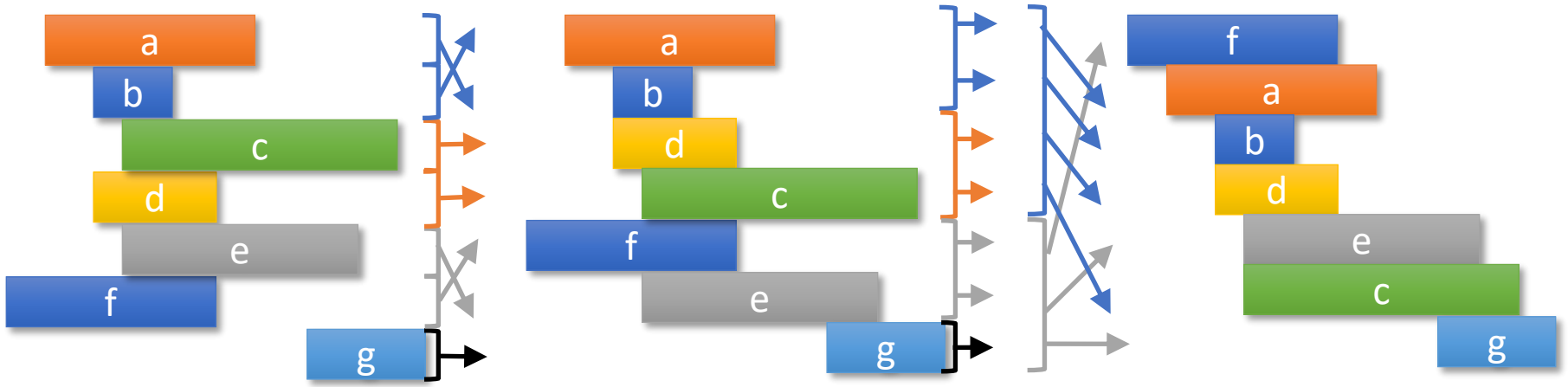
How About : Merge Sort for Intervals?

- Interval a $\leq$ Interval b $\Leftrightarrow$

  (a.*start-time* $\leq$ b.*start-time*) **or**

  (a.*start-time* = b.*start-time* **and** a.*end-time* $\leq$ b.*end-time*)

Running time: $T(n) = 2\,T(n/2) + n \leq C\,n \log(n)$

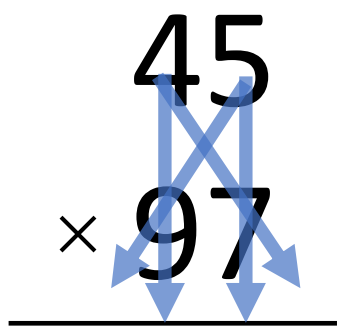


How to Multiply

- Multiplication is fundamental
  - Traditional approach
    - Digital-wise multiplication and multiplication table look-up
    - Not efficient for large-number multiplication
  - Efficient large-scale multiplication is needed
    - Computer graphics, e.g., 3D ray tracing
    - Cryptography, e.g., RSA cryptography
- Key Questions:
  - How do we multiply efficiently?
  - Any better way beyond simple multiplication table look-up?
  - Surprisingly, divide-and-conquer can apply well

Integer Multiplication 101

- Assumption: we skip the simple operations, e.g., additions and 1-digit multiplications (by multiplication table look-up)
- Long multiplication algorithm



Integer Multiplication 101

- Consider multiplying a pair of large numbers
- Long multiplication algorithm

$$\begin{array}{rr} & 1234567895931413 \\ \times & 4563823520395533 \\ \hline \end{array}$$

Integer Multiplication 101

- How many 1-digital multiplications are needed?
- There are about n^2 1-digit multiplications

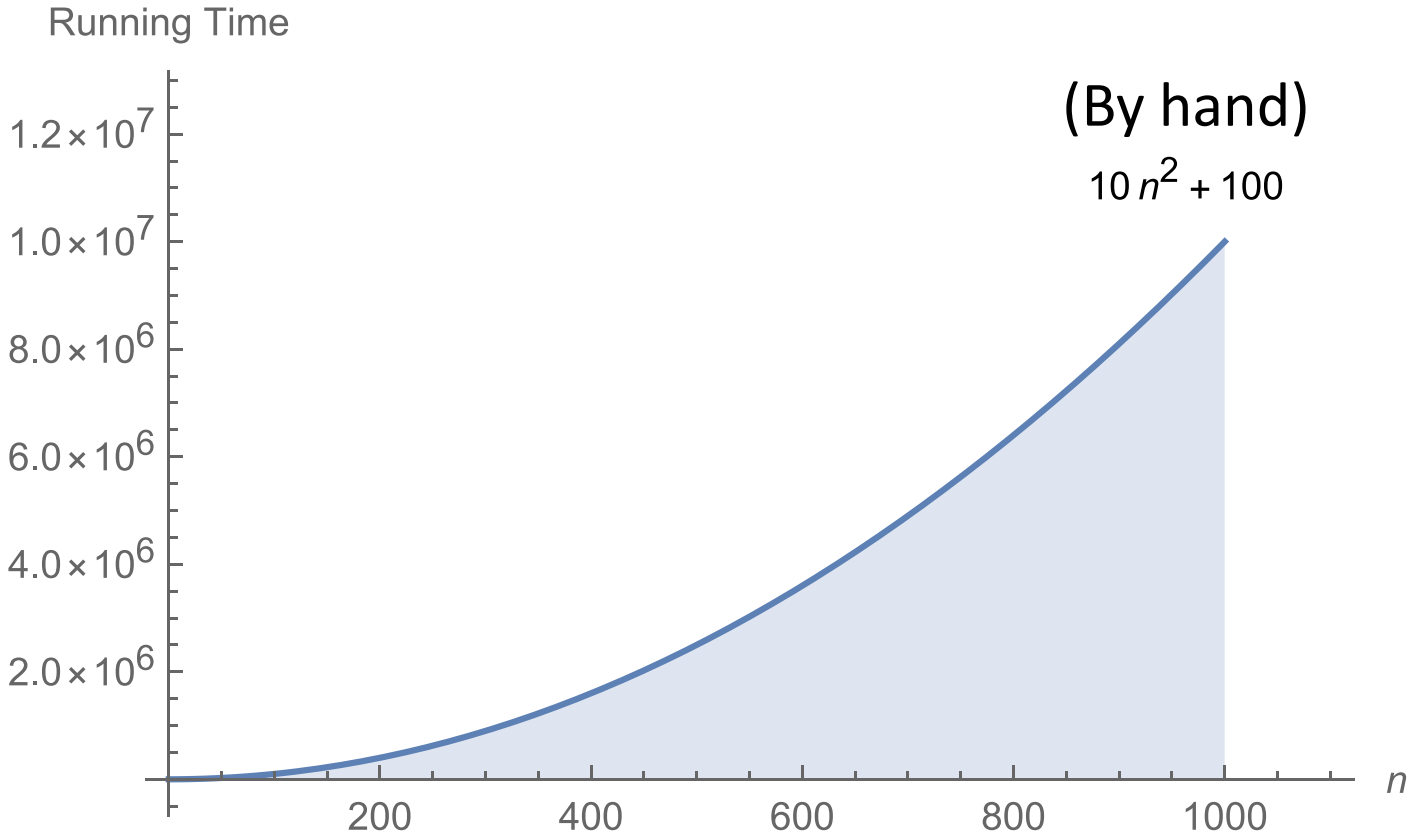
$$
\begin{array}{r}
\overbrace{1233925720752752384623764283568364918374523856298}^{n} \\
\times\ 456232358234239528562346723501913075013535001375 3 \\
\hline
\end{array}
$}

Measure of Running Time

- How do we measure the running time of an algorithm?
- Do we need test on different platforms?
  - Java? Python? C++?
  - How about future platforms?
- Can’t depend on the current platforms
- We should measure how the running time scales with the size of input
  - Big-O notation

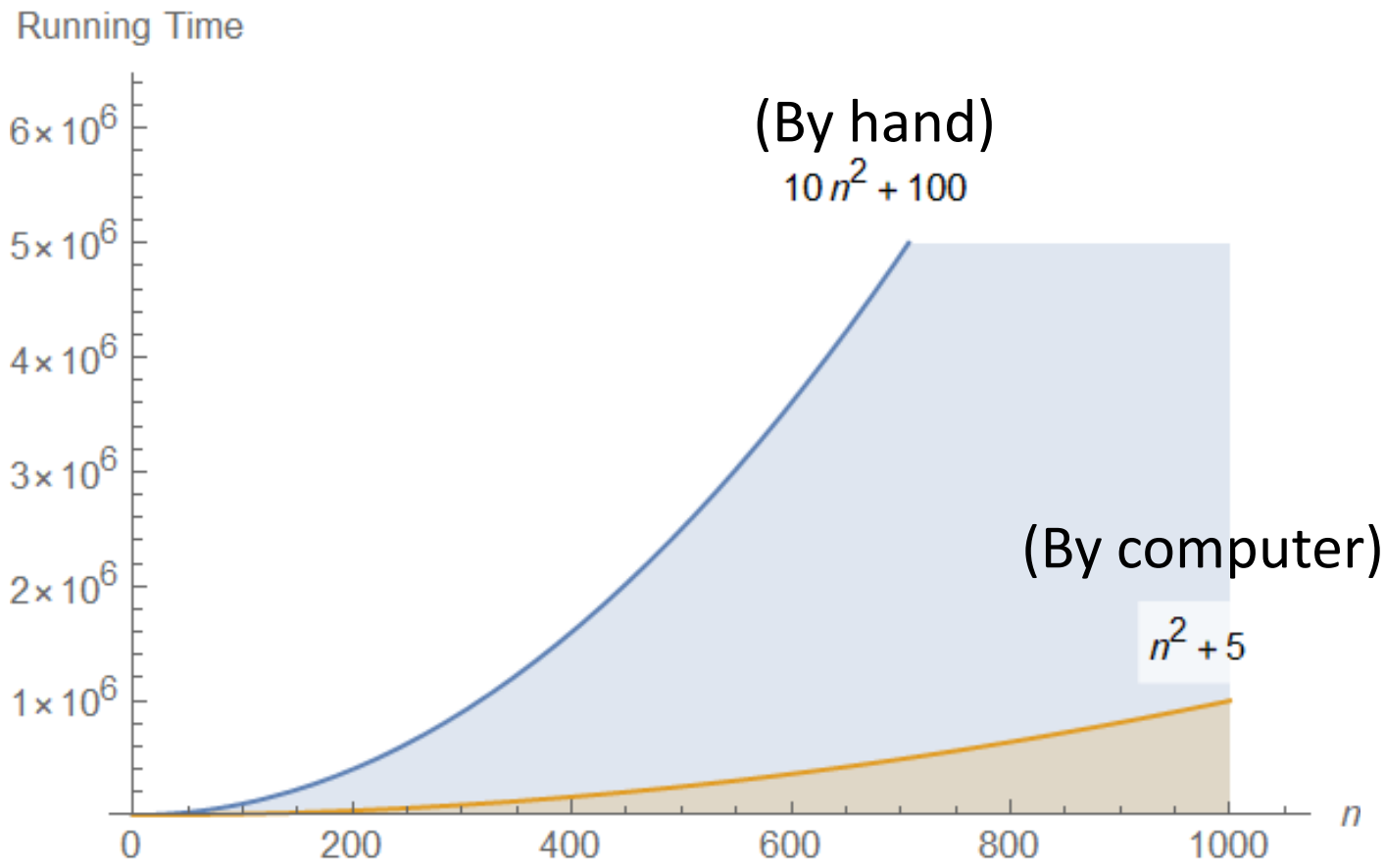
Measure of Running Time

- Rough approximation of running time of long multiplication algorithm by hand



Measure of Running Time

- Both scales like n^2 either way

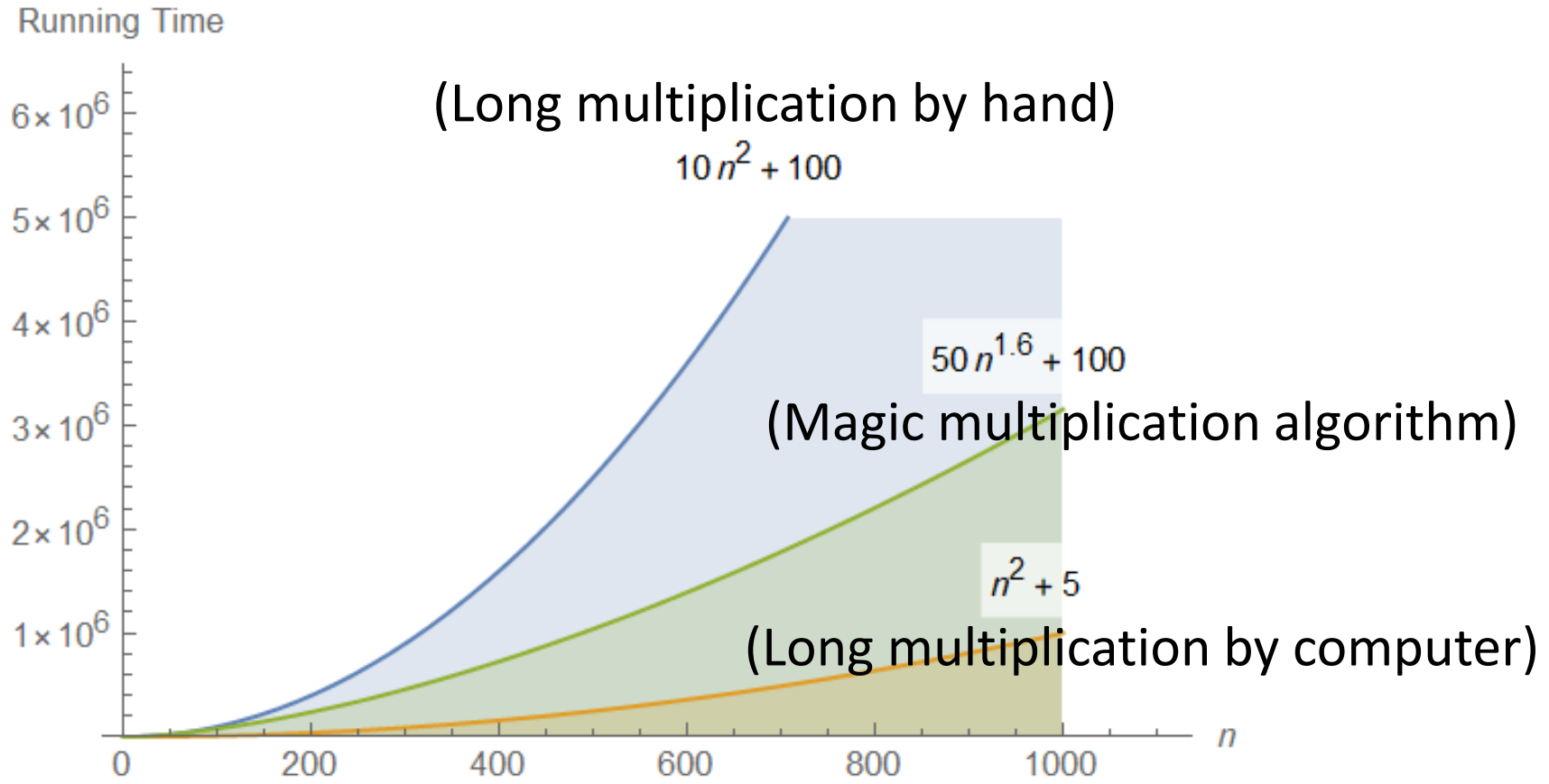


Asymptotic Analysis

- How does the runtime scale with the size of the input?
  - Running time of long multiplication algorithm scales like n^2
- You will learn how to use a formal notation (Big-O notation)

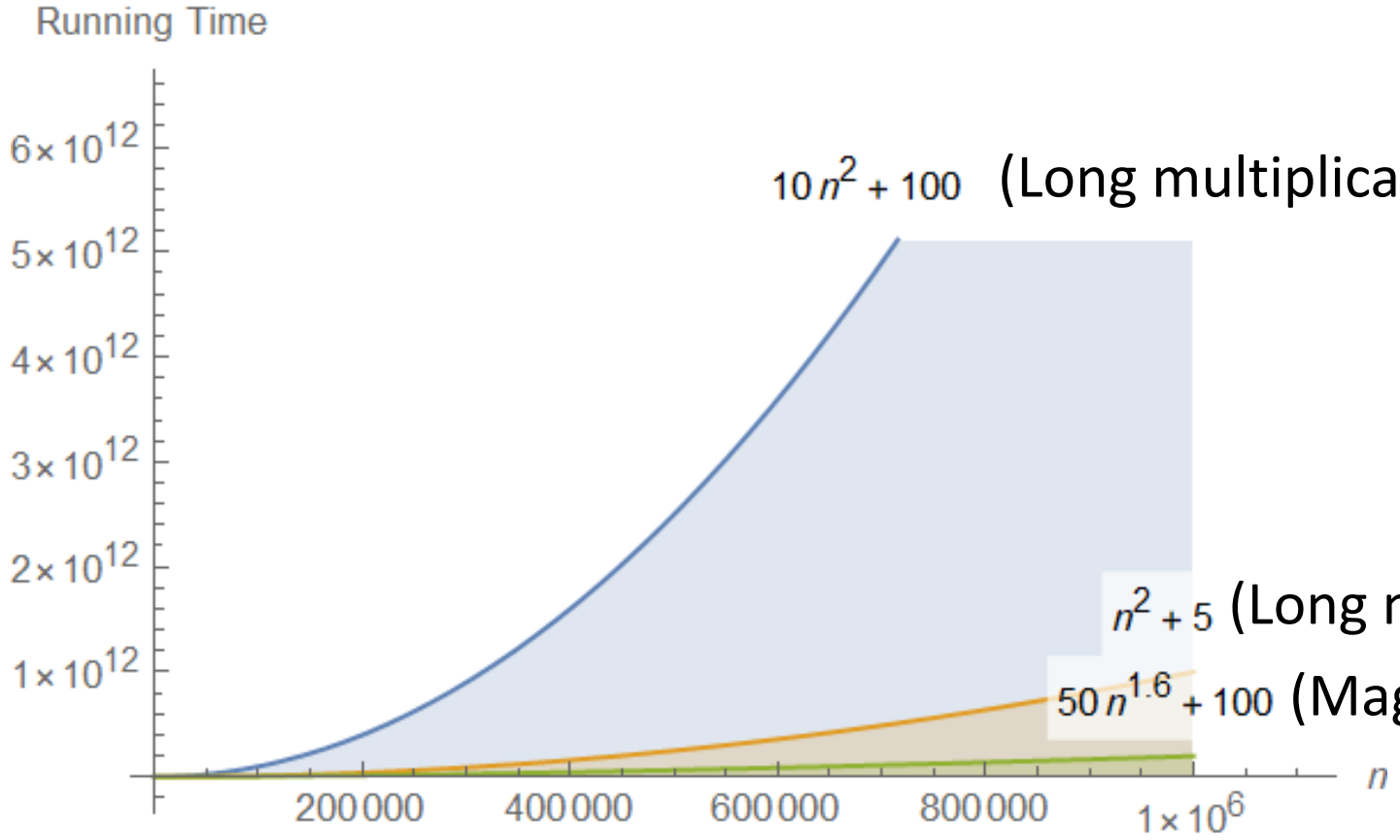
A New Multiplication Algorithm

- Hypothetically… A magic multiplication algorithm that scales like $n^{1.6}$



A New Multiplication Algorithm

- For large enough *n*, it is faster to do the magic algorithm by hand than long multiplication algorithm on computer



(Long multiplication by hand)

(Long multiplication by computer)

(Magic multiplication algorithm)

Divide-and-Conquer Multiplication

- Break up an integer: $1234 = 12 \times 100 + 34$
- One 4-digit multiplication = four 2-digit multiplications

$$1234 \times 5678$$

$$= (12 \times 100 + 34) (56 \times 100 + 78)$$

$$= \underbrace{(12 \times 56)}_{1} 10000 + \underbrace{(34 \times 56}_{2} + \underbrace{12 \times 78)}_{3} 100 + \underbrace{(34 \times 78)}_{4}$$

More Generally

- Break up an n-digit integer: $[x_1x_2\ldots x_n] = [x_1x_2\ldots x_{n/2}] \times 10^{n/2} + [x_{n/2+1}x_{n/2+2}\ldots x_n]$
- One n-digit multiplication = four $n/2$-digit multiplications

$$x \times y$$
$$= (a \times 10^{n/2} + b)\,(c \times 10^{n/2} + d)$$
$$= (a \times c) \times 10^{n} + (a \times d + c \times b) \times 10^{n/2} + (b \times d)$$

1 2 3 4

Divide-and-Conquer Multiplication Algorithm

- x, y are n-digit numbers
- Assume n is even
- a, b, c, d are $n/2$-digit numbers
- The algorithm recursively computes 4 multiplications with half of the number of digits

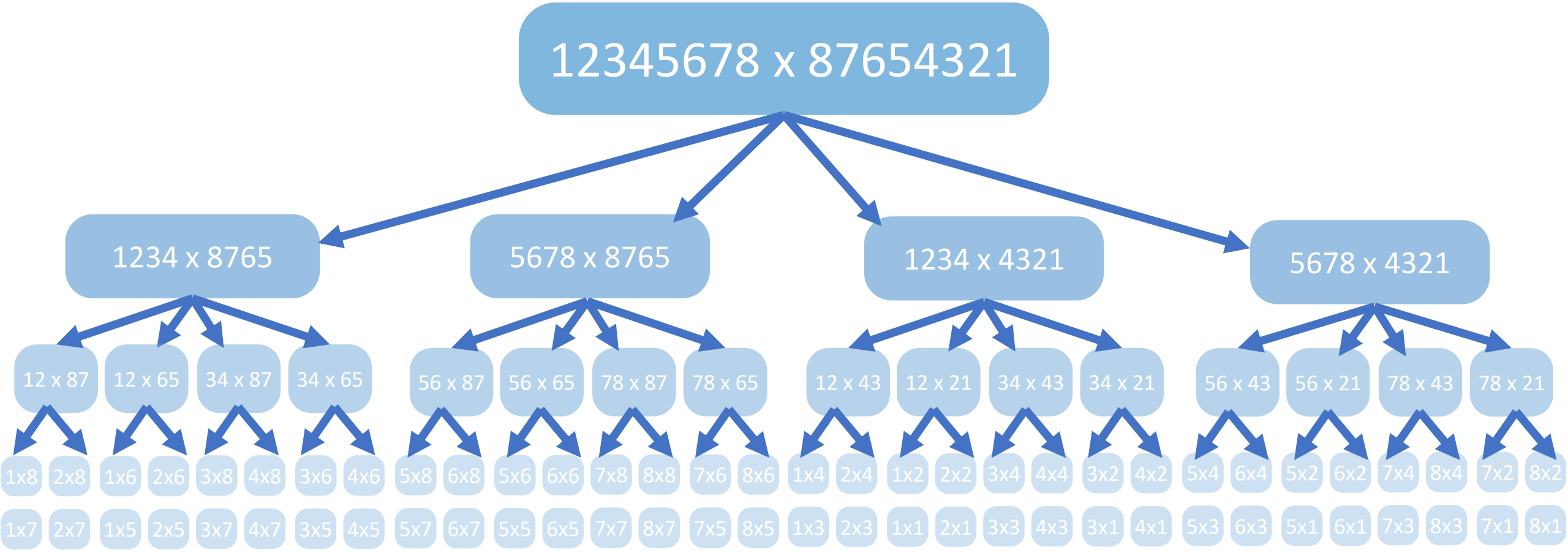
```
DnCMultiply[x, y]
//Divide-and-Conquer Multiplication

If n = 1 Then
    Return x × y

Write x = a × 10^(n/2) + b
Write y = c × 10^(n/2) + d

//Recursively compute a × c, a × d, c × b, b × d
a × c ← DnCMultiply[a, c], a × d ← DnCMultiply[a, d]
c × b ← DnCMultiply[c, b], b × d ← DnCMultiply[b, d]

// Add them up to get x × y
Return (a × c)×10^n + (a × d + c × b)×10^(n/2) + (b × d)
```

Running Time?

- Better or worse than the long multiplication algorithm?
  - That is, does the number of multiplications grow like n^2 ?
  - More or less than that?
- How do we answer this question?
  - Run it
  - Try to understand it analytically

- **Claim**:
  - The running time of `DnCMultiply` still has at least n^2 multiplications

Divide and Conquer Paradigm

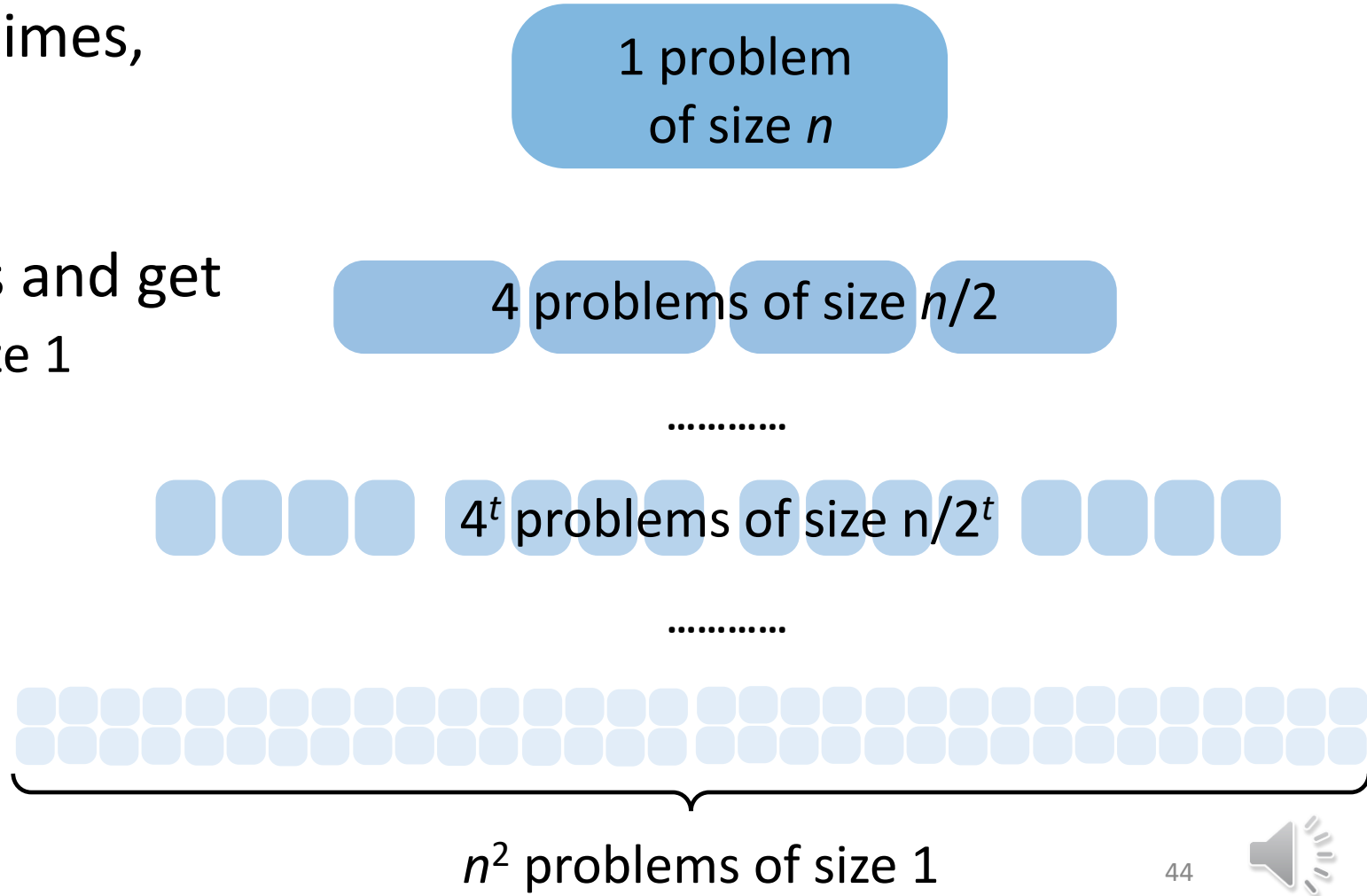


Every pair of digits still gets multiplied together separately

So the running time is still at least n^2

Another Way to See This

- If you cut n in half log(n) times, you get down to 1
  - $2^{\log(n)} = n$
- So we do this log(n) times and get
  - $4^{\log(n)} = n^2$ problems of size 1



Proof

- Let $T(n)$ be the running time to multiply two n-digit numbers
- Recurrence relation:

$T(n) = 4\,T(n/2) + (\text{about } n \text{ for additions})$

$T(n) \approx 4\,T(n/2)$

$= 4\,(4\,T(n/4)) = 4^2\,T(n/2^2)$

$= 4\,(4\,(4\,T(n/8))) = 4^3\,T(n/2^3)$

…

$= 4^t\,T(n/2^t)$

…

$= 4^{\log(n)}\,T(n/2^{\log(n)}) = n^2\,T(1)$

Karatsuba Multiplication

- Karatsuba figured out how to improve this!
- If only we recurse three times instead of four…

$$x \times y$$

$$= (a \times 10^{n/2} + b)\,(c \times 10^{n/2} + d)$$

$$= \underbrace{(a \times c)}_{1} \times 10^{n} + \underbrace{(a \times d \;+\; c \times b)}_{2} \times 10^{n/2} + \underbrace{(b \times d)}_{3}$$

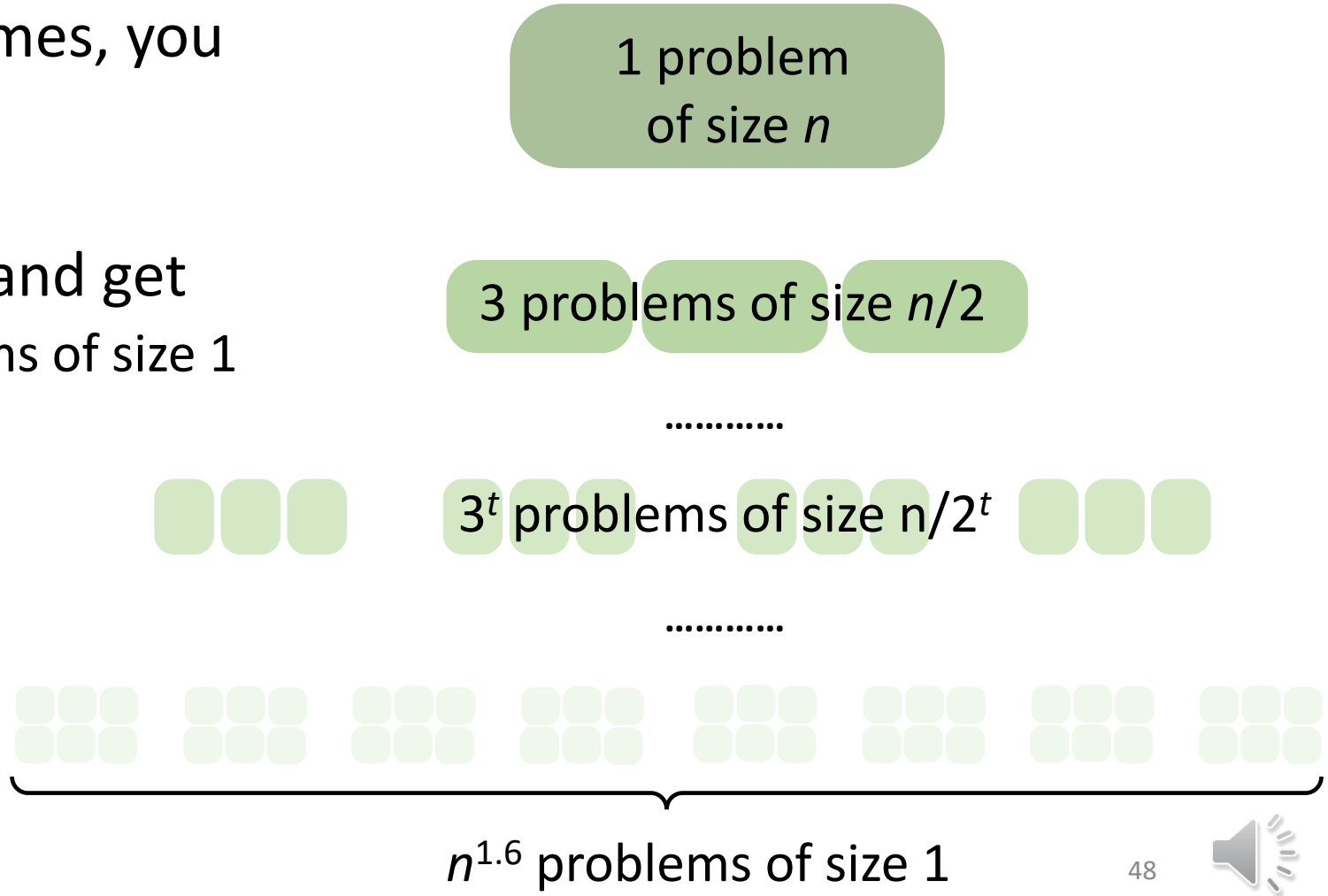
Karatsuba Multiplication

- Recursively compute these THREE things
  - $a \times c$
  - $b \times d$
  - $(a + b) \times (c + d)$
- Hence,
  - $a \times d + c \times b = (a + b) \times (c + d) - a \times c - b \times d$
  - $x \times y$

$$= \underbrace{(a \times c)}_{1} \times 10^{n} + \underbrace{((a + b) \times (c + d))}_{2} - a \times c - b \times d) \times 10^{n/2} + \underbrace{(b \times d)}_{3}$$

Running Time

- If you cut n in half log(n) times, you get down to 1
  - $2^{\log(n)} = n$
- So we do this log(n) times and get
  - $3^{\log(n)} = n^{\log(3)} = n^{1.6}$ problems of size 1

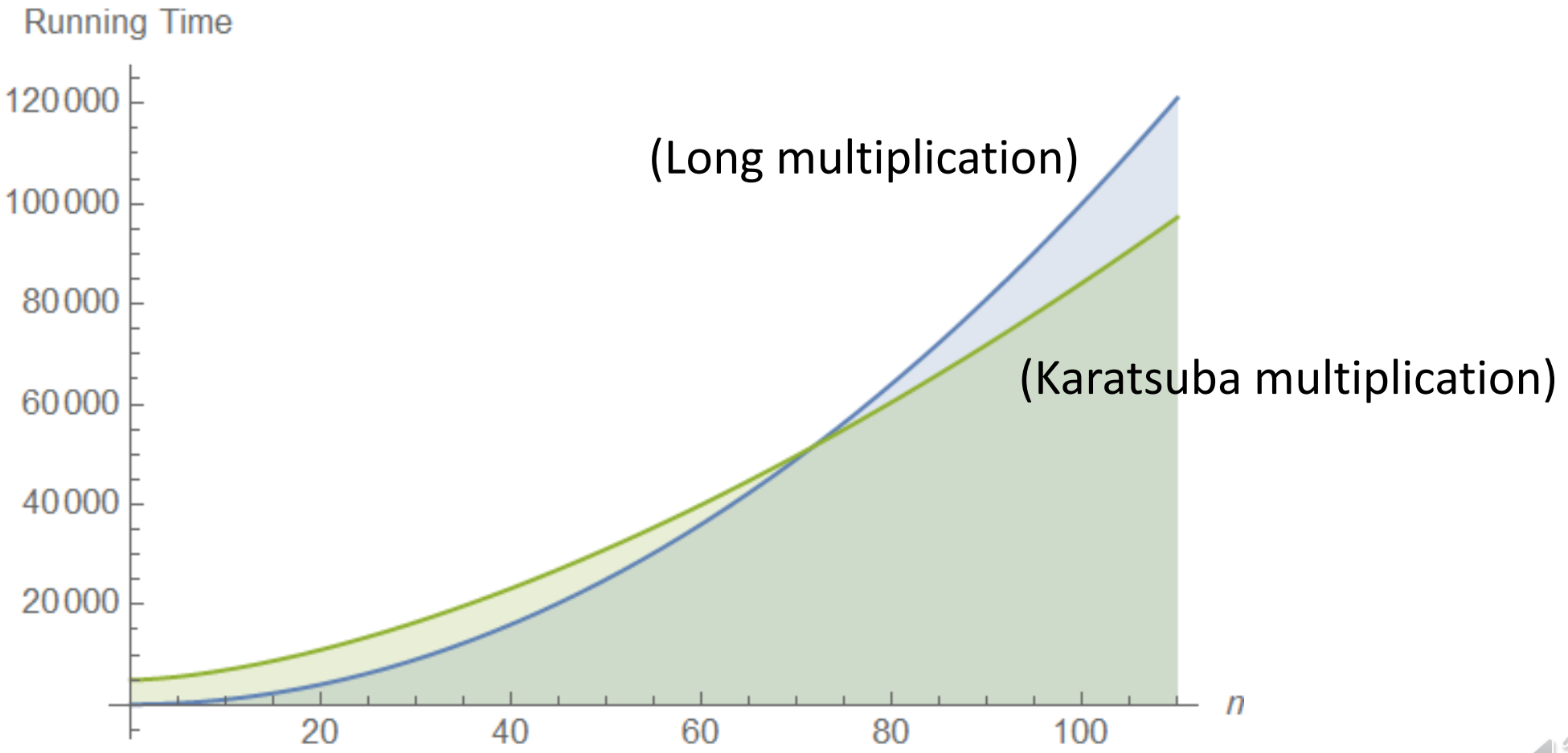


Karatsuba Multiplication Algorithm

- x, y are n-digit numbers
- Assume n is even
- a, b, c, d are $n/2$-digit numbers
- The algorithm recursively computes 3 multiplications with half of the number of digits

```
KMultiply[x, y]
//Divide-and-Conquer Multiplication

If n = 1 Then
    Return x × y

Write x = a × 10^(n/2) + b
Write y = c × 10^(n/2) + d

//Recursively compute a × c, b × d, (a + b) × (c + d)
a × c ← KMultiply[a, c], b × d ← KMultiply[b, d]
(a + b) × (c + d) ← KMultiply[a + b, c + d]
a × d + c × b ← (a + b) × (c + d) - a × c - b × d

// Add them up to get x × y
Return (a × c)×10^n + (a × d + c × b)×10^(n/2) + (b × d)
```

Karatsuba Multiplication is Faster



Can We Multiply Faster?

- **Toom-Cook** (1963)
  - Scales like $n^{1.465}$
- **Schönhage–Strassen** (1971)
  - Scales like $n \log(n) \operatorname{loglog}(n)$
- **Furer** (2007)
  - Scales like $n \log(n)\, 2^{\log^*(n)}$
- **Harvey-van der Hoeven** (2019)
  - Scales like $n \log(n)$
  - Impractical, requires gigantic n

Summary

- Divide and conquer
  - Binary search
  - Tower of Hanoi
  - Merge sort
  - Karatsuba multiplication
    - As fast as $O(n^{1.6})$
- Some understanding of measuring runtime in terms of scaling law as the input size
  - Applied the neat trick of divide and conquer

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```