Heaps and Heapsort
Strassen's Algorithm for Matrix Multiplication

CS 4102: Algorithms

Spring 2021

Mark Floryan and Tom Horton

Readings

- CLRS Chapter 6 on Heaps and Heapsort
- CLSR Section 4.2 on Strassen’s algorithm

Heapsort

Reminders, Terminology

- ADT Priority Queue
  - What’s an ADT?
  - What’s high priority?
  - Operations?
  - How is data stored?
- Heap data structure
  - The *heap structure*: an almost-complete binary tree
  - The *heap condition* or *heap order property:*
    - At any given node j, value[j] has higher priority than either of its child nodes’ values
    - Heaps are weakly sorted
  - Higher priority: large or small?
    - Max-heap vs min-heap

ADT Priority Queue

- An ADT that maintains a set of elements, each with an associated key
- Can have max or min priority queues
- Operations for max priority queue
  - Maximum
  - Extract-Max
  - Insert
  - Update-Priority
- Similar operations for min priority queue
- Data structures that implement this:
  - Usually: Binary heap in an array
  - Or, tree, Binomial Heap, Fibonacci Heap, …

Heapsort Basics

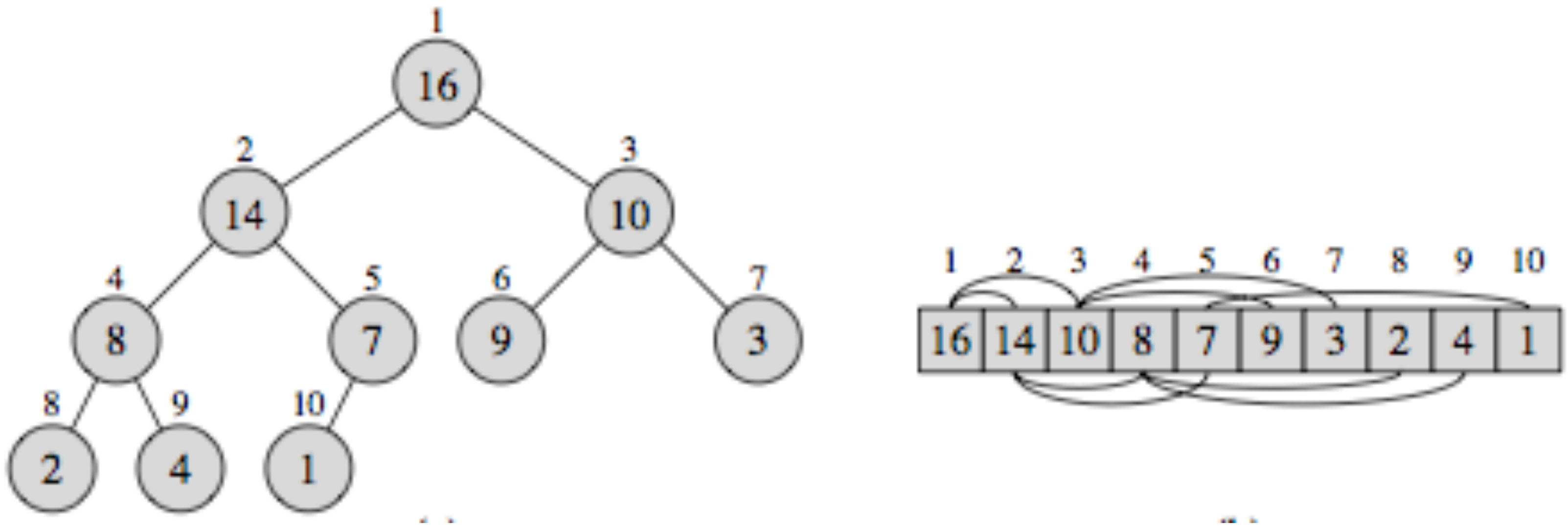
- Running time is O(n lg n) like merge sort, unlike insertion sort
- Sorts in-place (only a constant number of array elements stored outside the array at any time) like insertion sort, unlike merge sort
- Uses a “heap” data structure

Remember Heaps from CS2150?

- Remember (review) topics from CS2150
  - Slides on these from CLRS are at end of this deck if you need them
- Binary heap structure, stored in an array
- Operations for a max-heap:
  - Heap-Maximum: Returns max value $\Theta(1)$
  - Max-Heap-Insert: Insert new value into a heap $\Theta(\lg n)$
  - Max-Heapify: Restores heap-property if value changed at given index $\Theta(\lg n)$
  - Heap-Extract-Max: removes max item (uses heapify) $\Theta(\lg n)$
- We'll cover: Build-Max-Heap. Heapsort

Example of Heap Stored in Array

Review!



- p. 152 in text
- Height of a **node**: Length of the longest path from the node to a leaf
- Height of the **heap**: Height of the root ($\theta(\lg n)$)

How to Build a Heap

- Option 1:
  - Repeatedly insert a new item, start with a heap of 1 item
  - Cost: $\Theta(n \lg n)$ (Can you do the sum?)
- Option 2:
  - Take an unordered list, build the heap in place
  - Build-Max-Heap() algorithm, CLRS page 157
  - Strategy:
    - Work bottom up, starting with lowest sub-heaps
    - Call Max-Heapify() on each
  - Note: Some give this a different name, including (confusingly) "heapify"

Building a Heap using Heapify

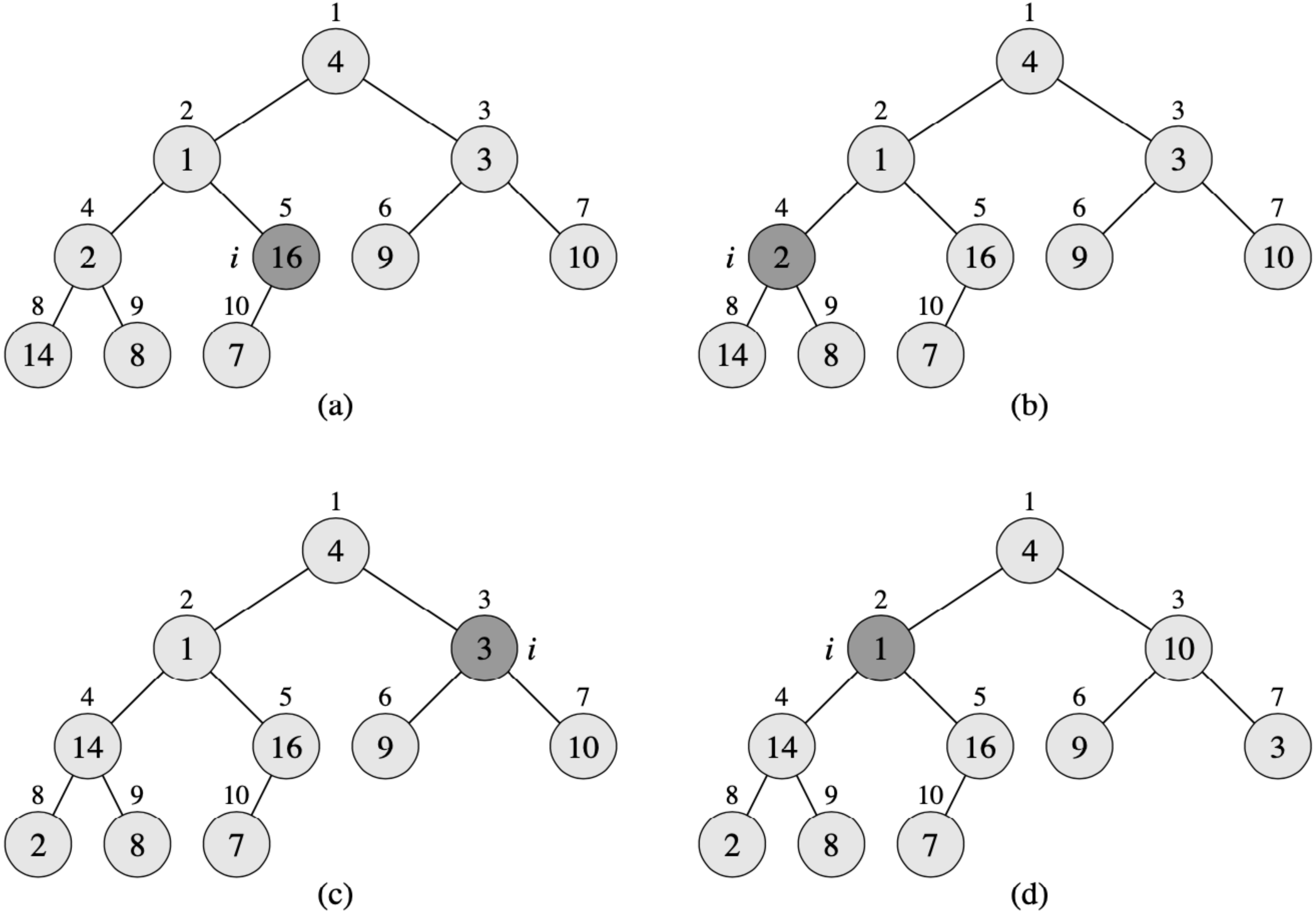
- Starts at lowest level non-leaf node, goes up to root, calling Max-Heapify for each node

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

- Do this example: [3, 1, 4, 2, 7, 11, 9, 8, 15, 12]
- Also see Figure 6.3 (next slide)
- Proof of correctness? Loop invariant is this:
  At the start of the for-loop, each node i+1, i+2,…, n is the root of a max-heap.

A	4	1	3	2	16	9	10	14	8	7



Runtime of Build-Max-Heap

- Each call to Max-Heapify costs O(lgn) time
- There are O(n) calls to Max-Heapify
- Upper bound on running time: O(nlgn)
- But the tight bound is: O(n)
  - Smaller sub-heaps at the bottom of tree are shorter, and there are more of them
  - See book for analysis

Applications of Heaps, Priority Queues

- Many!
- But (oh yeah) we were trying to sort
- What's our strategy?
  - – I'll tell you this: it's kind of like selection-sort
  - – Now, you tell me what you think it is
    - (If you already know, let someone else try to figure this out.)

Heapsort

- Maximum element is stored at the root (1^{st} item in the list)
- Exchange it with the last element
- Call Max-Heapify on the root, but with a heap-size that decrements
  - Note that we need a way to say how much of A is part of the current heap. How can we do this?

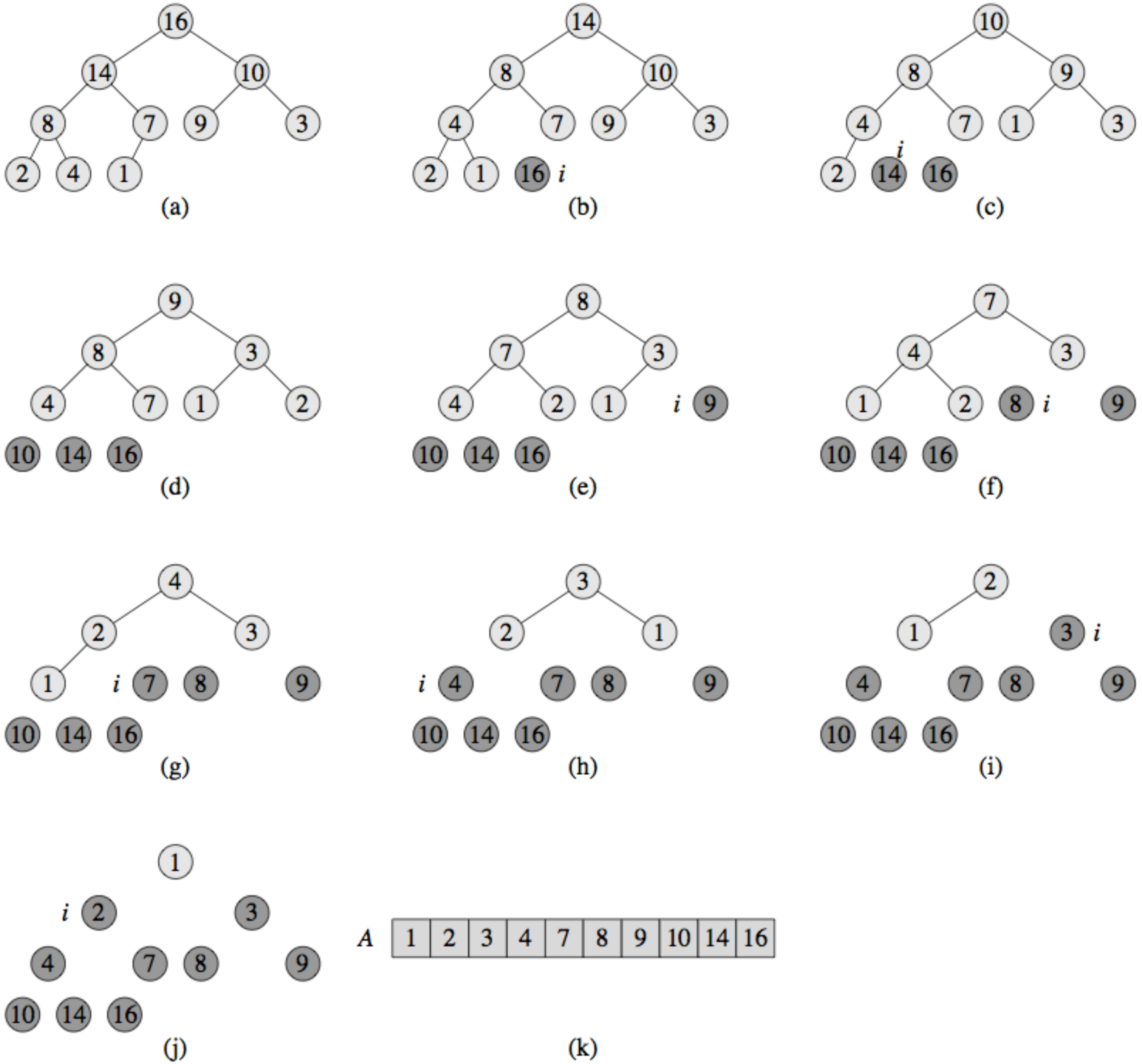
```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

Runtime of Heapsort

- Build-Max-Heap takes O(n) time
- Each of the n-1 calls to Max-Heapify takes O(lgn) time
- Total time: O(nlgn)

Do You Understand?

- Answer these:
  - Show how array is rearranged when heap-sorting the heap you got when you built one from this list: [3, 1, 4, 2, 7, 11, 9, 8, 15, 12]
  - Also, see Figure 6.4 (next slide)
  - Can you say anything about Heapsort's behavior if the array is already sorted?
  - Can you say anything about Heapsort's behavior if the array is in "reverse sorted" order?


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Summary

- ADT Priority Queue
  - Generally useful concept. We’ll see it again.
- Binary Heap Data Structure
  - An effective implementation of ADT Priority Queue
  - Most basic operations are O(lg n)
  - Heapify operation: used to changing a heap at its root, and then restoring it
  - One gotcha (which we’ll see later): how to update priority of item at position *i*
- Heapsort
  - W(n) is O(n lg n)
  - In-place

Matrix Multiplication

Matrix Multiplication

$$\begin{array}{cc} & n \\ n & \begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}\end{array} \times \begin{bmatrix}2&4&6\\8&10&12\\14&16&18\end{bmatrix}$$

$$= \begin{bmatrix} 2+16+42 & 4+20+48 & 6+24+54 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$= \begin{bmatrix} 60 & 72 & 84 \\ 132 & 162 & 192 \\ 204 & 252 & 300 \end{bmatrix}$$

Run time? $O(n^3)$ Lower Bound? $O(n^2)$

Matrix Multiplication D&C

Multiply $n \times n$ matrices (A and B)

Divide:

$$A = \left[\begin{array}{cc|cc} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ \hline a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{array}\right] \qquad B = \left[\begin{array}{cc|cc} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ \hline b_9 & b_{10} & b_{11} & b_{12} \\ b_{13} & b_{14} & b_{15} & b_{16} \end{array}\right]$$

Matrix Multiplication D&C

Multiply $n \times n$ matrices (A and B)

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Combine:

$$AB = \begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$

Run time? $T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2$

Case 1!

$T(n) = \Theta(n^3)$

Find an Algorithm with Better Recurrence?

$$T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2$$

- We've got a recurrence and want to improve things.
  You know how the Master Theorem works.
  What can we change to make it better?

  – Reduce the number of subproblems.

  – Reduce the order class of the non-recursive work.
  (OK to do more non-recursive work if new f(n) is same Θ)

Strassen's Algorithm

Multiply $n \times n$ matrices (A and B)

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Calculate:

$$Q_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$Q_2 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$Q_3 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$Q_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$Q_5 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$Q_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$Q_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

Find AB:

$$\begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$
$$=$$
$$\begin{bmatrix} Q_1 + Q_4 - Q_5 + Q_7 & Q_3 + Q_5 \\ Q_2 + Q_4 & Q_1 - Q_2 + Q_3 + Q_6 \end{bmatrix}$$

Number Mults.: 7 Number Adds: 18

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

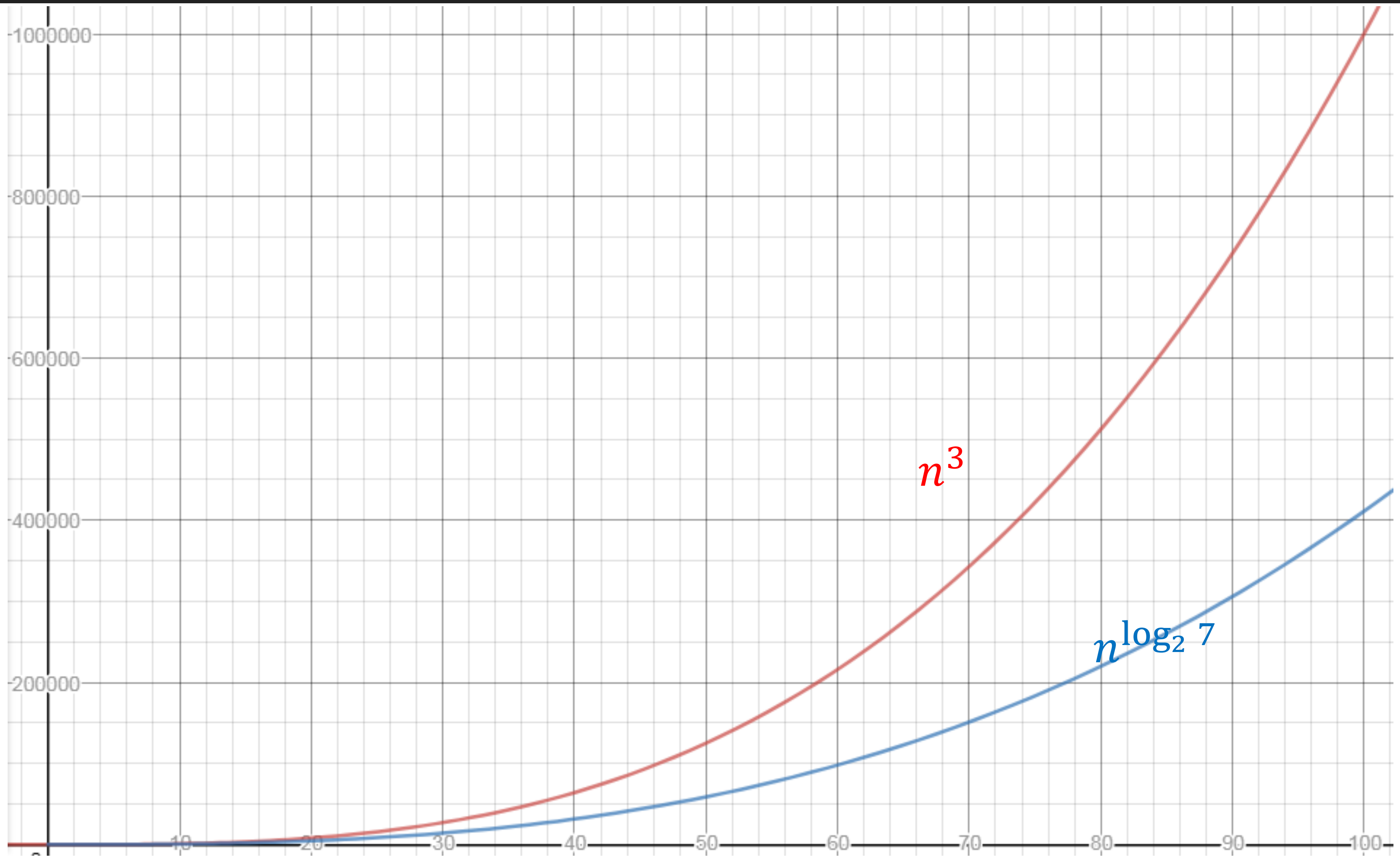
Strassen's Algorithm

$$T(n) = 7T\left(\frac{n}{2}\right) + \frac{9}{2}n^2$$

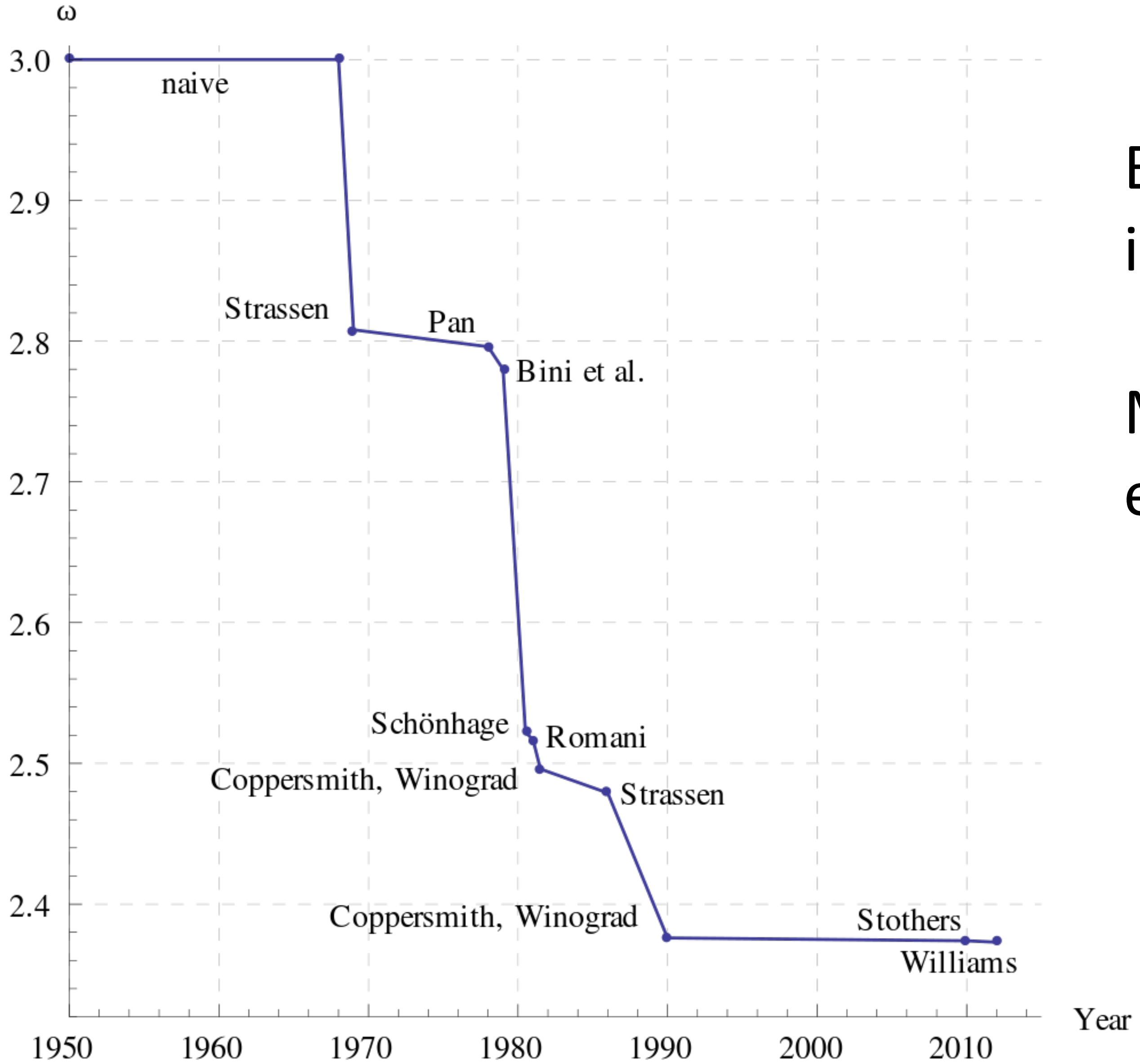
$$a = 7, b = 2, f(n) = \frac{9}{2}n^2$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$$ Case 1!

$$T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.807})$$


1000000
800000
600000
400000
200000
10
20
30
40
50
60
70
80
90
100
n^3
$n^{\log_2 7}$

Is this the fastest?



Best possible is unknown

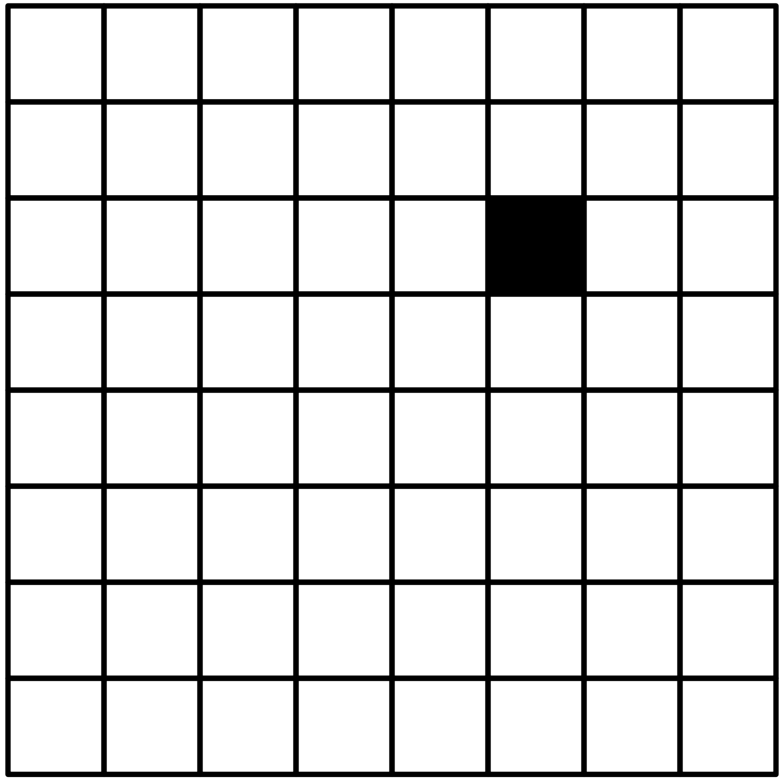
May not even exist!

Trominoes

A board puzzle with a nice divide
and conquer solution.
(For those of you tired of manipulating lists!)

Trominoes Board Puzzle

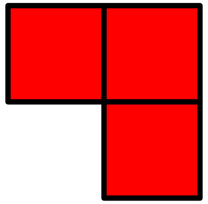
Can you cover this board?



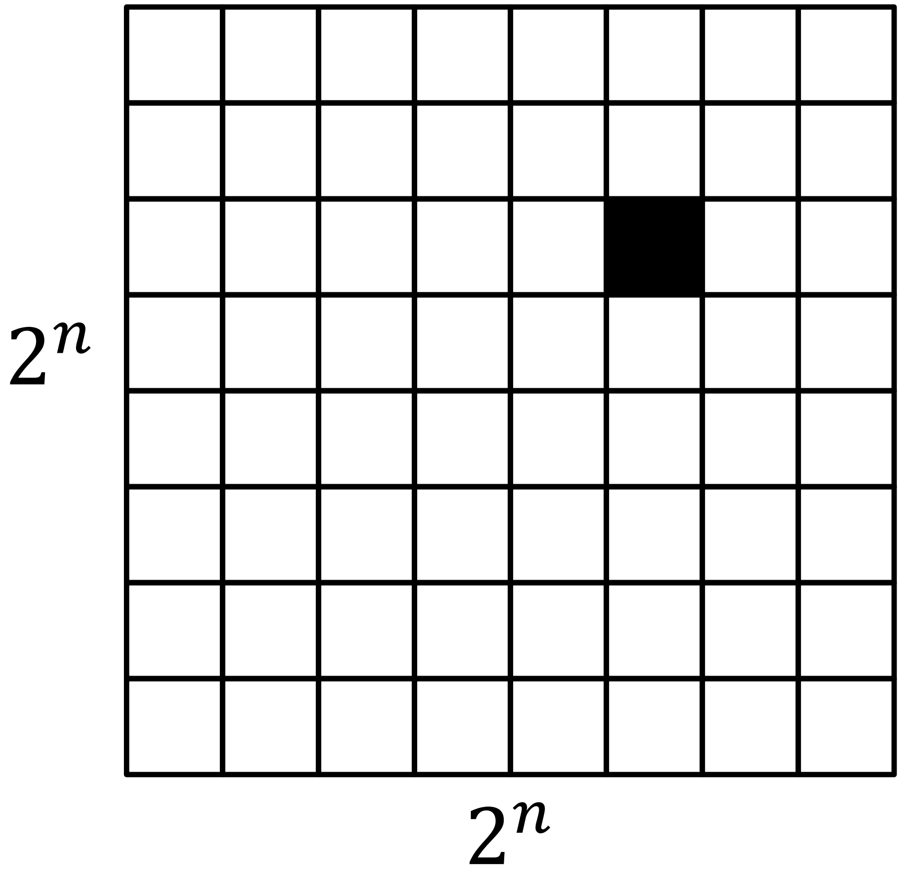
The Goal

Can you cover an 8×8 grid with 1 square missing using "trominoes?"

With these?

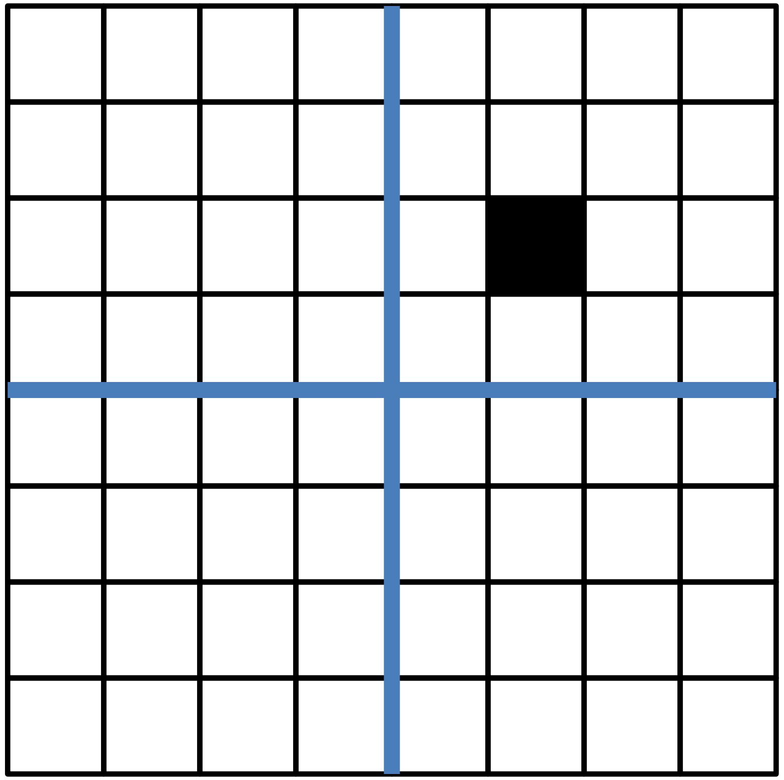


Trominoes Puzzle Solution



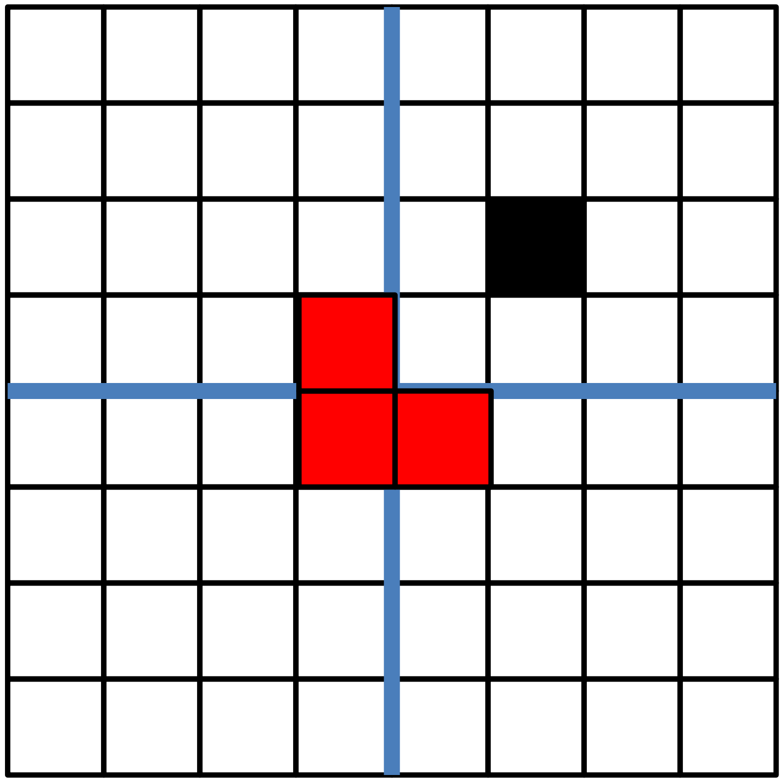
What about larger boards?

Trominoes Puzzle Solution



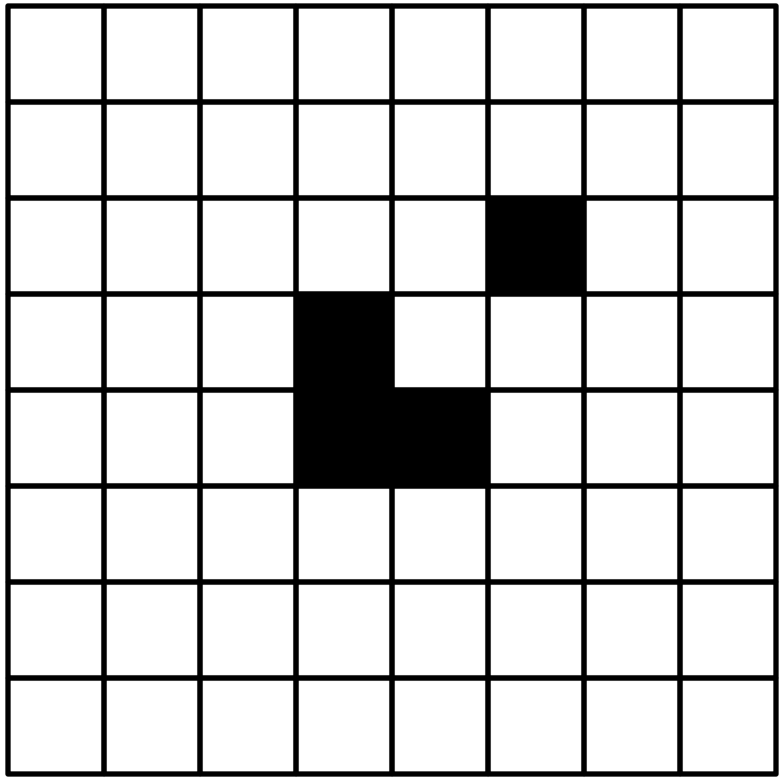
Divide the board into quadrants

Trominoes Puzzle Solution



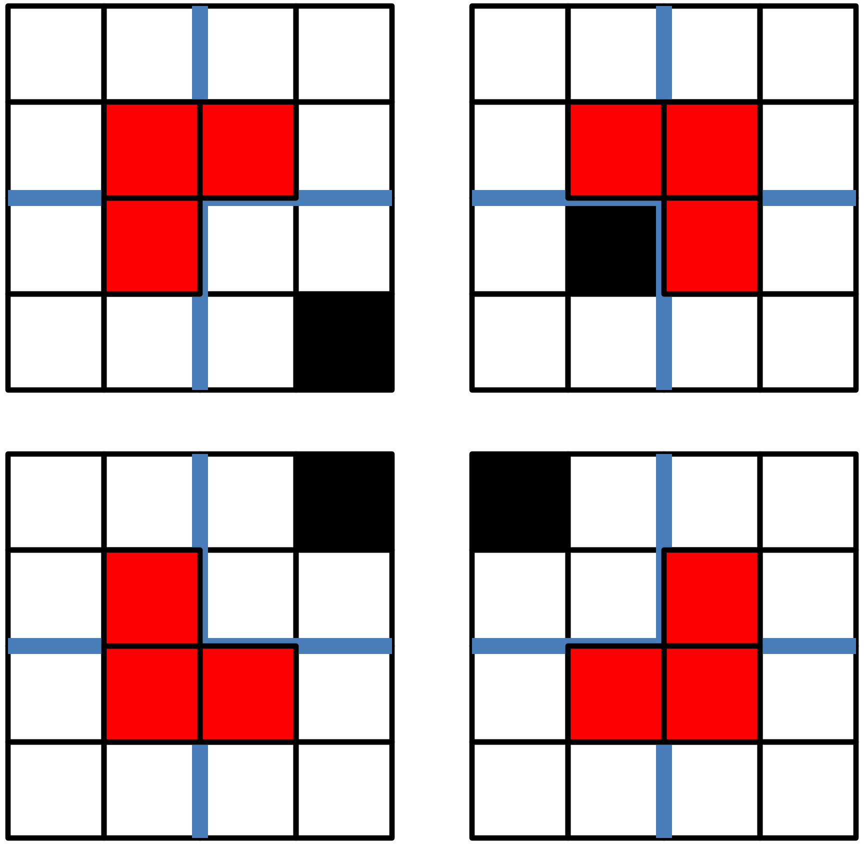
Place a tromino to occupy the three quadrants without the missing piece

Trominoes Puzzle Solution



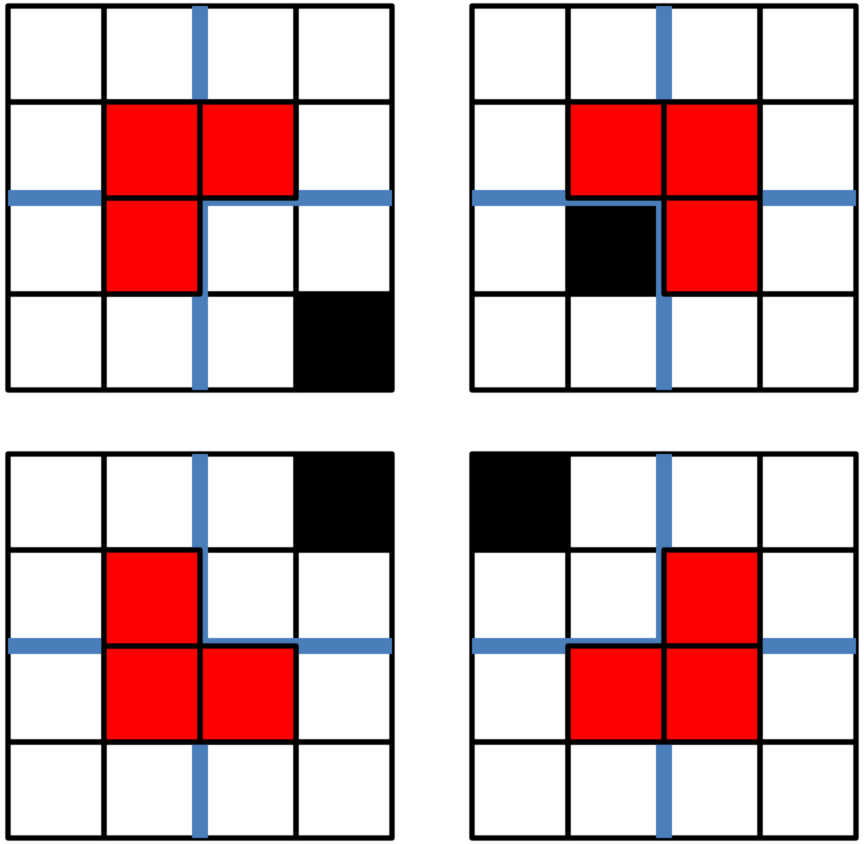
Each quadrant is now a smaller subproblem

Trominoes Puzzle Solution



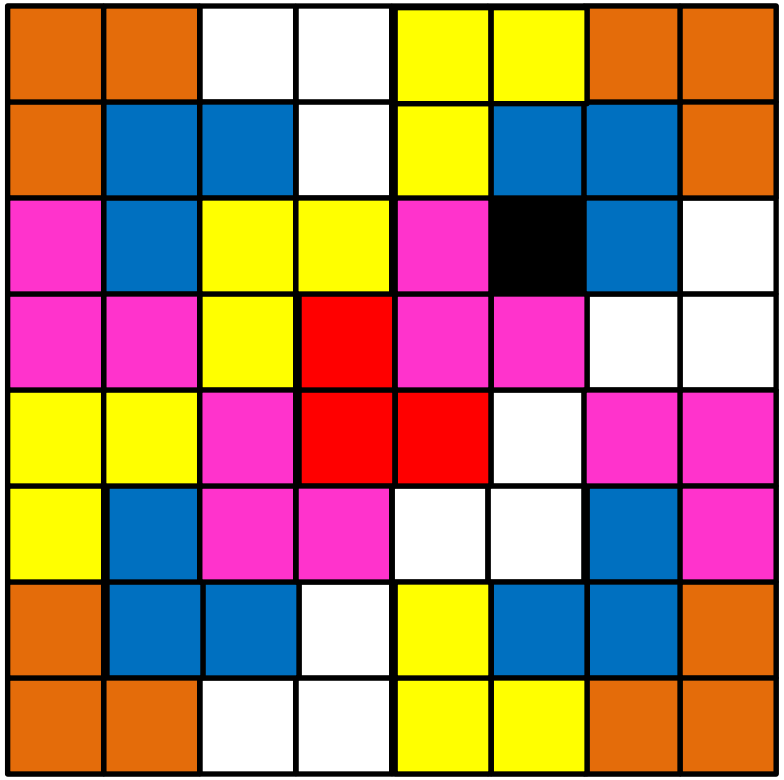
Solve **Recursively**

Divide and Conquer



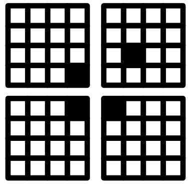
Our first algorithmic technique!

Trominoes Puzzle Solution

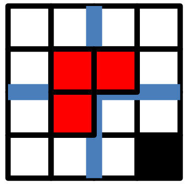


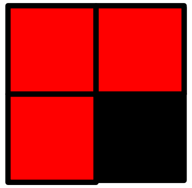
Divide and Conquer

- **Divide**:

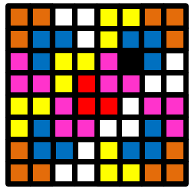


  - Break the problem into multiple subproblems, each smaller instances of the original
- **Conquer**:





  - If the suproblems are "large":
    - Solve each subproblem recursively
  - If the subproblems are "small":
    - Solve them directly (base case)
- **Combine**:



  - Merge together solutions to subproblems

Slides Reviewing Heaps from CS2150

Using code and terminology from CLRS Chapter on Heapsort

Binary Heap

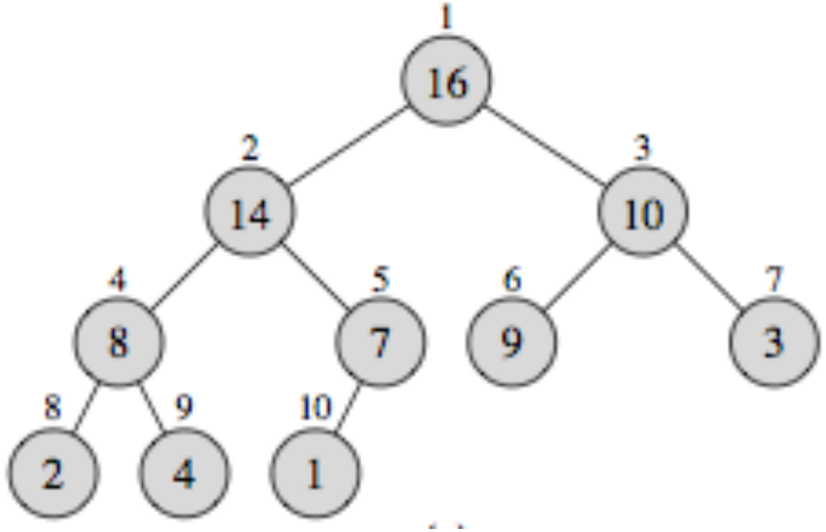
- Implemented as an array
- Viewed as a nearly complete binary tree
- Each tree node corresponds to an array element
- All tree levels filled except possibly the lowest
- Example in Figure 6.1

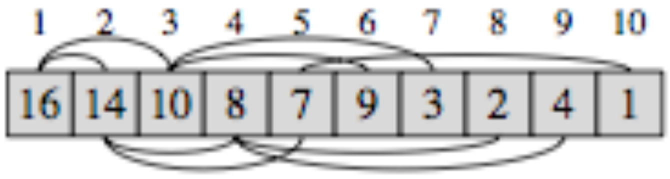
Indices of Key Nodes

- Root: 1
- Given index i of a node:
  - Parent(i) = floor(i/2)
  - Left-child(i) = 2i
  - Right-child(i) = 2i+1
- Multiplying by 2 goes down a level
- Dividing by 2 goes up a level

- Question: How do these change if we index the list from 0 (like Python, Java, etc.)?

Example of Heap Stored in Array





- p. 152 in text
- Height of a **node**: Length of the longest path from the node to a leaf
- Height of the **heap**: Height of the root (θ(lgn))

Basic Heap Algorithms

- Let's work with max-heaps for now
- Define a set of simple heap operations
  - Traverse tree, thus logarithmic complexity
- Highest priority item?
  - At the root. Just return it.
- Insert an item?
  - Add after the nth item (end of list)
  - Out of place? Swap with parent. Repeat, pushing it up the tree until in proper place
- Remove an item?
  - Hmm...

Book's Heap Operations

- Max-Heapify: Restores heap-property (O(lgn))
- Build-Max-Heap: Creates max-heap from unordered array (O(n))
- Heapsort: Sorts an array in place (O(nlgn))
- Max-Heap-Insert, Heap-Extract-Max, Heap-Maximum: Implement a priority queue (O(lgn))

Max-Heap-Insert

- Here's Python. Not quite than the same as in CLRS text, p. 164

```
def max_heap_insert(A, key):
    A.append(None)  # grow list by one
    i = len(A)
    while (i>1 and A[i//2] < key):
        A[i] = A[i//2]
        i = i//2
    A[i] = key
```

Maintaining the Heap Property

- Max-Heapify(A, i)
  - Also known as "fixheap" or "siftdown" or "percolate"
- Assumptions
  - left and right subtree of node i are max-heaps, but
  - A[i] might be smaller than its children
- Value at A[i] is "pushed down" the heap to restore the heap property. How?
  - Find larger of two children of current node
  - If current node is out-of-place, then swap with largest of its children
  - Keep pushing it down until in the right place or it's a leaf

Max-Heapify Algorithm

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

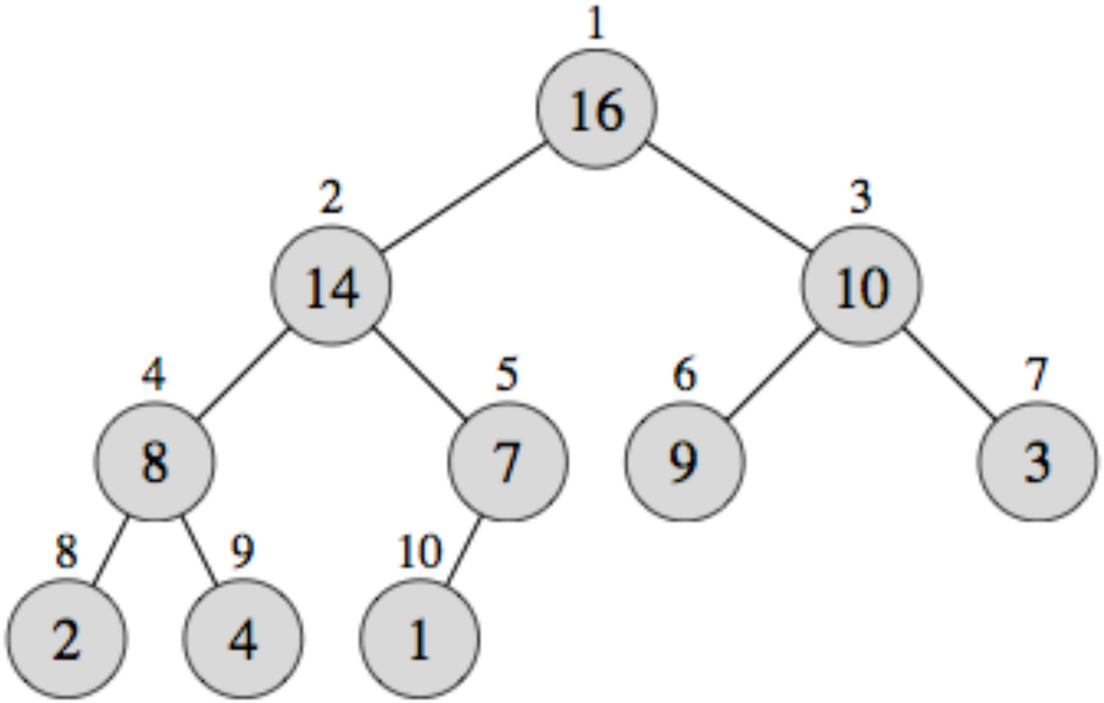
- Example: Figure 6.2

Max-Heapify in Python, with loop

- Python code that's closer to what you saw in CS2150, non-recursive

```
def max_heapify(A, i):
    temp = A[i]
    while 2*i <= len(A): # while left child exists
        max_child= 2*i
        # is there right child, and is it bigger?
        if max_child< len(A) and A[max_child+1] > A[max_child]:
            max_child= max_child+ 1
        # move child up?
        if A[max_child] > temp:
            A[i] = A[max_child]
        else:
            break # done, exit loop
        i = max_child
    A[i] = temp # after loop, put original item in correct spot
```

Practice with Max-Heapify



- Store 15 in A[3] and do Max-Heapify(A,3)
- Store 3 in A[2] and do Max-Heapify(A,2)
- Store 9 in A[1] and do Max-Heapify(A,1)

Runtime of Max-Heapify

- For a subtree of size n rooted at node i
  - $\Theta(1)$ for fixing node i and its children
  - Time to run Max-Heapify on a child's subtree: has size at most $2n/3$ (worst case when bottom level of tree is exactly half full)
  - Recurrence: $T(n) \le T(2n/3) + \theta(1)$
  - Solution: $T(n) = O(\lg n) = O(h)$
- Another analysis (without recursion):
  - At each level, does at most 2 comparisons
    - Find larger child, and then see if it's larger than current node
  - In worst case, "push down" h levels, where h is the height of the current node (distance to a leaf)
  - Overall from the root: $T(n) = O(2h) = O(\lg n)$

Back to Heap Operations: Extract-Max

- The max value is at A[1]
- A's size has to shrink by one, so A[n] has to move somewhere. So, A[1]=A[n]
- Still a heap? Probably not at position 1! Max-Heapify can fix that!

Heap-Extract-Max

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

- Worst-case complexity: θ(lgn)