CSE 250 - Data Structures: Recursion

Andrew Hughes

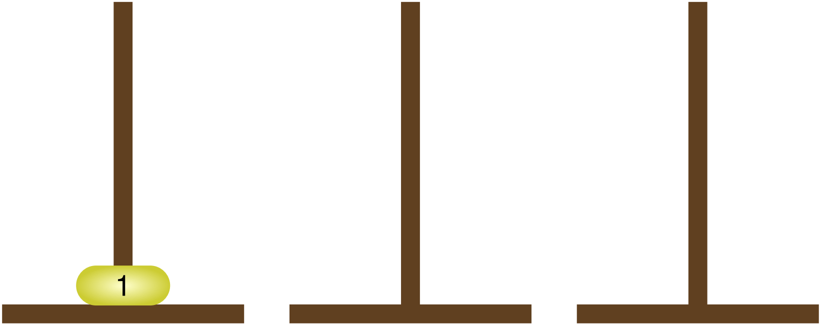
SUNY at Buffalo
Computer Science and Engineering

Fall 2019

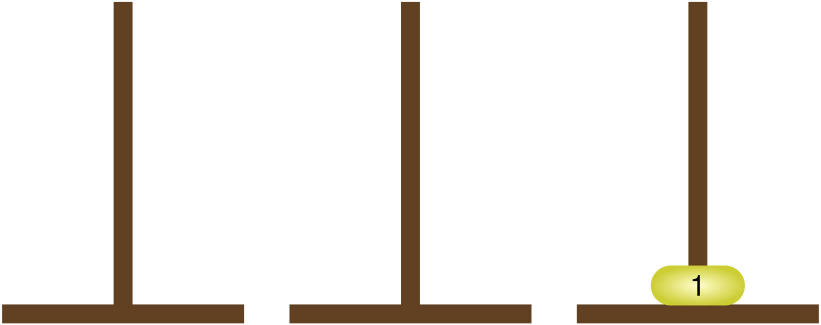
Outline

1. Recursive Functions
   - Towers of Hanoi
   - Fibonacci
   - Factorial
   - MergeSort
   - QuickSort

Tower of Hanoi – 1 Disc



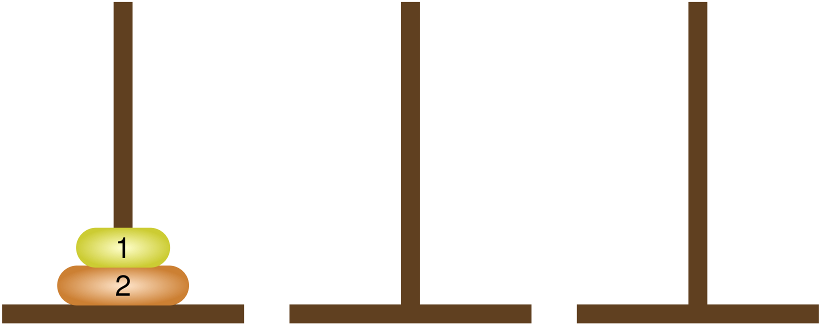
Tower of Hanoi – 1 Disc



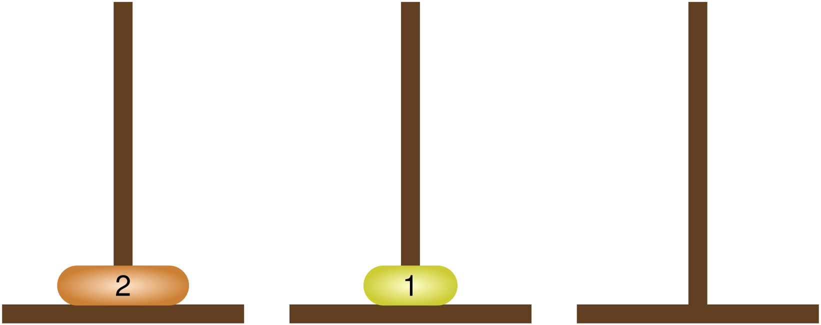
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 1 Disc

Tower of Hanoi – 2 Discs

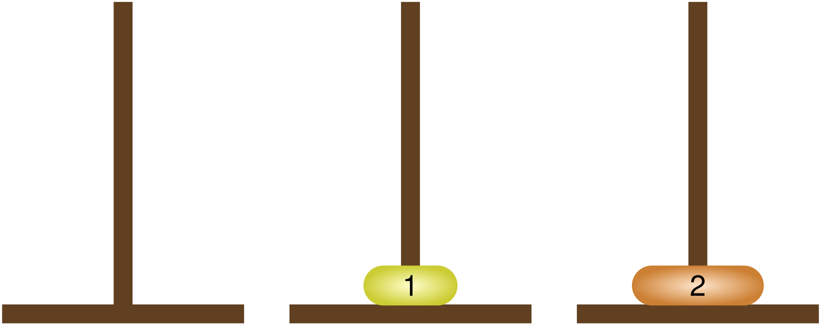


Tower of Hanoi – 2 Discs



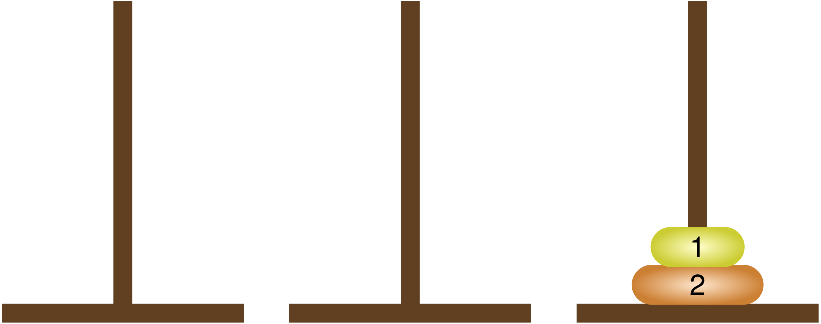
Moved disc from pole 1 to pole 2.

Tower of Hanoi – 2 Discs



Moved disc from pole 1 to pole 3.

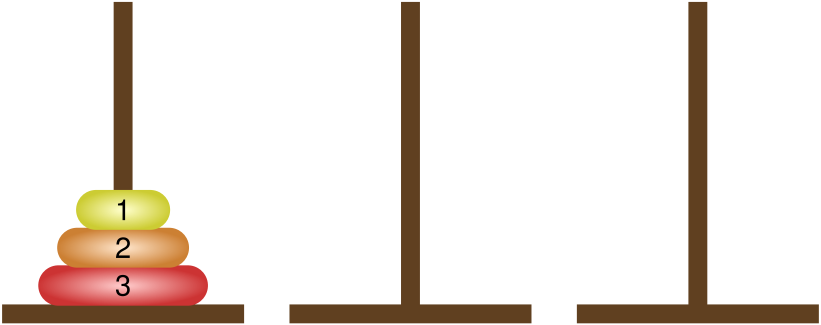
Tower of Hanoi – 2 Discs



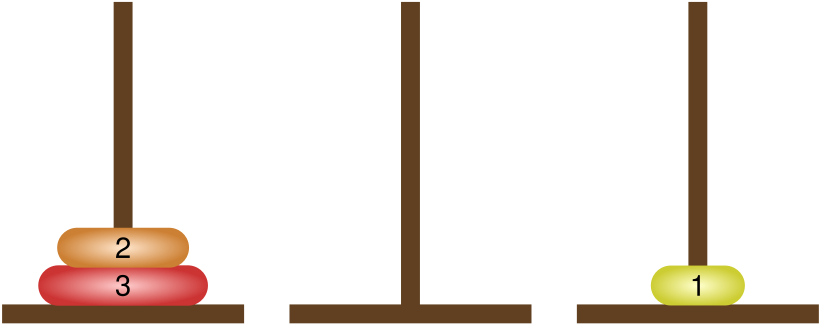
Moved disc from pole 2 to pole 3.

Tower of Hanoi – 2 Discs

Tower of Hanoi – 3 Discs

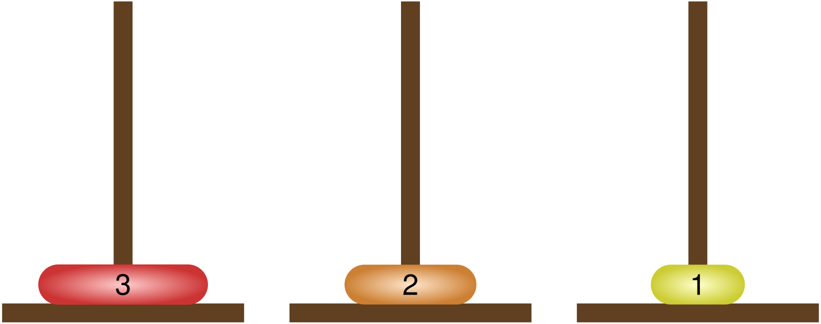


Tower of Hanoi – 3 Discs



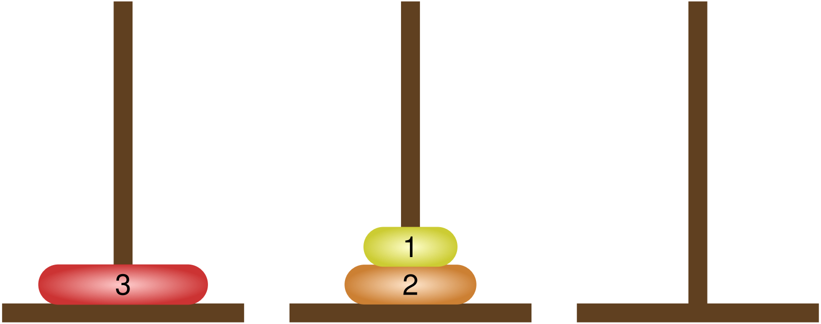
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs



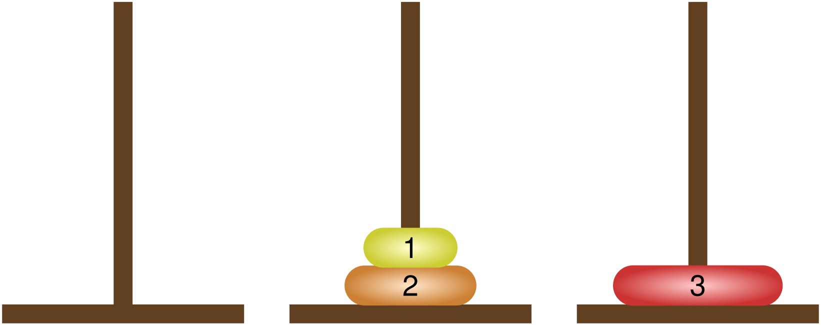
Moved disc from pole 1 to pole 2.

Tower of Hanoi – 3 Discs



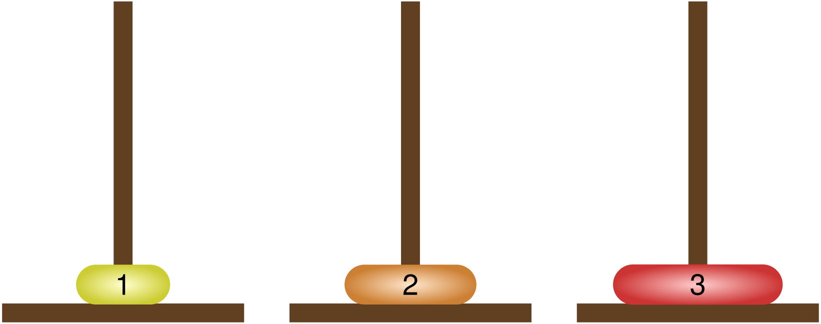
Moved disc from pole 3 to pole 2.

Tower of Hanoi – 3 Discs



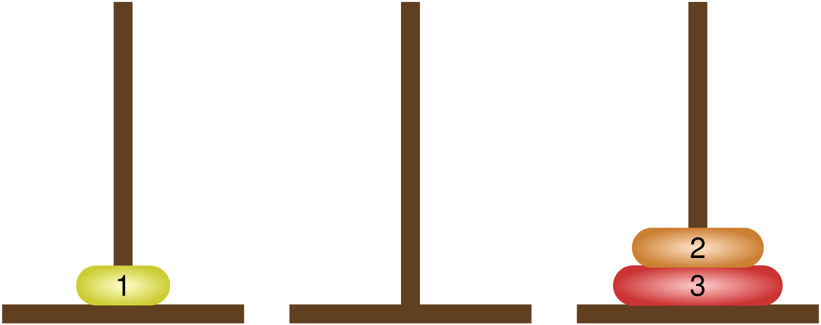
Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs



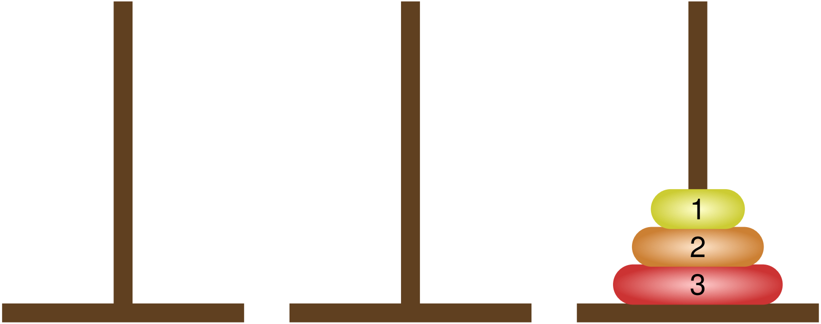
Moved disc from pole 2 to pole 1.

Tower of Hanoi – 3 Discs



Moved disc from pole 2 to pole 3.

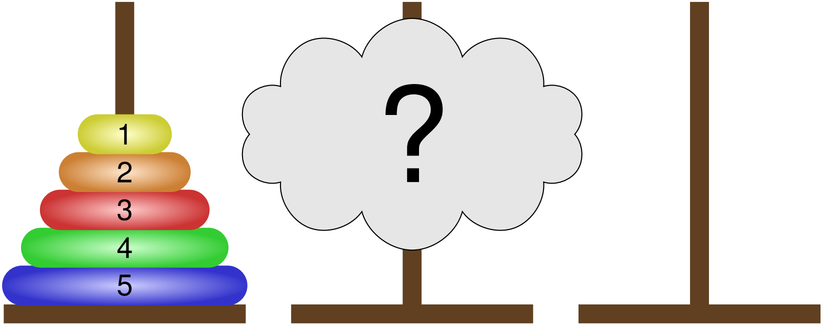
Tower of Hanoi – 3 Discs



Moved disc from pole 1 to pole 3.

Tower of Hanoi – 3 Discs

Tower of Hanoi – 5 Discs



Towers of Hanoi

Given a stack of n discs and three rods where:

- each disc is a unique size and
- the discs begin stacked in decreasing size on the left rod,

How many moves to move the stack from the left pole to the right pole without moving any larger disc on top of a smaller disc?

Towers of Hanoi

Move n discs from first to third pole by

- $n = 1$: move disc directly.
- $n > 1$: move $n - 1$ discs to middle pole, move largest to third pole, move $n - 1$ discs from middle to right pole.

Fibonacci Sequence

Recall the Fibonacci Sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Fibonacci Sequence

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

A **recurrence relation** is an equation that recursively defines a sequence.

- Easy to compute recursively.

```
def fib(n: Int): Long = {
  if(n == 0 || n == 1) n
  else fib(n-1) + fib(n-2)
}
```

Fibonacci Sequence

The runtime of a recursive function is easy to represent with a recurrence relation

```
def fib(n: Int): Long = {
  if(n == 0 || n == 1) n
  else fib(n-1) + fib(n-2)
}
```

Runtime recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \text{otherwise.} \end{cases}$$

Closed form? $T(n) = O(\frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}})$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618\ldots$

Factorial

Recall that $n! = n(n-1)\ldots(2)(1)$.

$$n! = \begin{cases} \boxed{1 \qquad \text{if } n = 0} \\ \boxed{n(n-1)(n-2)\cdots(2)(1) \qquad \text{if } n \geq 1} \end{cases}$$

```
def fact (n: Int): Long = {
  if(n <= 0) 1
  else n * fact(n-1)
}
```

Basis Case

$0! = 1$.

Recursive Case

- $1! = 1 = 1(0!)$
- $2! = 2(1)(0!) = 2(1!)$
- $3! = 3(2)(1)(0!) = 3(2!)$
- $\ldots$
- $$\begin{aligned} n! &= n(n-1)\cdots(2)(1)0! \\ &= n((n-1)!) \end{aligned}$$

Number of Operations

- $n = 0$: $O(1)$
- $n > 0$: $O(1)$ + ops for `fact`$(n-1)$.

$$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ T(n-1) + O(1) & \text{if } n \geq 1 \end{cases}$$

Factorial

Number of Operations

- $n = 0$: $O(1)$
- $n > 0$: $O(1)$ + ops for `fact(`$n-1$`)`.

$$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ T(n-1) + O(1) & \text{if } n \geq 1 \end{cases}$$

Closed form? Space allocated?

Tail-Recursive Factorial

How can we avoid the excessive space allocation?

Use tail recursion!

- The `@tailrec` annotation: require it, or else.
- Disclaimer: Only works if last action is a recursive call.
- The compiler *may* do this automatically without the annotation.
  - Annotation benefit: if non-tail-recursive, compilation fails.

Recall the factorial code:

```
def fact (n: Int): Long = {
  if(n <= 0) 1
  else n * fact(n-1)
}
```

What is the last operation?

Tail-Recursive Factorial

Idea

Pass accumulator to hold partial result instead of computing at end.

- This gives the following updated code:

```
import scala.annotation.tailrec
def factorial (n: Int): Long = {
  @tailrec def factorialAcc (n: Int, acc: Long): Long = {
    if (n <= 1) acc
    else factorialAcc(n-1, acc*n)
  }
  factorialAcc(n,1)
}
```

`acc` holds partial `factorial` result.

Last operation (in recursive case) is now `factorialAcc`.

Tail-Recursive Fibonacci

```
def fib(n: Int): Long = {
  if(n == 0 || n == 1) n
  else fib(n-1) + fib(n-2)
}
```

–VS–

```
def fibonacci(n: Int): Long = {
  @tailrec
  def fibAcc(n: Int, fibN1: Long, fibN: Long) = {
    if (n == 1) fibN
    else fibAcc(n-1,fibN,fibN1+fibN)
  }
  if(n <= 0) { n }
  else fibAcc(n,0,1)
}
```

Divide and Conquer Strategy

Recursive Solutions

Solve a problem building from solution(s) to smaller instances of the same problem.

Divide and Conquer Idea

- **Divide** problem into subproblem(s) (smaller instances of the same problem).
- **Conquer** subproblems by solving recursively or directly (if small enough).
- **Combine** solutions to subproblem(s) into solution for original instance.

Divide and Conquer Strategy

Towers of Hanoi

Move n discs from first to third pole by

- $n = 1$: move disc directly.
- $n > 1$: move $n - 1$ discs to middle pole, move largest to third pole, move $n - 1$ discs from middle to right pole.

Factorial

$n!$ =

- $n = 0$: 1
- $n > 0$: $n(n - 1)!$

No real “division” to speak of.

MergeSort

MergeSort

To sort a sequence of n values:

- If the sequence is 1 value or empty: done.
- If $n > 1$:
  - Divide: "Split" the sequence in half.
  - Conquer: Sort left and right half.
  - Combine: Merge sorted halves together.

```
def sort[A](data: Seq[A]): Seq[A] = {
  if(data.length <= 1) data
  else {
    val (left,right) = data.splitAt(data.length/2)
    // Sort each half, combine the sorted results.
    merge(sort(left),sort(right));
  }
}
```

MergeSort: Merge

```scala
def merge[A](left: Seq[A], right: Seq[A])(implicit comp:
    Ordering[A]): Seq[A] = {
    if (left.length == 0) right
    else if (right.length == 0) left
    else if (comp.lt(left.head, right.head))
      left.head +: merge(left.tail, right)
    else right.head +: merge(left, right.tail)
}
```

Recursive Analysis

Recursive Analysis

Runtime based on **recurrence equation** or **recurrence**.
Space usage is based on levels of the recursion tree.

Consider a recursive solution that performs:

- Division into a subproblems.
- Size of each subproblem is $1/b$ of the original size.
- Time $D(n)$ to divide and $C(n)$ to combine.

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c. \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

MergeSort Analysis

Suppose `data` is a sequence of size n.

- Assume n is a power of 2 to simplify analysis.

- Divide:
  "Split" the sequence in half.
- Conquer:
  Sort left and right half.
- Combine:
  Merge sorted halves together.

- $D(n) = \Theta(n)$.
- Division into $a = 2$ parts, each of size $b = 1/2$ the input size.
- $C(n) = \Theta(n)$.

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1. \\ 2T(\frac{n}{2}) + \Theta(n) + \Theta(n) = 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise.} \end{cases}$$

MergeSort: Recursion Tree

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1. \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise.} \end{cases}$$

One approach to finding the closed form: draw a **recursion tree.**

MergeSort Analysis

Substitution Method

1. Obtain a tight-bound somehow.
2. Use strong induction to show correctness of upperbound.**

We can verify that $T(n) = O(n \log(n))$.

Prove: $T(n) \leq cn \log(n)$ for a suitable $c > 0$.

Let $n > 1$ be some integer. Let $c > 0$ (defined later).
Assume: $\forall m < n, T(m) = cm \log(m)$.

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &\leq 2(cn/2 \log(n/2)) + \Theta(n) \\ &= cn \log(n) - cn \log(2) + \Theta(n) \\ &\leq cn \log(n) - cn + dn \text{ (some constant } d > 0) \\ &\leq cn \log(n) \text{ as long as } c \geq d. \end{aligned}$$

QuickSort

We can create another sorting algorithm that is similar to Mergesort.

Quicksort

- “Divide”: Partition the sequence by pivot element p (in-place).
- Conquer: Sort left and right half (in-place).
- Combine: ---.

QuickSort

Move all `data` elements $\leq$ `pivot` to the range `[lower,mid]`.

```
def partition[A](data: Array[A], lower: Int, upper:
    Int)(implicit comp: Ordering[A]): Int = {
  val pivot = data(upper-1)
  var mid = lower-1
  for (i <- lower until upper-1) {
    if (comp.lteq(data(i),pivot)) {
      mid += 1
      swap(data,mid,i)
    }
  }
  swap(data,mid+1,upper-1)
  mid+1
}
```

QuickSort

```
def sort[A](data: Array[A])(...): Unit = {
  def sortRange(data: Array[A], lower: Int, upper:
   Int): Unit = {
    if(lower < upper) {
      val pivotIndex = partition(data,lower,upper)
      sortRange(data,lower,pivotIndex)
      sortRange(data,pivotIndex+1,upper)
    }
  }
  sortRange(data,0,data.length)
}
```

QuickSort

Expect to split roughly in half each time.

- For arbitrary value p, expect $(n-1)/2$ values smaller than p.

To ease the unease: Select random p or randomly permute `data` prior.

Expected recurrence for runtime:

$$T(n) \leq \begin{cases} O(1) & \text{if } n \leq 1. \\ 2T(\frac{n}{2}) + O(n) & \text{otherwise.} \end{cases}$$

Expected runtime: $T(n) = O(n \log(n))$.

Benefits:

Performs sort **in-place**. Significantly reduces memory overhead.

- See also: https://www.scala-lang.org/api/current/scala/util/Sorting$.html

Bibliography

A. Alexander, *Scala Cookbook*.
O'Reilly Media, 2013.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein,
Introduction to Algorithms, Third Edition.
The MIT Press, 3rd ed., 2009.