

AP Computer Science A

Java Programming Essentials [Ver.4.0]

Unit 4: Data Collections

CHAPTER 19: ALGORITHM STUDY
AND NUMBER ALGORITHMS

DR. ERIC CHOU
IEEE SENIOR MEMBER



AP Computer Science Curriculum

- Not in AP CSA
- Yes int AP CSP

Objectives

- What is algorithm?
- Execution Time Versus Growth Rate
- Series (Mathematical Background)
- Big-O Notation
- Time Complexity

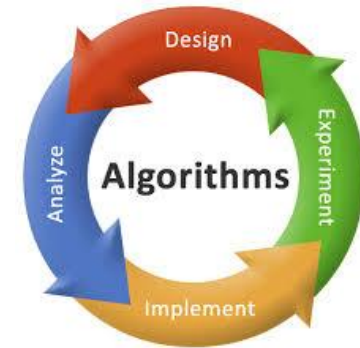


Overview of Algorithms

Lecture 1

Algorithm

How to solve a problem?



- A systematic process consisting of an ordered sequence of steps, each step depending on the outcome of the previous one.
- **Methodology (Problem Solving)**
- **Resource Estimation**
- Time Complexity
- Memory Complexity

Google Algorithm



Algorithm Study

How Efficiently the algorithm can solve a problem?

- **Problem Solving**

- **Sorting**
- Approximate Methods (ad hoc solutions)
- Randomized Solutions
- Dynamic Programming
- Linear Programming
- Cryptography
- Fourier Transform
- Computational Geometry
- Graph Theory

Algorithm Efficiency Analysis (Time Complexity)

Big-O Notation

NP-Completeness

Data Structure (Memory Complexity/Time Complexity Trade-offs)

Binary

B-Tree and other Tree

Stack, Queue, List, Map, Set, Heap, Sparse Matrix

Executing Time

Suppose two algorithms perform the same task such as search (linear search vs. binary search) and sorting (selection sort vs. insertion sort). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

Growth Rate

- It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their **growth rates**.

Memory Complexity is Similar to Time Complexity Except for the Goal Function

- $T(n)$: Time complexity function Time requirement when n grows
- $M(n)$: Memory complexity function Memory requirement when n grows

Why Study Algorithm?

- Remember the recursive versus iterative algorithm for Fibonacci number?
- Poor algorithm leads to time-consuming solution and waste of memory.
- Good algorithm leads to good solution which meets program latency requirement and does not take too much resource.
- Computer Scientist's job is to find good solution for problems.



Review of Sequence and Series

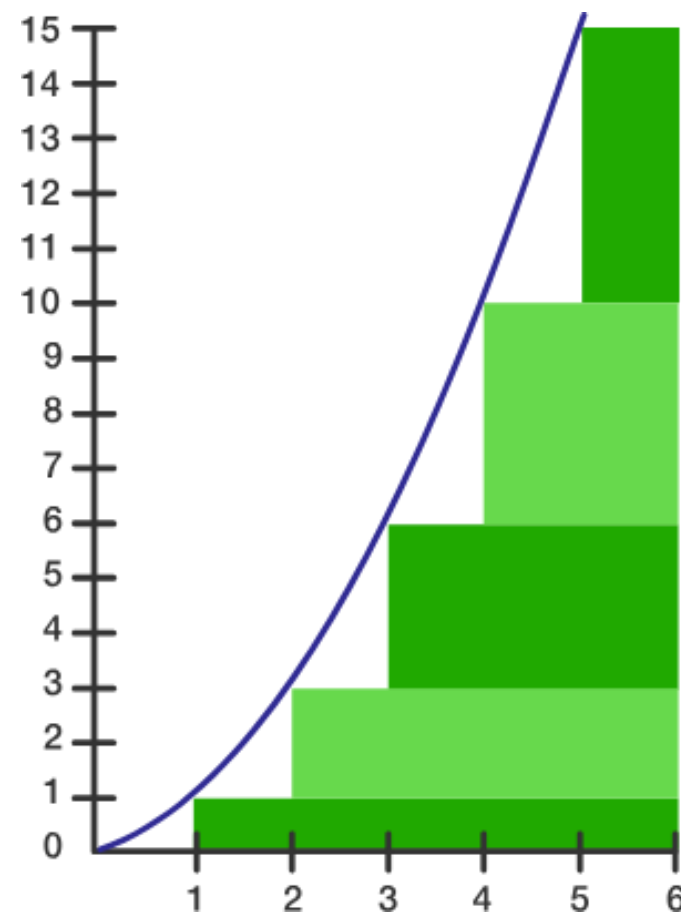
Lecture 2

Sum of n , n^2 , or n^3

The series $\sum_{k=1}^n k^a = 1^a + 2^a + 3^a + \cdots + n^a$ gives the sum of the a^{th} powers of the first n positive numbers, where a and n are positive integers. Each of these series can be calculated through a closed-form formula. The case $a = 1$, $n = 100$ is famously said to have been solved by [Gauss](#) as a young schoolboy: given the tedious task of adding the first 100 positive integers, Gauss quickly used a formula to calculate the sum of 5050.

The formulas for the first few values of a are as follows:

$$\begin{aligned}\sum_{k=1}^n k &= \frac{n(n+1)}{2} \\ \sum_{k=1}^n k^2 &= \frac{n(n+1)(2n+1)}{6} \\ \sum_{k=1}^n k^3 &= \frac{n^2(n+1)^2}{4}.\end{aligned}$$



Sum of the First n Positive Integers

Let $S_n = 1 + 2 + 3 + 4 + \cdots + n = \sum_{k=1}^n k$. The elementary trick for solving this equation (which Gauss is supposed to have used as a child) is a rearrangement of the sum as follows:

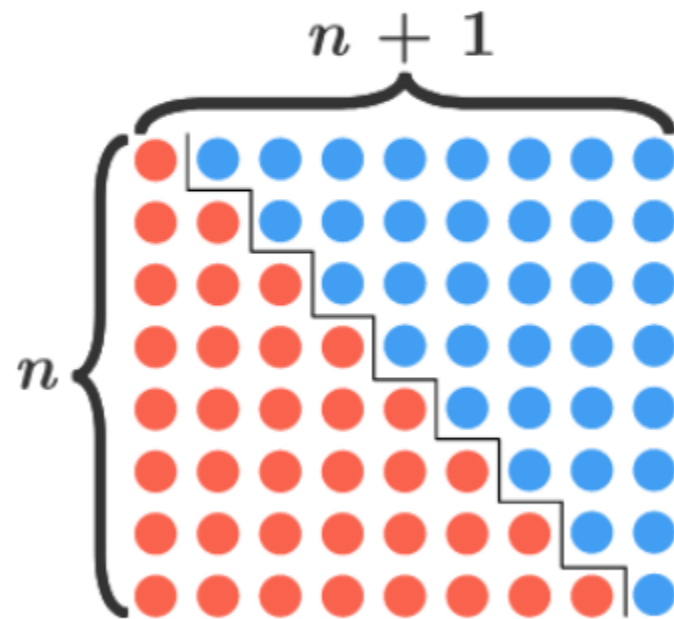
$$\begin{aligned} S_n &= 1 + 2 + 3 + \cdots + n \\ S_n &= n + (n-1) + (n-2) + \cdots + 1. \end{aligned}$$

Grouping and adding the above two sums gives

$$\begin{aligned} 2S_n &= (1+n) + (2+n-1) + (3+n-2) + \cdots + (n+1) \\ &= \underbrace{(n+1) + (n+1) + (n+1) + \cdots + (n+1)}_{n \text{ times}} \\ &= n(n+1). \end{aligned}$$

Therefore,

$$S_n = \frac{n(n+1)}{2}.$$



EXAMPLE

Find the sum of the first 100 positive integers.

Plugging $n = 100$ in our equation,

$$1 + 2 + 3 + 4 + \cdots + 100 = \frac{100(101)}{2} = \frac{10100}{2},$$

which implies our final answer is 5050. \square

EXAMPLE

Show that the sum of the first n positive odd integers is n^2 .

There are several ways to solve this problem. One way is to view the sum as the sum of the first $2n$ integers minus the sum of the first n even integers. The sum of the first n even integers is 2 times the sum of the first n integers, so putting this all together gives

$$\frac{2n(2n+1)}{2} - 2\left(\frac{n(n+1)}{2}\right) = n(2n+1) - n(n+1) = n^2.$$

Even more succinctly, the sum can be written as

$$\sum_{k=1}^n (2k-1) = 2 \sum_{k=1}^n k - \sum_{k=1}^n 1 = 2 \frac{n(n+1)}{2} - n = n^2. \quad \square$$

In a similar vein to the previous exercise, here is another way of deriving the formula for the sum of the first n positive integers. Start with the binomial expansion of $(k - 1)^2$:

$$(k - 1)^2 = k^2 - 2k + 1.$$

Rearrange the terms as below:

$$k^2 - (k - 1)^2 = 2k - 1.$$

Now sum both sides:

$$\sum_{k=1}^n (k^2 - (k - 1)^2) = 2 \sum_{k=1}^n k - \sum_{k=1}^n 1.$$

The left sum [telescopes](#): it equals n^2 . The right side equals $2S_n - n$, which gives $2S_n - n = n^2$, so $S_n = \frac{n(n+1)}{2}$.

This technique generalizes to a computation of any particular power sum one might wish to compute.

Sum of the Squares of the First n Positive Integers

Continuing the idea from the previous section, start with the binomial expansion of $(k - 1)^3$:

$$(k - 1)^3 = k^3 - 3k^2 + 3k - 1.$$

Rearrange the terms:

$$k^3 - (k - 1)^3 = 3k^2 - 3k + 1.$$

As before, summing the left side from $k = 1$ to n yields n^3 . This gives

$$\begin{aligned} n^3 &= 3 \left(\sum_{k=1}^n k^2 \right) - 3 \sum_{k=1}^n k + \sum_{k=1}^n 1 \\ n^3 &= 3 \left(\sum_{k=1}^n k^2 \right) - 3 \frac{n(n+1)}{2} + n \\ 3 \left(\sum_{k=1}^n k^2 \right) &= n^3 + 3 \frac{n(n+1)}{2} - n \\ \Rightarrow \sum_{k=1}^n k^2 &= \frac{1}{3} n^3 + \frac{1}{2} n^2 + \frac{1}{6} n \\ &= \frac{n(n+1)(2n+1)}{6}. \end{aligned}$$

EXAMPLE

Find the sum of the squares of the first 100 positive integers.

Plugging in $n = 100$,

$$1^2 + 2^2 + 3^2 + 4^2 + \cdots + 100^2 = \frac{100(101)(201)}{6} = \frac{2030100}{6} = 338350. \quad \square$$

Sum of the Cubes of the First n Positive Integers

Again, start with the binomial expansion of $(k - 1)^4$ and rearrange the terms:

$$k^4 - (k - 1)^4 = 4k^3 - 6k^2 + 4k - 1.$$

Sum from 1 to n to get

$$n^4 = 4s_{3,n} - 6s_{2,n} + 4s_{1,n} - n.$$

Here $s_{a,n}$ is the sum of the first n a^{th} powers. So

$$4s_{3,n} = n^4 + 6\frac{n(n+1)(2n+1)}{6} - 4\frac{n(n+1)}{2} + n$$

$$s_{3,n} = \frac{1}{4}n^4 + \frac{1}{2}n^3 + \frac{3}{4}n^2 + \frac{1}{4}n - \frac{1}{2}n^2 - \frac{1}{2}n + \frac{1}{4}n$$

$$s_{3,n} = \frac{1}{4}n^4 + \frac{1}{2}n^3 + \frac{1}{4}n^2$$

$$= \frac{n^2(n+1)^2}{4}.$$

EXAMPLE

Find the sum of the cubes of the first 200 positive integers.

Plugging $n = 200$ in our equation,

$$1^3 + 2^3 + 3^3 + 4^3 + 5^3 + 6^3 + 7^3 + 8^3 \cdots + 200^3 = \frac{200^2(201^2)}{4} = \frac{1616040000}{4} = 404010000. \square$$

TRY IT YOURSELF

$$\sum_{n=1}^{10} n(1+n+n^2) = ?$$

EXAMPLE

Simplify

$$2 + 4 + 6 + \cdots + 2n.$$

We have

$$\begin{aligned} 2 + 4 + 6 + \cdots + 2n &= \sum_{i=1}^n 2i \\ &= 2(1 + 2 + 3 + \cdots + n) \\ &= 2 \times \frac{n(n+1)}{2} \\ &= n(n+1). \quad \square \end{aligned}$$

EXAMPLE

Simplify

$$1 + 3 + 5 + \cdots + (2n - 1).$$

We have

$$\begin{aligned} 1 + 3 + 5 + \cdots + (2n - 1) &= \sum_{i=1}^n (2i - 1) \\ &= \sum_{i=1}^n 2i - \sum_{i=1}^n 1 \\ &= 2 \sum_{i=1}^n i - n \\ &= 2 \times \frac{n(n+1)}{2} - n \\ &= n(n+1) - n \\ &= n(n+1-1) \\ &= n^2. \quad \square \end{aligned}$$

EXAMPLE

Simplify

$$2^2 + 4^2 + 6^2 + \cdots + (2n)^2.$$

We have

$$\begin{aligned} 2^2 + 4^2 + 6^2 + \cdots + (2n)^2 &= \sum_{i=1}^n (2i)^2 \\ &= \sum_{i=1}^n (2^2 i^2) \\ &= 4 \sum_{i=1}^n i^2 \\ &= 4 \cdot \frac{n(n+1)(2n+1)}{6} \\ &= \frac{2n(n+1)(2n+1)}{3}. \quad \square \end{aligned}$$

EXAMPLE

Simplify

$$1^2 + 3^2 + 5^2 + \cdots + (2n-1)^2.$$

We have

$$\begin{aligned} 1^2 + 3^2 + 5^2 + \cdots + (2n-1)^2 &= (1^2 + 2^2 + 3^2 + 4^2 + \cdots + (2n-1)^2 + (2n)^2) - (2^2 + 4^2 + 6^2 + \cdots + (2n)^2) \\ &= \sum_{i=1}^{2n} i^2 - \sum_{i=1}^n (2i)^2 \\ &= \frac{2n(2n+1)(4n+1)}{6} - \frac{2n(n+1)(2n+1)}{3} \\ &= \frac{n(2n+1)((4n+1) - 2(n+1))}{3} \\ &= \frac{n(2n-1)(2n+1)}{3}. \quad \square \end{aligned}$$



Generalization

Lecture 4

Generalizations

As in the previous section, let $s_{a,n} = \sum_{k=1}^n k^a$. Then the relevant identity, derived in the same way from the binomial expansion, is

$$n^{a+1} = \binom{a+1}{1} s_{a,n} - \binom{a+1}{2} s_{a-1,n} + \binom{a+1}{3} s_{a-2,n} - \cdots + (-1)^{a-1} \binom{a+1}{a} s_{1,n} + (-1)^a n.$$

This recursive identity gives a formula for $s_{a,n}$ in terms of $s_{b,n}$ for $b < a$. It is the basis of many inductive arguments. In particular, the first pattern that one notices after deriving $s_{a,n}$ for $a = 1, 2, 3$ is the leading terms $\frac{1}{2}n^2, \frac{1}{3}n^3, \frac{1}{4}n^4$. Here is an easy argument that the pattern continues:

THEOREM

For a positive integer a , $s_{a,n}$ is a polynomial of degree $a + 1$ in n . Its leading term is $\frac{1}{a+1}n^{a+1}$.

PROOF

Induction. The statement is true for $a = 1$, and now suppose it is true for all positive integers less than a . Then solve the above recurrence for $s_{a,n}$ to get

$$s_{a,n} = \frac{1}{a+1}n^{a+1} + c_{a-1}s_{a-1,n} + c_{a-2}s_{a-2,n} + \cdots + c_1s_{1,n} + c_0n,$$

where the c_i are some rational numbers.

Now by the inductive hypothesis, all of the terms except for the first term are polynomials of degree $\leq a$ in n , so the statement follows. \square

Note the analogy to the continuous version of the sum: the integral $\int_0^n x^a dx = \frac{1}{a+1}n^{a+1}$. The lower-degree terms can be viewed as error terms in the approximation of the area under the curve $y = x^a$ by the rectangles of width 1 and height k^a .

Faulhaber's Formula

Having established that $s_{a,n} = \frac{1}{a+1}n^{a+1} + (\text{lower terms})$, the obvious question is whether there is an explicit expression for the lower terms. It turns out that the terms can be expressed quite concisely in terms of the [Bernoulli numbers](#), as follows:

THEOREM

Faulhaber's Formula:

$$\sum_{k=1}^n k^a = \frac{1}{a+1} \sum_{j=0}^a (-1)^j \binom{a+1}{j} B_j n^{a+1-j}.$$

That is, if $i = a + 1 - j$ is a positive integer, the coefficient of n^i in the polynomial expression for the sum is $\frac{(-1)^{a+1-i}}{a+1} \binom{a+1}{i} B_{a+1-i}$.

EXAMPLE

Show that $\sum_{k=1}^n k^a = \frac{1}{a+1} n^{a+1} + \frac{1}{2} n^a + (\text{lower terms})$.

This can be read off directly from Faulhaber's formula: the $j = 0$ term is $\frac{1}{a+1} n^{a+1}$, and the $j = 1$ term is

$$\frac{1}{a+1} (-1)^1 \binom{a+1}{1} B_1 n^a,$$

and since $B_1 = -\frac{1}{2}$, this simplifies to $\frac{1}{2} n^a$. \square

EXAMPLE

To compute $\sum_{k=1}^n k^4$ using Faulhaber's formula, write

$$\sum_{k=1}^n k^4 = \frac{1}{5} \sum_{j=0}^4 (-1)^j \binom{5}{j} B_j n^{5-j}$$

and use $B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_3 = 0, B_4 = -\frac{1}{30}$ to get

$$\sum_{k=1}^n k^4 = \frac{1}{5} \left(n^5 + \frac{5}{2}n^4 + \frac{10}{6}n^3 + 0n^2 - \frac{1}{6}n \right) = \frac{1}{5}n^5 + \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{6}n.$$

This happens to factor as

$$\sum_{k=1}^n k^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}.$$

Note that the $(-1)^j$ sign only affects the term when $j = 1$, because the odd Bernoulli numbers are zero except for $B_1 = -\frac{1}{2}$.

The proof of the theorem is straightforward (and is omitted here); it can be done inductively via standard recurrences involving the Bernoulli numbers, or more elegantly via the generating function for the Bernoulli numbers.



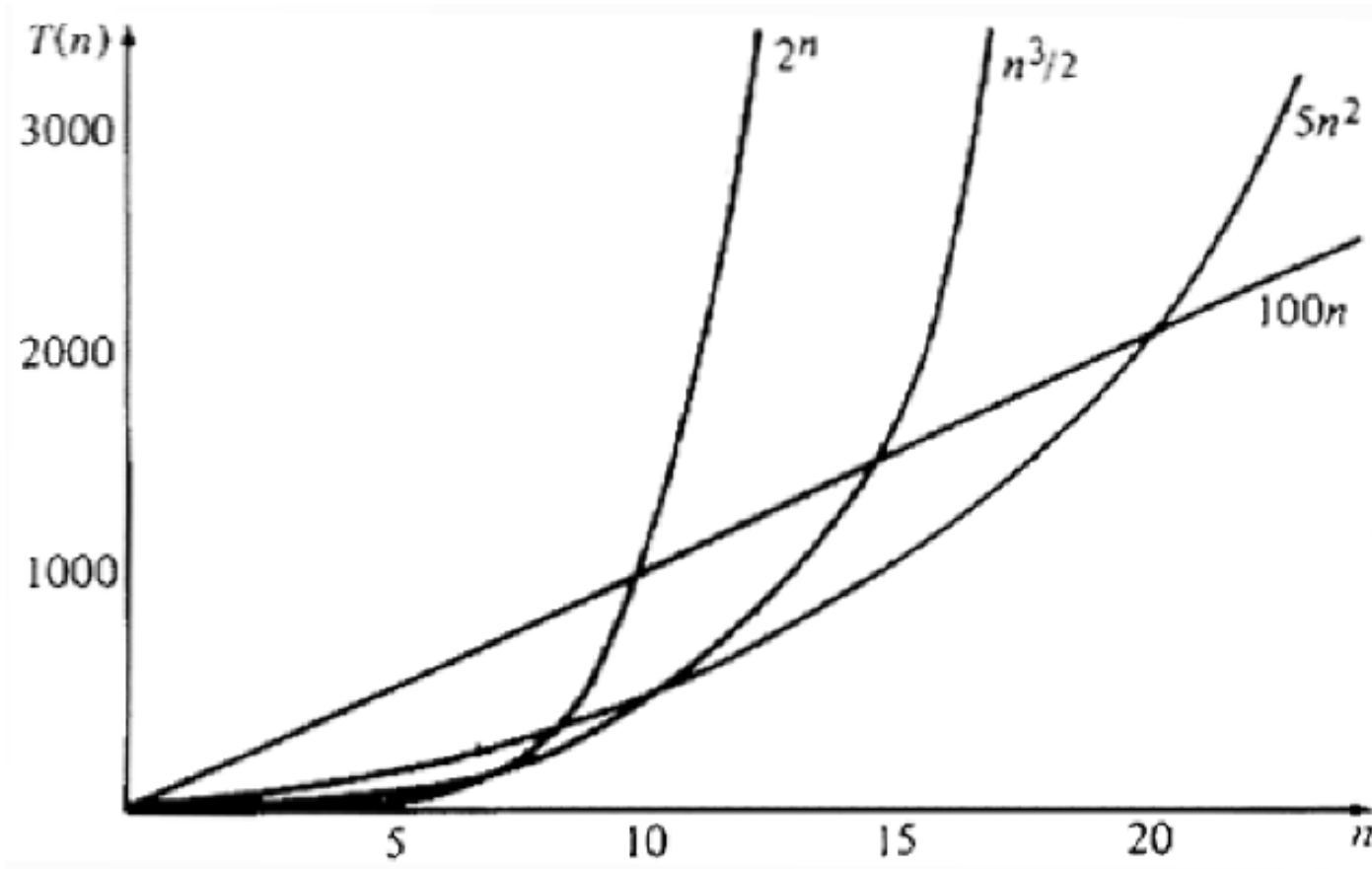
Analysis of Algorithms: Big-O Notation

Lecture 5

Big O Notation

- Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires n comparisons for an array of size n . If the key is in the array, **it requires $n/2$ comparisons on average.**
- The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of n . Computer scientists use the Big O notation to abbreviate for “order of magnitude.” Using this notation, the complexity of the linear search algorithm is **$O(n)$** , pronounced as “**order of n .**”

Approximate Growth Rate $T(n)$



Best, Worst, and Average Cases

- For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.
- **An average-case analysis attempts to determine the average amount of time among all possible input of the same size.** Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

Ignoring Multiplicative Constants

- The linear search algorithm requires n comparisons in the worst-case and $n/2$ comparisons in the average-case. Using the Big O notation, both cases require **$O(n)$** time. The multiplicative constant **$(1/2)$** can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.
- The growth rate for $n/2$ or $100n$ is the same as n , i.e., **$O(n)$** = **$O(n/2)$** = **$O(100n)$** .

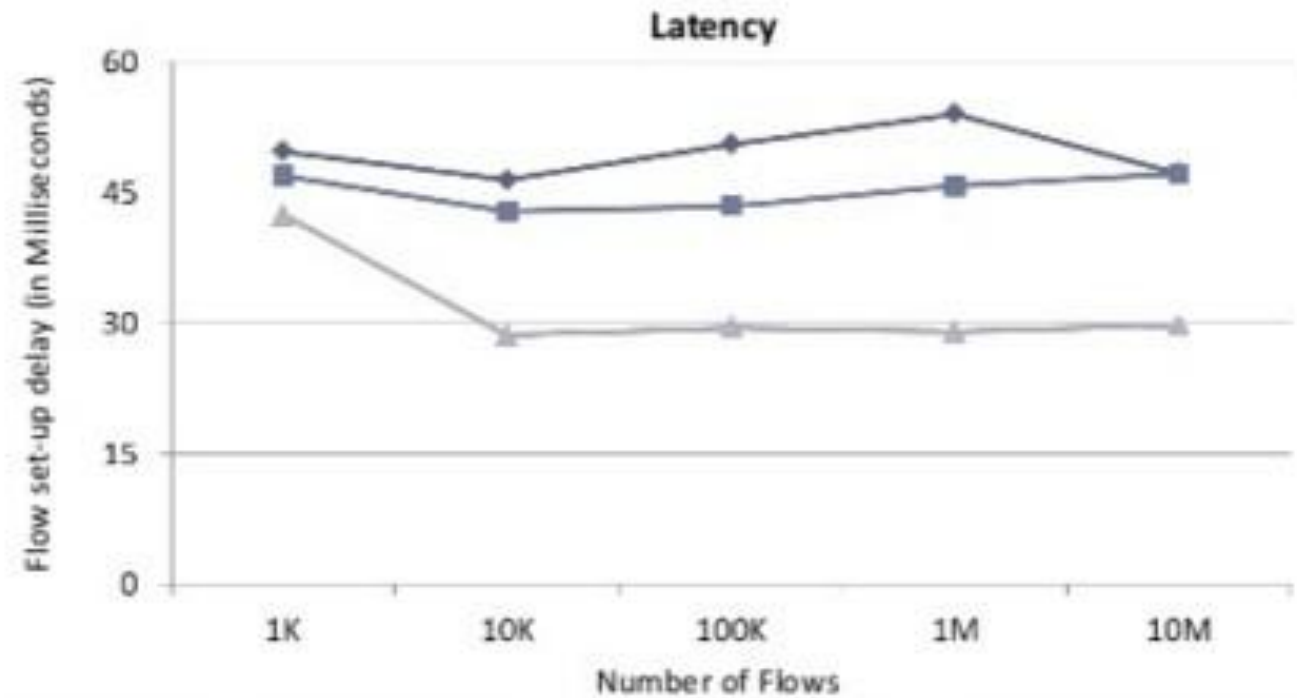
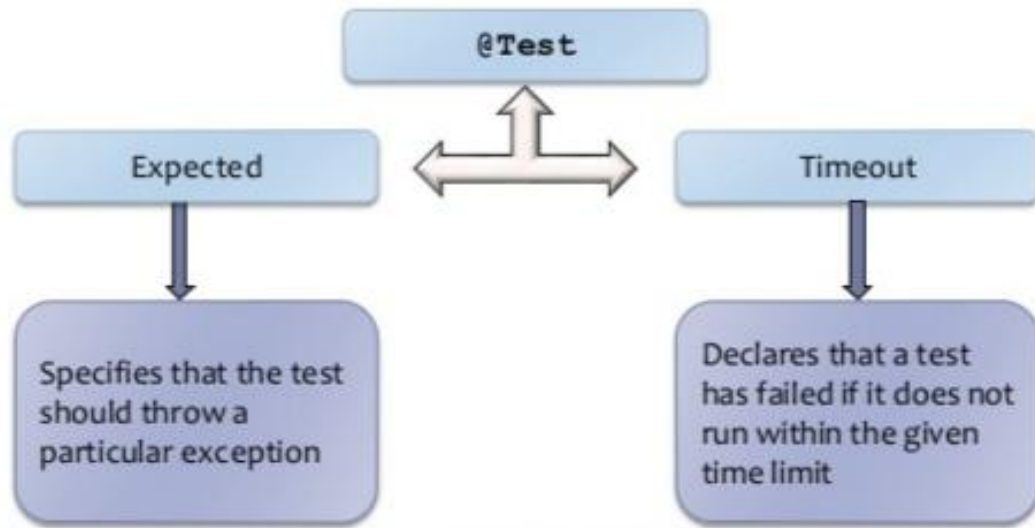
Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes $n-1$ times of comparisons to find maximum number in a list of n elements.
- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As n grows larger, the n part in the expression $n-1$ dominates the complexity.
- The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression $n-1$) and highlight the important part (e.g., n in the expression $n-1$). So, the complexity of this algorithm is $O(n)$.

Constant Time

- The Big **$O(n)$** notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation **$O(1)$** .
- For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

Timeout Test (JUnit Testing)





Time Complexity

Lecture 6

Determining Big-O

- Repetition
- Sequence
- Selection
- Logarithm

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

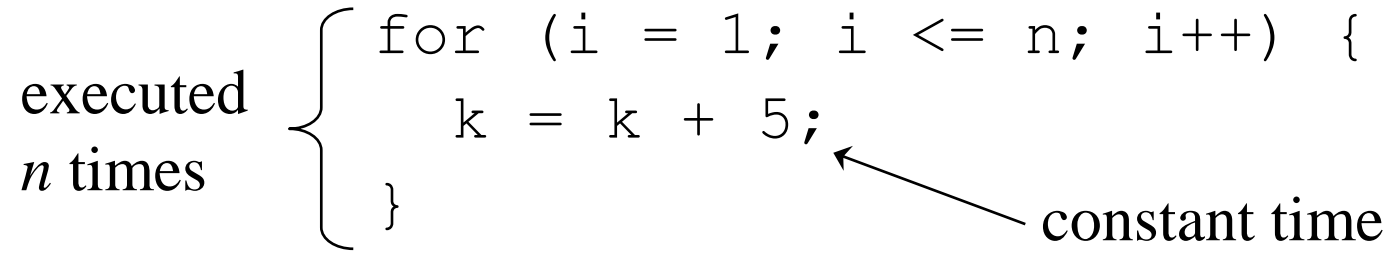
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

Repetition: Simple Loops

executed n times

```
{ for (i = 1; i <= n; i++) {  
    k = k + 5;  
}
```

constant time



Time Complexity

$$T(n) = (\text{a constant } c) * n = cn = \mathbf{O(n)}$$

Ignore multiplicative constants (e.g., "c").

Repetition: Nested Loops

executed
 n times

```
{ for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop
executed
 n times

constant time

Time Complexity

$$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$$

Ignore multiplicative constants (e.g., “c”).

Repetition: Nested Loops

executed n times {
 for ($i = 1; i \leq n; i++$) {
 for ($j = 1; j \leq i; j++$) {
 $k = k + i + j;$ inner loop executed i times
 }
 }
}

constant time

Time Complexity

$$T(n) = c + 2c + 3c + 4c + \dots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$$

Ignore non-dominating terms

Ignore multiplicative constants

Repetition: Nested Loops

executed n times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= 20; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed 20 times

constant time

Time Complexity

$$T(n) = 20 * c * n = O(n)$$

*Ignore multiplicative constants (e.g., $20*c$)*

Sequence

executed
10 times

```
{  
  for (j = 1; j <= 10; j++) {  
    k = k + 4;  
  }  
}
```

executed
n times

```
{  
  for (i = 1; i <= n; i++) {  
    for (j = 1; j <= 20; j++) {  
      k = k + i + j;  
    }  
  }  
}
```

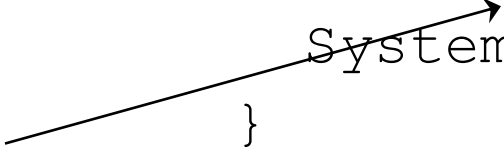
} inner loop
executed
20 times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

Selection

```
if (list.contains(e)) {  
    System.out.println(e);  
}  
else  
    for (Object t: list) {  
        System.out.println(t);  
    }
```

$O(n)$ 

Let n be
`list.size()`.
Executed
 n times.

Time Complexity

$$\begin{aligned} T(n) &= \text{test time} + \text{worst-case (if, else)} \\ &= O(n) + O(n) \\ &= O(n) \end{aligned}$$



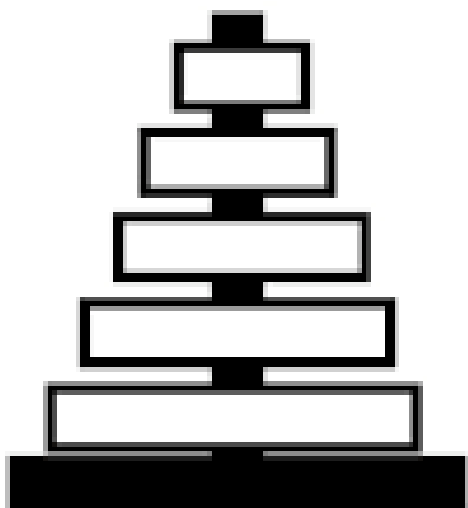
Tower of Hanoi

(Non-AP Topic)

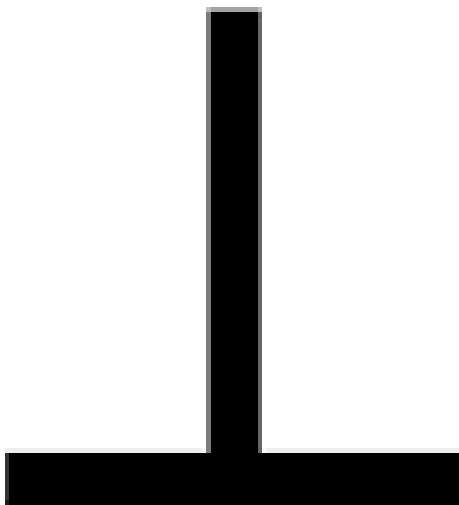
Lecture 9



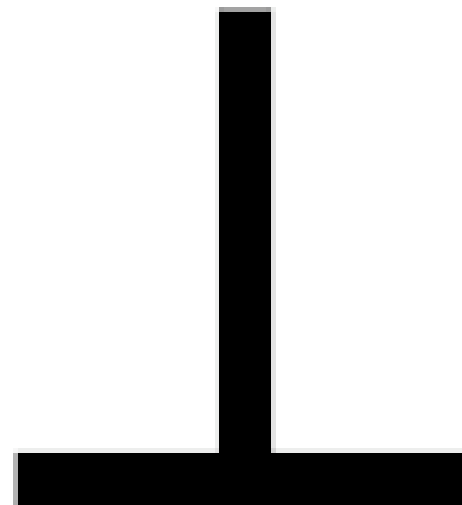
Towers of Hanoi



A



B

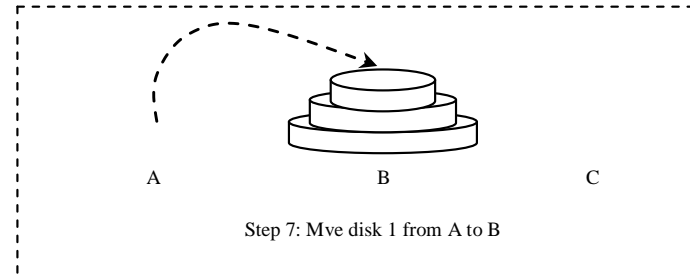
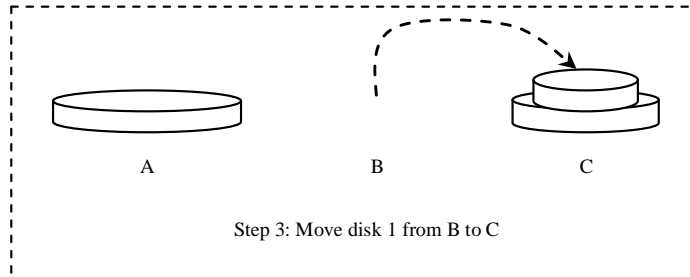
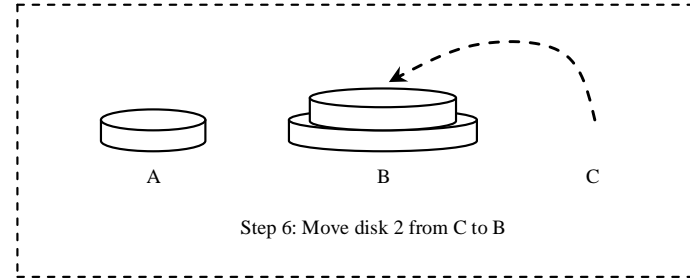
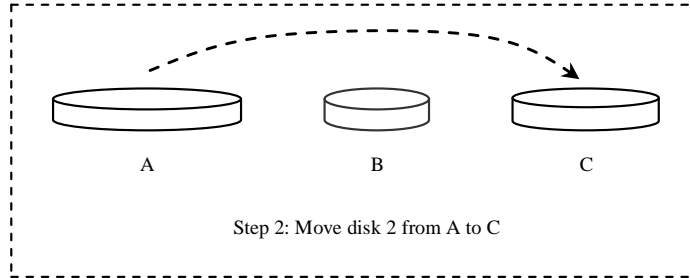
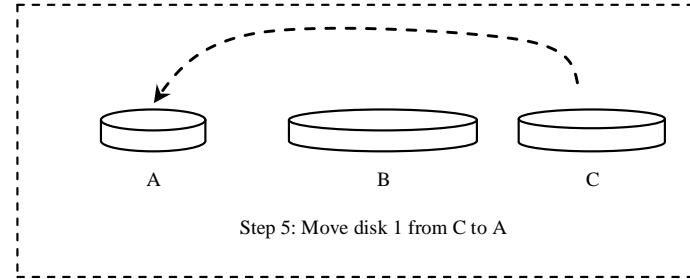
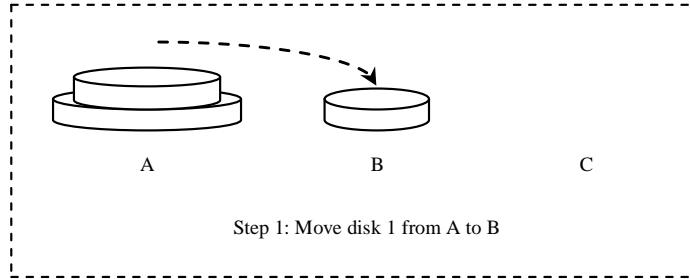
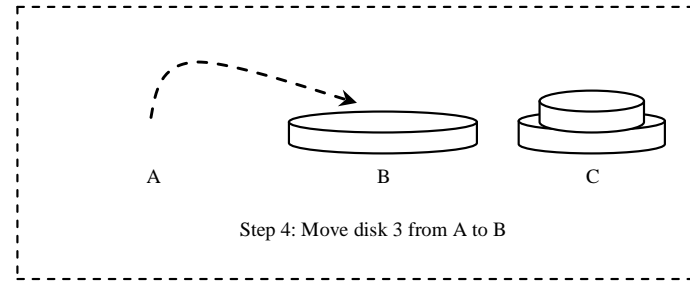
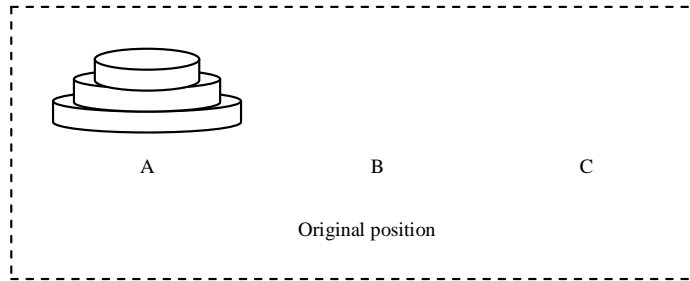


C

Towers of Hanoi

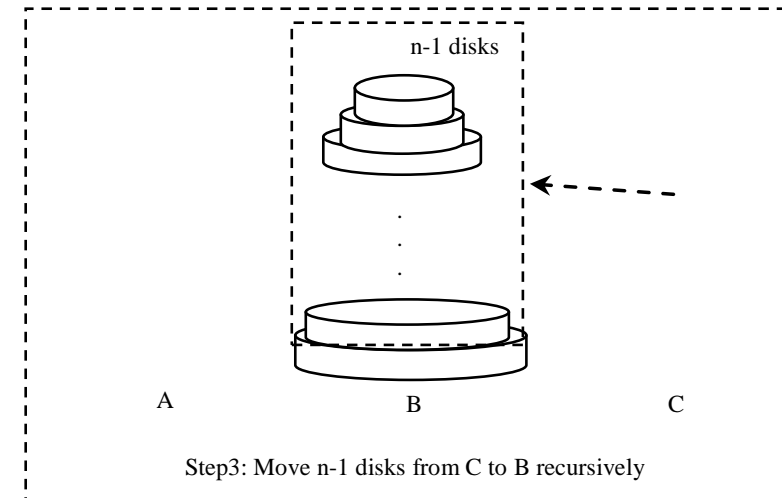
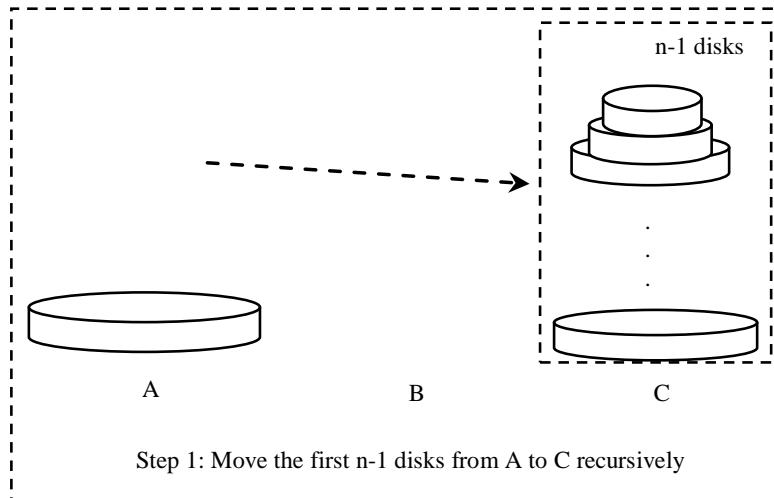
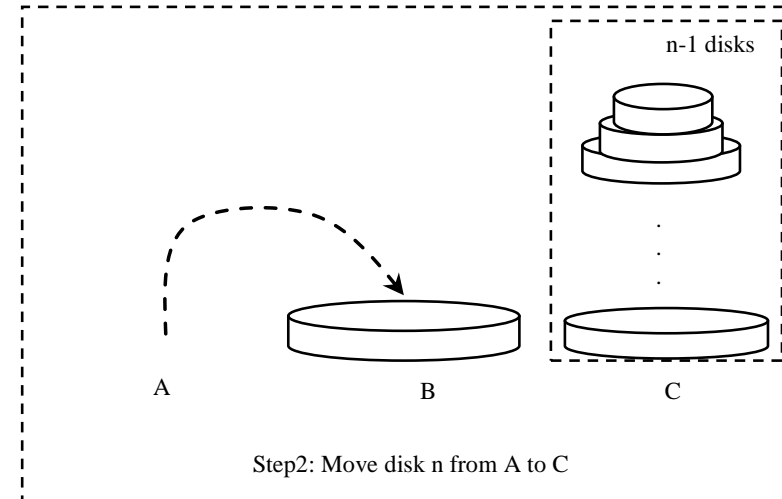
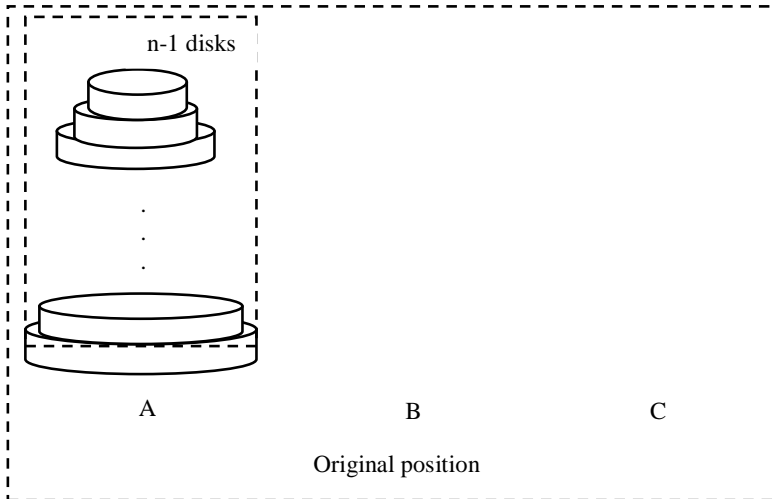
- There are n disks labeled $1, 2, 3, \dots, n$, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Towers of Hanoi



Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.



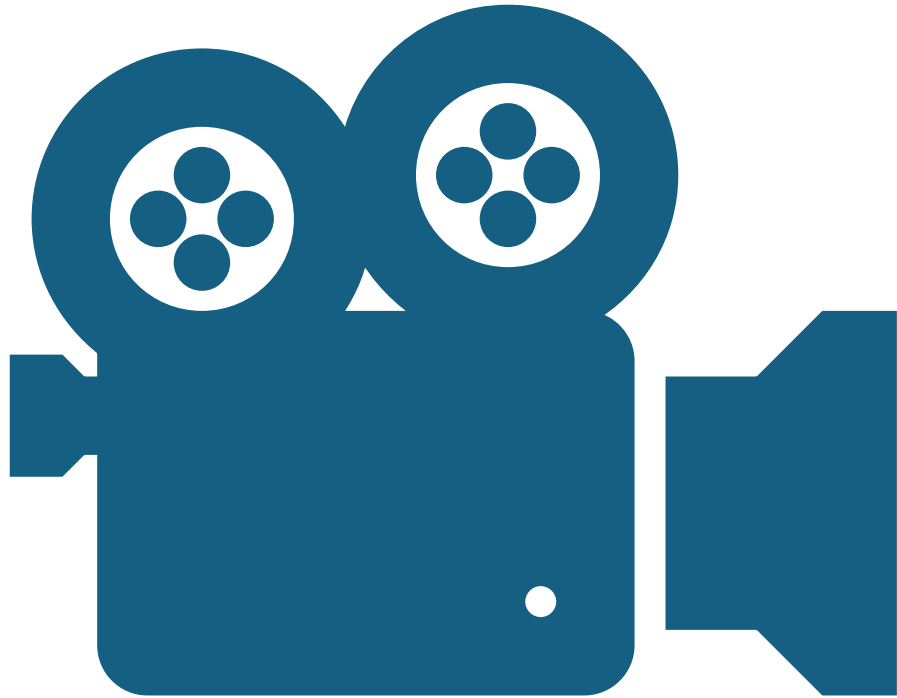
Solution to Towers of Hanoi

- Move the first $n - 1$ disks from A to C with the assistance of tower B.
- Move disk n from A to B.
- Move $n - 1$ disks from C to B with the assistance of tower A.



Demonstration Program

TOWERSOFHANOI.HAVA



Tower of Hanoi

VIDEO

[HTTPS://YOUTU.BE/SCEJ_KIQ7XE](https://youtu.be/SCEJ_KIQ7XE)

Analyzing Towers of Hanoi

The Towers of Hanoi problem presented in later lecture, TowersOfHanoi.java, moves n disks from tower A to tower B with the assistance of tower C recursively as follows:

Move the first $\underline{n - 1}$ disks from A to C with the assistance of tower B.

Move disk \underline{n} from A to B.

Move $\underline{n - 1}$ disks from C to B with the assistance of tower A.

Let $T(n)$ denote the complexity for the algorithm that moves disks and c denote the constant time to move one disk, i.e., $T(1)$ is c . So,

$$\begin{aligned} T(n) &= T(n - 1) + c + T(n - 1) = 2T(n - 1) + c \\ &= 2(2(T(n - 2) + c) + c) = 2^n T(1) + c2^{n-1} + \dots + c2 + c = \\ &= c2^n + c2^{n-1} + \dots + c2 + c = c(2^{n+1} - 1) = O(2^n) \end{aligned}$$