Bioinformatics Algorithms
COS-BIOL-530/630
Lecture08

Days & Times	Room	Meeting Dates
Tu 2:00PM - 3:50PM	Thomas Gosnell Hall (GOS)-2178	01/13/2025 - 04/28/2025
Th 2:00PM - 3:50PM	Thomas Gosnell Hall (GOS)-2178	01/13/2025 - 04/28/2025

Instructor:
Fernando Rodriguez
email: frvsbi@rit.edu
Office: Orange Hall 1311

Algorithms and pattern matching
- Lecture08-

Announcements

Week 8
Lab07
Lecture08/Lab08

Quiz6 (Lecture07-Gene ontologies) opens Friday, March 7th

Quiz7 (Lecture08-Algorithms and Motif analysis) opens Friday, March 14th

Algorithms and pattern matching - Lecture08-

Topics:

- Algorithms and complexity
- Algorithm design techniques
- Pattern matching search

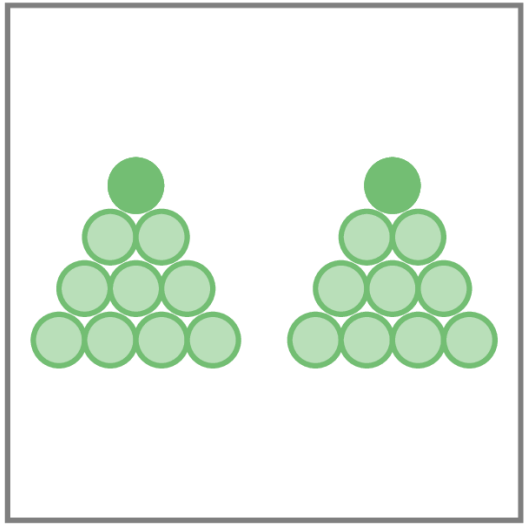
Algorithm definition

- Wikipedia: In mathematics and computer science, an algorithm is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation.
- Dictionary (Merriam-Webster): a step-by-step procedure for solving a problem or accomplishing some end.
- Algorithms can be designed to perform calculations, process data, or perform automated reasoning tasks.

Algorithm definition

Two Rocks Game

There are two piles of ten rocks. In each turn, you and your opponent may either take one rock from a single pile, or one rock from both piles. Your opponent moves first and the player that takes the last rock wins the game. Design a winning strategy.



Source :https://cogniterra.org

Algorithm definition

Two Rocks Game

- Since YOU know Bioinformatics Algorithms, you shouldn't have problems designing a winning strategy: recipe-style instructions are not a sufficiently expressive language for describing algorithms.

- Draw the following table filled with the symbols ↑, ←, ↖, and ∗.
  - The entry in position (i, j) (i.e., the i^{th} row and the j^{th} column) describes the moves that you will make in the i + j game, with i and j rocks in piles A and B respectively.
  - A ← indicates that she should take one stone from pile B.
  - A ↑ indicates that she should take one stone from pile A.
  - A ↖ indicates that she should take one stone from each pile
  - And ∗ indicates that you should not bother playing the game because you will definitely lose against an opponent who has a clue.

- This table, sounds familiar??

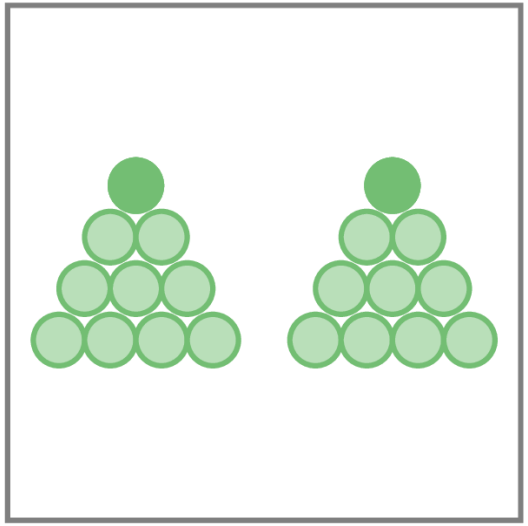
Algorithm definition

Two Rocks Game

	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
2	*	←	*	←	*	←	*	←	*	←	*
3	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
4	*	←	*	←	*	←	*	←	*	←	*
5	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
6	*	←	*	←	*	←	*	←	*	←	*
7	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
8	*	←	*	←	*	←	*	←	*	←	*
9	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
10	*	←	*	←	*	←	*	←	*	←	*

Algorithm definition

- The Rock game is in fact the ubiquitous sequence alignment problem in disguise.
- The computational idea used to solve both problems is the same.
- Now, you can try to play the game against the machine with our algorithm.



Play here :
https://discrete-math-puzzles.github.io/puzzles/take-the-last-stone/index.html

Algorithm definition

There are specific requirements that an algorithm must abide by:

- Definiteness: Each step in the process is precisely stated.
- Effective Computability: Each step in the process can be carried out by a computer.
- Finiteness: The program will eventually successfully terminate.
- May have an *Input*.
- Must produce an *Output*.

Some types of Algorithms

- Recursive algorithms
- Exhaustive search – Brut force
- Branch and Bound algorithms
- Greedy algorithms
- Dynamic programming
- Divide and Conquer algorithms
- Machine learning
- Randomized algorithms

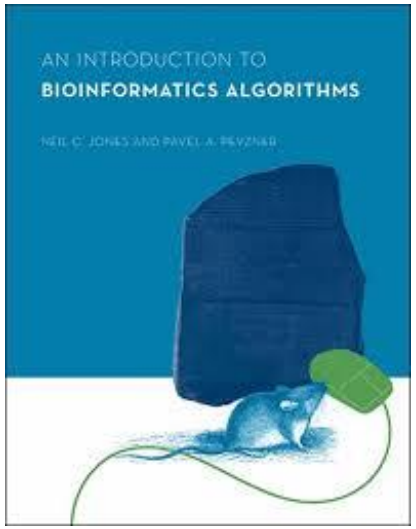
RIT Department of Computer Science

CSCI-665 Foundations of Algorithms Schedule

Week	Topics
1	Introduction, asymptotic notation, analyzing running times
2	O(n) and O(n log n) algorithms: Karatsuba, mergesort
3	Introduction to recurrences, Master Theorem
4	Omega(n log n) lower bound on sorting, linear-time k-Select algorithm
5	Greedy vs (simple) dynamic programming algorithms
6	Greedy vs dynamic programming algorithms (simple and with an added variable): variants of coin changing, knapsack
7	Greedy vs dynamic programming algorithms (simple and with an added variable): sequence problems (longest increasing, longest common)
8	Greedy vs dynamic programming algorithms (subproblem defined by "from-to" indices): Huffman coding, matrix-chain multiplication
9	Graphs: traversals and their applications: connected components, topological sort, etc.
10	Graphs: minimum spanning trees: Kruskal, Prim, union-find data structure
11	Graphs: shortest paths: Dijkstra, Floyd-Warshall, Bellman-Ford
12	Graphs: network flow
13	NP-completeness
14	NP-completeness continued, introduction to linear programming
15	Approximation algorithms and heuristics
16	Review, other topics, e.g., advanced data structures, pattern matching, or randomized algorithms

Some types of Algorithms

Subject	4 Exhaustive Search	5 Greedy Algorithms	6 Dynamic Programming Algorithms	7 Divide-and-Conquer Algorithms	8 Graph Algorithms	9 Combinatorial Pattern Matching	10 Clustering and Trees	11 Hidden Markov Models	12 Randomized Algorithms
Mapping DNA	○								
Sequencing DNA					○				
Comparing Sequences			○	○		○			
Predicting Genes			○						
Finding Signals	○	○						○	○
Identifying Proteins					○				
Repeat Analysis						○			
DNA Arrays					○				
Genome Rearrangements		○							
Molecular Evolution							○		

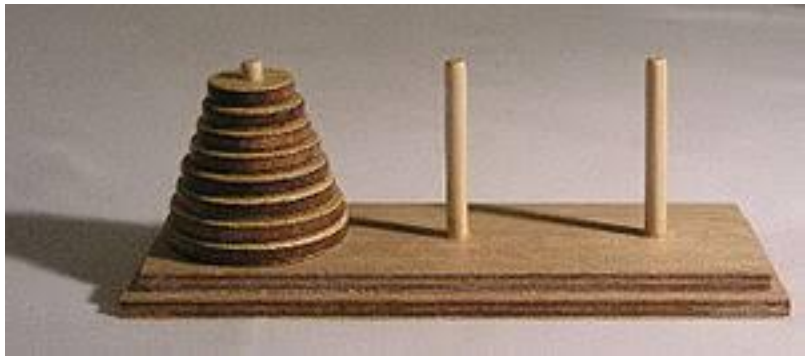


Source: *Introduction to bioinformatics algorithms*, Jones and Pevzner

Recursive Algorithms

- Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

- Towers of Hanoi problem: The game is played by moving **one disk at a time** between pegs. You are **only allowed to place smaller disks on top of larger ones**, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3.



Recursive Algorithms

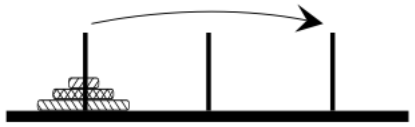
- Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

- Towers of Hanoi problem: The game is played by moving one disk at a time between pegs. You are **only allowed to place smaller disks on top of larger ones**, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3.

Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.

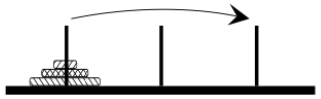
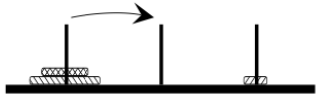
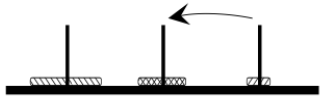
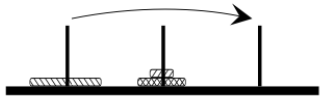
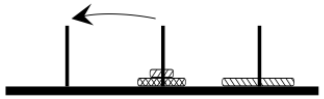
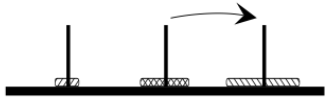
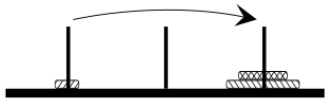
Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.



Source: *Introduction to bioinformatics algorithms*, Jones and Pevzner

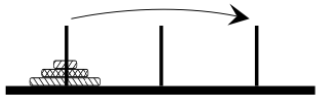
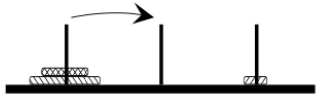
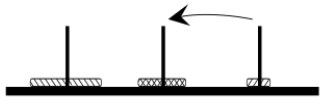
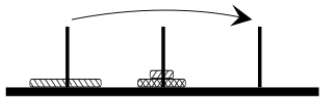
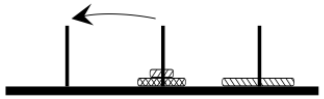
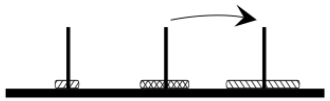
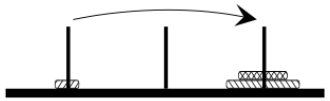
Recursive Algorithms

Three-disk: Seven moves:

- Move disk from peg 1 to peg 3 
- Move disk from peg 1 to peg 2 
- Move disk from peg 3 to peg 2 
- Move disk from peg 1 to peg 3 
- Move disk from peg 2 to peg 1 
- Move disk from peg 2 to peg 3 
- Move disk from peg 1 to peg 3 

Recursive Algorithms

Three-disk: Seven moves:

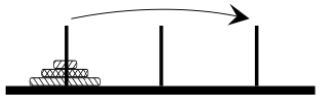
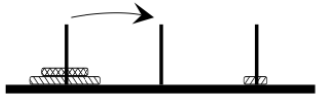
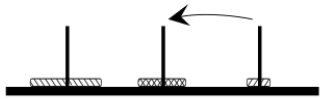
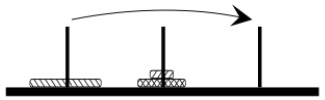
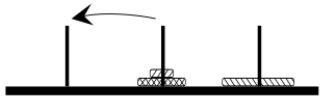
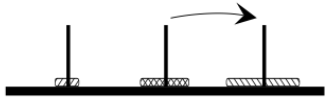
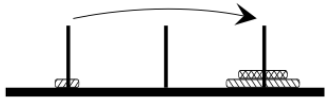
- Move disk from peg 1 to peg 3 
- Move disk from peg 1 to peg 2 
- Move disk from peg 3 to peg 2 
- Move disk from peg 1 to peg 3 
- Move disk from peg 2 to peg 1 
- Move disk from peg 2 to peg 3 
- Move disk from peg 1 to peg 3 

Four-disk puzzle?

- Move the top three to an empty peg (7 moves)
- Then, move the larger disk (one move)
- Again, move the three disk from their temporary peg to rest o top of the larger disk (another 7 moves)
- Total: 7+1+7=15 moves

Recursive Algorithms

Seven moves:

- Move disk from peg 1 to peg 3 
- Move disk from peg 1 to peg 2 
- Move disk from peg 3 to peg 2 
- Move disk from peg 1 to peg 3 
- Move disk from peg 2 to peg 1 
- Move disk from peg 2 to peg 3 
- Move disk from peg 1 to peg 3 

Stack of size n?

- First, move a stack of size n – 1 from left to middle
- Then, from the middle to the right peg, once you have moved the n^{th} disk to the right peg
- To move a stack of size n – 1 from the middle to the right, you first need to move a stack of size n – 2 from the middle to the left
- Then, move the (n – 1) disk to the right
- And move the stack of n – 2 from the left to the right peg
- And so on….

Recursive Algorithms

- At first glance, the Towers of Hanoi problem looks difficult.
- However, the following recursive algorithm solves the Towers of Hanoi problem with n disks.
- The solution can be expressed in 8 lines of pseudocode

```
HANOITOWERS(n, fromPeg, toPeg)
1  if n = 1
2       output "Move disk from peg fromPeg to peg toPeg"
3       return
4  unusedPeg ← 6 − fromPeg − toPeg
5  HANOITOWERS(n − 1, fromPeg, unusedPeg)
6  output "Move disk from peg fromPeg to peg toPeg"
7  HANOITOWERS(n − 1, unusedPeg, toPeg)
8  return
```

The variables (*fromPeg, toPeg, unusedPeg*) refer to the three different pegs.

So HANOITOWERS(n, 1, 3) moves n disks from the first peg to the third peg

To solve 100-disk tower: it require more moves than atoms in the universe.

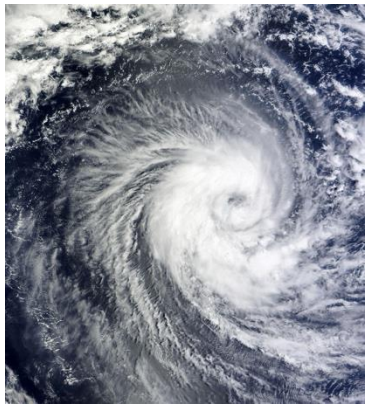
Recursive Algorithms

- Recursive algorithms can often be written to use iterative loops instead, and vice versa.
- Recursion is often the most natural way to solve many computational problems.
- However, recursion can often lead to very inefficient algorithms.
- Let's see this with the example of the **Fibonacci sequence**:

  - Leonardo Pisano Fibonacci (Italy, XIII century) tried to compute the number of offspring of a pair of rabbits over the course of a year.
  - Other examples:

Nautilus shell

Sunflower seeds



Storm systems

Recursive Algorithms

- Recursive algorithms can often be written to use iterative loops instead, and vice versa.
- Recursion is often the most natural way to solve many computational problems.
- However, recursion can often lead to very inefficient algorithms.
- Let's see this with the example of the Fibonacci sequence:
  - Leonardo Pisano Fibonacci (Italy, XIII century) tried to compute the number of offspring of a pair of rabbits over the course of a year.
  - One pair of adults could create a new pair of rabbits in the same time it takes baby rabbits to grow into adults.
  - F_n represents the number of rabbits in period n (adults plus babies)
  - $F_0 = 0$ AND $F_1 = 1$
  - $F_n = F_{(n-1)} + F_{(n-2)}$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Fibonacci sequence

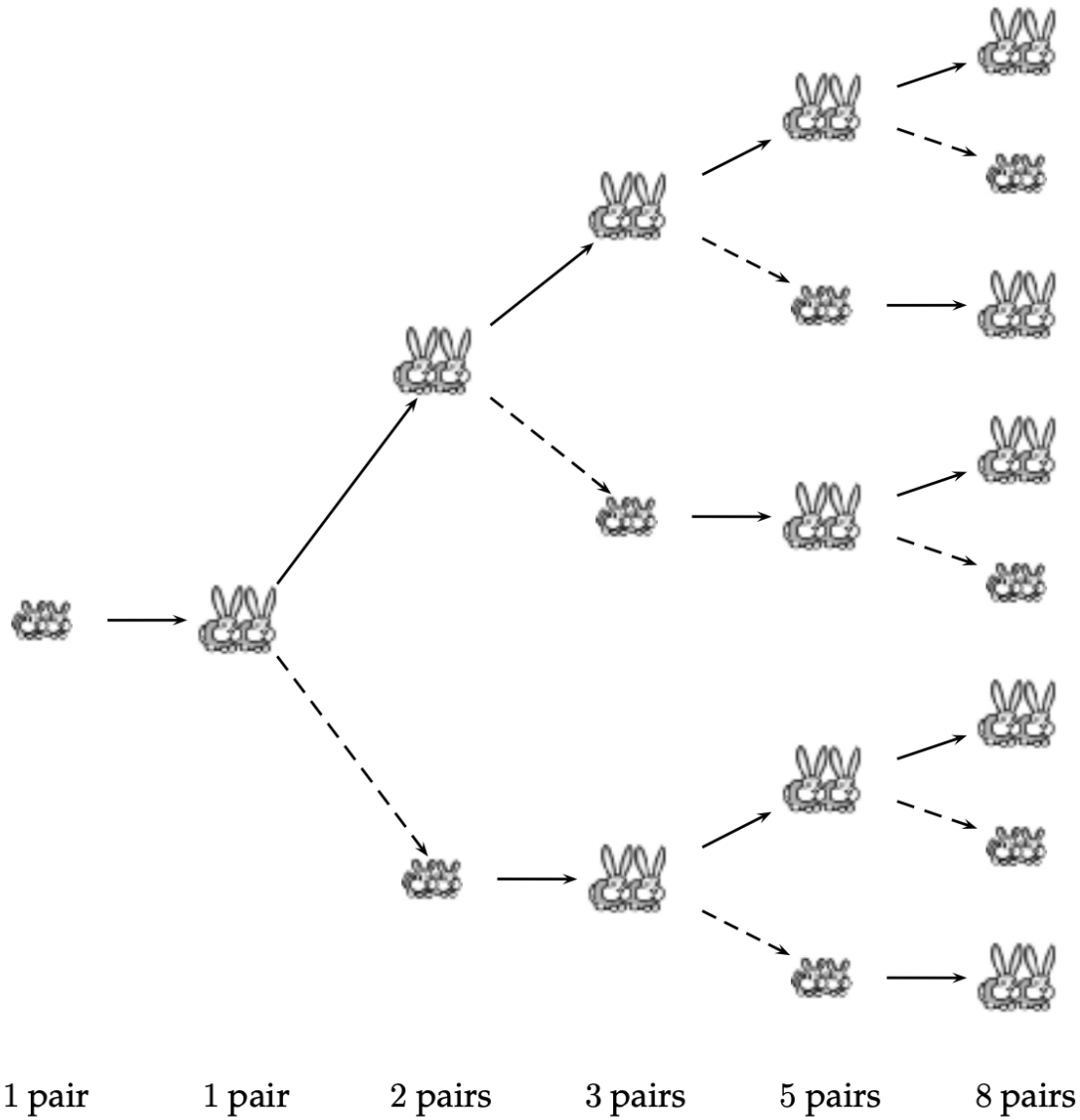


Figure 2.4 Fibonacci's model of rabbit expansion. A dashed line from a pair of big rabbits to a pair of little rabbits means that the pair of adult rabbits had bunnies.

- The simplest recursive algorithm calculate F_n by calling itself to compute $F_{(n-1)}$ and $F_{(n-2)}$.
- This approach results in a large amount of duplicated effort: in calculating $F_{(n-1)}$ we find the value of $F_{(n-2)}$.
- But we calculate it again from scratch in order to determine F_n.
- It is a waste of computing resources.

Source*: Introduction to bioinformatics algorithms*, Jones and Pevzner

Fibonacci sequence

```
RECURSIVEFIBONACCI(n)
1  if n = 1 or n = 2
2      return 1
3  else
4      a ← RECURSIVEFIBONACCI(n − 1)
5      b ← RECURSIVEFIBONACCI(n − 2)
6      return a + b
```

```
FIBONACCI(n)
1  F_1 ← 1
2  F_2 ← 1
3  for i ← 3 to n
4      F_i ← F_{i−1} + F_{i−2}
5  return F_n
```

- However, by using an array to save previously computed Fibonacci numbers, we can calculate the n^{th} Fibonacci number without repeating work.
- RECURSIVEFIBONACCI is an exponential-time algorithm
- FIBONACCI is a linear-time algorithm
- The example shows that an iterative algorithm is superior to a recursive algorithm.
- They require different amounts of time to resolve the same problem.

Algorithm efficiency

- When we develop or use an algorithm, we would like to know how its run time and memory requirements will scale with respect to data size.
- We need to know if a given algorithm can be completed (terminate) within a reasonable amount of time.
- Computer scientists use the **Big-O notation** to describe concisely the running time of an algorithm.
- It provides a standardized way to compare the efficiency of different algorithms in terms of their worst-case performance.
- Understanding **Big-O notation** is essential for analyzing and designing efficient algorithms.

Algorithm efficiency

- **Big-O notation** is used to notate the upper bound of how slow a function, *f(n)*, grows with respect to *n*.
- For example, consider the function in Python:

```
def f(n):
    print("Hello World")
```

- No matter what n is, this function will always run in *constant* time and is independent of the input n. This function use the notation *O(1)*.
- But with:

```
def f(n):
    for i in range(n):
    print(i)
```

- Now, this function is dependent on *n*. If *n* is 1, it will print 1 line. If *n* is 9000, it will print 9000 lines. We say that the growth of this function is **linear** and use the notation *O(n)* to upper bound its growth.

How fast is an algorithm?

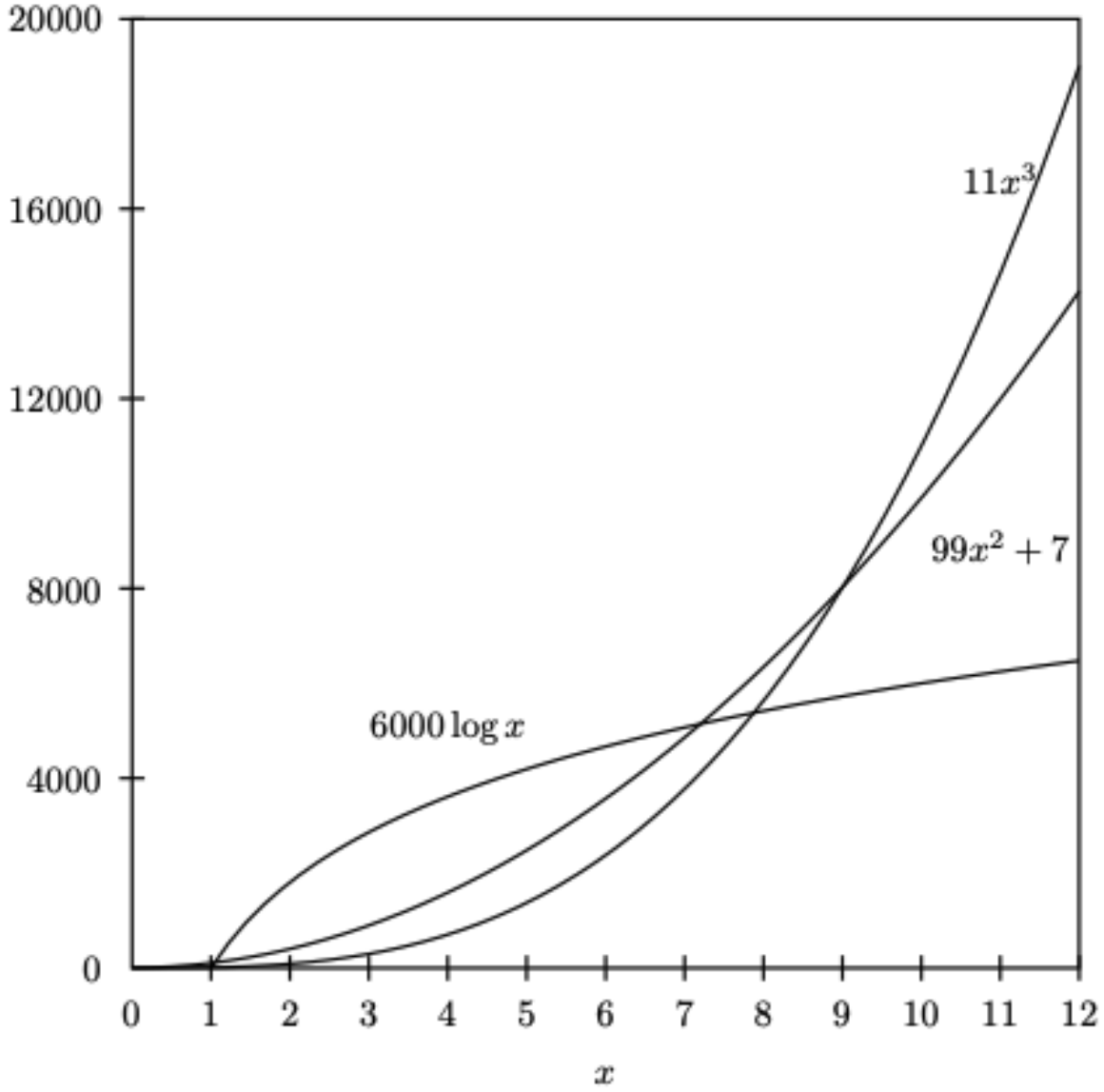
- If we know how to compute the number of basic operations that an algorithm performs, then we have a basis to compare it against a different algorithm that solves the same problem.

- Rather than tediously count every multiplication and addition, we can perform this comparison by gaining a high-level understanding of the growth of each algorithm's operation count as the size of the input increases.

- To illustrate this, suppose an algorithm **A** performs $11x^3$ operations on an input of size x, and a different algorithm, **B**, solves the same problem in $99x^2 + 7$ operations. Which algorithm, **A** or **B**, is faster?

- **A** may be faster than **B** for some small x. But x^3 is a faster growing function than x^2 with respect to x.

How fast is an algorithm?



A comparison of a logarithmic (*h(x) = 6000 log x*), a quadratic (*f (x) = 99x2 + 7*), and a cubic (*g(x) = 11x*^{3}) function.

Any three (positive-valued) functions with leading terms of *log x, x2,* and *x3* respectively would exhibit the same basic behavior, though the crossover points might be different.

Source: *Introduction to bioinformatics algorithms*, Jones and Pevzner

Big-O Notation

- **Big-O**, commonly referred to as "**Order of**", is a way to express the **upper bound** of an algorithm's time complexity, since it analyses the **worst-case** situation of algorithm.
- It provides an **upper limit/bound** on the time taken by an algorithm in terms of the size of the input.
- It's denoted as **O(f(n))**, where **f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.
- **Big-O notation** only describes the **asymptotic** behavior of a function, not its exact value.
- The **Big-O notation** can be used to compare the efficiency of different algorithms or data structures.

Asymptotic: approaching a given value as an expression containing a variable tends to infinity.

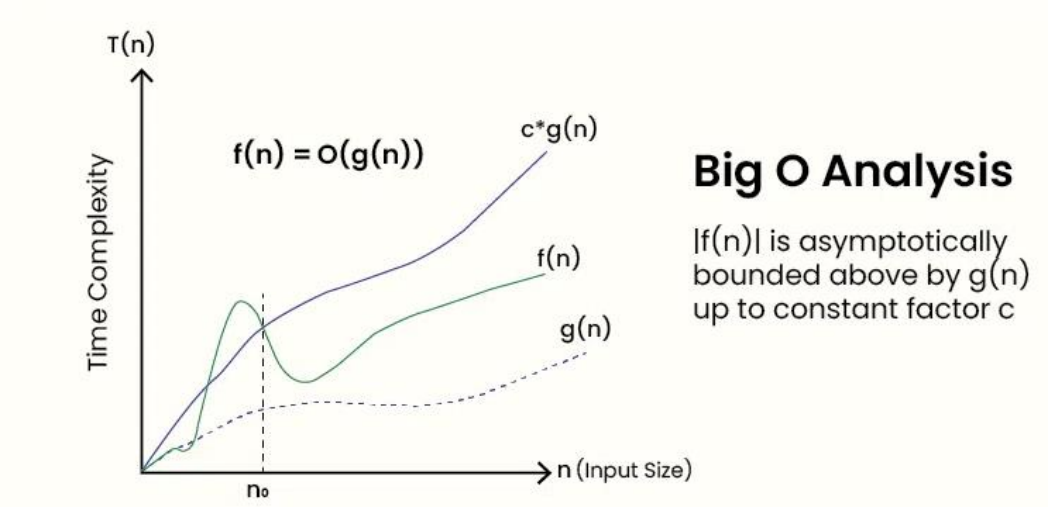
Big-O notation is used to describe the performance or complexity of an algorithm. Specifically, it describes the **worst-case scenario** in terms of **time** or **space complexity.**

Source: https://www.geeksforgeeks.org/

Definition of Big-O Notation

- Given two functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there exist constants **c > 0** and **n_0 >= 0** such that **f(n) <= c*g(n)** for all **n >= n_0**.

- In simpler terms, **f(n)** is **O(g(n))** if **f(n)** grows no faster than **c*g(n)** for all n >= n_0 where c and n_0 are constants.



Source: https://www.geeksforgeeks.org/

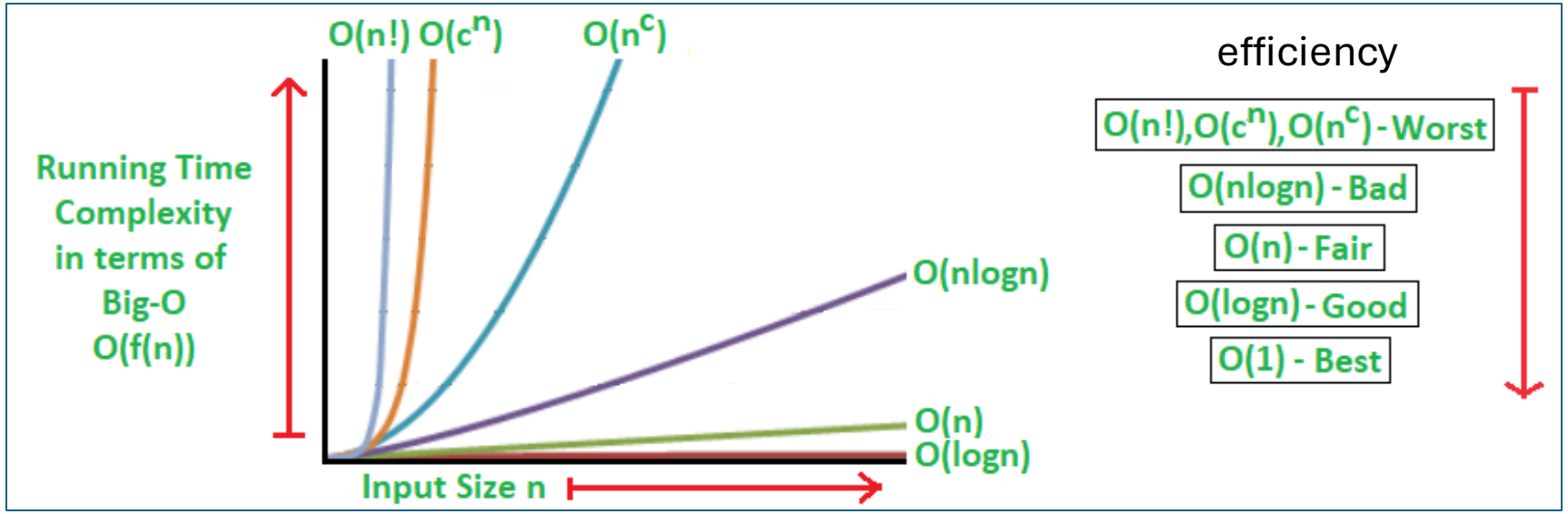
Big-O Notation

Big O notation is important for several reasons:

- Big O Notation is important because it helps analyze the efficiency of algorithms.
- It provides a way to describe how the **runtime** or **space requirements** of an algorithm grow as the input size increases.
- Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
- Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
- Enables developers to optimize code and improve overall performance.

Big-O Notation

If we plot the most common Big O notation examples, we would have a graph like this:



But:
$n^2 = O(n^2)$
$n^2 + n = O(n^2)$
$n^2 + 1000n = O(n^2)$
$5000n^2 + 1000n = O(n^2)$
Constants do not matter!

Source: https://www.geeksforgeeks.org/

Big-O Notation

- Big-O notation tells you how much slower an algorithm gets as the input gets larger.

- Examples:
  - Returning the first element of an array takes a constant amount of time. If you double the length of an array, it still takes the same amount of time. That code does not scale with input size.
  - Summing all the elements of an array takes a linear amount of time with regard to the length of the array. Double the length, the code takes roughly twice as long
  - Searching for an element in a sorted array? Performance *does* get worse as the array length increases, but less than linearly**.**

- The "notation" of Big-O notation is just concise shorthand for describing the above patterns. O(1) for constant time, O(n) for linear time (where n is the length of the array), O(log n) for logarithmic time, etc.

Polynomial vs. Exponential

```
RecursiveFibonacci(n)
1  if n = 1 or n = 2
2      return 1
3  else
4      a ← RecursiveFibonacci(n − 1)
5      b ← RecursiveFibonacci(n − 2)
6      return a + b
```

O(2^n)

Exponential algorithms: run time is bounded by an exponential function, where the exponent is n:
n^n, 2^n, etc.

```
Fibonacci(n)
1  F_1 ← 1
2  F_2 ← 1
3  for i ← 3 to n
4      F_i ← F_{i−1} + F_{i−2}
5  return F_n
```

O(n)

Polynomial algorithms: run time is bounded by a polynomial function (addition, subtraction, multiplication, division, non-negative integer exponents:
n, n2, n5000, etc

Algorithm design techniques

- Many algorithms share similar ideas, even though they solve very different problems.
- There appear to be relatively few basic techniques that can be applied when designing an algorithm.
- To illustrate the design techniques, we will consider a very simple problem:

Source: *Introduction to bioinformatics algorithms*, Jones and Pevzner

Exhaustive search – brute force

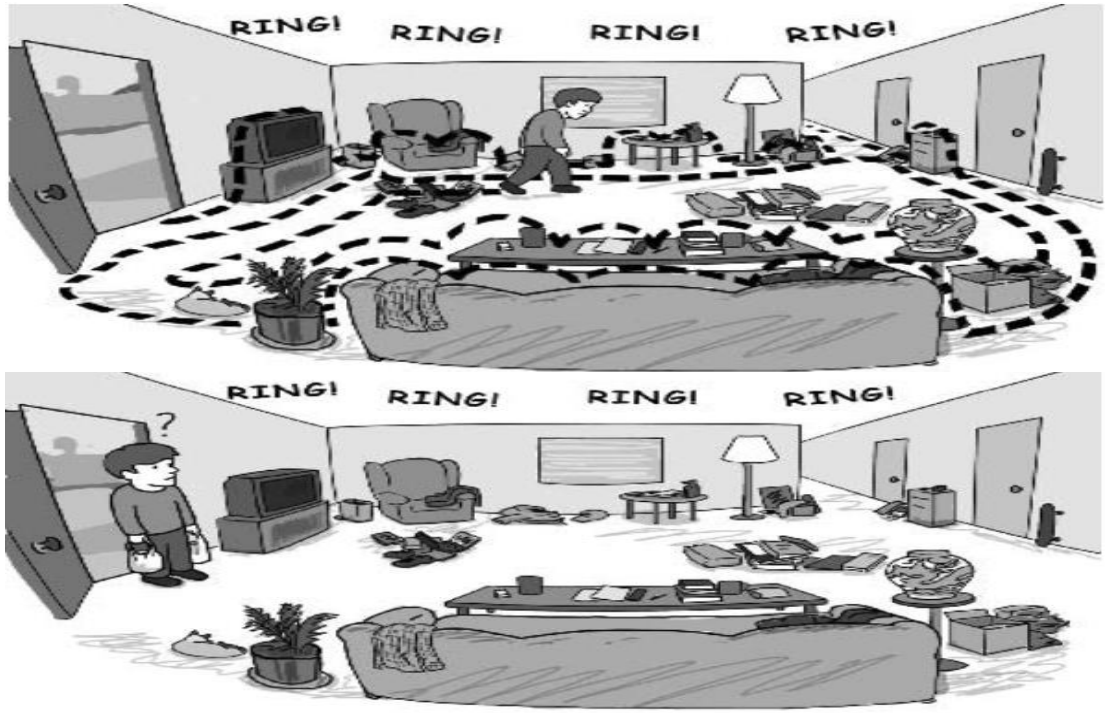
- Examine every possible alternative to find one particular solution.
- These are the easiest algorithms to design and understand.
- Sometimes they work acceptably for certain practical problems in biology.
- But brute force algorithms are too slow to be practical for anything.

Branch and Bound algorithms

- Omit a large number of alternatives when performing brute force.
- Suppose you are exhaustively searching the first floor and hear the phone ringing above your head.

second floor

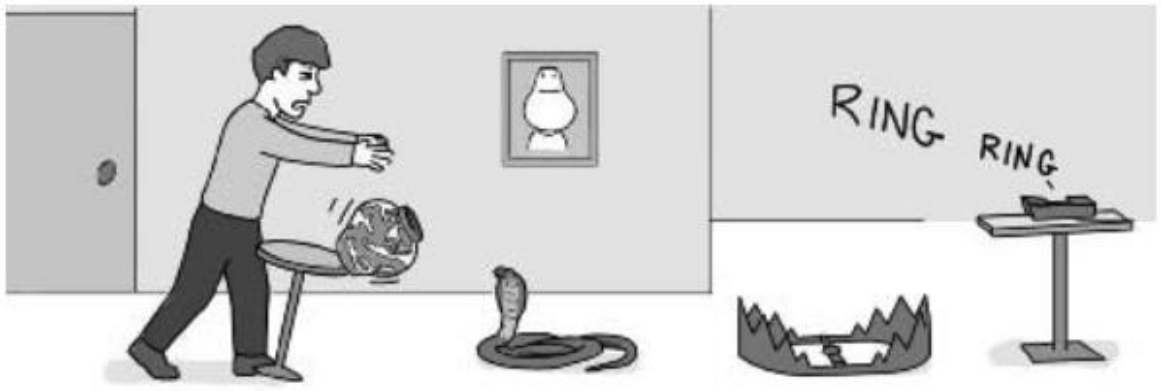
first floor



Half the time!!

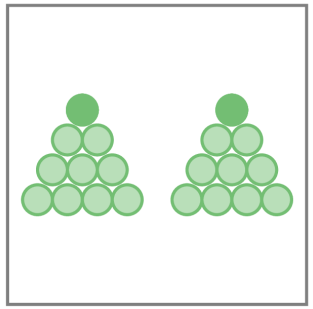
Greedy algorithms

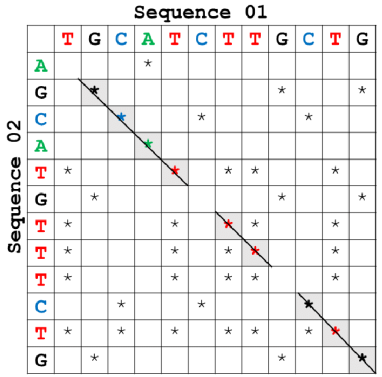
- Many algorithms are iterative procedures that choose among a number of alternatives at each iteration.
- Greedy algorithms choose the “most attractive” alternative at each iteration.
- In the telephone example, the corresponding greedy algorithm would simply be to walk in the direction of the telephone’s ringing until you found it.
- In many cases, a greedy approach will seem “obvious” and natural, but will be subtly wrong.



Dynamic Programing

- Break problems into subproblems; solve subproblems; merge solutions of subproblems to solve the real problem.
- Keep track of computations to avoid recomputing values that you already solved.
- Dynamic programming (traceback) table. Example: Rocks game, Global sequence alignment.





Algorithm design techniques

- **Divide and conquer**:
  - Split, solve, merge
  - Merge sort: is one of the most efficient sorting algorithms. It works on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
- **Machine learning**:
  - Analyze previously available solutions, calculate statistics, apply most likely solution
- **Randomized algorithms**:
  - Pick a solution randomly, test if it works. If not, pick another random solution

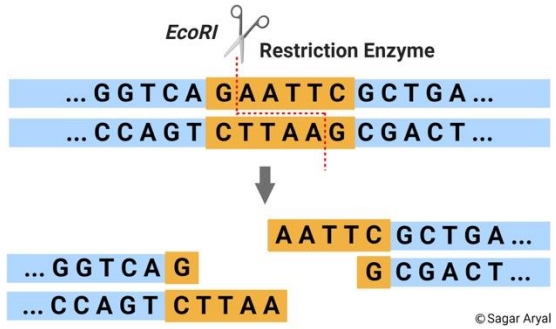
Pattern Matching

- In biology we have patterns everywhere!
  - TATA box (“TATAAA”)
  - Protein motifs
  - Restriction enzyme cleaves sites
  - Exon-Intron splice sites
  - Termination signals
  - Transcription factor binding sites
  - …

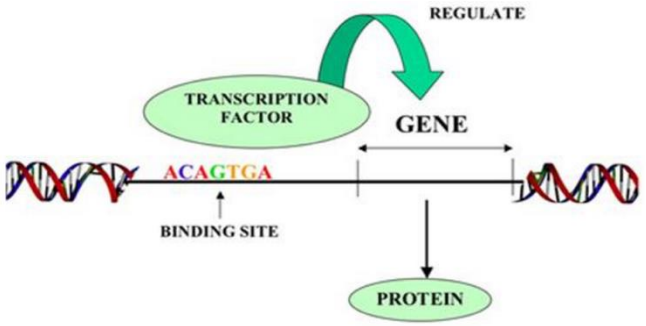
All have regular patterns!

Pattern Matching

- Exact patterns
  - Not that common in biology
  - Restriction enzyme recognition sites



- Inexact patterns
  - Common in biology
  - Transcription factor binding sites
  - Splicing recognition sequences
  - Termination sequences



Transcription Factor binding sites

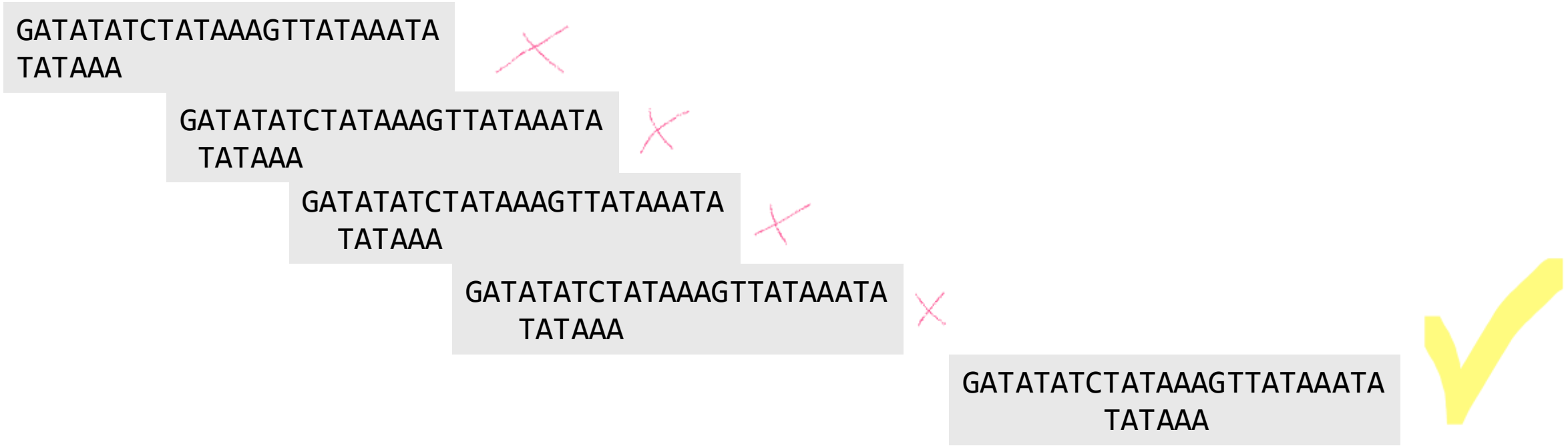
Binding Site	Description	Publications
	Cytoadherence, sporozoite-regulated. Bound by the AP2 transcription factor PF14_0633.	De Silva et al. (2008) *PNAS* 105: 8393-8398. Young et al. (2008) *BMC Genomics* 9: 70.
	Heat shock, trophozoite-specific genes, ribosomal proteins.	Militello et al. (2004) *Mol. Biochem. Parasitol.* 134:75-88. Young et al. (2008) *BMC Genomics* 9: 70.
	Protein modification, phosphorylation, apical complex and invasion. Regulated by the AP2 transcription factor PFF0200c.	De Silva et al. (2008) *PNAS* 105: 8393-8398. Young et al. (2008) *BMC Genomics* 9:70. Voss et al. (2003) *Mol. Micro.* 48: 1593-1608.
	Ookinete-regulated. Bound by the AP2 transcription factor PB000572.01.0.	Yuda et al. (2008) *Mol. Micro.* 71: 1402-1414.

The height of the nucleotide (A T C G) is proportional to how often it is observed at that position in the TF binding site.

Pattern Matching search

How to search for patterns?

- **Brute force**:
  - Given the pattern “TATAAA”
  - Search in the sequence “GATATATCTATAAAGTTATAAATA…”



Pattern Matching search

How to search for inexact matches?

- We need to represent the pattern with a syntax
- We can use regular expressions, which are rather standard across different tools
    - Use a period (“.”)

```
GATATATATATAAAGTTATAAATA
  TATA.A
```

    - Case sensitive
    - Match the pattern at the beginning of a string (“^”)
        - **“**^TATA**”**

```
TATATATAAAGTTATAAATA
TATA
```

    - Match at the end of the string only (“$”)
        - “TA$”

```
TATATATAAAGTTATAAATA
                 TA
```

    - String starting with “Q”, “G” **or** “W” (amino acids)
        - “^[AGW]”

Pattern Matching search

- There are several systems of representing inexact patterns.
  - Grep
  - Perl
  - Tools for motif analysis
    - transcription factor binding sites
    - RNA motif analysis
    - …
  - Graph matching for molecular compounds

Motif sequence search

- Sequence motifs are short segments of conserved protein or nucleic acid sequences, that are present in many proteins or and have specific functional significance.

  - In some cases, the entire set of amino acids or nucleic acids in the sequence **is conserved** and required to perform the specific function.
  - In other cases, only amino acids or nucleic acids at specific locations in the sequence motif **may be conserved** and significant for the function.

- The sequence motif search option allows you to query for amino acid or nucleotide sequence fragments.

- But sequence motif searches are indeed different from a regular similarity-based sequence search (e.g., BLAST).

  - Proteins
  - DNA

Motif sequence syntax

- Simple mode: Input a sequence of one or more of one-letter codes.
  - Ambiguous nucleotide codes: wildcard symbol (**X**)
  - Use **<** and **>** to match the N- and C-termini, respectively.

- PROSITE mode: for more complex queries.
  - Any-of ([ ]) One or more codes enclosed in [ ], such as [ATC]. This matches exactly 1 residue whose code is listed.

  - None-of ({ }) One or more codes enclosed in { }, such as {ATC}. This matches exactly 1 residue whose code is not listed.

  - N-terminus (<)
  - C-terminus (>)

  - Quantifiers
    - Exact A(2) matches exactly AA.
    - Minimum A(2,) matches at AA, AAA, … .
    - Range A(2,4) matches AA, AAA, and AAAA.

Motif sequence match

In various motif-based analysis tools, sequences are searched for matches to **a set of motifs**, and the sequences are sorted by the best combined match to all motifs.

- Input:
  - Sequence (query) file
  - Motif file:

matrix profile MA0035.1 in JASPAR DB

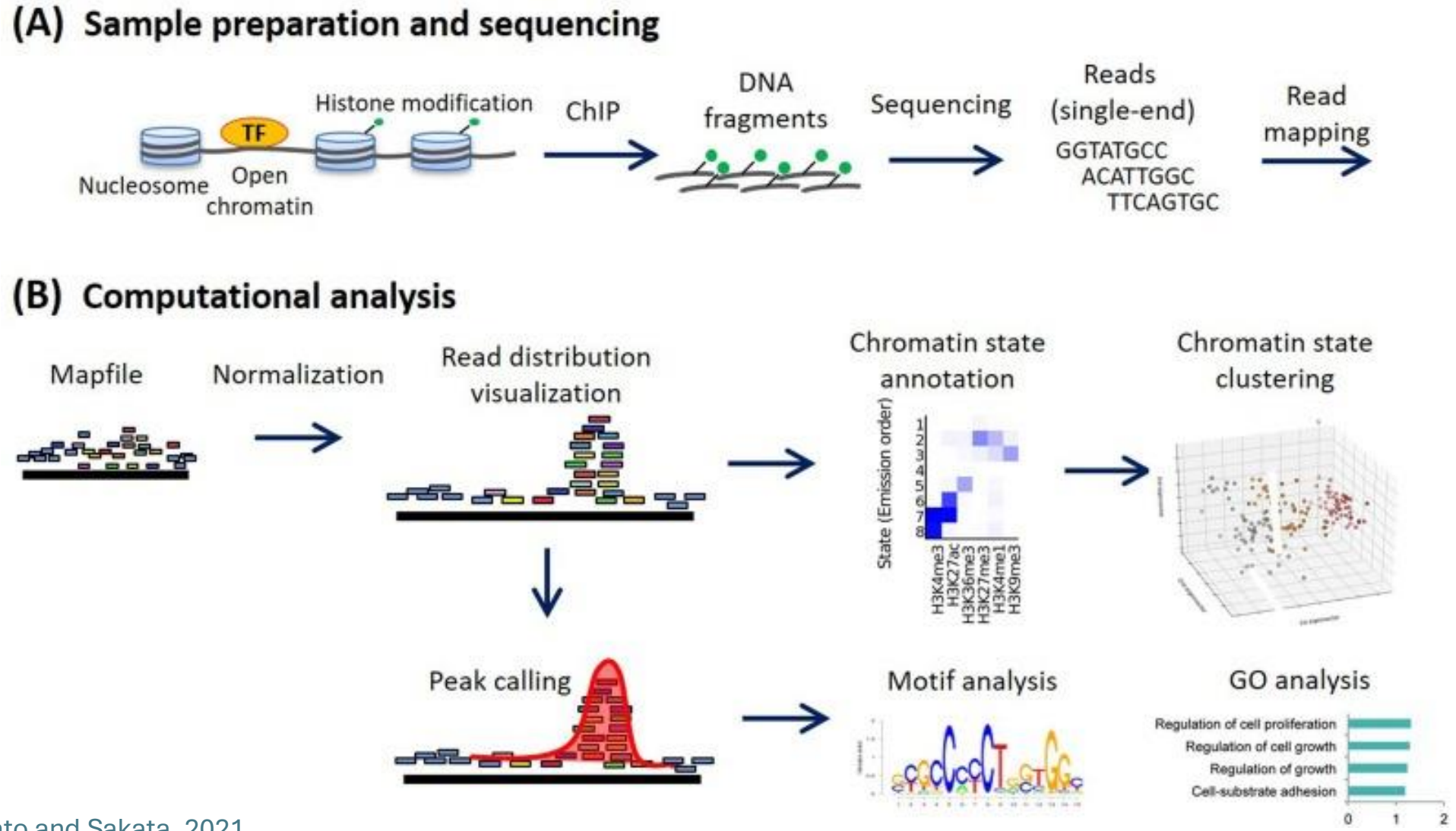
Frequency matrix

Position:	1	2	3	4	5	6	
A[	13	0	50	1	13	6	]
C[	14	0	2	0	6	16	]
G[	15	53	0	1	25	22	]
T[	11	0	1	51	9	9	]

>query1
AATAAAAG**AGATAA**AAATGGAAATATCAAGGC
>query2
AATAAAAG**TGATGA**AAATGGAAATATCAAGGC

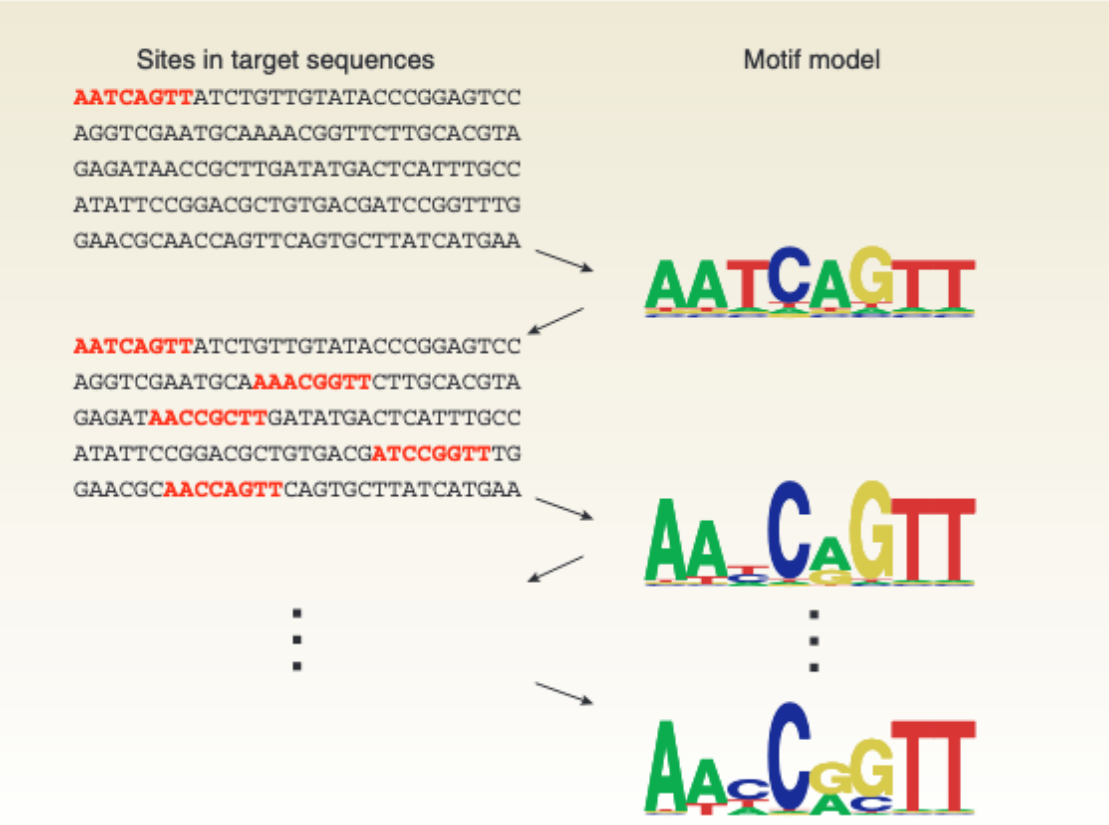
-Calculates a Match Score for every position in the motif
-Sequence location with highest scores

ChIP-seq analysis workflow



Source: [Nakato and Sakata, 2021]

How does DNA sequence motif discovery work?



Source: [D'haeseleer, 2006]

Starting from a single site, expectation maximization algorithms such as MEME alternate between assigning sites to a motif (left) and updating the motif model (right).

The position weight matrix for the motif is initialized with a single n-*mer* subsequence.

Next, for each n-mer in the target sequences, we calculate the probability that it was generated by the motif.