Introduction to Algorithm

Yuzhen Ye (yye@indiana.edu)
School of Informatics and Computing, IUB
Computational problems

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?

- Example 1:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically

- Example 2:
  - Input: A picture in digital format
  - Output: An English description of what the picture shows

- From biological problems to computational problems
  - Similarity between two protein sequences -> edit distance between two strings
Algorithms

- An algorithm is an exact specification of how to solve a computational problem.
- An algorithm must specify every step completely, so a computer can implement it without any further “understanding.”
- An algorithm must work for all possible inputs of the problem.
- Algorithms must be:
  - Correct: For each input produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later
- There can be many different algorithms for each computational problem.
Designing algorithms

- There is no single recipe for inventing algorithms
- There are basic rules:
  - Understand your problem well – may require much mathematical analysis!
  - Use existing algorithms (reduction) or algorithmic ideas
Algorithmic paradigms

- Greedy: build up a solution incrementally, myopically optimize some local criteria
- Divide and conquer: break up a problem into non-overlapping sub-problems, solve sub-problems independently, and then combine solutions to the sub-problems to form solution to the original problem.
- Dynamic programming: break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger and larger sub-problems
Divide and conquer

- In its simplest (and most useful) form it is simple induction
  - In order to solve a problem, solve a similar problem of smaller size

- The key conceptual idea:
  - Think only about how to use the smaller solution to get the larger one
  - Do not worry about how to solve to smaller problem (it will be solved using an even smaller one)
Recursion

- A recursive method is a method that contains a call to itself
- Technically:
  - All modern computing languages allow writing methods that call themselves
  - We will discuss how this is implemented later
- Conceptually:
  - This allows programming in a style that reflects divide-n-conquer algorithmic thinking
  - At the beginning recursive programs are confusing – after a while they become clearer than non-recursive variants
Fibonacci numbers

\[ F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0 \quad \text{and} \quad F_1 = 1. \]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584...

```python
def Fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```
Elements of a recursive program

- **Basis:** a case that can be answered without using further recursive calls
  - In our case: if (n==0) return 1;
- **Creating the smaller problem, and invoking a recursive call on it**
  - In our case: factorial(n-1)
- **Finishing to solve the original problem**
  - In our case: return n * /*solution of recursive call*/
Dynamic programming

- You will see dynamic programming is used very often in bioinformatics
  - Smith-Waterman for genetic sequence alignment (the fundamental algorithm for bioinformatics)
  - Viterbi for hidden Markov models (theoretical basis for wide-ranging applications including cell phone communication, DNA analysis and speech recognition)
  - Algorithms for shortest path computing in biological networks.
Greedy algorithm sometimes works

- Cashier's algorithm is optimal for U.S. coinage (coin change problem)
  - But postal worker's algorithm is not optimal for U.S. postage
- Minimum spanning tree (MST) can be solved by greedy algorithms (e.g., Prim's algorithm)
How fast will your program run?

- The running time of your program will depend upon:
  - The algorithm
  - The input
  - Programming language & compiler & OS
  - Your computer hardware (supercomputer, Cloud)
  - Other programs on your computer…

- Each algorithm performs a sequence of basic operations:
  - Arithmetic: \((\text{low} + \text{high})/2\)
  - Comparison: \(\text{if } (\ x > 0 \ ) \ …\)
  - Assignment: \(\text{temp} = x\)
  - Branching: \(\text{while } (\text{true} ) \{ \ … \}\)
  - Not all operations take the same amount of time.
  - Operations take different times with different hardware or compilers
You may choose different programming languages for different jobs

- Python is good for parsing, but could be slow
- C++ (c) is probably is one of the choices when speed is the top priority, complied programming language
- Statistics? MatLab & R (free)
- Web-programming? php (or asp for windows)
Asymptotic running times

- Complexity of the algorithm is $O(N^2)$
- The running time of the algorithm is $O(N^2)$.
- Asymptotic lower-bound
- Asymptotic upper-bound
- Big-O notation
- Can this problem be solved in polynomial time?

```python
def myMethod(N):
    sq = 0;
    for i in range(N):
        for j in range(N):
            sq += 1
            #and other operations
    return sq
```
Big-O notation

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size $n$, which is usually the number of items

$f(n) = O(g(n))$ means there are positive constants $c$ and $k$, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$
From word ladder to sequence alignment

- Remember what the plays can do (at each step) in word ladder game
  - Add a letter
  - Remove a letter
  - Change a letter
  - Use the same letters in different order

- Same (or slightly different) operations are used in biology. You will see later that we need to use insertion, deletion, replacement, and rearrangement when we compare (align) sequences.
Combinatorial problems

“Ever since my undergraduate days I have had a fascination with combinatorial algorithms. These puzzle-like problems involve searching through a finite but vast set of possibilities for a pattern or structure that meets certain requirements or is of minimum cost. Examples in bioinformatics include sequence assembly, multiple alignment of sequences, phylogeny construction, the analysis of genomic rearrangements, and the modeling of gene regulation. For some combinational problems, there are elegant and efficient algorithms that proceed like clockwork to find the required solution, but most are less tractable and require either a very long computation or a compromise on a solution that may not be optimal.” (Richard Karp, UC berkeley)
Approximation algorithms to hard problems

- Not all problems have efficient algorithms
- TSP problem: the “traveling-salesman problem” is NP-complete. Given a set of $n$ nodes (cities) and distances for each pair of nodes (cities), find a roundtrip of minimal total length (lowest overall travel distance) visiting each node (city) exactly once.
- There is no efficient algorithm for solving NP-complete problems (so-far) (i.e., worst case running time for any algorithm for TSP increases exponentially with the number of nodes, or cities)
Combinatorial problems in bioinformatics

- There are $20^{100}$ possible protein sequences of length 100. Combinatorial explosion
  - Protein structure prediction & protein sequence design

- Multiple alignment
  - SW algorithm $O(N^2)$ (for pairwise alignment) (N is the length of the input sequences)
  - SW algorithm applied to multiple alignments? Probably not, $O(N^m)$ (N: the average length, m is the number of sequences)
  - Heuristic approaches to multiple alignments (e.g., Clustalw)

- Simulation of gene regulation
  - Interaction between genes and their transcription factors
  - Histone code: the post-translational modifications of histones, alone or in combination, function to direct specific and distinct DNA-templated programs
Machine learning algorithms

- Machine learning is a scientific discipline that is concerned with the design and development of algorithms that allow computers to learn from data. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data.

- Machine learning algorithms:
  - Supervised learning (e.g., classification)
  - Unsupervised learning (e.g., clustering)
  - Semi-supervised learning
  - Reinforcement learning
Machine learning methods in bioinformatics

- Examples of classification tasks
  - Classifying secondary structures of protein as alpha-helix, beta-sheet, or random coil
  - Fold recognition
  - Predicting tumor cells as benign or malignant
  - Identifying (non-synonymous) SNPs that may affect molecular function

- Examples of cluster analysis
  - Clustering of microarray data
  - Protein/gene family

- Examples of association analysis
  - Gene-disease associations; Biomarker discovery
Machine learning methods in bioinformatics

Figure from: http://bib.oxfordjournals.org/cgi/content/full/7/1/86/F1
Data mining

Figure 1.3. Four of the core data mining tasks.
Two notes

- You will learn more about algorithms from 1500
- You will learn more algorithms in bioinformatics in this course
What’s in the textbook

- Appendix A: Probability, Information, and Bayesian Analysis
- Appendix B: Molecular Energy Functions
- Appendix C: Function Optimization
  - Minimization & maximization problem
    - One of the formulations that you often see in bioinformatics
      - The alignment score in sequence alignment
      - Energy-related functions in modeling and threading
    - Full search method (dynamic programming and branch-and-bound) vs local optimization