Computational Biology (MAT4930/MAT5932)

Lecture 2: Algorithms and Complexity

Abdulmelik Mohammed

University of South Florida

01/14/2021

Modeling biological problems computationally

- How do we solve the following problems computationally?
  - Find a gene in a genome?
  - Compare the genomes of two individuals or two species?
  - Predict the tertiary structure of a protein from its primary structure?
- To begin to address such questions, we first need to model the problems as computational problems.

Computational problems

- A *computational problem* consists of an *input* and an *output*.
- The input and output are well-defined mathematical objects such as strings, numbers, graphs. The input and output are related according to the goals of the computational problem.

Example (Sorting numbers)

Input: A (finite) list L of numbers $a_1, a_2, \ldots, a_n$.
Output: A permutation $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ of L such that $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$.

- A computational problem is the computational analog of a mathematical function, with the goal of processing on a computing machine.
- A *problem instance* is one specific case of a computational problem. For instance, the list $5, 4, 1, 2$ would be a concrete input instance of the sorting problem, with an output $1, 2, 4, 5$.

Decision problems

- A *decision problem* is a computational problem with a YES/NO (1/0) output.

Example (Deciding whether a number is in a list)

Input: A list of numbers $a_1, a_2, \ldots, a_n$, and a number b.
Output: YES if there exists an integer i such that $a_i = b$, NO otherwise.

- Decision problems can be formalized as languages, which are subsets of the power set of a finite alphabet.
- We will not need such formalization.

Search and optimization problems

- Another type of problem is a *search problem*, where, instead of simply returning yes/no, the goal is to find a solution if such a solution exists. For instance, in the above problem, the output would be the integer i which satisfies the criterion $a_i = b$, if such an i exists.
- Yet another type of problem is an *optimization problem*. In an optimization problem, there are many candidate solutions, which are assigned costs/values, and the goal is to find the solution with the minimum cost or maximum value.

Example (US Change Problem)

Input: An amount of money M, in cents.
Output: The smallest number of quarters q, dimes d, nickels n and pennies p whose value adds to M.

Algorithms

- An *algorithm* is a set of instructions to solve a computational problem.
- The notion of an algorithm can be made more rigorous with computational models such as a Turing machine, but it is common to work with higher-level descriptions to avoid tedious formalism.
- Algorithms are specified using pseudocode, which are human-readable representations of computer programs.

Algorithm for US Change Problem

Input: An amount of money M, in cents.
Output: The smallest number of quarter q, dimes d, nickels n and pennies p whose value adds to M.

1 $r \leftarrow M$;
2 $q \leftarrow r/25$; #Integer division.;
3 $r \leftarrow r - 25 \cdot q$;
4 $d \leftarrow r/10$;
5 $r \leftarrow r - 10 \cdot d$;
6 $n \leftarrow r/5$;
7 $r \leftarrow r - 5 \cdot n$;
8 $p \leftarrow r$;
9 **return** (q, d, n, p);

Algorithm 1: UsChange algorithm.

- For $M = 77$, the algorithm returns $(3, 0, 0, 2)$.

Generalized Change Problem and a greedy algorithm for the problem

Input: An amount of money M and an array d of denominations $\mathbf{c} = (c_1, \ldots, c_d)$ in decreasing order of value.

Output: A list of d integers $i_1, \ldots, i_d$ such that $\sum_j^d c_j i_j = M$ and $\sum_j^d i_j$ is minimized.

```
1 r ← M;
2 for k ← 1 to d do
3 |   i_k ← r/c_k;
4 |   r ← r − c_k · i_k;
5 end
6 return (i_1, ..., i_d);
```

Algorithm 2: BETTERCHANGE algorithm.

Algorithm correctness

- An algorithm is *correct* if it gives the correct solution to all problem instances. If it gives an incorrect solution to one problem instance, it is considered incorrect.
- The greedy change algorithm is incorrect. Consider the program output on the input $M = 40$ and $\mathbf{c} = (25, 20, 10, 5)$.
- For hard optimization problems, approximation algorithms which have *provable gap* to the optimum are common and valuable, despite technically being incorrect.

A correct brute force algorithm for the Generalized Change Problem

Input: An amount of money M and an array d of denominations $\mathbf{c} = (c_1, \ldots, c_d)$ s.t $c_1 > \ldots > c_d$.
Output: A list of d integers $i_1, \ldots, i_d$ such that $\sum_j^d c_j i_j = M$ and $\sum_j^d i_j$ is minimized.

```
1  BruteForceChange(M, c, d)
2      min ← ∞, change ← impossible;
3      foreach (i_1, ..., i_d) from (0, ..., 0) to (M/c_1, ..., M/c_d)
        do
4          value ← Σ_j^d c_j i_j;
5          if value = M then
6              size ← Σ_j^d i_j;
7              if size < min then
8                  min ← size;
9                  change ← (i_1, ..., i_d);
10             end
11         end
12     end
13     return change;
```

Algorithm 3: BruteForceChange algorithm.

BruteForceChange algorithm

- The algorithm is correct because it goes through all possible combinations of changes and records the minimum combination along the way.
- The algorithm's correctness can be formally proved through induction. In particular, we can prove that, at the end of each iteration, the *min* variable holds the minumum number of coins with value M among all the possible combinations from $(0, \ldots 0)$ to $(i_1, \ldots, i_d)$, if any such combination exists.
- Likewise, *change* can be proven to hold the actual coin denominations satisfying the condition at the end each iteration.

Computing functions

- Another category of computational problems is the computation of mathematical functions.
- The Fibonacci number F_n is a recursively defined function with $F_n = F_{n-1} + F_{n-2}$, and $F_1 = 1, F_2 = 1$.
- Italian mathematician Leonardo P. Fibonacci first studied the Fibonacci numbers as a model of rabbit population growth.
- The numbers grow very quickly. The first few Fibonacci numbers are: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$.

A recursive algorithm for computing Fibonacci numbers

Input: A positive integer n.
Output: The nth Fibonacci number F_n

```
1 RecursiveFibonacci(n)
2 if  n = 1 or n = 2 then
3 |   return 1;
4 else
5 |   a ← RecursiveFibonacci(n − 1);
6 |   b ← RecursiveFibonacci(n − 2);
7 |   return a + b;
8 end
```

Algorithm 4: RecursiveFibonacci algorithm.

Recursion tree of the recursive Fibonacci algorithm

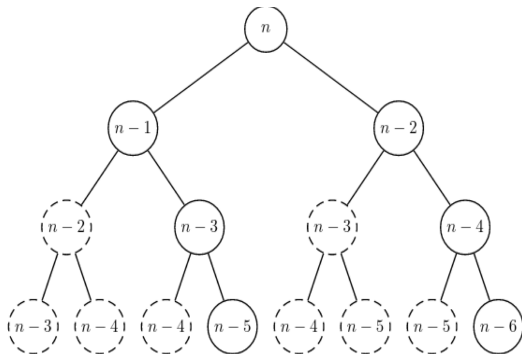

Figure: Recursion tree of the algorithm. The dotted circles represent duplicated computations. Image reprinted from Jones and Pevzner 2004.

Analyzing the running time of the recursive algorithm

- How many computations does the recursive algorithm do?
- Let $T_R(n)$ be the number of elementary computations in RecursiveFibonacci.
- If $n \geq 3$, only lines 5, 6, 7 are executed. Thus $T_R(n) = T_R(n-1) + T_R(n-2) + c_1$, where c_1 is a constant that abstracts away the computations related to the assignment and addition. Here, we are simplifying a bit by assuming the addition in Line 7 takes constant time.

- If $n < 3$, $T_R(n) = c_2$, where $c_2 \geq 1$ is a constant that represents the number of computations in Lines 2 and 3.
- We can see that $T_R(n) \geq F_n$. By using induction, we can prove that $F_n \geq 2^{0.5n}$, for $n \geq 6$. So $T_R(n) \geq 2^{0.5n}$.
- The bound can also be observed directly from the size of the recursion tree.
- Can we do better?
- Let us next look at an iterative algorithm for the same problem and compare their *running time*.

Iterative Fibonacci algorithm

Input: A positive integer n.
Output: The nth Fibonacci number F_n

```
1 IterativeFibonacci(n)
2 F ← an array of size n;
3 F[1] ← 1;
4 F[2] ← 2;
5 for i from 3 to n do
6 |   F[i] ← F[i − 1] + F[i − 2];
7 end
```

Algorithm 5: IterativeFibonacci algorithm.

Analyzing the running time of the iterative algorithm

- The iterative algorithm avoids recomputing previous values, and thus intuitively should take less amount of total computations.
- Let $T_I(n)$ be the number of elementary computations in the iterative Fibonacci algorithm.
- The array allocation in Line 2 can be done in 1 computation with a pointer, since we do not care about the values in the array (they are overridden appropriately).

Analyzing the running time of the iterative algorithm

- Array indexing and assignment, such as in Lines 3 and 4 to a memory location are considered elementary operations and take unit time.
- The for loop is computed $n - 2$ times.
- Thus, $T_I(n) = c_1 + c_2 \cdot (n - 2)$, where c_1 is a constant representing the total number of computations outside the loop, and c_2 is a constant representing the total number of computations inside the loop.

Comparing the recursive and iterative algorithms

- In the theory of algorithms, we are interested in how the behavior of algorithms depends on the input.
- We are particularly interested in the growth of functions as the input grows large, in the case of the Fibonacci algorithms as n grows large.
- For the Fibonacci problem, the run-time of the recursive algorithm grows exponentially, while that of the iterative algorithm grows linearly.
- How do the two run-times compare as n grows large?

Comparing the recursive and iterative algorithms

- Let us be pessimistic towards our analysis of the iterative algorithm and set $c_1 = 100$ and $c_2 = 100$.
- For $n = 10$, $T_R(n) \geq 2^5 = 32$, and $T_I(n) = 100 + 100 \cdot (10 - 2) = 900$, and so we cannot make a conclusion.
- For $n = 64$, $T_R(n) \geq 2^{32} = 1073741824$, while $T_I(n) = 100 + 100 \cdot 62 = 6200$. Clearly, the iterative algorithm is better, despite our pessimistic setting of constants for $T_I(n)$.
- This is true for larger n and other settings of the constants for $T_I(n)$. So, the iterative Fibonacci is considered better than the recursive algorithm.

Run time of algorithms

- The wall-clock time will depend on the machine one is running the program on.
- In algorithms theory, we abstract away these differences and consider the number of *elementary operations* that the algorithm performs. This is the *run-time* of an algorithm.
- What is considered an elementary operation depends on the problem. Some general examples of elementary operations are assignment, array indexing, addition, multiplication and logical checks (such as "and" and "or").
- However, for instance, when considering algorithms for multiplication of two integers, multiplication is not considered as an elementary operation; otherwise all such algorithms would take unit time!

- The run-time of an algorithm is expressed in terms of the *size of the input*. The size of the input is also problem dependent.
- For e.g., in the sorting problem, the size of the input is the number of items in the list.
- For integer inputs, the input size is commonly considered to be the number of bits needed to represent the integer. Thus, for instance, the size of the input for the US Change Problem is $log_2(M)$.
- An algorithm typically will have different number of computations for different input instances. When talking about the running time of an algorithm, we consider the *worst* possible inputs, i.e. those inputs for which the algorithm takes the most time. This is called a *worst-case* analysis, and the run-time is the *worst-case* running time of the algorithm.

Constants are not important asymptotically

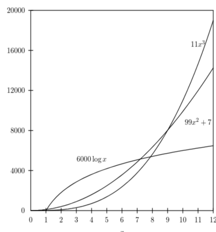

Figure: Growth of functions. Image reprinted from Jones and Pevzner 2004.

- When comparing the run-times of algorithms, multiplicative constants are ignored.
- Thus, $6000 \log x$ is considered better than $11x^3$.
- This is formalized using the big-O notation.

Big-O notation: Upper bounds

- The big-O notation captures the growth of functions, ignoring additive and multiplicative constants.

Definition (Big-O notation)

A function $f(n)$ is $O(g(n))$ if there is a positive real constant c and a positive integer n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Intuitively, $O(g(n))$ is the class of all functions that asymptotically (i.e. in the limit to infinity) grow no more than $g(n)$. Then $f(n)$ is $O(g(n))$ if it belongs to that class. A descriptive notation would be $f(n) \in O(g(n))$, but in computer science the standard notation is $f(n) = O(g(n))$.
- In other terms, $g(n)$ is an asymptotic upper bound to $f(n)$.

Big-O notation: Upper bounds

Example (Linear function)

$T_I(n) = c_1 + c_2 \cdot (n-2) = O(n)$.

Proof.

$$
\begin{aligned}
T(n) &= c_2 n + c_1 - 2c_2 \\
&\leq c_2 n + c_3 n, \text{ for some } c_3 > c_1 - 2c_2 \text{ and } c_3 > 0, \text{ and } n > 1 \\
&\leq cn, \text{for } c = c_2 + c_3 \text{ and } n > 1
\end{aligned}
$$

Hence, $T_I(n) = O(n)$, with $c = c_2 + c_3$ and $n_0 = 1$. □

Asymptotic lower bounds

- A related concept, but for lower bounds is the big omega notation.

Definition (Big omega notation)

A function $f(n)$ is $\Omega(g(n))$ if there is a positive real constant c and a positive integer n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

- Intuitively, if $f(n) = \Omega(g(n))$, then $f(n)$ grows asymptotically faster than $g(n)$.

Example (Recursive Fibonacci algorithm)

$T_R(n) \geq 2^{0.5n} = \Omega(2^{0.5n})$.

Asymptotically equivalent

- How about if $f(n)$ and $g(n)$ grow similarly in the limit?

Definition (Theta notation)

$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Example (Iterative Fibonacci algorithm)

$T_I(n) = \Theta(n)$.

Polynomial time algorithms

Definition (Polynomial time algorithms)

A *polynomial time algorithm* is an algorithm whose (worst-case) running time is $O(n^d)$, where n is the size of the input and d is some positive constant not related to the input size.

Example

The iterative Fibonacci algorithm is a polynomial time algorithm with respect to n.

Complexity of computational problems (advanced material)

- Computational problems for which there are polynomial time algorithms are said to be *tractable* and *efficiently* solvable. The computational complexity class P encapsulates all problems with polynomial-time algorithms.
- Other problems are suspected to not admit *any* polynomial time algorithm, these are the NP-HARD problems.
- For some NP-HARD problems, one can efficiently verify a candidate solution if such as solution is provided. These are the so-called NP-COMPLETE problems. One such problem is the so-called Traveling Salesman Problem.
- Solving a problem vs verifying a solution to a problem is analogous to composing a symphony vs appreciating a composed symphony!

Recurrence relations

- Recurrence relations frequently arise in the analysis of recursive algorithms.
- We saw one recurrence relation with the recursive Fibonacci algorithm.
- Here is another recurrence relation $T(n) = n + T(n-1)$ with $T(1) = 1$. This recurrence appears in the analysis of a recursive algorithm for sorting (see textbook, Chapter 2).
- We frequently need to solve such recurrence relations to obtain closed-form formulas.

Solving recurrences

- A common technique for solving recurrence relations is to unfold and see a pattern.
- If we consider the previous recurrence and set the base case $T(1) = 1$, we get:

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
&= n + n - 1 + T(n-2) \\
&= n + n - 1 + n - 2 + T(n-3) \\
&= n + n - 1 + n - 2 + \cdots + 1 \\
&= \frac{n(n+1)}{2} \\
&= O(n^2).
\end{aligned}
$$

Summary

- Computational problems and algorithms, correctness of algorithms.
- Recursive and iterative algorithms.
- Run-time of algorithms, fast and slow algorithms.
- Big-O, Big-Ω and Θ notations.
- Polynomial time algorithms, tractability and hardness of some problems.
- Recurrences and how to solve them.
- Next lectures: various algorithm design principles.