Introduction to the Design and Analysis of Algorithms

Stefan D. Bruda

CS 317, Fall 2024

Problems and Algorithms

- It all starts with a problem: a task to be performed or a question to be answered
  1. Sort a sequence S of n numbers in increasing order
  2. Determine whether the number x is in the sequence S of n numbers
  3. Find the nth term in the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, . . .
- Parameters: variables that are not assigned values in the statement of the problem
  1. S, n
  2. S, n, x
  3. n
- Algorithm: A rigorous, step-by-step procedure to solve a problem for all possible values of the parameters
  - Named after Abū ’Abdallāh Muḥammad ibn Mūsā al-Khwārizmī, or Mohammed Al-Khorezmi for short (Baghdad, 780–850)

Problems and Algorithms (cont'd)

- Instance: A specific assignment of values to the parameters
  1. Sorting instance: $S = \langle 8, 3, 5, 6, 3, 9, 2 \rangle$, $n = 7$
  2. Searching instance: $S = \langle 8, 3, 5, 6, 3, 9, 2 \rangle$, $n = 7$, $x = 9$
  3. Fibonacci calculation instance: $n = 4$
- Solution: The answer to the question posed in the problem on the given instance
  1. $S = \langle 2, 3, 3, 5, 6, 8, 9 \rangle$
  2. Yes/True
  3. 3
- A traditional algorithm:
  - Receives an input
  - Produces an output
  - Is deterministic i.e., all the intermediate results are unambiguously determined by the previous steps and input
  - It is correct (aka partial correctness)
  - It always terminates (aka total correctness)
  - It is general in the sense that it works for any set of input values

1. Algorithm design (the Art)
   Can consider different techniques such as
   - Divide and conquer
   - Greedy
   - Dynamic programming
   - Backtracking
   - Branch and bound
2. Algorithm analysis (the Science)
   - Proof of partial and total correctness
   - Performance analysis (time and space)

- Throughout the course we will describe algorithms using pseudocode
  - Flexible enough to allow for concise descriptions, but rigorous enough to be easily translated into actual code in any half-decent programming language

- There are multiple algorithms for most problems
- Find the largest of four values a, b, c, d
- Which algorithm is
  - More time efficient?
  - More space efficient?
  - More elegant?

```
largest ← a
if b > largest then
    largest ← b
if c > largest then
    largest ← c
if d > largest then
    largest ← d
return largest
```

```
if a > b then
    if a > c then
        if a > d then
            return a
        else return d
    else
        if c > d then
            return c
        else return d
else
    if b > c then
        if b > d then
            return b
        else return d
    else
        if c > d then
            return c
        else return d
```

Algorithm Design

- There are multiple algorithms for most problems
- Find the largest of four values a, b, c, d
- Which algorithm is
  - More time efficient? (number of comparisons!)
  - More space efficient?
  - More elegant?

```
largest ← a
if b > largest then
    largest ← b
if c > largest then
    largest ← c
if d > largest then
    largest ← d
return largest
```

```
if a > b then
    if a > c then
        if a > d then
            return a
        else return d
    else
        if c > d then
            return c
        else return d
else
    if b > c then
        if b > d then
            return b
        else return d
    else
        if c > d then
            return c
        else return d
```

- There are multiple algorithms for most problems
- Find the largest of four values a, b, c, d
- Which algorithm is
  - More time efficient? (number of comparisons!)
  - More space efficient?
  - More elegant? (e.g., simpler)

```
largest ← a
if b > largest then
    largest ← b
if c > largest then
    largest ← c
if d > largest then
    largest ← d
return largest
```

```
if a > b then
    if a > c then
        if a > d then
            return a
        else return d
    else
        if c > d then
            return c
        else return d
else
    if b > c then
        if b > d then
            return b
        else return d
    else
        if c > d then
            return c
        else return d
```

- Multiplication of two integers

Traditional

(0981 $\times$ 0123)

```
   981
 x 123
 -----
  2943
 1962
 981
------
120663
```

Divide and conquer

(09|81 $\times$ 01|23)

```
  09 81
x 01 23
--------
  18 63  (23 x 81)
 207     (23 x 09)
  81     (01 x 81)
09       (01 x 09)
-------
1206 63
```

Peasant multiplication

($m \times n$)

result $\leftarrow 0$
repeat
  if m is odd **then**
    result $\leftarrow$ *result* $+ n$
  $m \leftarrow m$ `div` 2
  $n \leftarrow n + n$
until $m < 1$:

- Computing the nth Fibonacci number
  - Recursive:

```
algorithm Fib (n):
    if n ≤ 1 then
        return n
    else
        return Fib (n − 1) + Fib (n − 2)
```

  - Iterative:

```
algorithm Fib (n):
    f[0] ← 0
    if n > 0 then
        f[1] ← 1
        for i = 2 to n do
            f[i] ← f[i − 1] + f[i − 2]
        return f[n]
```

- Which algorithm is more elegant?
- Which algorithm is faster?

- Searching for a given value in a sequence of values
  - Sequential search:

    algorithm SEQSEARCH(x, S, l, h):
    - $i \leftarrow l$
    - **while** $i \leq h$ **do**
      - **if** $S[i] = x$ **then return** i
      - **else** $i \leftarrow i + 1$
    - **return** -1
  - Binary search:

    algorithm BINSEARCH(x, S, l, h):
    - $i \leftarrow l$
    - $j \leftarrow h$
    - **while** $i \leq j$ **do**
      - $m \leftarrow (i + j)/2$
      - **if** $S[m] = x$ **then return** m
      - **else if** $S[m] > x$ **then** $j \leftarrow m - 1$
      - **else** $i \leftarrow m + 1$
    - **return** -1
  - Speed? Restrictions?

- Searching for a given value in a sequence of values
  - Sequential search:

```
algorithm SeqSearch(x, S, l, h):
    i ← l
    while i ≤ h do
        if S[i] = x then return i
        else i ← i + 1
    return −1
```

  - Binary search:

```
algorithm BinSearch(x, S, l, h):
    i ← l
    j ← h
    while i ≤ j do
        m ← (i + j)/2
        if S[m] = x then return m
        else if S[m] > x then j ← m − 1
        else i ← m + 1
    return −1
```

  - Speed? Restrictions?
    (BINSEARCH is not an algorithm unless preconditions are stated)

- The performance of an algorithm (running time, space requirements) must be a function of input size
  - Critical to define a meaningful input size
    - Running time may vary widely when different concepts of size are considered
  - Example: The running time of an algorithm for multiplying two $n \times n$ matrices
    - Compare the running time as a function of n (the dimension of the matrix) vs. a function of $n \times n$ (the number of values in one matrix) vs. a function of $2 \times n \times n$ (the total number of values involved)
    - What would be a fair notion of input size?
  - Example: Consider an algorithm that determines whether the input N is a prime number
    - Compare the running time as a function of N (the input number itself) vs. a function of the number of digits of N
    - What would be a fair notion of input size?

TIME COMPLEXITY

- During the course we will mostly analyze algorithms with respect to their running time
  - Arguably the most significant measure of performance
- Running time can vary depending on many other factors than the size of the input, such as the power of the machine the implementation will run on
  - We want a measure of performance that is independent of such factors
  - We will split the running time of algorithms into classes that ignore this kind of factors (multiplicative or additive constants) $\Rightarrow$ time complexity
  - We will further analyze the time complexity of an algorithm as the input size keeps increasing indefinitely $\Rightarrow$ asymptotic time complexity
- The running time of many algorithms depends of the particular instance the algorithm runs on, so one may consider
  - worst-case time complexity (used the most often)
  - average-case time complexity (used sometimes)
  - best-case time complexity (not very meaningful, rarely used if ever)
- Amortized complexity determines the running time an algorithm is statistically likely to need (under various definitions of “likely”)
  - Most useful for operations over data structures and also online algorithms