CALIFORNIA STATE UNIVERSITY FULLERTON

CPSC 535: Advanced Algorithm

Spring 2025

Introduction

Instructor: Dr. Sampson Akwafuo

1

Introduction

Course Details

Syllabus

2

Algorithms

- What is an algorithm?

"a sequence of unambiguous, clear, and correct instructions for solving a problem, that are able to terminate"

Note the attributes of a good algorithm!

- Why do we study algorithms?
  - Developing our analytical skills
    - Does it solve the problem?
    - Does it use resources efficiently?
  - Necessary for solving problems/coding

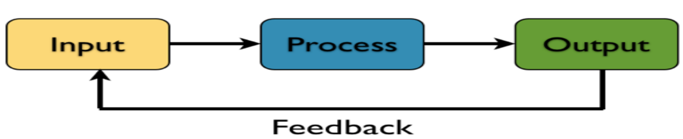
3

Terminologies - Algorithm

Algorithm = an ordered sequence of process/ steps – which produces a solution to a problem. 3 attributes:

- ***Clarity***: contains clear description for implementation
- ***Correctness***: produces a correct solution, always
- ***Termination***: takes a finite amount of time/steps.

- Example of algorithms:
  - Algorithm for adding two integers
  - Algorithm for finding the shortest path in a network
  - Algorithm to search for an article on the web



4

Terminologies - Algorithm

Algorithm = an ordered sequence of process/ steps – which produces a solution to a problem.

Remark:

- There can be more than one algorithm to solve a given problem.
- Sorting problem can be solved by using any of the following algorithms
  - Insertion sort
  - Heap sort
  - Quick sort
  - Etc.

Terminologies

- A **problem** is defined by a set of input instances and a task to be performed on the input instances.
- A problem definition may introduce mathematical variables whose scope is limited to that problem definition.
- ***Notation:***

  The <*problem name*> problem is:
  input: <definition of input objects>
  Output: <definition of output objects>

- Example:

  The ***minimum* problem** is:
  input: a list L of n > 0 comparable objects
  output: the least element in L

- **Data**: Consists of finite mathematical objects that can be represented by strings of binary 0 and 1 digits

Terminologies

- A ***problem instance*** (or an *instance*) = a specific, concrete input to a problem.
- *A solution for a specific problem and instance is a valid concrete output corresponding to the problem and instance.*
    - Ex: lists [7, 2, 1, 9] is an instance of the minimum problem. So is [1, 2]. Empty list [] is not a valid instance of that problem since the problem's input statement requires that n > 0.
- A ***process*** is a series of actions directed to some end.
    - Driving directions, cooking recipes, and computer programs are all processes

CALIFORNIA STATE UNIVERSITY FULLERTON

7

Pseudocode

- ***Pseudocode*** = a human-readable format for communicating algorithms that may include code-like syntax, math notation, and prose.
- Pseudocode is similar to program source code in a language such as Python or C,
    - but is not required to be syntactically-perfect code.
- The pseudocode for *minimum* problem is:

```
Let min = first element of L (there is one since n > 0)
For each element in L do
    if (min > element) then let min = element
Return min
```

CALIFORNIA STATE UNIVERSITY FULLERTON

8

Pseudocode Checklist

Helps in "sanity-checking" of our pseudocodes"

- *If a piece of pseudocode fails any of these tests, it is not good enough to specify an algorithm and needs more work.*
- *If pseudocode passes all these tests, it is probably, but not necessarily, at least adequate.*

1. ***Input and output***: Are the algorithm's inputs and outputs clear, and explicitly separated from other variables? Arguments should correspond to problem inputs; return value corresponds to problem output.
2. ***Undefined variables***: Are any variables used before they are defined or initialized?
3. ***Variable meanings***: Is the intended meaning of every variable clear? Potentially-confusing variables should be explained with a comment
4. ***Defined return value***: Does every execution path have a defined return value?
5. ***Return value data type***: Does the data type of every returned value match the output in the problem definition?
6. ***Handles all cases***: Does your algorithm have the potential to return every kind of valid output?
7. ***Loop termination***: Does every loop have a termination condition that prevents infinite loops?
8. ***Base case***: Does every recursive function have a clearly-defined base case?
9. ***Repetitive code***: Repetitive code should be moved into a helper function or loop.
10. ***Dead code***: Delete code that is never executed.
11. ***Vagueness***: Are any steps vague? Check that every line of pseudocode could be translated into program code without elaboration

9

Designing an Algorithm

Iterative process

– involves working through many drafts

1. Be clear about what problem your algorithm will solve
2. Pick a pattern.
3. Produce a first draft, using the selected pattern
4. Revise the draft by filling in blanks, eliminating potential infinite loops, or clarifying unclear passages.
5. Repeat until your algorithm is correct, clear and able to terminate
6. Write a final and clean draft, ensuring your algorithm will be clear to others
7. Prove that your algorithm is correct, working from your final draft.
8. Prove the efficiency of your algorithm

10

Implementation

- An ***implementation*** of an algorithm is executable computer code that follows the process defined by the algorithm.
- Ex: implementation in C++

```
float min;
int i;
min = L[0];
for (i=1; i < n; i++)
    if (min > L[i])
        min = L[i];
return min;
```

CALIFORNIA STATE UNIVERSITY FULLERTON

11

Analysis of Algorithms

CALIFORNIA STATE UNIVERSITY FULLERTON

12

12

Random-Access Machine (RAM)

In order to predict running time, a (simple) computational model is used: the Random-Access Machine (RAM)

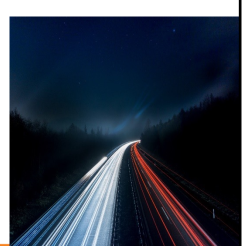
- Instructions are executed sequentially
  - No concurrent operations
- Each ***basic*** instruction takes a constant amount of time
  - arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling, shift left/shift right
  - data movement: load, store, copy
- Loops and subroutine calls are not *basic* operations. They depend upon the size of the data and the contents of a subroutine.
  - "Sort" is not a single step operation.
- Each memory access takes exactly 1 step.
  - We measure the run time of an algorithm by counting the number of steps

13

Functions for Measuring Resources

An algorithm is efficient when it consumes few resources

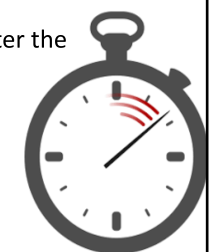
- ***Time:*** measured in units of seconds, CPU instructions, or generic steps;
- ***Space***: measured in units of bits, bytes, gigabytes, or generic words;
- ***input/output bandwidth (I/O):*** measured in units of bytes or blocks;
- ***Cache:*** measured in units of integers; or
- ***Energy***: measured in units of kilowatt-hours.



14

Measuring the quality of an algorithm

- Quality of an algorithm is measured in terms of the resources it consumes when it is executed on a computer
- Of interest in this class are the execution time and the memory need
  - Shorter the execution time, better the quality of the algorithm – *time complexity*
  - Smaller the amount of memory needed for execution, better the quality of the algorithm – *space complexity*



Measuring the quality of an algorithm

- The *complexity* of an algorithm *A*, with respect to a specific instance ***I*** and resource ***R***, is a non-negative real number representing the amount of ***R*** that is consumed by ***A*** when run on ***I***.
- ***Time complexity*** is an indication of the run time of an algorithm in terms of how quickly it grows relative to the input $'n'$
  - Denoted by an efficiency class, it is the amount of time taken by an algorithm to run, as a function of the length of the input.
  - It measures the time taken to execute each statement of code in an algorithm.
- It is not useful to measure the execution time of an algorithm in absolute terms (like 10 sec, 30 µ sec, 1 hr, etc.) since the execution time depends on on a lot of factors

Improving the efficiency of an algorithm

Optimization?

```
def firstTrial_sum(S):
  total = 0
  for x in S:
      total += x
  return total
```

```
def secondTrial_sum(S):
  if len(S) == 0:
     return 0
  else:
   total = 0
   for x in S:
      total += x
   return total
```

- The second trial of the algorithm will be slightly faster in the case of an empty list
  - The algorithm does not initialize any variables or incur any loop overhead in this case.

Time complexity function for the number of milliseconds taken by this algorithm will be

$$T(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2n + 4 & \text{if } n > 0. \end{cases}$$

Counting Operations

- ***Step count*** **(or *Running Time*)** = the number of primitive operations or "steps".
- For our RAM model, we assume that executing each instruction (or statement) takes a constant amount of time. The constant values may differ, but they are the same for two similar instructions:
  - e.g. adding two numbers takes the same time if the numbers are integer/floating point, but addition may take less time than subtraction
- Step count (or running time):
  - *The sum of steps (or running times) for each executable statement*

General Rules for Computing the S.C.

- Simple operations take (1) unit of time:
  - addition, multiplication, assignment, comparison, read/write a value
- Consecutive statements add up.
- If/Else: for the fragment

  If (condition)
  S1
  else
  S2

- The s.c. is equal to the number of steps to evaluate the condition plus the maximum of the s.c. of S1 and S2
- For loops: The s.c. of a for-loop is at most the s.c. of the statements inside the for-loop times the number of iterations.
- Nested loops are analyzed inside out.

Step Counts

- The amount of computing represented by one step may be different from that represented by another.

For example, the entire statement:

- return a+b+b*c+(a+b-c)/(a+b)+4
  - can be regarded as a single step *if its execution time is independent of the problem size*.
- We may also count a statement such as x = y; as a single step

Examples

```python
def arithmetic(lst):
    x = lst[0]
    y = lst[1]
    lst[2] = x + y
    return lst[0] / lst[1]
```

Each line here = 1 step

T(n) = 6
constant

Step Counts in Non-recursive Function Calls

```python
def setup():
    forward = make_list(20)
    backward = make_list(30)
    return forward, backward

def make_list(n):
    L = []   # an empty list
    for i in range(n):
      L.append(0)
    return L
```

What is the resulting step count?

Time Complexity of Setup() =

$T(n)$ = 1 + T_{ml}(20) + 1 + T_{ml} (30) + 1

Time Complexity of Make_list() =

$T_{ml}(n)$ = 1 + $2n$+ 1
= $2n$ +2

Loops

- The total number of steps executed by a *for* loop is the sum of the number of steps executed in each individual iteration. More formally,

$$T(n) = \sum_{x \in X} t_x$$

```
for (i = 0; i < n; i++)
{
  // 3 atomics
}
```

Complexity = O(4n) = O(n)

Loops in Sequence

```
for (j = 0; j < n; ++j)
{
  // 3 atomics
}
for (j = 0; j < n; ++j)
{
  // 5 atomics
}
```

Complexity = *O(4n + 6n) = O(n)* *(efficiency Class)*

Nested Loops

```
for (i = 0; i < n; ++i)
{
   // 2 atomics
   for (j = 0; j < n; ++j)
   {
      // 3 atomics
   }
}
```

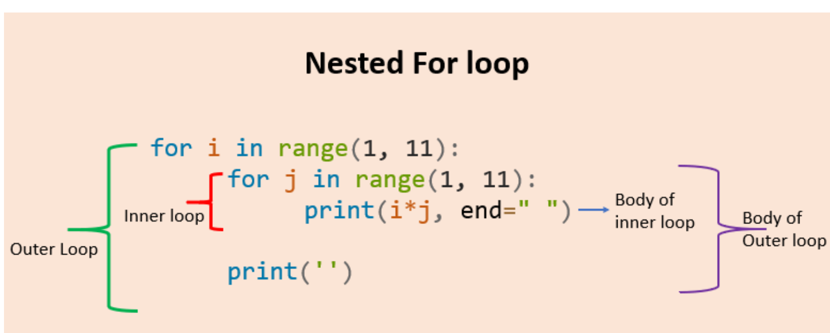
Complexity = $O(3n \times 4n)$ $= O(n^2)$

CALIFORNIA STATE UNIVERSITY FULLERTON

25

25

Nested Loops



CALIFORNIA STATE UNIVERSITY FULLERTON

https://pynative.com/python-nested-loops/

26

26