CS 441: Algorithms and Pseudocode

PhD. Nils Murrugarra-Llerena
nem177@pitt.edu

University of Pittsburgh

Today's topics

- Algorithms and pseudocode
  - What is an algorithm?
  - Pseudocode allows us to describe an algorithm in a semi-structured way
  - Algorithms for basic problems

What is an algorithm?

Definition: An algorithm is a finite sequence of precise instructions for solving a problem

Note these important features!

- Finite: In order to execute, it must be finite
- Sequence: The steps needs to be in the correct order
- Precise: Each step must be unambiguous
- Instructions: Each step can be carried out
- Solving a problem: ?

What is a problem?

A “problem” should be general enough to be broadly useful, but specific enough to dictate strategies

- “Sort the numbers (6, 8, 5, 3)” is too specific; let’s allow the input to be specified
- “Return the numeric solution to the input query” is too general; different instances may not be solved similarly

In computing, we usually represent a problem using its desired solutions

- Inputs: What is provided to identify the instance of the problem?
- Outputs: What are the correct outputs for each input?

An example problem: Sorting

Sorting: Put the provided values in ascending order

- Input: A sequence of numeric values $\{n_i\}$
- Output: A sequence $\{r_i\}$ containing the same elements as $\{n_i\}$, arranged such that $\forall i, j\left(i < j \rightarrow r_i < r_j\right)$

If we specify the input(s), we create an instance of the problem

- We consider each input sequence to be **one instance** of the more general problem, not a different problem
- A solution to an instance is an output that corresponds to the input
    - Could a problem instance have multiple correct outputs?

The searching problem

Intuition: Given a list, find a specific element (if possible)

- How would you solve this?
- To help with describing a solution, let's assign our inputs to have *variable names* that we can refer to, and describe the desired output more precisely

Problem: Given a sequence of values a_1, a_2, …, a_n, and a target value, x, return an index i such that $a_i = x$ (or 0 if no such index exists)

- One possible algorithm to solve this:

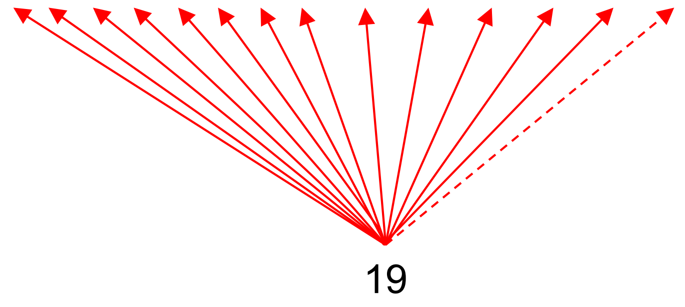
  Compare each a_i to x, starting from a_1 and proceeding sequentially. Return the first i which yields $a_i = x$.

Search: Example

Example: Search 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Values:	1	2	3	5	6	7	8	10	12	13	15	16	18	19	20	22



Pseudocode provides a structured way to represent an algorithm

If we start describing a way to solve sorting, it may not feel very precise

- Examples?

Pseudocode is a semi-structured set of notations

- More precise than describing in prose
- Less overhead than a programming language

Compare to proof techniques

- A formal proof is analogous to a program in (say) Java
  - Meant to be executed by a machine, very precise
- An informal proof is analogous to an algorithm in pseudocode
  - Meant to be understood by a human, so steps can be abstracted. Different communities may prefer different conventions.

An algorithm for the searching problem

Algorithm name ***Input variables*** ***Input data types***

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
    $i := 1$                                  Assigning a variable
    while ($i \leq n$ and $x \neq a_i$)       Repetition structure
        $i := i + 1$
    if $i \leq n$ then                        Selection structure
        location := $i$
    else
        location := 0
    return location {location is the subscript of the term that equals $x$, or is 0 if $x$ is not found}
```

Return the solution ***Comment (exposition; no effect)***

Selection structure executes instructions conditionally

```
if condition then statement 1
else statement 2
```

optional else case

```
if condition then
        statement 1
        statement 2
        …
        statement n
statement n+1
```

multiple conditional statements

statements after are always executed

```
if condition 1 then statement 1
else if condition 2 then statement 2
…
else if condition n then statement n
else statement n+1
```

mutually-exclusive cases

Repetition structure executes instructions multiple times

```
for variable := initial value to final value
    statement
```

for-loop: execute statement(s) for each value in a sequence

```
while condition
    statement
```

while-loop: if condition is true, execute statement(s), then check again

variable is assumed to update automatically for each iteration

```
sum := 0
for i := 1 to n
    sum := sum + i
```

```
sum := 0
i := 1
while i ≤ n
    sum := sum + i
    i := i + 1
```

statements must update variable(s) to make condition false

In-class exercises

Problem 1: Consider a version of the search problem where the sequence to search is in sorted order. Write an updated version of the linear search algorithm that stops searching once it is determined that the target element is not present.

Problem 2: What is the output of this algorithm given input 100?

```
procedure problem 2($t$: integer)
    $x := 1$
    while ($x \le t$)
        $x := x * 2$
    return $x$
```

More efficient approaches to searching?

If the input is sorted, we can implement an even faster search

Binary search, main idea

- Track an interval within the sequence where the target must be
  - Initially set to the entire sequence
- Divide the interval in half and determine whether the target would be in the left or right half
- Repeat until the item is found or the interval is empty

~~3 6 11 18 22 26 32~~ $\boxed{\cancel{40}}$ ~~52 55 58~~ $\boxed{\cancel{60}}$ $\enclose{circle}{64}$ $\boxed{68}$ ~~77 80~~

Search for 64

Binary search, pseudocode

```
procedure binary search($x$: integer, $a_1, a_2, \ldots, a_n$: integers in non-
decreasing order)
    start := 1
    end := n
    while start < end
        mid := $\lfloor (start + end) / 2 \rfloor$
        if $x > a_{mid}$ then
                start := mid + 1
        else if $x < a_{mid}$ then
                end := mid – 1
        else
            start := mid
            end := mid
    if $x = a_{start}$ then
        location := start
    else
        location := 0
    return location
```

Binary search: Example

Example: The steps taken by a binary search for 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. The list has 16 elements, so the midpoint is 8. The value in the 8th position is 10. Since 19 > 10, further search is restricted to positions 9 through 16.

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Values:	1	2	3	5	6	7	8	**10**	12	13	15	16	18	19	20	22

2. The midpoint of the list (positions 9 through 16) is now the 12th position with a value of 16. Since 19 > 16, further search is restricted to the 13th position and above.

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Values:	1	2	3	5	6	7	8	10	12	13	15	**16**	18	19	20	22

Binary search: Example

3. The midpoint of the current list is now the 14th position with a value of 19. Since 19 ≯ 19, further search is restricted to the portion from the 13th through the 14th positions .

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Values:	1	2	3	5	6	7	8	10	12	13	15	16	18	**19**	20	22

4. The midpoint of the current list is now the 13th position with a value of 18. Since 19> 18, search is restricted to the portion from the 14th position through the 14th.

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Values:	1	2	3	5	6	7	8	10	12	13	15	16	**18**	19	20	22

5. Now the list has a single element and the loop ends. Since **19=19**, the location 14 is returned.

Let’s revisit a problem we specified earlier

Sorting: Put the provided values in ascending order

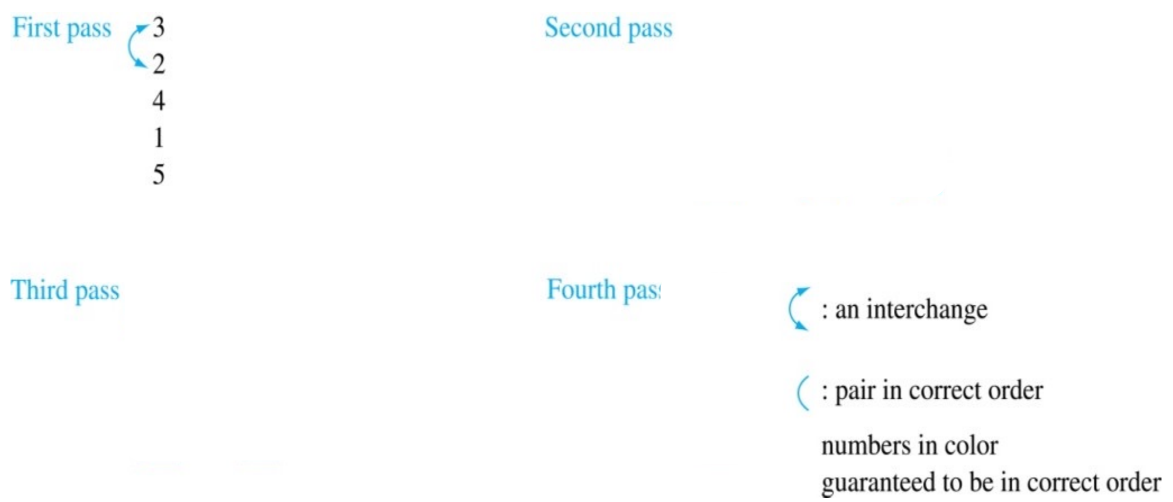
- Input: A sequence of numeric values $\{n_i\}$
- Output: A sequence $\{r_i\}$ containing the same elements as $\{n_i\}$, arranged such that $\forall i(i < j \rightarrow r_i < r_j)$

Bubble sort, intuition

- When comparing two adjacent elements, they should be *relatively sorted* (if not, let’s ~~swap~~ interchange them)
- Traverse the sequence left to right, swapping whenever necessary
- Repeat as many times as necessary
  - How many times?

Bubble Sort: Example

Example: Show the steps of bubble sort with 3 2 4 1 5



At the first pass the largest element has been put into the correct position
At the end of the second pass, the 2nd largest element has been put into the correct position.
In each subsequent pass, an additional element is put in the correct position.

Bubble sort pseudocode

```
procedure bubble sort($a_1$, $a_2$, …, $a_n$: real numbers)
    for $i$ := 1 to $n$-1
        for $j$ := 1 to $n$-1
            if $a_j > a_{j+1}$ then
                swap $a_j$ and $a_{j+1}$
```

Do we need to go all the way to the end?

Any optimizations?

- Inner loop stopping value: Consider how much progress we can guarantee from each pass
- Outer loop responsive termination: Might we finish sorting before $i = n-1$?

Specific types of problems and algorithms

Definition: An optimization problem is a problem in which the goal is to find the solution (among a set of possible solutions) that maximizes or minimizes the value of some parameter

Examples:

- Given the locations of a set of cities, what visitation order minimizes total travel? (Traveling salesperson problem)
- Given an amount of change, what denominations should be given to minimize the number of coins? (Change-making problem)
- Given an actor, what is the minimum number of “hops” (each to an actor that appeared with the previous in some movie) to arrive at Kevin Bacon?

One strategy for optimization problems: Greedy algorithms

Definition: A greedy algorithm is an algorithm that makes what seems to be the "best" choice at each step while iteratively constructing a solution.

A greedy approach to change-making:

- At each step, select the greatest denomination that can fit within the remaining change to give

A greedy approach to traveling salesperson:

- At each step, visit the (unvisited) city closest to your current location

Note that, depending on the problem, a greedy algorithm **might not** find the optimal solution!

Cashier's algorithm, pseudocode

```
procedure cashier's change ($c_1, c_2, \ldots, c_r$: values of denominations in decreasing order; $n$: a positive integer)
    for $i$ := 1 to $r$
        $d_i := 0$  {counts how many coins of denom. $i$}
        while $n \geq c_i$
            $d_i := d_i + 1$  {add a coin of denom. $i$}
            $n := n - c_i$  {remove value from remaining change to give}
    return $\{d_i\}$  {this sequence specifies how many of each denom.}
```

Does this always return the optimal solution?

For US currency, the cashier’s algorithm is optimal

Lemma: When giving change with US currency, the optimal solution contains at most 2 dimes, at most 1 nickel, at most 4 pennies, and cannot have 2 dimes and a nickel.

- Intuition: If we had (e.g.) more than 2 dimes, we could replace them with fewer coins but equivalent value

Theorem: The cashier’s algorithm makes change using the fewest possible coins when using US denominations

- Briefly: Quarters are selected first, and selecting fewer quarters always increases total coins. Dimes are selected next, and fewer dimes always increases total coins based on the above lemma…

Final thoughts

- Algorithms are a major foundation of computer science
  - More structured than prose, easier to write than “real” code

- Greedy algorithms are convenient for optimization problems, but don’t always give optimal results

- Next time:
  - Growth rates and Big-O notations (Section 3.2)