Fundamental algorithm design strategies

William Hendrix

Lecture 3

Today

- Review

- Algorithm design strategies
  - Exhaustive search
  - Greedy algorithms

Correctness and incorrectness

- To prove an algorithm is correct:
  - Prove that it produces the correct output for *every* input
    - Trace input to find algorithm's output
    - Prove output is correct
  - Frequently proof by induction
  - May also use proof by contradiction

- To prove an algorithm is incorrect:
  - Find a *counterexample*
    - Instance where the algorithm computes an incorrect solution

Example: BadSort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- "Compare every i^{th} element, swapping any that are out of order, for i from n-1 to 1"

$i = 4$

- Input: 1 3 5 2 4

Example: BadSort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- "Compare every i^{th} element, swapping any that are out of order, for i from n-1 to 1"

$i = 3$

- Input: 1 3 5 2 4

Example: BadSort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- "Compare every i^{th} element, swapping any that are out of order, for i from n-1 to 1"

$i = 2$

- Input: 1 3 5 2 4

Example: BadSort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

```
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- "Compare every i^{th} element, swapping any that are out of order, for i from n-1 to 1"

$i = 1$

- Input: 1 3 4 2 5

Example: BadSort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- "Compare every i^{th} element, swapping any that are out of order, for i from n-1 to 1"

$\text{data}[2] > \text{data}[3]$

- Input: 1 3 2 4 5

Incorrectness exercise

- Prove that BinSort (next slide) is not a correct sorted search algorithm.

- **Problem:** search (sorted)
  - **Input:** an array of values (`data`) in ascending order and a target value (`t`)
  - **Output:** an index `i` such that `data[i] = t`, or `0` if `t` is not in `data`

Incorrectness exercise

```
data:  a sorted array of integers to search
n:     the size of data
t:     the target value
lo = 1
hi = n
while lo < hi
  mid = floor((hi + lo) / 2)
  if data[mid] = t
    return t
  else if data[mid] > t
    hi = mid
  else
    lo = mid
  end
end
if data[lo] = mid
  return lo
else
  return 0
end
```

Incorrectness exercise solution

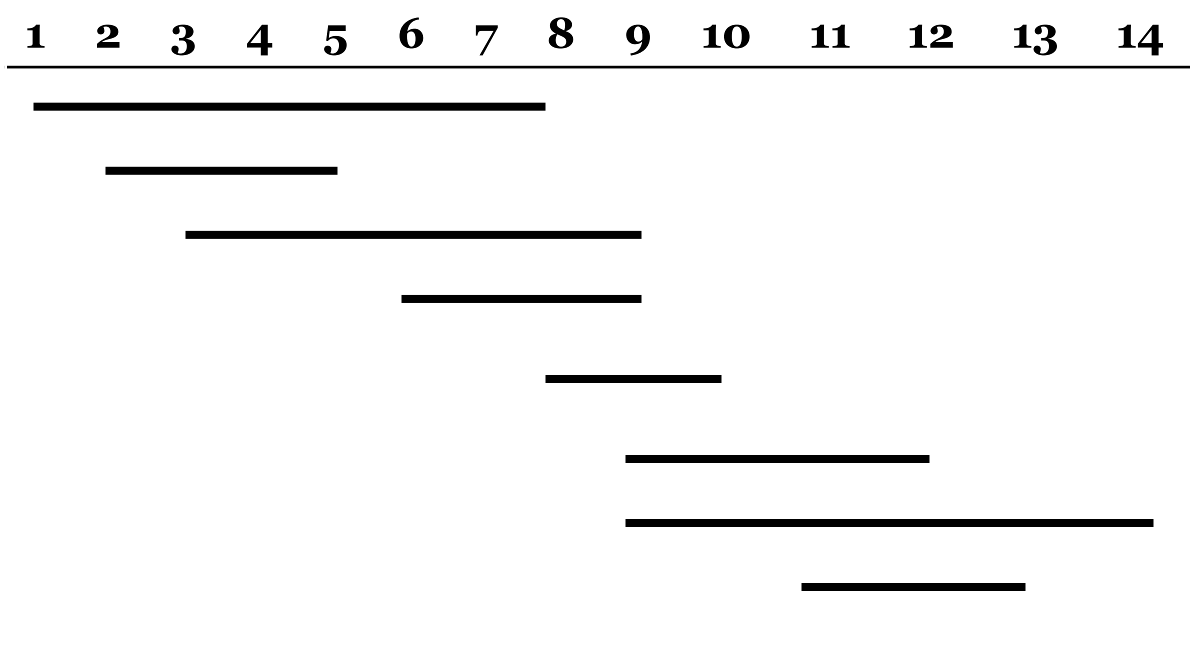
```
data:  a sorted array of integers to search
n:     the size of data
t:     the target value
1  lo = 1
2  hi = n
3  while lo < hi
4    mid = floor((hi + lo) / 2)
5    if data[mid] = t
6      return t
7    else if data[mid] > t
8      hi = mid - 1
9    else
10     lo = mid + 1
11   end
12 end
13 if data[lo] = mid
14   return lo
15 else
16   return 0
17 end
```

Proof. Consider the instance where:
`data = (10, 20)`,
`n = 2`, and
`t = 20`.
Lines 1 and 2 set `lo = 1` and `hi = 2`.
In the first iteration, `mid = 1`.
So, `data[mid] = 10`, which fits the third case of the `if` statement. As a result `lo = 1`.
However, these are the same values of `lo` and `hi` as the start of loop, so the loop repeats infinitely.
Since BinSort does not terminate on this input, it is not correct. □

Algorithm design example

- Consider the following problem:
- **Problem:** workshop scheduling
  - **Input:** the start times and durations for a set of workshops
  - **Output:** the largest number of workshops whose times do not overlap
- **Example instance:**

	Start	**Dur.**
A	1	7
B	2	3
C	3	6
D	6	3
E	8	2
F	9	3
G	9	5
H	11	1
I	13	1



Algorithm design

- How do we come up with algorithms to solve problems?

- Try to understand the problem
  - Solve small examples
- Consider various *algorithm design strategies* and which one is best
- Prove that your algorithm is correct

- **Strategy:** exhaustive search
  - A.k.a "brute force" method
  - Test all possibilities for the solution
  - Report the correct/best solution

Exhaustive search example

- Exhaustive scheduling:

```
start:  an array of start times for workshops
duration:  an array of durations for workshops
n:     the number of workshops
```

Pseudocode:

```
best = 0
workshops = (1, 2, ..., n)
while there are more subsets of workshops to test
  sub = next subset of workshops
  overlap = false
  for every pair of workshops (i, j) in sub
    if workshops i and j overlap
      overlap = true
    end
  end
  if overlap = false and |sub| > best
    best = |sub|
  end
end
return best
```

Analysis: exhaustive search

Pros

- Applicable to most problems
- *Always* gets the correct/optimum answer
- Easy to design and describe
- Makes proof of correctness easy

Cons

- Almost always slowest solution
- Often infeasible
- Exponential or factorial number of tests are impractical for most realistic purposes

Strategy: greedy algorithms

- Usually applied to *optimization* problems
  - "Find the best/largest/smallest/etc ..."
- **Outline**
  - Select the best possible element to add to the solution
    - Or eliminate the worst possible element
  - Repeat until there are no more possible elements to add/remove

- **Pros**
  - Usually easy to implement and describe
  - Generally good efficiency
  - Very good "first attempt" for optimization problems
  - May be "good enough" even if not correct
- **Cons**
  - Not always a correct solution (!!)
  - Need to decide how to determine "best/worst" element

GreedySearch

```
start:  an array of start times for workshops
duration:  an array of durations for workshops
n:      the number of workshops
```

Pseudocode:

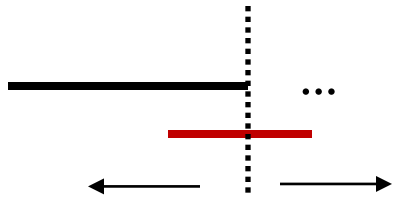
```
best = 0
subset = {}
Sort start and duration by ______________________
while start and duration not empty
  Add (start[1], duration[1]) to subset
  Remove all workshops from start and duration that
overlap this workshop
  best = best + 1
end
return best
```

GreedySearch variants

- Earliest workshop first
- **Counterexample:**

- Shortest workshop first
- **Counterexample:**

- Earliest workshop end first
- **Proof idea:** when we select the first workshop to add, every workshop we eliminate must overlap everything else we eliminate, and possibly more workshops. None of these could improve our solution.



Starts before here Ends after here

Formal proof of correctness

Proof. We prove the claim by contradiction. Suppose that GreedySearch (by earliest end time) is not correct. Let start and duration represent an instance that GreedySearch solves incorrectly, let $bestset$ be the set with the largest number of workshops, and let $found$ be the set identified by GreedySearch.

Since $found$ is incorrect, $bestset$ must have at least one element more than $found$. As a result, it must contain a workshop that $found$ does not. We use y to denote the earliest workshop that is in $bestset$ but not $found$. Since x is not in $found$, there must be some workshop y in $found$ with an earlier end time that overlaps x.

Consider the set formed by removing y from $bestset$ and adding x. Since all of the workshops that end before y in $bestset$ are also in $found$ and $found$ contains no overlaps, x cannot overlap any earlier workshop in $bestset$. Also, since x has an earlier end time than y and y does overlap any later workshop in $bestset$, x cannot either. So, this new set contains no overlaps and the same number of workshops as $bestset$. Thus, we can construct a solution of the same size that contains x.

If we repeat this process, we can eliminate every element (y) of $bestset$ not in $found$ without changing its size. This is impossible, as the resulting set would contain only the elements of $found$, but it would still have more elements than found. $\Rightarrow\Leftarrow$

Thus, our assumption that a larger solution exists must not be correct. $\square$

Coming up

- Complexity
- Big-Oh notation
- Logarithms

- **Homework 3** is due Tuesday
- **Homework 2** is due Thursday

- **Recommended readings:** Sections 2.1-2.4 and 2.6-2.7
- **Practice problems:** solve several problems from "Big Oh" in Chapter 2 (p. 58) and 1-2 from "Program Analysis" and "Logarithms"