CS2521: Introduction

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Today

- Course Practicalities
- What is an algorithm?
- How do we write and define an algorithm?
- Algorithm Correctness
- Notation

Algorithms

- What is an algorithm?

Algorithms

- What is an algorithm?

  "A well defined computational procedure that takes some input (values) and produces an output (i.e. one or more values)"

- An algorithm transforms an input into an output through some specific (c.f. well defined) process.
- What does well defined mean (given that computers, and often people too, are stupid)?

Defining Algorithms

- A well defined algorithm is one that is one that has a precise specification, which can be
  - written in standard English (very hard to make precise).
  - written as a computer program (precise, but then it can be hard to use or understand or use in other langauges).
  - written using a formal language (also hard to do, but typically easier to write than a program).
- The specification must deal with the input, output and process itself.
- Pseudo-code (essentially, like a programming language, but with no set syntax) is what we will use in this course. Pseudo-code provides a good balance of precision and ease of understanding.

Example: Sorting

- Let's consider the sorting problem, which (unsurprisingly) requires the creation of an algorithm to perform sorting (for example, of numbers, things with different sizes, etc).
- What are the inputs and outputs of this?

Example: Sorting

- Let's consider the sorting problem, which (unsurprisingly) requires the creation of an algorithm to perform sorting (for example, of numbers, things with different sizes, etc).
- What are the inputs and outputs of this?
  - **Inputs:** Some numbers
  - **Outputs:** The numbers from the input, but sorted

Example: Sorting

- Let’s consider the sorting problem, which (unsurprisingly) requires the creation of an algorithm to perform sorting (for example, of numbers, things with different sizes, etc).
- What are the inputs and outputs of this?
    - **Inputs:** Some numbers
    - **Outputs:** The numbers from the input, but sorted
- What are the problems with this?

Example: Sorting

- Let's consider the sorting problem, which (unsurprisingly) requires the creation of an algorithm to perform sorting (for example, of numbers, things with different sizes, etc).
- What are the inputs and outputs of this?
  - **Inputs:** Some numbers
  - **Outputs:** The numbers from the input, but sorted
- What are the problems with this?
  - How can we know whether we've solved the problem? What does the output look like? What does "sorted" mean?
  - What if the numbers are not unique?

Example: Sorting

- Let's consider the sorting problem, which (unsurprisingly) requires the creation of an algorithm to perform sorting (for example, of numbers, things with different sizes, etc).
- What are the inputs and outputs of this?
- **Inputs:** A sequence[1] of n numbers $N = \langle a_1, a_2, \ldots, a_n \rangle$
- **Outputs:** a reordering of N of the form $\langle a'_1, \ldots a'_n \rangle$ such that $a'_i \in N$ where $1 \leq i \leq n$; and $a'_1 \leq a'_2, \ldots \leq a'_n$.
- For example, given $\langle 3, 7, 223, 3, 4, 9 \rangle$ as an input, the algorithm should output $\langle 3, 3, 4, 7, 9, 223 \rangle$.
- The example is a <u>instance</u> of the sorting problem. A <u>problem instance</u> is thus the set of inputs (as defined by the problem statement) which is needed in order to compute a solution.

[1] Why not a set?

Defining Algorithms

- The process is where the ingenuity (and often difficulty) comes in - its here where you describe how to transform inputs to outputs in a way others can understand in a compact, precise manner.
- Suggested approach:
  - Describe ideas of an algorithm in English.
  - Clarify tricky bits in pseudo-code. Do not use pseudo-code to make bad ideas look formal!

The heart of any algorithm is an idea. If your idea is not clearly revealed when you express an algorithm, then you are using too low-level a notation to describe it.

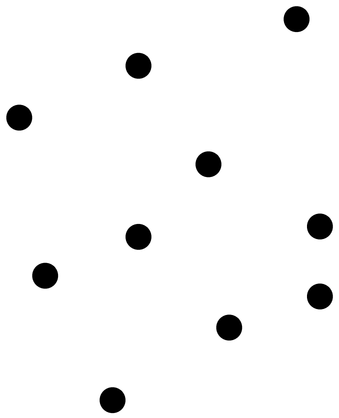
What are we looking for?

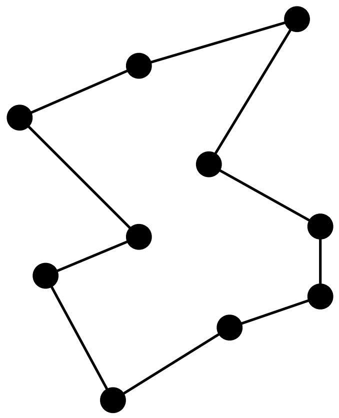
- A “good” algorithm has three desirable properties:
  - Correct
  - Efficient
  - Easy to implement
- These goals may not all be achievable together.
- Sometimes, we’ll be looking for something “good enough” rather than perfect.

Correctness

- An algorithm is correct if it always returns the desired output for all legal instances of the problem.
- In other words, an algorithm is said to be correct if, for every possible problem instance, it halts with a correct output.
- **Halts:** does not run forever.
- A correct algorithm solves the problem.
- Note that an incorrect algorithm can also find a solution to the problem, but might not halt on some inputs, or might halt with an incorrect answer.
- It is often difficult to show that an algorithm is correct!
- Showing an algorithm is correct is done via a proof. "It's obvious" is not good enough.

Example





You're in charge of writing a program to drive the robot arm. You need to provide an algorithm to find the best tour[2].

[2]A tour tells us which points to visit in which order

Example

- The first thing that's needed is a precise specification of the problem:
  - Problem: Robot Tour Optimisation
  - Input: Points on a circuit board
  - Output: The order in which we must move between points

Example

- The first thing that's needed is a precise specification of the problem:
    - Problem: Robot Tour Optimisation
    - Input: A set S of n points in the plane
    - Output: The shortest cycle tour that visits each point in the set S?
        - Is this definition sufficiently precise?

Example

- The first thing that's needed is a precise specification of the problem:
  - Problem: Robot Tour Optimisation
  - Input: A set S of n points in the plane
  - Output: The shortest cycle tour that visits each point in the set S?
    - Is this definition sufficiently precise?
  - Input': A set S of n points $\{(x_1, y_1), \ldots (x_n, y_n)\}$ such that $x_i, y_i \in \mathbb{R}$ for any $1 \leq i \leq n$
  - Output': A tour $M \in \mathcal{T}$, where $\mathcal{T}$ is the set of possible tours. Here, a tour $T \in \mathcal{T}$ is a sequence $T = [s_0, \ldots, s_{n-1}]$ such that $\forall 1 \leq i \leq n$, $s_i \in S$ and $s_i = s_j$ if and only if $i = j$, and for which $|T| = \sum_i = 0^n |s_i - s_{(i+1)\%n}|$. Then $|M| \leq |T|$ for any $T \in \mathcal{T}$.
- Suggestions for a solution?

A Solution

- Starting from some point on the board (let's call it p_0), move to its nearest neighbour, p_1. From p_1 move to the nearest unvisited neighbour p_2. Repeat this until we run out of unvisited points, after which we return to p_0 to close the tour.

A Solution

```
Require: $P$, a set of $n$ points
 1: function NearestNeighbour($P$)
 2:     pick and visit an initial point $p_0$
 3:     i=0
 4:     while there are still unvisited points do
 5:         i=i+1
 6:         Let $p_i$ be the closest unvisited point to $p_{i-1}$
 7:         Visit $p_i$
 8:     end while
 9:     Visit $p_0$
10: end function
```

- The good:
  - simple to understand and implement.
  - Makes sense intuitively.
  - Efficient, it only looks at each pair of points twice (once when adding visiting p_i and again when adding p_j).

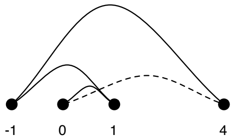
A Solution

```
Require: $P$, a set of $n$ points
 1: function NearestNeighbour($P$)
 2:     pick and visit an initial point $p_0$
 3:     i=0
 4:     while there are still unvisited points do
 5:         i=i+1
 6:         Let $p_i$ be the closest unvisited point to $p_{i-1}$
 7:         Visit $p_i$
 8:     end while
 9:     Visit $p_0$
10: end function
```

- The bad: It doesn't work.
    - A tour is always found (algorithm halts)
    - But the tour isn't the shortest (halts with incorrect answer)

A Solution

```
Require: $P$, a set of $n$ points
 1: function NearestNeighbour($P$)
 2:     pick and visit an initial point $p_0$
 3:     i=0
 4:     while there are still unvisited points do
 5:         i=i+1
 6:         Let $p_i$ be the closest unvisited point to $p_{i-1}$
 7:         Visit $p_i$
 8:     end while
 9:     Visit $p_0$
10: end function
```



- Starting from the left doesn't solve the problem.

An Alternative Solution

- Repeatedly connect the closest pair of endpoints whose connection will not create a problem.
- Each point begins as its own single vertex chain. We merge shortest distance vertex chains ending up with a single chain containing all points. Joining the final two endpoints gives us a cycle.
- We can call this the closest pair heuristic.
- **Homework:** Write pseudo-code for the closest pair heuristic. This will be discussed in the practical.

The Closest Pair Heuristic

- This heuristic is less efficient (and more complicated) than the nearest-neighbour approach suggested earlier.
- An algorithm should always provide a correct result. A heuristic does a good job, but provides no guarantee.
- It still fails to give the correct answer in many situations.

An Optimal Solution

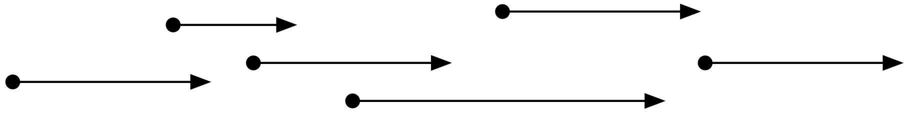
```
function OptimalTSP($P$)
    $d = \infty$
    $P_{min} = \emptyset$
    for all $n!$ permutations $P_i$ of $P$ do
        if $distance(P_i) \leq d$ then
            $d = distance(P_i)$
            $P_{min} = P_i$
        end if
    end for
    return $P_{min}$
end function
```

- This algorithm is correct, as it selects the best of all possible choices.
- It's slow. Given 20 points, $20! \sim 2 \times 10^{18}$. How long would this take a 1GHz machine?
- On a real circuit board, $n > 1000$.
- No efficient correct algorithm for this problem is known!
- Note: A faster algorithm running on a slower computer will always be faster for large instances (Often, large isn't actually very big)!

Scheduling

Imagine you're a famous actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus you can't simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.



Scheduling

- Formally:
  - Input: A set I of n intervals on the line
  - Output: The largest subset of mutually non-overlapping intervals which can be selected from I

Exhaustive Solution

```
function ExhaustiveScheduling($I$)
    $j = 0$
    $S_{max} = \emptyset$
    for all $S_i \in 2^I$ do
        if $\nexists S_a, S_b \in S_i$ s.t. $S_a \cap S_b \neq \emptyset \& size(S_i) > j$ then
            $j = size(S_i)$
            $S_{max} = S_i$
        end if
    end for
    return $S_{max}$
end function
```

Exhaustive Solution

```
1:  function ExhaustiveScheduling($I$)
2:      $j = 0$
3:      $S_{max} = \emptyset$
4:      for all $S_i \in 2^I$ do
5:          if $\nexists S_a, S_b \in S_i$ s.t. $S_a \cap S_b \neq \emptyset \& size(S_i) > j$ then
6:              $j = size(S_i)$
7:              $S_{max} = S_i$
8:          end if
9:      end for
10:     return $S_{max}$
11: end function
```

Test all 2^n subsets of intervals from I and return the largest subset consisting of mutually non-overlapping intervals.

Don't use maths for maths sake!

Possible Solutions

- An exhaustive solution would consider all 2^n possible subsets of I and find the maximal one.
- This would not take as long as finding all $n!$ orders from TSP.
- Nevertheless, $2^{100} > 20!$
- However, there is an algorithm which solves the scheduling problem efficiently and correctly.

Possible Solutions

1. Accept the earliest job j from I which does not overlap with any previously accepted jobs. Repeat until no such jobs remain.
2. Repeatedly accept the shortest job j from I and delete any jobs which intersect with it.
3. Repeatedly accept the job with the earliest completion time and delete any jobs which overlap with it.

Attempt 1

- Accept the earliest job j from I which does not overlap with any previously accepted jobs. Repeat until no such jobs remain.
- A long, early job will block lots of short jobs which overlap with it.

Attempt 2: Shortest Job First

```
1: function ShortestJobFirst($I$)
2:     while $I \neq \emptyset$ do
3:         accept shortest job $j$ from $I$
4:         delete $j$ and any jobs which intersect with $j$ from $I$
5:     end while
6: end function
```

- Accepting a short job might block us from taking 2 other jobs.

An efficient, optimal algorithm

```
function OptimalSchedule($I$)
    $acceptedJobs = \emptyset$
    while $I \neq \emptyset$ do
        add job $j$ from $I$ with earliest completion date to $acceptedJobs$
        delete $j$ and any jobs which intersect with $j$ from $I$
    end while
end function
```

What have we learned?

- Correct, efficient algorithms do exist, and this course intends to train you in how to find them.
- Reasonable looking algorithms can easily be incorrect.
- The easiest way to show an algorithm is incorrect is to find a counter-example. Such a counter-example must be verifiable
  - Calculate the answer the algorithm gives for the counter-example.
  - Display a better answer to show the algorithm did not find it.
- A counter-example should be simple, making clear why the proposed algorithm fails.

Finding Counter-Examples

- Use small examples: easier to verify and think about.
- If an algorithm always uses the biggest/smallest value, try work out when this could be the wrong thing to do.
- Try find extremes in your data: very big, very small, ties.
- Try think about all possibilities that should be verified (e.g. disjoint intervals, overlaps and nesting intervals for the scheduling problem).
- Not finding a counter-example does not mean an algorithm is correct. Correctness must be demonstrated through a proof.

Demonstrating Correctness

- An algorithm description is not enough to show correctness.
- A description of the problem that is being solved is also required.
  - The set of allowable input instances.
  - The required properties of the algorithm's output.
- Ask the wrong question and you'll get the wrong answer....
- Narrowing the permitted input can sometimes yield an efficient algorithm when none exists for the general case (e.g. consider the case where gaps in the schedule are allowed).
- When specifying output
  - Be precise. Asking for "the best route" from a navigation algorithm. Does best mean shortest, fastest, least turns?
  - Use simple goals. Don't say "find path from *a* to *b* that uses no more than *n* turns than necessary".

Correctness of the scheduling algorithm (Intuition)

- Let j be the first job to terminate.
- Any job, e.g. j' that overlaps it must terminate later, (possibly) blocking opportunities after j.
- So we should pick j (and delete all overlaps with j).
- In other words, picking j blocks the least jobs, allowing us to maximise the number of jobs we perform.

Proving Correctness

Problem: Sorting
Input: A sequence of n keys $a_1, \ldots, a_n$
Output: A permutation of the input sequence such that $a'_1 \leq a'_2, \ldots \leq a'_n$

- Instances: $\{6,3,7\} \rightarrow \{3,6,7\}, \{Ed, Al, Claire\} \rightarrow \{Al, Claire, Ed\}$

```
Require: a sequence $L = a_1, \ldots, a_n$
 1: function insertion sort(L)
 2:     for all $i = 1$ to length($L$)-1 do
 3:         $j = i$
 4:         while $j > 0$ and $a_j < a_{j-1}$ do
 5:             swap $a_j$ and $a_{j-1}$
 6:             $j = j - 1$
 7:         end while
 8:     end for
 9:     return L
10: end function
```

- We can prove that this sorting algorithm works by working from the simplest case upwards.
- This is a proof by induction

Induction

- Base case: consider an input consisting of only a single element. In this case, the sequence is already sorted.
- Inductive Assumption: Let's assume that the first n elements have already been sorted.
- General case:To deal with the $n+1$-th element (let's call it x), we need to find where it goes, which is the spot between the biggest element less than or equal to x and the smallest element greater than x. This is done by moving all greater elements up by one position, creating room for x where it needs to go.
- We also need to show that the algorithm terminates. Since the inner loop will eventually reach 0, and the outer loop will eventually have i=length(L)-1, termination must occur.

Induction Example 2

Prove the correctness of the following algorithm to increment a number (Input: y Output: y+1):

```
function Increment(y)
    if y=0 then
        return 1
    else if y mod 2 = 1 then
        return 2· Increment(⌊ y/2 ⌋)
    else return y+1
    end if
end function
```

Base case: y=0, obviously correct.

Induction Example 2

Prove the correctness of the following algorithm to increment a number (Input: y Output: y+1):

```
1: function Increment(y)
2:     if y=0 then
3:         return 1
4:     else if y mod 2 = 1 then
5:         return 2· Increment(⌊ y/2 ⌋)
6:     else return y+1
7:     end if
8: end function
```

Assume it works for the general case where y $\leq$ n-1. So we need to show it works for the case where y=n.

- If y mod 2 = 0 then it's trivial as line 6 returns the correct answer.

Induction Example 2

Prove the correctness of the following algorithm to increment a number (Input: y Output: y+1):

```
function Increment(y)
    if y=0 then
        return 1
    else if y mod 2 = 1 then
        return 2· Increment(⌊ y/2 ⌋)
    else return y+1
    end if
end function
```

- If y mod 2 = 1 then let n=$2 \times m$. Then $y = 2m + 1$ and so...

  2 ·Increment(⌊ (2m+1)/2 ⌋)
  = 2 · Increment(⌊ m+1/2 ⌋)
  = 2 · Increment(m)
  = 2(m+1)
  = 2m +2 = y+1
- Since that's our expected output, we've proven the general case.

Proof by induction is useful for proving the correctness of many algorithms

Loop Invariants

- A common approach to proving correctness of loops is through the use of a loop invariant - something that is true at the start, and end of the loop.
- The idea:
  - Show that the loop invariant holds before the start of the loop
  - Assume that it holds at the start of the loop
  - Show that it holds at the end of the loop
- Compare to proof by induction:
  - Base case - show that property holds for some initial case
  - Inductive assumption - assume true for arbitrary value
  - General case - show that it works for one step

Loop invariants - TSP

```
1:  function OptimalTSP($P$)
2:      $d = \infty$
3:      $P_{min} = \emptyset$
4:      for all $n!$ permutations $P_i$ of $P$ do
5:          if $distance(P_i) \leq d$ then
6:              $d = distance(P_i)$
7:              $P_{min} = P_i$
8:          end if
9:      end for
10:     return $P_{min}$
11: end function
```

- d is the minimal distance found so far (or P_{min} is the shortest path so far).
- Clearly holds before loop
- assume that it holds at start of loop
- lines 5-7 mean that it holds at end of loop

Loop Invariants - Scheduling

```
1: function OptimalSchedule($I$)
2:     $acceptedJobs = \emptyset$
3:     while $I \neq \emptyset$ do
4:         add job $j$ from $I$ with earliest completion date to $acceptedJobs$
5:         delete $j$ and any jobs which intersect with $j$ from $I$
6:     end while
7: end function
```

- What's the loop invariant?

Loop Invariants - Scheduling

```
1: function OptimalSchedule($I$)
2:     $acceptedJobs = \emptyset$
3:     while $I \neq \emptyset$ do
4:         add job $j$ from $I$ with earliest completion date to $acceptedJobs$
5:         delete $j$ and any jobs which intersect with $j$ from $I$
6:     end while
7: end function
```

- *acceptedJobs* contains the most jobs possible up to the time of the end of the last job within it.
- This clearly holds before the loop starts.
- Assume true at the start of the loop.
- At the end of the loop the new job added has the earliest completion date, anything else must end after that date (and we could not have added it). Therefore, we have added a job, which makes *acceptedJobs* hold more jobs than it otherwise would.

Summations

- We will be analysing algorithms later on in the course, and will often make use of summation.

$$\sum_{i=1}^{n} f(i) \equiv f(1) + f(2) + \ldots + f(n)$$

- There are lots of simple closed forms for summation.

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i = n(n+1)/2$$

$$G(n,a) = \sum_{i=0}^{n} a^i = (a^{n+1} - 1)/(a-1)$$

- When $|a| < 1$ $G(n,a)$ converges to a constant even as $n \to \infty$

Induction Example 3

- Prove that $\sum_{i=1}^{n} i \times i! = (n+1)! - 1$ by induction
  - Base case: for $n = 1$ we get $1 \times 1! = 1 = (1+1)! - 1 = 2 - 1 = 1$
  - Inductive case, i.e. assume true for n, show for $n+1$
  - Then LHS becomes

$$\sum_{i=1}^{n+1} i \times i! = (n+1)(n+1)! + \sum_{i=1}^{n} i \times i!$$

  - Substitute original formula's RHS into the new expression

$$\begin{aligned}\sum_{1=1}^{n+1} i \times i! &= (n+1)(n+1)! + (n+1)! - 1 \\ &= (n+1)!((n+1)+1) - 1 \\ &= (n+1)!(n+2) - 1\end{aligned}$$

  - Now RHS is $(n+1+1)! - 1 = (n+2)! - 1 = (n+2)(n+1)! - 1$
- Pulling out the largest term for a summation is a common trick.

Describing Algorithms

- Recall that when describing an algorithm, we dealt with precise concepts, e.g. n points on the plane, intervals on a line, sequences of numbers etc.
- Modelling the problem concisely using these concepts is a critical skill.
- Correct modelling may reduce your problem to an existing one, meaning that others may have done the hard work for you already.
- Many real world objects map to the same abstract concept. E.g. routing traffic over a road network, or trying to draw circuits are both problems that can be reduced to graphs.
- We're going to briefly survey some common objects that will commonly appear in various algorithms.

Common Objects[3]

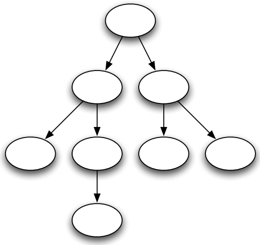
- Sets represent a collection of distinct items. Order is irrelevant. Thus, $\{1, 2, 3, 4\}$ is the same as $\{4, 3, 2, 1\}$. Refer to CS1 notes for more details. Sets that can contain multiple copies of an object are called multi sets.
- Permutations are orderings of some items. E.g. $(1, 2, 4, 3)$ and $(4, 1, 3, 2)$ are two permutations of the same integers.
- Tuples are ordered sets of objects with the same size and type. E.g. $(a, 5)$ and $(c, 9)$ are tuples of a letter and number. A tuple containing two objects is called a pair, and a tuple containing three objects is a triple. The cartesian product of n sets is the set of $n-$tuples.
- Subsets are a selection from a set of items. E.g. $\{2\}$ and $\{4, 1\}$ are distinct subsets of the natural numbers less than 4. Subsets are themselves sets. Thus, $\{4, 2\} \subset \{1, 2, 3, 4\}$

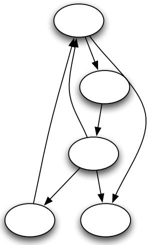
[3] Refer to Appendix B of Corman for more details

Further Reading

- http://www.drdobbs.com/cpp/mathematical-induction-makes-extrapolati/240168969
- http://www.drdobbs.com/cpp/loop-invariants-abbreviate-induction-pro/240169015
- http://www.drdobbs.com/cpp/using-a-loop-invariant-to-help-think-abo/240169056
- http://www.drdobbs.com/cpp/a-loop-invariant-can-be-an-optimization/240169097

Common Objects





- Trees represent a hierarchical relationship between items. They are a special type of graph.
- Graphs represent relationships between arbitrary pairs of objects. Graphs may be directed or undirected, and have nodes (or vertices) and edges.
- Graphs can be encoded using tuples.

Common Objects

- Points are pairs which represent locations in some space.
- Polygons represent regions in some space.
- Intervals represent regions on a line.
- Strings represent sequences of characters or patterns.

Recursion

- Many of the objects we've just described are composed of smaller versions of themselves.
  - Remove an element from a permutation of size n, and you have a permutation of size $n - 1$.
  - A subset is itself a set.
  - Remove a leaf from a tree, and you have a tree. Delete any other node, and you have 2 trees.
  - Delete a vertex or edge from a graph, and you have one or more graphs.
  - Sets of points can be split by drawing a line.
  - Adding a chord to a polygon creates a new polygon.
  - Removing a character from a string results in a shorter string.

Recursive Objects

- Describing an object <u>recursively</u> (i.e. in terms of smaller forms of itself) requires
  - A decomposition rule (i.e. how the object can be broken up)
  - And a base case (i.e. the smallest/simplest type of object which can't be broken up).
- Examples
  - Strings/Sets/permutations: the empty set ($\emptyset$ or $\{\}$)
  - Graphs/Trees: single vertex (or even the empty graph)
  - Point set: a single point
  - Polygons: a triangle
- Such recursive definitions are useful, and commonly used in proofs (especially by induction).

What have we Covered?

- Algorithms and heuristics - correctness, efficiency, ease of understanding.
- Writing algorithms - input, output, operations, pseudo-code.
- Algorithm correctness - returns desired output for any instance; halts.
- Incorrectness of algorithm - counter-example.
- Correctness - proof of solution and proof of halting.
- Proof by induction.
- Summations.
- Common objects, recursion.