Lecture 1: Introduction

Last updated: Aug 25, 2025

References:

- The Algorithm Design Manual, Skiener, Chapter 1
- Algorithm Design Techniques, Programming Pearls, Jon Bentley, ACM 1984
- Algorithms, Jeff Erickson. Chapter 0
- Algorithms, Gopal Pandurangan, Chapter 2.1

What is an algorithm?

An algorithm is an **explicit**, **precise**, **unambiguous**, **mechanically executable** sequence of **elementary instructions**, intended to accomplish a **specific purpose**.

- Explicit: can be described in words and mathematical notations
- Precise: only one interpretation
- Mechanically executable: only use elementary instructions that machines support
- Specific purpose: what does it accomplish?

Problem vs Problem Instance

To be interesting, an algorithm must solve a **general** problem. An algorithmic problem is specified by describing the complete set of **instances** it must work on and of its output.

The distinction between problem and problem instance is fundamental.

For example, the algorithmic problem known as *sorting* can be described:

Problem: sorting
Input: A sequence of n keys $a_1, \ldots, a_n$.
Output: The permutation (reordering) of the input sequence such that $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

An instance of the sorting problem might be an array of integers {2,3,1}.

Consider the following "procedure":

```
fun magic_sort(a[1..3]){
   output({1,2,3});
}
```

It correctly "sorts" the **problem instance** {2,3,1}, but fails on pretty much every other instances (defined by inputs).

This procedure is not an (correct) algorithm. An algorithm must solve a general problem (all possible instances), instead of one or part.

Most of the time, it's not clear whether the algorithm actually does solve all instances, so we must supply a **proof** of correctness of the algorithm to make it useful. The proof is a certification of the correctness of an algorithm.

Bad Example

Here's a curious algorithm:

```
fun BeAMillionareAndNeverPayTaxes():
   Get a million dollars
   if the tax man shows up
      say "I forgot"
```

What's the problem with this "algorithm"?

Practical Working Definition

A practical working definition an algorithm is some procedure that can be described in a text program, on some kind of representitive computer. Could be our typical x64, arm computer ISA, a theoretical device like turing machine, or a more commonly, a high-level programming language.

For convenience (and also relevance to digital computer) we use the following RAM machine that the Rhythm pseudocode language bytecodes.

Let's jump into an example:

```
1 // x,y (input): two positive integers
2 // returns the product x*y
3 fun mysterious_multiply(x, y) {
4     var prod = 0;
5     while (x>0) {
6         if (x%2 == 1)
7             prod = prod + y;
8         x = floor(x/2);
9         y = y + y;
10    }
11    return prod;
12 }
13 print mysterious_multiply(123,456);
```

generates bytecode

```
== BeatFunc: mysterious_multiply ==
0000    4 OP_CONSTANT         0 '0'
0002    5 OP_GET_LOCAL        0
0004    | OP_CONSTANT         1 '0'
0006    | OP_GREATER
0007    | OP_JUMP_IF_FALSE    7 -> 58
0010    | OP_POP
0011    6 OP_GET_LOCAL        0
0013    | OP_CONSTANT         2 '2'
0015    | OP_MODULO
0016    | OP_CONSTANT         3 '1'
0018    | OP_EQUAL
0019    | OP_JUMP_IF_FALSE   19 -> 34
0022    | OP_POP
0023    7 OP_GET_LOCAL        2
0025    | OP_GET_LOCAL        1
0027    | OP_ADD
0028    | OP_SET_LOCAL        2
0030    | OP_POP
0031    6 OP_JUMP            31 -> 35
0034    | OP_POP
0035    8 OP_GET_GLOBAL       4 'floor'
0037    | OP_GET_LOCAL        0
0039    | OP_CONSTANT         5 '2'
0041    | OP_DIVIDE
0042    | OP_CALL             1
0044    | OP_SET_LOCAL        0
0046    | OP_POP
0047    9 OP_GET_LOCAL        1
0049    | OP_GET_LOCAL        1
0051    | OP_ADD
0052    | OP_SET_LOCAL        1
0054    | OP_POP
0055    0 OP_LOOP            55 -> 2
0058    5 OP_POP
0059   11 OP_GET_LOCAL        2
0061    | OP_RETURN
0062    4 OP_POP
0063    0 OP_NIL
0064    | OP_RETURN
```

There are only 32 opcodes in rhythm currently; see the full list at https://github.com/robbwu/rhythm/blob/1212577d8f8613cf21fe378c3d92d9ccb9a33742/src/vm/chunk.hpp#L8

Each opcode counts as 1 step in time complexity analysis

Additional conveniences:

In Rhythm there are high level data structures: `Array` and `Map`.

`Array` is dynamic array like `std::vector` in C++; accessing, assigning, or pushing to the end cost amortized 1 step.

`Map` is unordered hash table like `std::unordered_map` in C++; accessing, testing key existence, assigning val to key all cost 1 step.

In the `beat` (bytecode interpreter) command line flag `-c` will enable printing out the total number of opcodes executed in the program like so:

```
pwu@Panruos-MacBook-Pro code % ~/Code/rhythm/build/beat -c peasant.rhy
56088
=== total opcodes executed: 228
```

The total opcodes executed could be the "number of elementary steps" that this algorithms takes on this problem instance.

Example: Multiplication

We would like to multiply two positive integers. First let's figure out what data representation we use. We mostly use the decimal positional notation: basically 123 means $1 \times 100 + 2 \times 10 + 3$. Formally, we represent an integer x,y as array of decimal digits $X[0 \ldots m-1]$, $Y[0 \ldots n-1]$

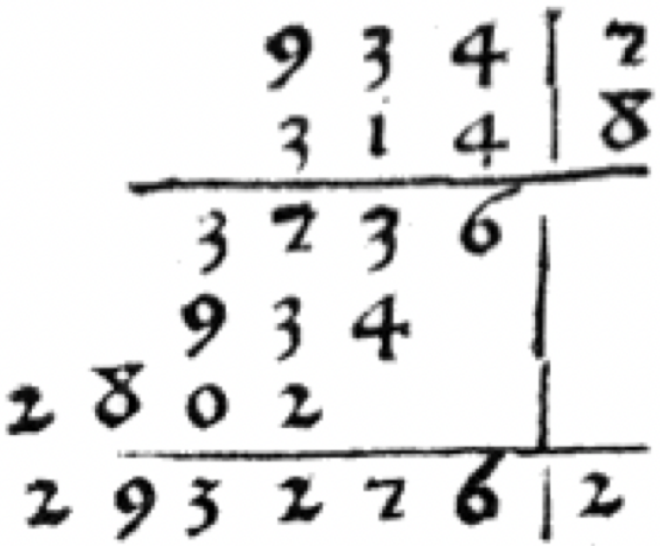
$$x = \sum_{i=0}^{m-1} X[i] \times 10^i, y = \sum_{j=0}^{n-1} Y[j] \times 10^j$$

We would like to compute $z = x \cdot y$ which is represented $Z[0 \ldots m+$

$n-1]$:

$$z = \sum_{k=0}^{m+n-1} Z[k] \times 10^k$$

I heard that Americans are most familiar with the Lattice algorithm, which is illustrated as



Why is it correct? Can we give a proof of the Lattice multiplication algorithm? Hint:

$$z = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} X[i]Y[j] \times 10^{i+j}$$

The algorithm is based on **elementary operations**: single digit multiplication (can be done by looking up in a table, from memory of a computer, etc) and addition.

Now we know it's correct, the next question is: is it efficient?

First we derive the time complexity in terms of the elementray operations. By some accounting, we see that the Lattice multiplication algorithm takes $O(mn)$ steps (single digit multiplication/addition).

There's an even older and maybe simpler algorithm goes by many names including **peasant multiplication** which reduces to four

operations; 1) determining parity; 2) addition; 3) duplation (doubling); 4) mediation (halving).

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

```
fun peasant_multiply(x, y) {
    var prod = 0;
    while (x>0) {
          if (x%2 == 1)
             prod = prod + y;
          x = floor(x/2);
          y = y + y;
    }
    return prod;

}

print peasant_multiply(123, 456); // expect 56088
```

Now why is this algorithm correct? It's based on the following recursive identity:

$$xy = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor (y+y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor (y+y) + y & \text{if } x \text{ is odd} \end{cases}$$

This is an recursive algorithm! (implemented as iterations).

Now what's the time complexity of this algorithm?

Without loss of generality, assume $x \leqslant y$. Clearly, the algorithm does $\log x$ parity, addition, and mediation. What's the cost of each operation?

Assuming any reasonable place-value (positional) representation of numbers (binary, decimal, Roman numeral, bead positions on abacus, etc)

each operation requires $O(\log x + \log y) = O(\log y)$ (because x has $O(\log x)$ digits). Therefore the total time complexity is $O(\log x \cdot \log y) = O(mn)$ time, the same as the Lattice algorithm!

This algorithm is arguably easier for humans to execute, because the basic operations are simpler (if you can't remember single digit multiplication table!).

In fact, for binary representation, the peasant algorithm is identical as lattice mulitplication algorithm.

The recursive formulation of PeasantMultiply:

```
fun peasant_recursive(x,y) {
  if (x==0) return 0;
  else if (x%2==0)
    return peasant_recursive(x/2, y+y);
  else
    return y + peasant_recursive(floor(x/2), y+y);
}
print peasant_recursive(123, 456); // expect 56088
```

Euclid's Multiplication: Compass and Straightedge

Ancient Greek geometers as "Computer"; elementary instructions:

- Draw the unique line passing through two distinct points: LINE(A,B)
- Draw the unique circle centered at a point C and pass through another point P: CIRCLE(C,P)
- Identify the intersection point of two lines
- Identify the intersection points of a line and a circle
- Identify the intersection points of two circle

How to do multiplication on the Greek computer?

Inputs are represented as two line segments, output is also a line segment. Very different representation of data!

```
⟨⟨Construct the line perpendicular to ℓ passing through P.⟩⟩
RightAngle(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← Intersect(Circle(P, A), ℓ)
  C, D ← Intersect(Circle(A, B), Circle(B, A))
  return Line(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MultiplyOrDivide(A, B, C, D):
  α ← RightAngle(Line(A, C), A)
  E ← Intersect(Circle(A, B), α)
  F ← Intersect(Circle(A, D), α)
  β ← RightAngle(Line(E, C), F)
  γ ← RightAngle(β, F)
  return Intersect(γ, Line(A, C))
```

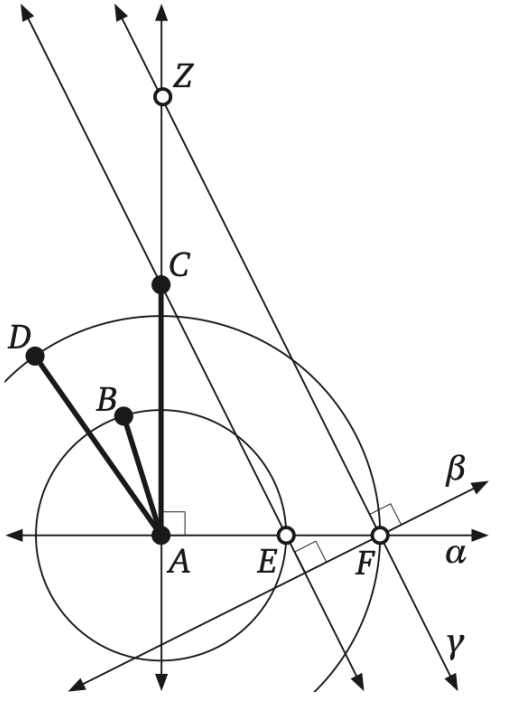
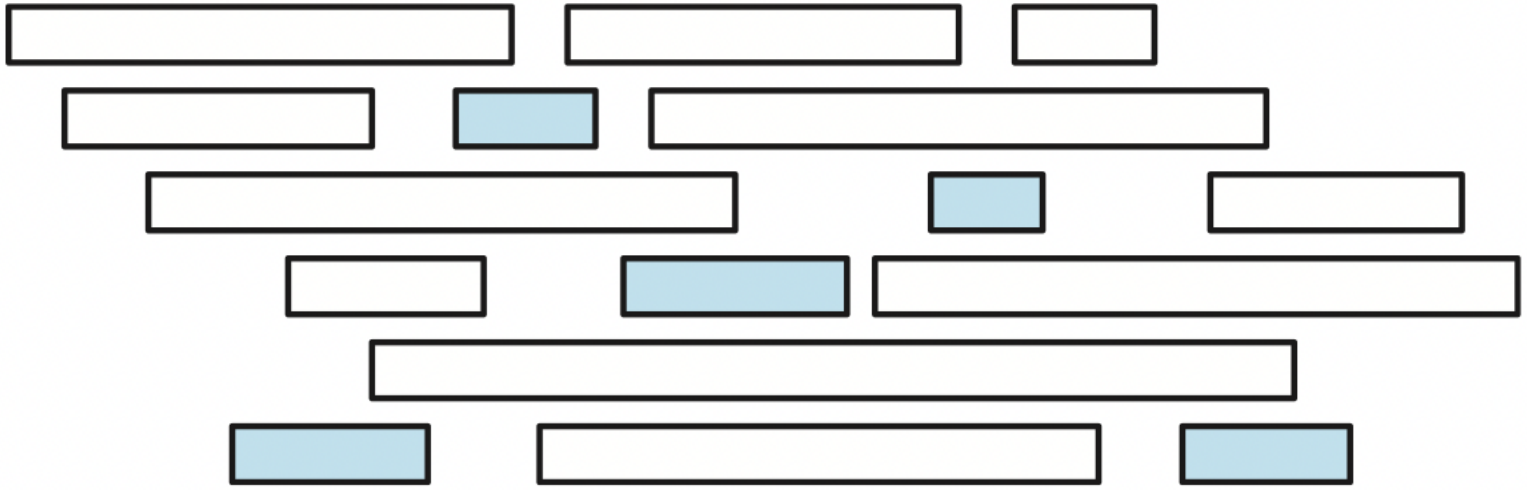


Figure 0.4. Multiplication by compass and straightedge.

Scheduling Classes

1 2 3 4 5 6 7 8 9 10 11 12 13



Suppose we are given n classes with potentially overlapping lecture time. Class i starts at time $S[i]$ and finishes at $F[i]$. Find the maximum number of non-overlapping classes.

We can visualize the class as blocks on time axis. The goal is to find the largest subset of blocks with no vertical overlap.

Think of an algorithm to solve it!

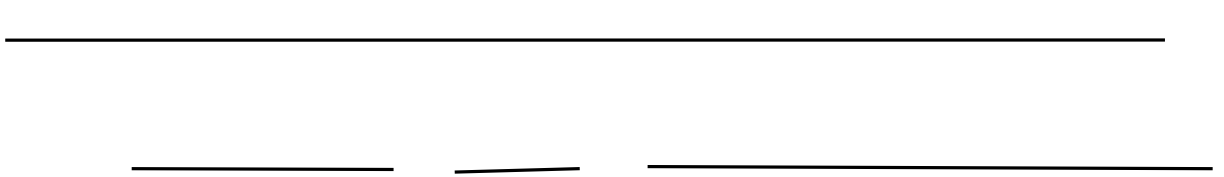
Let's try some ideas. Suppose the inputs are given by a set I of intervals ([start,finish]).

```
fun disjoint(int1, int2) {
    if (int1[1] < int2[0] or int2[1] < int1[0]) return true;
    return false;
}
// returns an array of classes to take
fun earliest_class_first(intervals) {
    qsort(intervals, 0, len(intervals), fun(a,b){return a[0]<b[0];});
    var res = [];
    for (var i=0; i<len(intervals); i=i+1) {
        if (len(res) == 0)
            push(res, intervals[i]);
        else if (disjoint(res[len(res)-1], intervals[i]))
            push(res, intervals[i]);
    }
    return res;
}

var intervals = [ [1,2], [3, 4], [0, 9] ];
print "earliest lass first result";
print earliest_class_first(intervals); // expects [[0, 9],]
```

Is this correct? If not, can you give an counter-example?

Counterexample



OK, let's try another one...

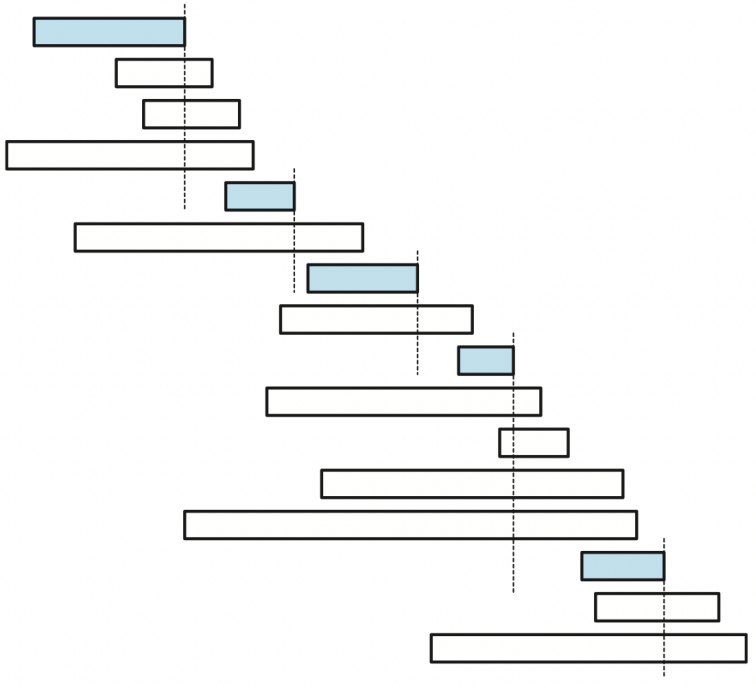
```
fun shortest_class_first(intervals) {
    qsort(intervals, 0, len(intervals), fun(a,b){return (a[1]-a[0])<(b[1]-b[0]);});
    var res = [];
    for (var i=0; i<len(intervals); i=i+1) {
        if (len(res) == 0)
            push(res, intervals[i]);
        else if (disjoint(res[len(res)-1], intervals[i]))
            push(res, intervals[i]);
    }
    return res;
}

print "shortest lass first result";
print shortest_class_first(intervals); // expects [[1,2],[3,4]]
```

This is still not correct...

How about this one:

```
fun earliest_finish_class_first(intervals) {
    qsort(intervals, 0, len(intervals), fun(a,b){return a[1]<b[1];});
    var res = [];
    for (var i=0; i<len(intervals); i=i+1) {
        if (len(res) == 0)
            push(res, intervals[i]);
        else if (disjoint(res[len(res)-1], intervals[i]))
            push(res, intervals[i]);
    }
    return res;
}
print "earliest finish lass first result";
print earliest_finish_class_first(intervals); // expects [[1,2],[3,4]]
```



Can you find any counter example?

But how do you know this algorithm is correct? In later lectures, we are going to **prove** that the EarliestFinishClassFirst() algorithm is guarantteed to give an optimal solution. **The proof is non-trivial**. This example shows that algorithms are not obvious to be correct. Finding counterexample proves the algorithm is incorrect;

but absence of counterexample does not prove it correct. We need much stronger argument.

Basic proof techniques include:

- Mathematical induction and recursion. They go like this:
  1. Basic case is obvious correct: n=0, n=1, for example.
  2. If the statement is true for all $k \leqslant n-1$, then we show that the statement is also true for $k = n$.
  3. We proved that the statement is true for any n.

Example: Find the Celebrity

Problem: A *celebrity* among a group of n people is someone who knows no one, but is known by everyone else. Identify a celebrity by only asking this question to a person: "do you know that person?".

Think about a solution.

Strategy: reduce (decrease) and conquer. What happens if we ask if A knows B? Two scenarios:

- A knows B. Then A cannot be celebrity. We reduce the problem by removing A from our later consideration.
- A does not know B. Then B cannot be celebrity. We reduce the problem by removing B from our later consideration.

Repeating this process for $n-1$ times, and we are left with only 1 person. If there is a celebrity, this one person must be it. (There cannot be more than 1 celebrities. Why?). But we are not sure whether the last one person is a celebrity or not; we must ask him $n-1$ questions to see if he knows anyone, and also if everyone knows him. This algorithm costs $(n-1) \times 3 = 3n-3$ questions.

A rigorous proof can be given by induction. How?

Describing Algorithms

There are three primary ways to descreibe algorithms:

- English words
- Pseudocode
- Computer programming language

in the order of increasing precision, but in decreasing conciseness and generality. In this course, we are going to primarily use combination of English and Rhythm ~~pseudocode~~

Rhythm is our "executable" pseudocode!

Learn about Rhythm in a single paper cheatsheet

Contents of This Course

This course consists of the following contents:

- Algorithm design techniques: recursion, divide and conquer, backtracking, greedy, randomized, ...
- Algorithm analysis: the proof of correctness, runtime/space complexity
- Selected illustrative or important algorithms & data structures: combinatoric problems, games, tree, graphs, networks, hash, disjoint sets...
- Problem solving: how to solve a problem? From distilling a specification of problem, to designing algorithm, to analyzing its correctness and performance, and to turn that into computer programs.

The RAM Model of Computation

Why can we analyze the performance of algorithms independent of software systems and the machine? That's because we rely on the **RAM model of computation**, a simple abstract machine that captures the first-order performance characteristics of real machines:

- Each simple operation (+,-,*,/,if,call) takes 1 step. In reality, not all steps are equal; / is usually much slower than others.
- Loops and subroutine calls are not simple operation.
- Each random memory access takes 1 step. In reality, there's cache, data locality, and memory is not really random access.

By abstracting the machine, we can simply count the number of "steps" an algorithm needs. It captures the asymptotic behavior of an

algorithm very well. (faster algorithm is definitely faster on machine if problem size is sufficiently big).

Worst case Complexity

Unless otherwise stated, we assume in this course that we are always referring to the worst case complexity. Why prefer worst case than average case, or best case?

- Worst case complexity is easy to analyze
- It's easy to use—no need to assume any specialness of input
- It's conservative—guaranttee to be working as described, if not better.

Average case complexity is sometimes more appropriate, especially in randomized algorithms. But it's more involved in analysis, because it needs assumptions on input probablistic distribution. We'll consider average case complexity on as-needed basis.

Asymptotic notations (review):

Big-O: upper bound of the functions, when problem size n is big (asymptotic behavior of functions).

- $g(n) = O(f(n))$ means that there exists constant C such that $g(n) \leqslant Cf(n)$, for all sufficiently larege n. Put it in another way, $g(n)$ grows no faster than $f(n)$.
- Similarly, $g(n) = \Omega(f(n))$ means lower bound.
- Similarly, $g(n) = \Theta(f(n))$ means lower and upper bounded.

Asymptotic Dominance: (assuming 1 step takes 1ns)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Dominance Rankings:

- $n!$
- c^n
- n^3,
- $n \log n$
- n
- $\sqrt{\mathsf{n}}$
- $\log \log n$
- $\log^2 n$

Proof by (Mathematical) Induction

Reference chapter: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/98-induction.pdf

Induction (short for mathematical induction) is a method for proving universally quantified propositions–statements about **all** elements of a (usually infinite) set. It's the single most useful tool for reasoning about, developing, and analyzing algorithms.

- **divisor** of a positive integer n is a positive integer p such that n/p is an integer. Trivially, 1, and n are divisors of n.
- A positive integer (>1) is **prime** if it has exactly two divisors, 1 and itself. Otherwise, it's **composite.**

Theorem 1. *Every integer greater than 1 has a* ***prime divisor****.*

How to prove? (this is a statement about all integers, which are infinite). Try proof by contradiction?

Proof. By Induction. (Mathematician can stop right here, but you are unlikely a professional mathematician, you need to continue).

The claim obviously work for small integers such as 2,3,4,5. Let n be an arbitrary integer greater than 1. Assume that every integer k such that $1 < k < n$ has a prime divisor (the claim is true for subset: induction hypothesis). If n is prime, then n is a divisor and prime, therefore claim holds for n. If n is composite, then n must have a proper divisor $d < n$. Since $1 < d < n$, by **induction hypothesis**, d has a prime divisor p, which must also be a prime divisor of n.

By mathematical induction, the claim holds for all positive integer $n > 1$. □

If you think this proof looks like "cheating" in that it references itself, well it is (not cheating, but rather recursive). A well written induction proof **looks very much like a recursive program**.

Theorem 2. *Given an unlimmited supply of 5-cents, and 7-cents stamps, we can make any amount of postage larger than 23 cents.*

Proof. By induction. Let n be an integer larger than 23.

Inductive hypothesis: assume for any integer k such that $23 <$

$k < n$, we can make k cents postage.

Inductive cases:

- If $n > 28$, then $n - 5 > 23$ and $n - 5 < n$. Therefore we can make $n - 5$ cents postage (invoke induction hypothesis). Add 5 cents we get n cents postage.

- (base cases) If $n < 28$, then there are only 6 cases: $n = 24, 25, 26, 27, 28$. We can just solve them one by one:

  $24 = 7 + 7 + 5 + 5, 25 = 5 + 5 + 5 + 5 + 5, 26 = 7 + 5 + 5 + 5, \ldots$

□

Note a program that prints out the composition of postages:

```
fun postage(n) {
    assert(n > 23);
    if (n == 24)
        printf("7+7+5+5");
    else if  (n == 25)
        printf("5+5+5+5+5");
    else if (n == 26)
        printf("7+7+7+5");
    else if (n == 27)
        printf("7+5+5+5+5");
    else if (n == 28)
        printf("7+7+7+7");
    else {
        postage(n - 5);
        printf("+5");
    }
}
postage(42); //expects 7+5+5+5+5+5+5+5
```

Is the recursive algorithm in fact… an induction proof?

Exercise 1. Prove

$$\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$$

for every non-negative n.

Exercise 2. Prove

$$\left(\sum_{i=0}^{n} i\right)^2 = \sum_{i=0}^{n} i^3$$