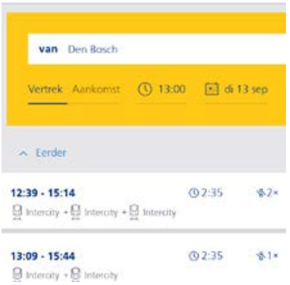
Data Structures & Algorithms

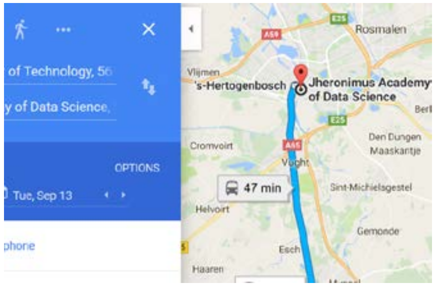
Lecture 1: Linear Search & Proofs

Many slides by Kevin Buchin

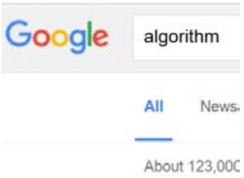
Algorithms: Examples

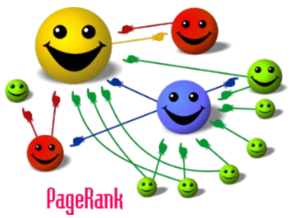
- Route planning
  shortest-path algorithms

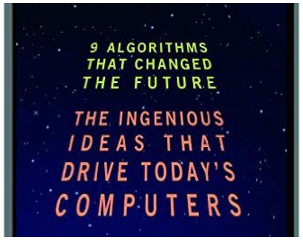




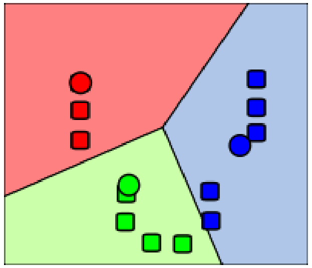
- Search engines
  matching and ranking algorithms

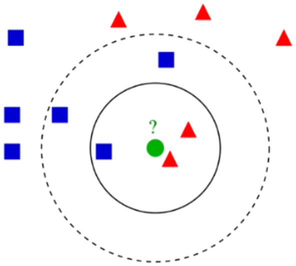






- Data Analysis
  e.g., k-means clustering algorithm,
  k-nearest neighbor algorithm, …





- Algorithms run everywhere: cars, smartphones, laptops, servers, climate-control systems, elevators …

Algorithms on computers

- How do you "teach" a computational device to perform an algorithm?
- Humans can tolerate imprecision, computers cannot

  "*if traffic is bad, take another route …*"
- Computers do not have intuition or spatial insight

> "An algorithm is a set of steps to accomplish a task that is described precisely enough that a computer can run it."

Algorithms

Algorithm

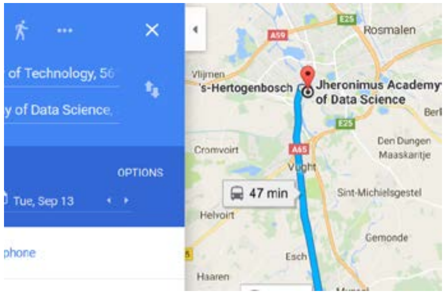
a well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output.

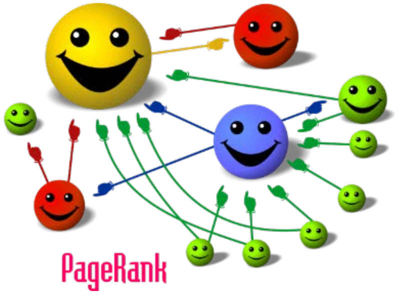
Algorithm

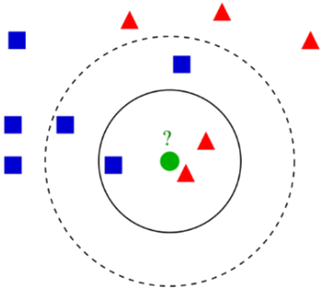
sequence of computational steps that transform the input into the output.

Algorithms & Data Structures

fast algorithms require the data to be stored in a suitable way.







Data structures

Data Structure

a way to store and organize data to facilitate access and modifications.

Abstract data type

describes functionality (which operations are supported).

Implementation

a way to realize the desired functionality

- how is the data stored (array, linked list, …)
- which algorithms implement the operations

The course: Overview

- Design and analysis of efficient algorithms for some basic computational problems.
  - Basic algorithm design techniques and paradigms
  - Efficiency and algorithms analysis: O-notation, recursions, …
  - Basic data structures
  - Basic graph algorithms
  - Intro to NP-Completeness

The course: Objectives

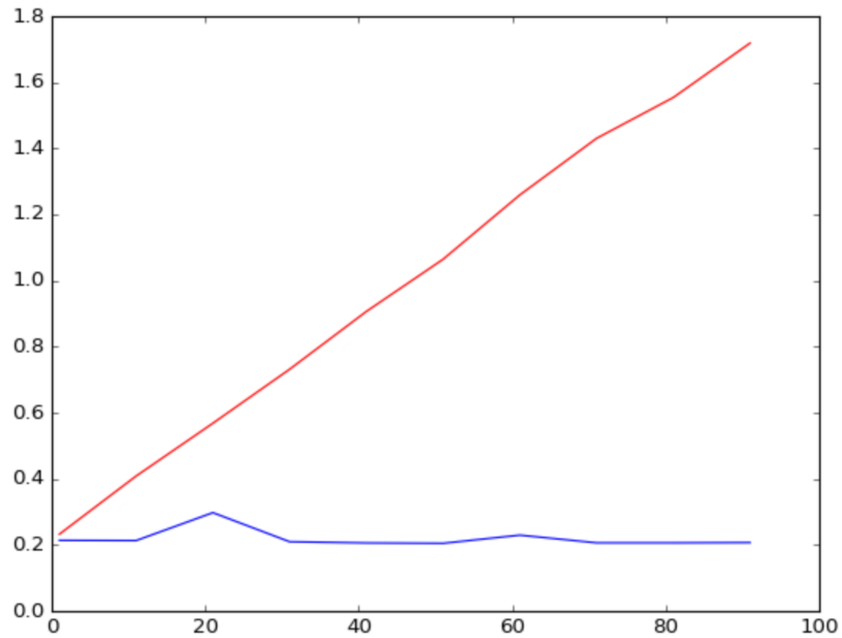
For any computational task on data you need an algorithm to solve it, and you need to store the data in a suitable data structure to access the data. You need these algorithms and data structures to be efficient.

At the end of this course, you should be able:

- select a suitable basic algorithm and data structure for a given task
- design efficient algorithms for simple computational tasks

Example: Select a data structure

- If you often need to search your data, simply storing it in an array/list will considerably slow down the computations

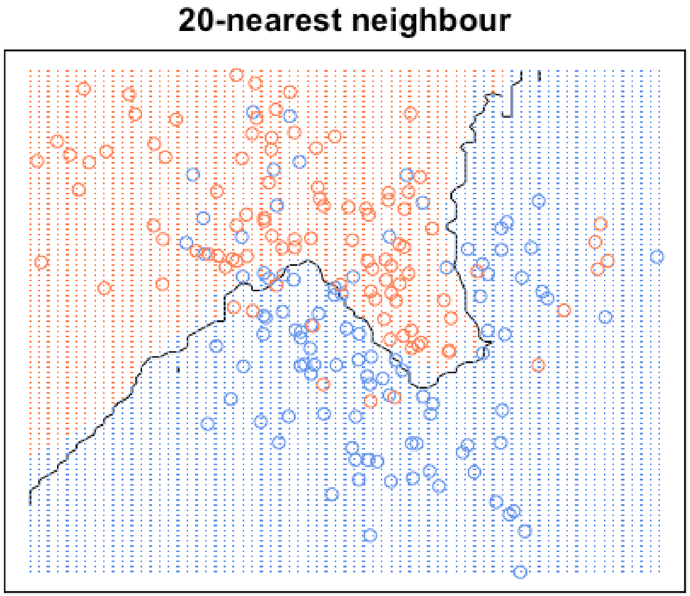



```
In [5]: timeit.timeit(stmt='1 in A', setup='A = list(range(2, 300))')
Out[5]: 4.774117301917343

In [6]: timeit.timeit(stmt='1 in A', setup='A = set(range(2, 300))')
Out[6]: 0.0499541983165841
```

Example: Select an algorithm

- Algorithms for finding the k nearest neighbors are used for analysis tasks like classification



- Which algorithm/implementation is suitable for your data?

Example: Select an algorithm

http://scikit-learn.org/stable/modules/neighbors.html

1.6.4. Nearest Neighbor Algorithms

1.6.4.1. Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: for N samples in D dimensions, this approach scales as $O[DN^2]$. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples N grows, the brute-force approach quickly becomes infeasible. In the classes within `sklearn.neighbors`, brute-force neighbors searches are specified using the keyword `algorithm = 'brute'`, and are computed using the routines available in `sklearn.metrics.pairwise`.

1.6.4.2. K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, *without having to explicitly calculate their distance*. In this way, the computational cost of a nearest neighbors search can be reduced to $O[DN\log(N)]$ or better. This is a significant improvement over brute-force for large N.

An early approach to taking advantage of this aggregate information was the *KD tree* data structure (short for *K-dimensional*

Example: Design a simple algorithm

Exploring weather data: Iterative and recursive algorithms by the example of CountInInterval

In this notebook we want to explore weather data of the weather in Eindhoven. Specifically, we will develop a routine CountInInterval(
an array A how many values are between p and q.

In [1]:
```
import csv

with open('weather-eindhoven.csv') as csvfile:
    reader = csv.reader(csvfile)
    next(reader)  # skip header row
    datatype = int
    date = []
    max_temperature = []
    for row in reader:
        date.append(datatype(row[1])) #column 1 (that is, the second) contains the date
        max_temperature.append(datatype(row[14])/10) # column 14 contains the max temperature times 10
```

In [2]:
```
def countInInterval(A, p, q):
    answer = 0
    for i in range(0, len(A)):
        if A[i]>= p and A[i]<= q: answer = answer+1
    return answer
```

In [3]:
```
print(countInInterval(max_temperature, -5, 5))
```

What do we expect from an algorithm?

What do we expect from an algorithm?

- Computer algorithms solve computational problems
- Computational problems have well-specified input and output
  - Question: Is the following problem well-specified?

    ~~*"Given a collection of values, find a certain value x."*~~

array: sequential collection of elements, which allows constant-time access to an element by its index. (*Note: We (and Python) use as first index 0, in the textbooks, they start at 1.*)

35	30	19	30	8	12	11	17	2	5
0	1	2	3						

"Given an array A of elements and another element x, output either an index i for which A[i] = x, or Not-Found"

There are 2 requirements on the algorithm:

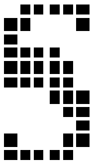
1. Given an input the algorithm should produce the correct output
2. The algorithm should use resources efficiently

Correctness

Given an input the algorithm should produce the correct output

- What is a correct solution?
  For example, the shortest-path … but given traffic, constructions … input might be incorrect

- Not all problems have a well-specified correct solution



➡ we focus on problems with a clear correct solution

- Randomized algorithms and approximation algorithms
  special cases with alternative definition of correctness

Efficiency

The algorithm should use resources efficiently

- The algorithm should be reasonably fast (elapsed time)
- The algorithm should not use too much memory
- Other resources: network bandwidth, random bits, disk operations …

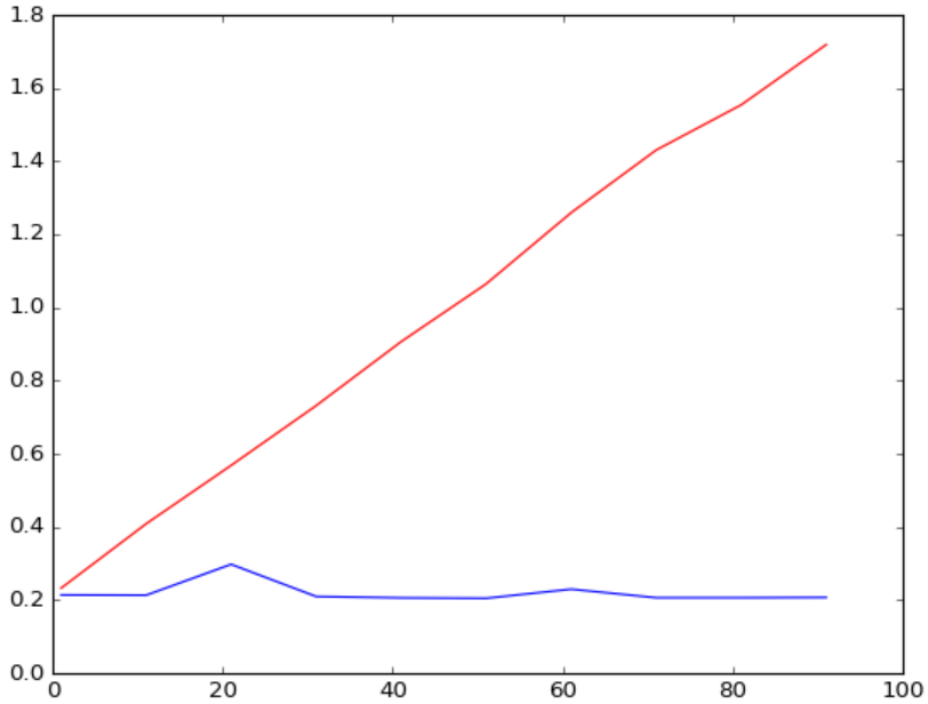
➡ we focus on time

- How do you measure time?

Experiments?

```
In [5]: timeit.timeit(stmt='1 in A', setup='A = list(range(2, 300))')
Out[5]: 4.774117301917343

In [6]: timeit.timeit(stmt='1 in A', setup='A = set(range(2, 300))')
Out[6]: 0.0499541983165841
```



Efficiency

The algorithm should use resources efficiently

- How do you measure time?

- Extrinsic factors: computer system, programming language, compiler, skill of programmer, other programs …

➡ implementing an algorithm, running it on a particular machine and input, and measuring time gives very little information

Efficiency analysis

Two components:

1. Determine running time as function $T(n)$ of input size n
2. Characterize rate of growth of $T(n)$

- Focus on the order of growth
  ignore all but the most dominant terms

Examples

Algorithm A takes $50n + 125$ machine cycles to search a list

- $50n$ dominates 125 if $n \geq 3$, even factor 50 is not significant
  ➡ the running time of algorithm A grows linearly in n

Algorithm B takes $20n^3 + 100n^2 + 300\,n + 200$ machine cycles

➡ the running time of algorithm B grows as n^3

Intermezzo: Logarithms

- $\lg n$ denotes $\log_2 n$

- We have for $a, b, c > 0$:

1. $\log_c (ab) =$

   $\log_c a + \log_c b$

2. $\log_c (a^b) =$

   $b \log_c a$

3. $\log_a b =$

   $\log_c b \,/\, \log_c a$

Exercise

- Compare growth rates

Rank the following functions of n by order of growth (starting with the slowest growing). Functions with the same order of growth should be ranked equal.

$$\log n^3,\ n,\ n^2 \log n,\ 4^{\log n},\ \log \sqrt{n},\ n + \log n^4,\ 2^{\log 16},\ n^{-1},\ 16,\ n^{\log 4}$$

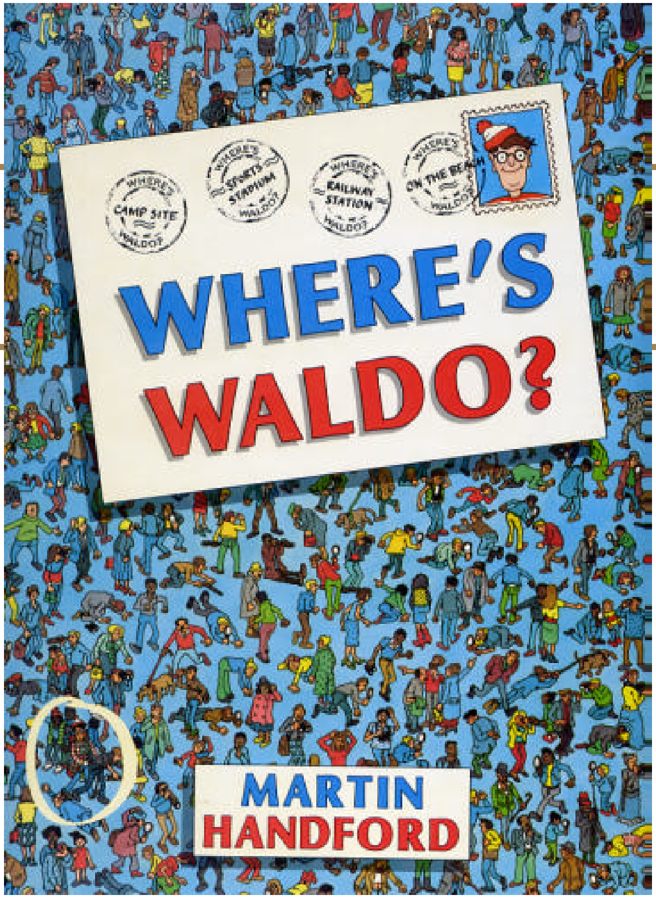
Comparing orders of growth

- $\log^{35} n$ vs. $\sqrt{n}$?
  - logarithmic functions grow slower than polynomial functions
  - $\lg^a n$ grows slower than n^b for all constants $a > 0$ and $b > 0$
- n^{100} vs. 2^n ?
  - polynomial functions grow slower than exponential functions
  - n^a grows slower than b^n for all constants $a > 0$ and $b > 1$

Describing algorithms

- A complete description of an algorithm consists of three parts:
  1. the algorithm
     - *expressed in whatever way is clearest and most concise,*
     - *can be English and / or "readable code",*
     - *readable: pseudo-code, python code, etc.*
     - *code will nearly always need a short high-level description in words*
  2. a proof of the algorithm's correctness
  3. a derivation of the algorithm's running time

Searching


WHERE'S CAMP SITE WALDO?
WHERE'S SPORTS STADIUM WALDO?
WHERE'S RAILWAY STATION WALDO?
WHERE'S ON THE BEACH WALDO?
WHERE'S WALDO?
MARTIN HANDFORD

Linear Search – Pseudo-Code

Linear-Search(A, n, x)

Input and Output specification

Input:

- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

Linear Search

Linear-Search(A, n, x)

array A[0 … n-1]

35	30	19	30	8	12	11	17	2	5
0	1	2	3						n-1

A.length = n

Input:
- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. Set answer to Not-Found
2. For each index i, going from 0 to n-1, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Linear Search in Python

```
In [6]: def linear_search(A, x):
            answer = -1
            for i in range(0, len(A)):
                if A[i] == x: answer = i
            return answer

In [7]: linear_search([10, 5, 9, 9], 10)

Out[7]: 0

In [8]: linear_search([10, 5, 9, 9], 9)

Out[8]: 3

In [9]: linear_search([10, 5, 9, 9], 8)

Out[9]: -1
```

Linear Search

Linear-Search(A, n, x)

Input:

- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

Loop with variable i

1. Set answer to Not-Found
2. For each index i, going from 0 to n-1, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Body of the loop

This loop always runs until n-1. Is that necessary?

Linear Search

Better-Linear-Search(A, n, x)

35	30	19	30	8	12	11	17	2	5
0	1	2	3						n-1

Input:

- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

Better Linear Search in Python

```
In [10]: def better_linear_search(A, x):
             for i in range(0, len(A)):
                 if A[i] == x: return i
             return -1

In [11]: better_linear_search([10, 5, 9, 9], 10)

Out[11]: 0

In [12]: better_linear_search([10, 5, 9, 9], 9)

Out[12]: 2

In [13]: better_linear_search([10, 5, 9, 9], 8)

Out[13]: -1
```

Describing algorithms

- A complete description of an algorithm consists of three parts:

  1. the algorithm
     - *expressed in whatever way is clearest and most concise,*
     - *can be English and / or "readable code",*
     - *readable: pseudo-code or python code*
     - *code will nearly always need a short high-level description in words*

  2. a proof of the algorithm's correctness

     requires writing a mathematical proof!

  2. a derivation of the algorithm's running time

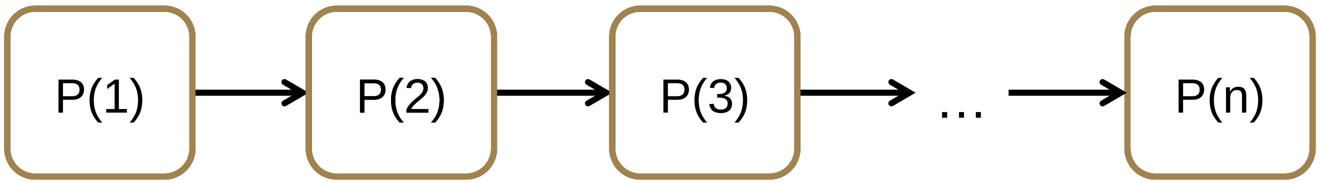
General Tips for Writing Proofs

1. State the proof techniques you're using (e.g. induction, loop invariant proof, …)
2. Keep a linear flow
3. Describe every step clearly in words
4. Don't use complicated notation
5. Make sure your axioms are actually "obvious"
   - What is obvious to you may not be obvious to the reader
6. Finish your proof
   - Connect everything with what you were trying to prove

MATHEMATICAL INDUCTION

The Idea

The Base Idea of Induction



Base Case

One (or more) very simple cases that we can trivially proof.

Induction Hypothesis

The statement that we want to prove (for any n).

Step

Prove that if the statement holds for a small instance, it must also hold for a larger instance.

Usage

When to use?

- Whenever you need to prove something is true for all values of n.
  - (or all values $\geq x$)
  - Infinite possibilities!
- When there is a clear structure in the problem (e.g., trees)
  - We will see more about trees later in the course

Basic Example

Theorem

If n dominos are placed in a row and I push the first; they all fall.

Proof:

We use induction on n.

Base Case $(n = 1)$:

If there is only 1 domino, it must also be the first.

I will push the first over, so trivially they all fall.

Basic Example

Proof:

We use induction on n.

Base Case $(n = 1)$:

If there is only 1 domino, it must also be the first.

I will push the first over, so trivially they all fall and the IH holds.

Induction Hypothesis:

If n dominos are placed in a row and I push the first; they all fall.

Step:

Assume the IH holds for n dominos.

If there were $n+1$ dominos in a row, the first n form a row of length n.

By IH the first n dominos will all fall. As all n dominos fall, so must the n^{th} domino. If the n^{th} domino falls, then it will tip over the $(n+1)^{\text{th}}$. The first n dominos fall over and the $(n+1)^{\text{th}}$ domino also falls over.

So all $n+1$ dominos fall over. Thus, the IH holds for n+1.

Example

Theorem

For all positive integers n, $3^n - 1$ is even.

Proof:

We use induction on n.

Base Case (n = 1): $3^1 - 1 = 2$, which is indeed even.

IH: $3^n - 1$ is even.

Induction Step (n >= 1):

Assume that $3^n - 1$ is even. (IH)

We need to show that $3^{n+1} - 1$ is even.

We have: $3^{n+1} - 1 = 3 * 3^n - 1 = (2 * 3^n) + (3^n - 1)$.

A multiplication with an even number is always even ($2 * 3^n$).

By IH, $(3^n - 1)$ is also even. The sum of two even numbers is also even.

Thus, $3^{n+1} - 1$ must be even. The IH holds for n+1. □

Strong vs. weak induction

- In weak induction, the step goes from *P(k)* to *P(k+1)*
- In strong induction, the step goes from *P(c),…,P(k)* to *P(k+1)*

Strong induction

Principle:

Let $P(n)$ be a statement involving a positive integer n. If

[Base] $P(c)$ is true for some c, (e.g., $c = 1$) and

[Induction Step] $P(c),\ldots,P(k)$ implies the truth of $P(k+1)$ for every positive $k \geq c$,

then $P(n)$ must be true for all positive integers $n \geq c$.

In the Step, we assume the

[Hypothesis] $P(c),\ldots,P(k)$ holds

and use this to show that then also $P(k+1)$ holds.

Strong induction Example

Claim: Every integer greater than 1 is divisible by a prime number.

Proof by induction.

[Base] The result is true for 2, since 2 is prime and 2|2.

[Hypothesis] Assume all integers m, 1<m<n, are divisible by a prime.

[Induction Step]

- If n is prime, then n is divisible by itself -- a prime.
- If n is not prime, then it is composite. Thus n has a divisor m, with 1<m<n and m|n. By the induction hypothesis, m is divisible by a prime number p. So we have p|m and m|n, which implies p|n.

Practice Exercise 2

Theorem

For every integer $n \geq 5$, $2^n > n^2$

Practice Exercise 2

Proof:

We use induction on n.

Base case ($n = 5$): $2^n = 2^5 = 32 > 25 = 5^2 = n^2$

IH: $2^n > n^2$

Induction Step ($n \geq 5$):

Suppose that $2^n > n^2$ (IH).

We need to show that $2^{n+1} > (n+1)^2$.

We have:

$$2^{n+1} = 2 * 2^n > 2 * n^2 \text{ (by IH)}$$

So it is sufficient to show that $2 * n^2 \geq (n+1)^2 = n^2 + 2n + 1$ for $n \geq 5$. This can be simplified to $n^2 - 2n - 1 \geq 0$ or $(n-1)^2 \geq 2$.

This is clearly true for $n \geq 5$.

So it follows by induction that $2^n > n^2$ for $n \geq 5$. □

Exercises

1. Prove that for all $n \geq 1$: $\sum_{i=1}^{n} 2(3^{i-1}) = 3^n - 1$.

2. Prove that any number $n > 7$ can be written as sum of 3's and 5's.

3. What is wrong in the following proof?

   Claim: $6n = 0$ for all $n \geq 0$.

   Proof by induction:

   Clearly, if $n = 0$, then $6n = 0$.

   Now, suppose that $n > 0$. Let $n = a + b$.

   By the induction hypothesis, $6a = 0$ and $6b = 0$.

   Therefore, $6n = 6(a+b) = 6a+6b = 0+0 = 0$.

Correctness

Correctness proof

It's easy to see that Linear-Search works …
it's not always that easy …

- There are several methods to prove correctness

➡ today we focus on loop invariants

Loop invariant

an assertion that we prove to be true each time a loop iteration starts

Correctness proof

- To proof correctness with a loop invariant we need to show three things:

Initialization

Invariant is true prior to the first iteration of the loop.

Maintenance

If the invariant is true before an iteration of the loop, it remains true before the next iteration.

Termination

The loop terminates, and when it does, the loop invariant, along with the reason that the loop terminated gives us a useful property.

Linear Search

Better-Linear-Search(A, n, x)

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

to show

1. if index i is returned then A[i] = x ✔
2. if Not-Found is returned then x is not in the array

Loop invariant

At the start of each iteration of step 1, if x is present in the array A, then it is present in the subarray from A[i] through A[n-1]

Linear Search

Better-Linear-Search(A, n, x)

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

Loop invariant

At the start of each iteration of step 1, if x is present in the array A, then it is present in the subarray from A[i] through A[n-1]

Initialization

Initially, i=0 so that the subarray in the loop invariant is A[0] through A[n-1], which is the entire array.

Linear Search

Better-Linear-Search(A, n, x)

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

Loop invariant

At the start of each iteration of step 1, if x is present in the array A, then it is present in the subarray from A[i] through A[n-1]

Maintenance

If at the start of an iteration x is present in the array, then it is present in the subarray from A[i] though A[n-1]. If we do not return then A[i] ≠ x. Hence, if x is in the array then is it in the subarray from A[i+1] though A[n-1]. i is incremented before the next iteration, so the invariant will hold again.

Linear Search

Better-Linear-Search(A, n, x)

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

Loop invariant

At the start of each iteration of step 1, if x is present in the array A, then it is present in the subarray from A[i] through A[n-1]

Termination

If A[i] = x ✓. If i > n-1 consider contrapositive of invariant.

"if A then B" ⇔ "if not B then not A"

Linear Search

Better-Linear-Search(A, n, x)

1. For $i = 0$ to $n-1$:
   A. If $A[i] = x$, then return the value of i as the output
2. Return Not-Found as the output

Loop invariant

At the start of each iteration of step 1, if x is present in the array A, then it is present in the subarray from $A[i]$ through $A[n-1]$

Termination

If $A[i] = x$ ✔. If $i > n-1$ consider contrapositive of invariant.

"if x is not present in the subarray from $A[i]$ through $A[n-1]$, then x is not present in A."

$i > n-1$ ➡ subarray from $A[i]$ through $A[n-1]$ is empty ➡ x is not present in an empty subarray ➡ x is not present in A

Exercise: Loop Invariant Proofs

We define the problem CountInInterval as follows: *Given an array* A *of* n *integers, and two integers* p *and* q*, count the number of elements of* A *that are at least* p *and at most* q*.*

Give an algorithm for CountInInterval. *Don't forget to prove correctness and analyze the running time.*

Recap and preview

Today

- Describing algorithms
- Efficiency Analysis (informal)
- Linear Search
- Mathematical induction
- Correctness proofs via loop invariants

Next lecture

- Efficiency analysis (formal)
- Binary Search
- Recursive algorithms