CSC165 Week 7

Larry Zhang, October 21, 2014

announcements

- **A1** marks on MarkUs
  - class average: 83%
  - re-marking form: available at course web page
- **A2** is out, due on Nov. 3, 10:00pm
  - manage your time wisely!
- (lots of) practice questions with solutions
  - http://www.cdf.toronto.edu/~heap/165/F14/A65_Practice_Questions_Solns.pdf

an informal, anonymous and very useful
survey of your learning experience

http://goo.gl/forms/AyC01bEk7g

EXCLUSIVE FOR TUESDAY EVENING SECTION

today's outline

➔ proof by cases

➔ review of proofs

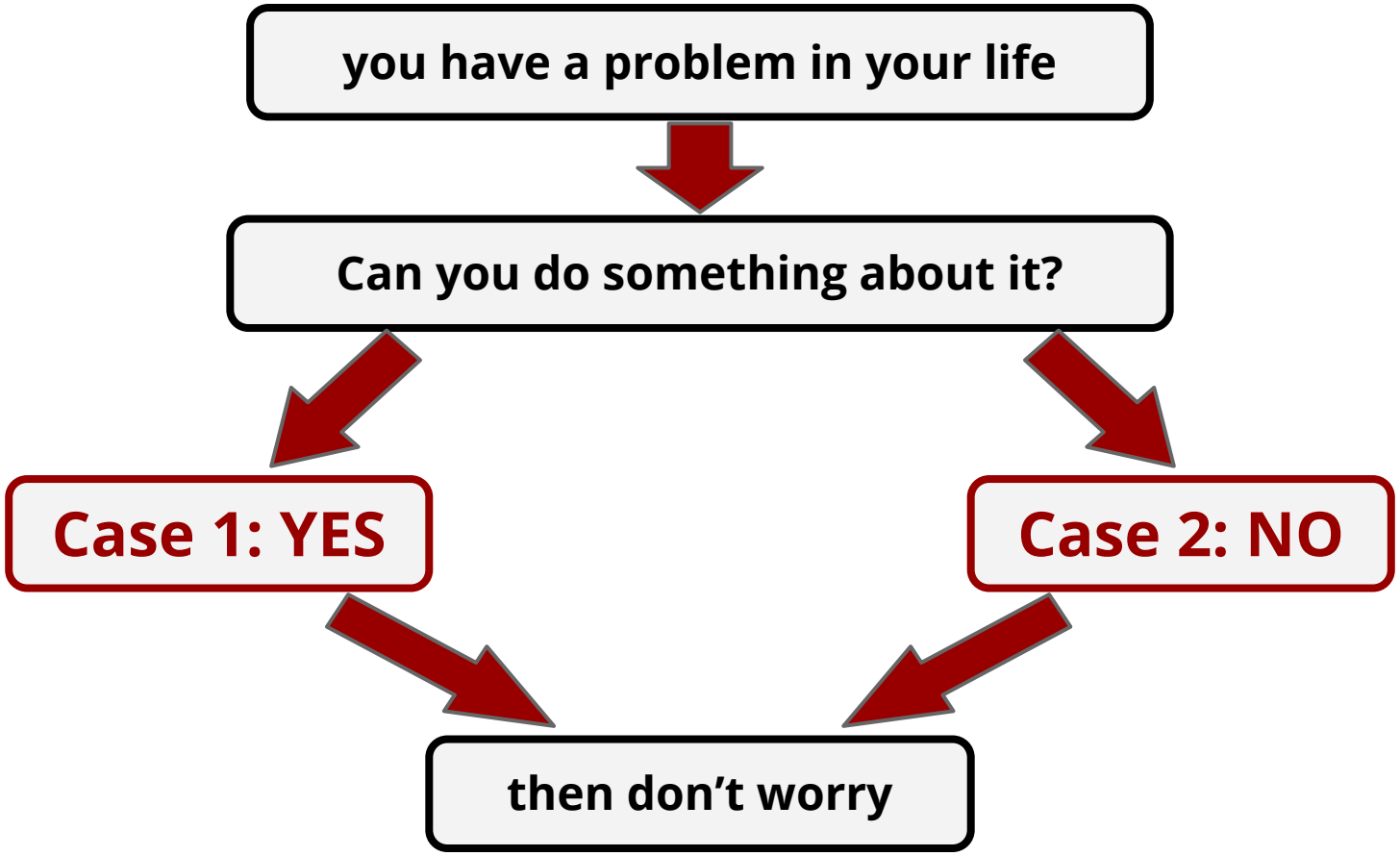
➔ algorithms

proof by cases

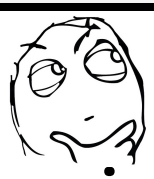
proof by cases

- ➔ **split** your argument into differences cases
- ➔ prove the conclusion **for each** case

Prove:
If you have a problem in your life, then don't worry.



What makes it a valid proof?

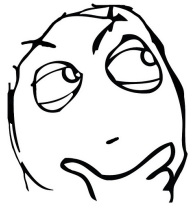


The union of the cases are covering ALL possibilities.

$$\forall n \in \mathbb{N}, n^2 + n \text{ is even}$$

thoughts

$$n^2 + n = n(n+1)$$



n **and** ***n+1*****, one of them has to be even, so their product must be even**

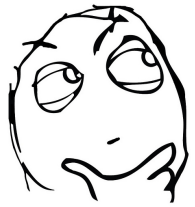
Case 1: n is even, then n is even.

Case 2: n is odd, then n+1 is even.

$$\forall n \in \mathbb{N}, n^2 + n \text{ is even}$$

another way of thinking

$$n^2 + n$$



use some results that we have proven before

Case 1: n is even => n^2 is even, their sum must be even

Case 2: n is odd => n^2 is odd, their sum must be even

Proof:

$\forall n \in \mathbb{N}, n^2 + n$ is even

assume $n \in \mathbb{N}$ # n is a generic natural number

then $(\exists k \in \mathbb{N}, n = 2k + 1) \vee (\exists k \in \mathbb{N}, n = 2k)$

n is either odd or even, only two possible cases

Case 1: assume $\exists k \in \mathbb{N}, n = 2k + 1$ # n is odd

then $n^2 + n = n(n+1) = (2k+1)(2k+2)$

$= 2[(2k + 1)(k + 1)]$

then $\exists k' \in \mathbb{N}, n^2 + n = 2k'$ # k' = (2k+1)(k+1)

Case 2: assume $\exists k \in \mathbb{N}, n = 2k$ # n is even

then $n^2 + n = n(n + 1) = 2k(2k + 1)$

$= 2[k(2k + 1)]$

then $\exists k' \in \mathbb{N}, n^2 + n = 2k'$ # k' = k(2k+1)

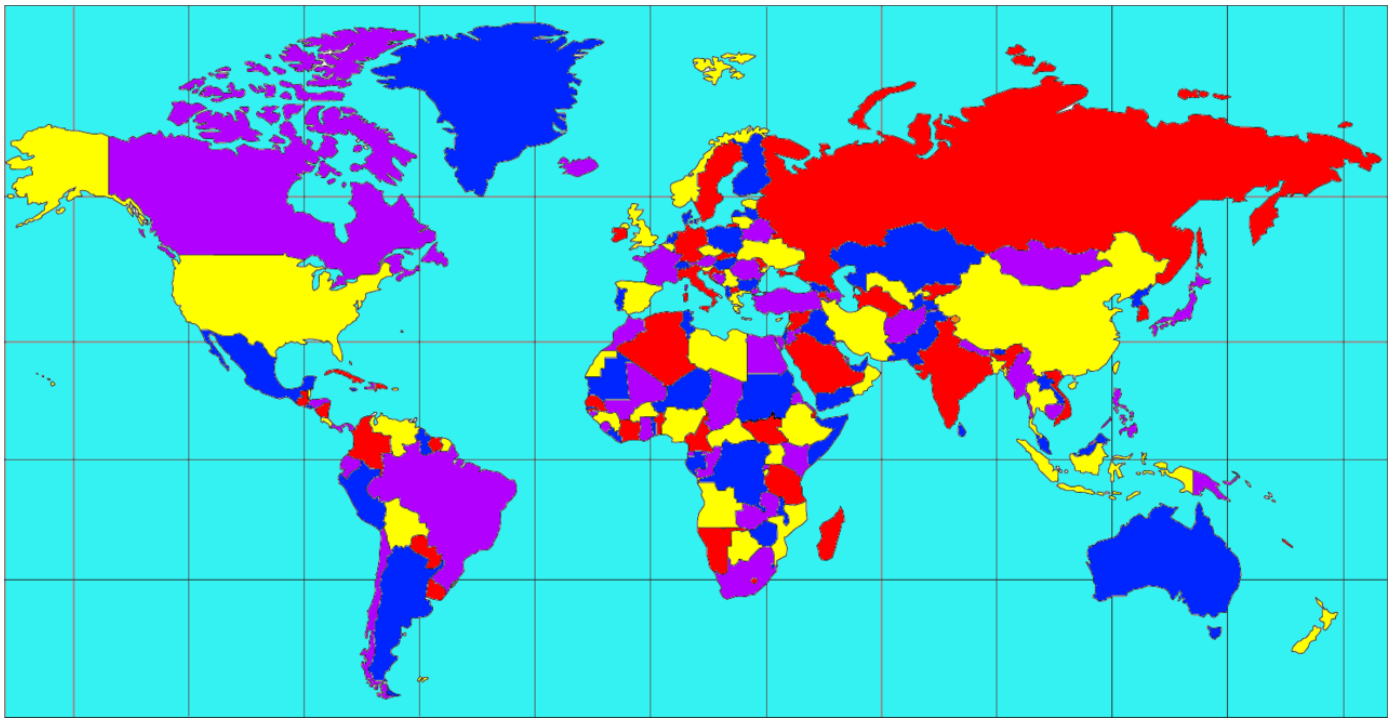
then $\forall n \in \mathbb{N}, \exists k \in \mathbb{N}, n^2 + n = 2k$ # introduce universal

then $\forall n \in \mathbb{N}, n^2 + n$ is even # definition of even

an extreme example of proof by cases

(just for fun)

Four colour theorem



***Four** colours are enough to colour any map (separation of a plane), so that no two adjacent regions have the same colour.*

First conjectured in 1852 by

Francis & Frederick Gurthrie, and their advisor

Augustus De Morgan

First proven in 1976 by

Kenneth Appel & Wolfgang Haken, their proof

➔ *reduces all possible counter-examples into **1936 cases***

➔ *just need to find a valid 4-colouring for each case*

➔ *but solving these cases **by hand** would take **forever***

➔ *so a **computer program** is used, which took about **1000 hours** to solve all cases*

➔ *the world's first **computer-assisted proof***

More story: http://en.wikipedia.org/wiki/Four_color_theorem

one more practice

Define predicate $T(n)$ by

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

Prove:

$$S1 : \forall n \in \mathbb{N}, T(n) \Rightarrow T(n^2)$$

Disprove:

$$S2 : \forall n \in \mathbb{N}, T(n^2) \Rightarrow T(n)$$

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

thoughts Prove: $S1 : \forall n \in \mathbb{N}, T(n) \Rightarrow T(n^2)$

$$n = 7i + 1$$

$$n^2 = (7i + 1)^2$$

$$= 49i^2 + 14i + 1$$

$$= 7(7i^2 + 2i) + 1$$

$$\text{let } k = 7i^2 + 2i$$

$$n^2 = 7k + 1$$

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

Proof: Prove: $S1 : \forall n \in \mathbb{N}, T(n) \Rightarrow T(n^2)$

assume $n \in \mathbb{N}$ *# n is a generic natural number*

assume $T(n)$, i.e., $\exists i \in \mathbb{N}, n = 7i + 1$

then $n^2 = (7i + 1)^2$ *# square both sides*

$= 49i^2 + 14i + 1$

$= 7(7i^2 + 2i) + 1$

then let $k = 7i^2 + 2i$

then $k \in \mathbb{N}$ *# i, 7, 2 are all natural numbers*

then $n^2 = 7k + 1$

then $T(n^2)$ *# definition of T(n)*

then $T(n) \Rightarrow T(n^2)$ *# introduce antecedent*

then $\forall n \in \mathbb{N}, T(n) \Rightarrow T(n^2)$ *# introduce universal*

Define predicate $T(n)$ by

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

Prove:

$$S1 : \forall n \in \mathbb{N}, T(n) \Rightarrow T(n^2)$$

Disprove:

$$S2 : \forall n \in \mathbb{N}, T(n^2) \Rightarrow T(n)$$

uniqueness (math prerequisite)

if ***m*** and ***n*** are natural numbers, with $n \neq 0$, then there is **exactly one** pair natural numbers ***(q, r)*** such that:

$$m = qn + r, \qquad n > r \geq 0$$

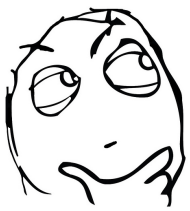
divide *m* by *n*, the quotient and remainder are unique

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

thoughts Disprove: $S2 : \forall n \in \mathbb{N}, T(n^2) \Rightarrow T(n)$

$$\neg S_2 : \quad \exists n \in \mathbb{N}, \quad T(n^2) \wedge \neg T(n)$$

just find a $n^2 = 7k + 1$, and $n \neq 7i + 1$



$1^2 = 7 \cdot 0 + 1$ but $1 = 7 \cdot 0 + 1$

$6^2 = 7 \cdot 5 + 1$ and $6 \neq 7i + 1$

use n = 6

$$\forall n \in \mathbb{N}, \quad T(n) \Leftrightarrow \exists i \in \mathbb{N}, n = 7i + 1$$

Disproof: Disprove: $S2 : \forall n \in \mathbb{N}, T(n^2) \Rightarrow T(n)$

pick $n = 6$

then $n \in \mathbb{N}$ *# 6 is a natural number*

then $n^2 = 36 = 7 \cdot 5 + 1$

then $T(n^2)$ *# definition of T(n)*

also $n = 6 = 7 \cdot 0 + 6$

then $\nexists i \in \mathbb{N}, n = 7i + 1$ *# uniqueness of remainder*

then $\neg T(n)$

then $\exists n \in \mathbb{N}, T(n^2) \wedge \neg T(n)$ *# introduce ∃*

then $\neg(\forall n \in \mathbb{N}, T(n^2) \Rightarrow T(n))$ *# negation*

a review of proof patterns

patterns of inference

what's known

---

what can be inferred

Two categories of inference rules

- ➔ ***introduction****: smaller statement => larger statement*
- ➔ ***elimination****: larger statement => smaller statement*

introduction rules

negation introduction

assume A

…

contradiction

¬A

assume there are finitely many prime numbers

…

contradiction

---

there are infinitely many prime numbers

conjunction introduction

A

B

A ∧ B

c is red

c is fast

---

c is red and fast

disjunction introduction

A

A ∨ B

B ∨ A

(nothing here)

A ∨ ¬A

(nothing)

c is red

c is red or fast

c is fast or red

it's a boy or a girl

implication introduction

assume A

...

B

A => B

(direct)

assume ¬B

...

¬A

A => B

(indirect, contrapositive)

equivalence introduction

$A \Rightarrow B$

$B \Rightarrow A$

$A \Leftrightarrow B$

n odd $\Rightarrow$ n^2 odd

n^2 odd $\Rightarrow$ n odd

---

n odd $\Leftrightarrow$ n^2 odd

universal introduction

assume $a \in D$

…

$P(a)$

$\forall\ x \in D,\ P(x)$

assume a generic car c

…

c is red

---

all cars are red

existential introduction

$P(a)$

$a \in D$

$\exists\ x \in D,\ P(x)$

c is red

c is a car

---

there exists a car that is red

elimination rules

negation elimination

$\neg\neg A$

A

"the car is not red" is false

---

the car is red

negation elimination

$$\frac{A \qquad \neg A}{\textbf{contradiction}}$$

$$\frac{\text{there are 51 balls} \qquad \text{there are not 51 balls}}{\text{contradiction!}}$$

conjunction elimination

$$\frac{A \wedge B}{\begin{array}{l}\mathbf{A} \\ \mathbf{B}\end{array}}$$

$$\frac{\text{the car is red and fast}}{\begin{array}{l}\text{the car is red} \\ \text{the car is fast}\end{array}}$$

disjunction elimination

$A \vee B$

$\neg A$

B

it's a boy or a girl

it's not a boy

---

it's a girl

$A \vee B$

$\neg B$

A

implication elimination

$$\frac{A \Rightarrow B \qquad A}{B}$$

$$\frac{\text{If you work hard, you get A+} \qquad \text{You work hard}}{\text{You get A+}}$$

implication elimination

$A \Rightarrow B$

$\neg B$

$\neg A$

If you work hard, you get A+

You don't get A+

You don't work hard

equivalence elimination

$$\frac{A \Leftrightarrow B}{\textbf{A => B} \qquad \textbf{B => A}}$$

$$\frac{n \text{ odd} <=> n^2 \text{ odd}}{n \text{ odd} => n^2 \text{ odd} \qquad n^2 \text{ odd} => n \text{ odd}}$$

universal elimination

$\forall\ x \in D, P(x)$

$a \in D$

P(a)

all cars are red

X is a car

X is red

existential elimination

$\exists\ x \in D,\ P(x)$

Let a $\in$ *D such that P(a)*

$\exists\ n \in N,\ n$ *divides 32*

Let m $\in$ N such that m divides 32

how to be good at proofs

➔ become familiar with these patterns, by **lots of practice**

➔ recognize these patterns in your proof, use the manipulation rules to get closer to your target

Fermat’s last theorem

(just for fun)

Fermat's last theorem

$$\text{no natural numbers } x, y, z \text{ satisfy}$$
$$x^n + y^n = z^n \quad \text{for } n > 2$$

First conjectured in 1637 by Pierre de Fermat

Proven by Andrew Wiles in 1994

http://www.cs.uleth.ca/~kaminski/esferm03.html

an extraordinary story

- *"a truly marvellous proof" by Fermat that's never written down because "this margin is too narrow"*
- *358 years of marathon to rediscover Fermat's proof*
- *frustrated King of Mathematics -- Leonhard Euler*
- *a woman who has to take on the identity of a man in order to conduct her research, makes breakthrough*
- *a suicidal man coincidentally saved by the problem leaves million-dollar prize for a proof*
- *a 10-year old boy's obsessive dream comes true*

the right emotion about finding a proof

Chapter 4

Algorithm Analysis and Asymptotic Notation

Computer scientists talk like…

"The worst-case runtime of bubble-sort is in $O(n^2)$."

"I can sort it in n log n time."

"That's too slow, make it linear-time."

"That problem cannot be solved in polynomial time."

compare two sorting algorithms

bubble sort

merge sort

demo at *http://www.sorting-algorithms.com/*

Observations

➔ **merge** is faster than **bubble**

➔ with larger input size, the advantage of **merge** over **bubble** becomes larger

compare two sorting algorithms

	20	40
bubble	8.6 sec	38.0 sec
merge	5.0 sec	11.2 sec

*when input size **grows** from 20 to 40…*

➔ *the **"running time"** of merge roughly doubled*

➔ *the **"running time"** of bubble roughly quadrupled*

what does "running time" really mean in computer science?

- ➔ It does **NOT** mean how many **seconds** are spent in running the algorithm.
- ➔ It means **the number of steps** that are taken by the algorithm.
- ➔ So, the running time is **independent of the hardware** on which you run the algorithm.
- ➔ It only depends on the algorithm itself.

You can run **bubble** on a super-computer and run **merge** on a mechanical watch, that has nothing to do with the fact that **merge** is a faster sorting algorithm than **bubble**.

describe algorithm running time

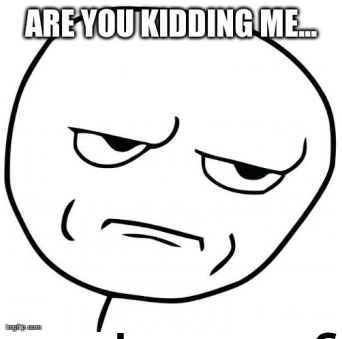
	20	40
bubble	***200 steps***	***800 steps***
merge	***120 steps***	***295 steps***

number of steps as a function of ***n***, the size of input

➔ the running time of bubble could be ***0.5n²*** *(steps)*

➔ the running time of merge could be ***n log n*** *(steps)*

but, we don't really care about the number of steps...



- ➔ what we really care: how the number of steps **grows** as the size of input **grows**
- ➔ we **don't care** about the **absolute** number of steps
- ➔ we care about: "*when input size **doubles**, the running time **quadruples***"
- ➔ so, ***0.5n²*** and ***700n²*** are **no different**!
- ➔ **constant factors do NOT matter!**

constant factor does not matter, when it comes to growth

$$T_1(n) = 0.5\,n^2 \qquad T_2(n) = 700\,n^2$$

$$\frac{T_1(2n)}{T_1(n)} = \frac{0.5\,(2n)^2}{0.5\,n^2} = \frac{2n^2}{0.5n^2} = 4$$

$$\frac{T_2(2n)}{T_2(n)} = \frac{700\,(2n)^2}{700\,n^2} = \frac{2800\,n^2}{700\,n^2} = 4$$

we care about large input sizes

- ➔ we don't need to study algorithms in order to sort **two** elements, because different algorithms make no difference
- ➔ we care about algorithm design when the input size ***n*** is very large
- ➔ so, n^2 and n^2+n+2 are no different, because when n is really large, $n+2$ is negligible compared to n^2
- ➔ ***only the highest-order term matters***

low-order terms don't matter

$$T_1(n) = n^2 \qquad T_2(n) = n^2 + n + 2$$

$$T_1(10000) = 100,000,000$$

$$T_2(10000) = 100,010,002$$

$$\text{difference} \approx 0.01\%$$

summary of running time

- ➔ we count the number of steps
- ➔ constant factors don't matter
- ➔ only the highest-order term matters

so, the followings functions are ***of the same class***

$$n^2 \qquad 2n^2 + 3n \qquad \frac{n^2}{165} + 1130n + 3.14159$$

that class could be called $O(n^2)$

O(n²) is an asymptotic notation

O(f(n)) *is the asymptotic upper-bound*

➔ *the set of functions that grows* ***no faster*** *than* ***f(n)***

➔ *for example, when we say*

$$5n^2 + 3n + 1 \text{ is in } O(n^2)$$

we mean

$5n^2+3n+1$ grows no faster than n^2, asymptotically

other things to be studied later

*Ω(**f(n)**): the asymptotic lower-bound*

*Θ(**f(n)**): the asymptotic tight-bound*

more precise definitions of O, Ω, and Θ

a high-level look at asymptotic notations

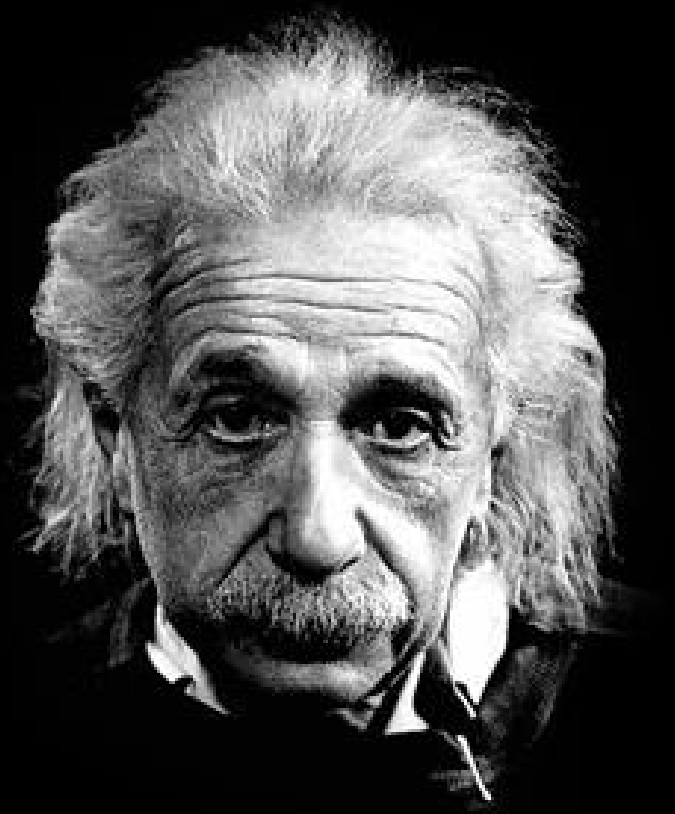
*It is a **simplification** of the "real" running time*

➔ *it does not tell the whole story about how fast a program runs in real life.*

- *in real-world applications, constant factor matters! hardware matters! implementation matters!*

➔ *this simplification makes possible the development of the whole **theory of computational complexity**.*

- ***HUGE idea!***



"Make everything as simple as possible, but not simpler."

—Albert Einstein

a quick note

In CSC165, sometimes we use **asymptotic notations** such as ***O(n²)***, and sometimes, when we want to be more accurate, we use the **exact forms**, such as ***3n² + 2n***

It depends on what the question asks.

analyse the time complexity of a program

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0
2.  while i < len(A):
3.      if A[i] == x:
4.          return i
5.      i = i + 1
6.  return -1
```

What's the running time of this program?

Can't say yet, it depends on the input (A, x)

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                    🙂
2.  while i < len(A):        🙂 🙂
3.      if A[i] == x:        🙂 🙂
4.          return i             🙂
5.      i = i + 1            🙂
6.  return -1
```

Count time complexity

(# of lines of code executed)

`LS([2, 4, 6, 8], 4)`

$t_{LS}([2, 4, 6, 8], 4) = 7$

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                    🙂
2.  while i < len(A):        🙂 🙂 🙂
3.      if A[i] == x:        🙂 🙂 🙂
4.          return i             🙂
5.      i = i + 1            🙂 🙂
6.  return -1
```

Count time complexity

`LS([2, 4, 6, 8], 6)`

$t_{LS}([2, 4, 6, 8], 6) = 10$

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                          🙂
2.  while i < len(A):              🙂 🙂 🙂
3.      if A[i] == x:              🙂 🙂 🙂
4.          return i                   🙂
5.      i = i + 1                  🙂 🙂
6.  return -1
```

what is the running time of **LS(A, x)**

if the first index where ***x*** is found is ***k***

i.e., **A[k] == x**

$$t_{LS}(A, x) = 1 + 3(k+1)$$
$$= 3k + 4$$

$$t_{LS}([2, 4, 6, 8], 6) = 10$$

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                    🙂
2.  while i < len(A):        🙂 🙂 🙂 🙂 🙂
3.      if A[i] == x:        🙂 🙂 🙂 🙂
4.          return i
5.      i = i + 1            🙂 🙂 🙂 🙂
6.  return -1                          🙂
```

Count time complexity

LS([2, 4, 6, 8], 99)

$t_{LS}([2, 4, 6, 8], 99) = 15$

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                          🙂
2.  while i < len(A):              🙂 🙂 🙂 🙂 🙂
3.      if A[i] == x:              🙂 🙂 🙂 🙂
4.          return i
5.      i = i + 1                  🙂 🙂 🙂 🙂
6.  return -1                                  🙂
```

what is the running time of **LS(A, x)**

if ***x*** is not in **A** at all

let **n** be the size of **A**

$t_{LS}([2, 4, 6, 8], 99) = 15$

$t_{LS}(A, x) = 1 + 3n + 2$

$= 3n + 3$

takeaway

➔ program running time varies with inputs

➔ among inputs of a given size, there is a **worst case** in which the running time is the longest

worst-case time complexity

$t_P(x)$: running time of program P with input x

the worst-case time complexity of P
with input $x \in I$ of size n

$$W_P(n) = \max\{\, t_P(x) \mid x \in I \wedge \text{size}(x) = n \,\}$$

linear search

```
def LS(A, x):
""" Return index i, x == A[i].
    Otherwise, return -1 """
1.  i = 0                          🙂
2.  while i < len(A):              🙂 🙂 🙂 🙂 🙂
3.      if A[i] == x:              🙂 🙂 🙂 🙂
4.          return i
5.      i = i + 1                  🙂 🙂 🙂 🙂
6.  return -1                                  🙂
```

what is the **worst-case** running time of **LS(A, x)** given that **len(A) == n ?**

$$W_{LS}(n) = 1 + 3n + 2$$
$$= 3n + 3$$

$t_{LS}([2, 4, 6, 8], 99) = 15$

the case where
x is not in **A** at all

worst-case: performance in the worst situation, what we typically do in CSC165, and in CSC236

best-case: performance in the best situation, not very interesting, rarely studied

average-case: the expected performance under random inputs following certain probability distribution, will study in CSC263

next week

➔ more on asymptotic notations

➔ more algorithm analyses