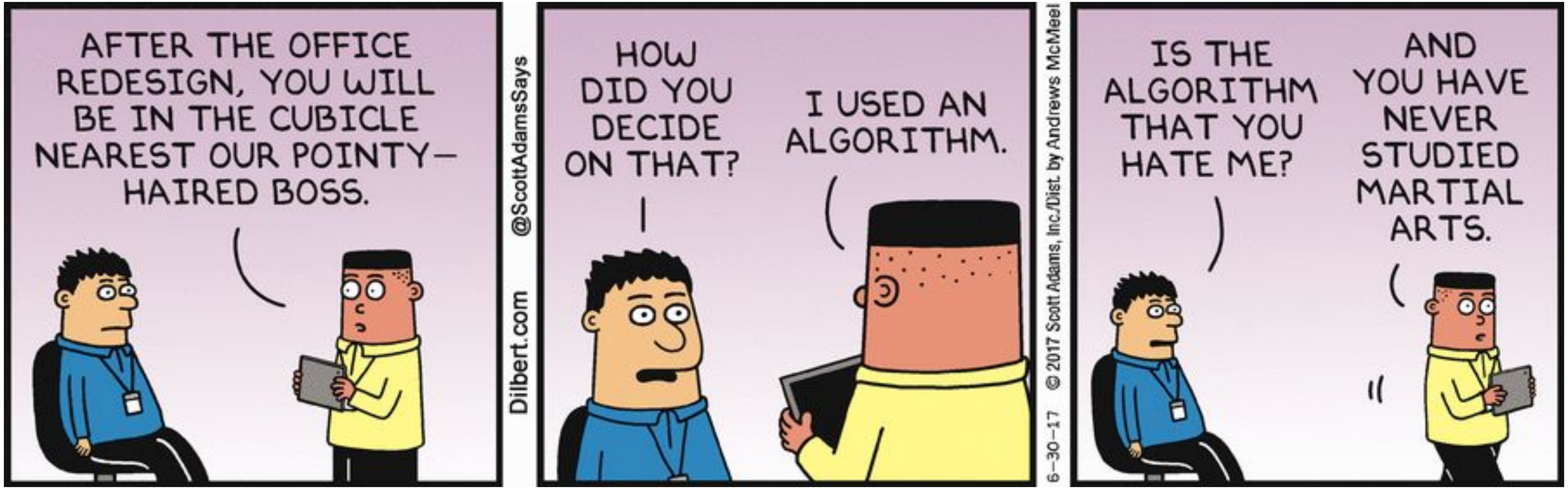
Announcements and reminders

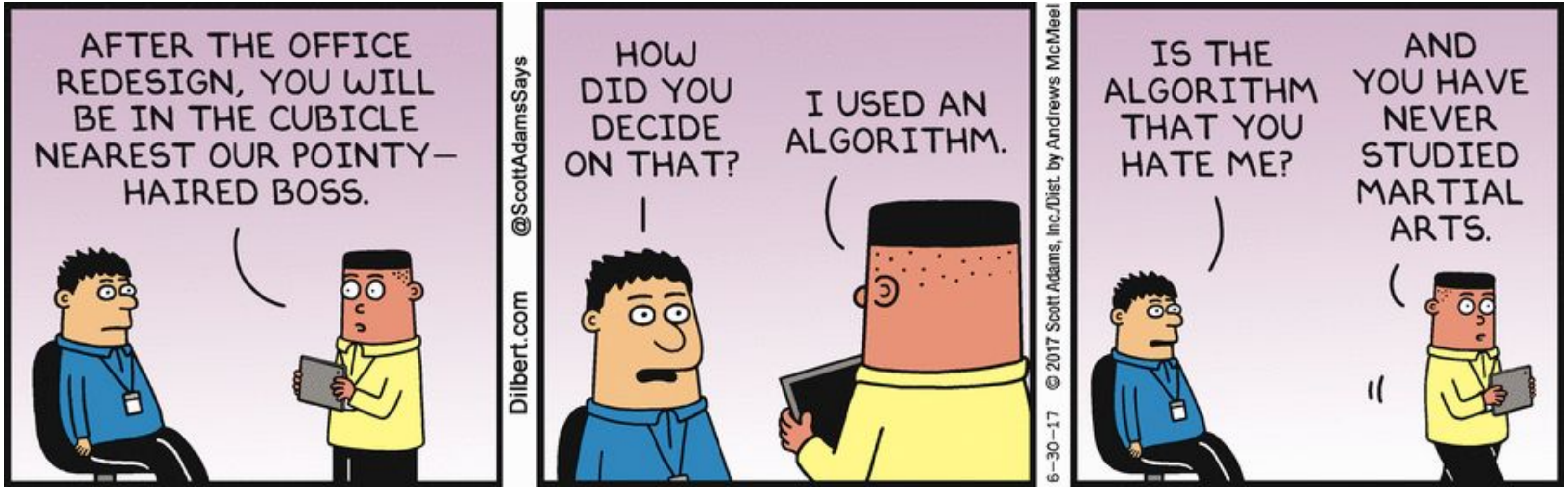
Quizlet 6: extended to tomorrow @ 8a

- Homework 6 (written) is posted and is due Friday 12 Oct at 12 PM Noon
- The CU final exam schedule is up. You must take your final exam during your scheduled final exam time.

  Tony's section: 7:30 - 10 PM, Sunday 16 Dec
  Rachel's section: 1:30 - 4 PM, Wednesday 19 Dec



Lecture 16: Algorithms



What did we do last time?

- We learned about ***sequences…***
  - What are they?
  - Special kinds of sequences (geometric, harmonic, arithmetic, Fibonacci)
  - Nifty ways to define sequences (recursion vs explicit formula)

Today:

- We talk ***algorithms***

Algorithms

Definition: An **algorithm** is a finite sequence of precise instructions for performing a computation or solving a problem.

A motivating example: Find the maximum (largest) element in a finite sequence.

For example: Given the sequence {1, 22, 5, 8, 1, 2, 13}, return 22.

Solution in words: Guidance: ALWAYS sketch ~~it~~ out in words/pseudocode First!

1. Initialize `max` to the first element in the sequence.
2. Compare this to the second element in the sequence. If the second element is larger, reset `max` to the second element.
3. Repeat the previous step for all of the remaining elements in the sequence.
4. Stop when there are no more terms.
5. `max` is now set to the largest element.

Algorithms

Definition: An **algorithm** is a finite sequence of precise instructions for performing a computation or solving a problem.

A motivating example: Find the maximum (largest) element in a finite sequence.

For example: Given the sequence {1, 22, 5, 8, 1, 2, 13}, return 22.

Solution in pseudocode:

```
def findMax( {a[0], a[1], a[2], a[3], ...} ):
    max = a[0]
    for i in 1 to length(a) :
        if(a[i] > max) max = a[i]
    return (max)
```

Algorithms

Properties of algorithms:

- **Input:** An algorithm has inputs from a particular set
- **Output:** From each set of inputs, the algorithm produces outputs from a particular set. The *outputs* are the solution to the problem the algorithm tackles.
- **Definiteness:** The steps in the algorithm are defined precisely.
- **Correctness:** The algorithm should produce the correct output for each set of inputs.
- **Finiteness:** An algorithm should terminate in finite time. test w/ smaller input
- **Generality:** An algorithm should be applicable for all problems of the desired form.

writing code in a modular way

Algorithms

in #'s: 12321 is also a palindrome

123321

FYOG: A ***palindrome*** is a string that reads the same forward and backward (for example, "stressed desserts"). Describe an algorithm for determining whether a string of *n* characters is a palindrome.

def clever_fcn_name_here(string):

check if first / last char are same

check if second / penultimate char are same

loop here?

⋮

Need a check: if even, -- . .

if odd, . - - - .

Searching algorithms

Task: Find the location of a specific element within a list, or determine that the element is not contained in the list.

Applications: Check if a word is in a dictionary.
Check if a particular behavior was observed.

1 2 3 4 5

Example: Find 15 in the sequence {1, 2, 7, 11, 15, 19, 30, 31}
(should return "5")
(index)

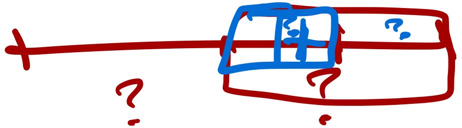
We will look at two algorithms: (1) ***linear search***
(2) ***binary search***

Searching algorithms

In English:

- Linear search: start from the first element and check each element in the list until you find what you're looking for

- Binary search: leverage the fact that our list is *ordered* (or sorted); split the list into two halfs and see which half would contain what we're looking for; then do this again, and again, and again…



Linear search

In pseudocode:

{a1, a2, a3, … aN}

search this to find x

```
def LinearSearch(x, a):
    i = 0
    while i≤len(a) and x!=a[i]:
        i = i+1
    if i ≤ len(a):
        location = i
    else:
        location = -1
    return location
```

FYOG: Code up two versions of the linear search: one using a “for” loop and another using a “while” one.

FYOG: Which is faster?

Natural question: How fast is this?

Binary search

Task: Find the location of a specific element within an *ordered* list, or determine that the element is not contained in the list.

Example: Find x = 15 in the sequence a = {1, 2, 7, 11, 15, 19, 30, 31}

Binary search leverages the fact that the list is sorted.

Sep. into 2 halves: [1, 2, 7, 11] & [15, 19, 30, 31]

→ which should our elt. be in? → compare $x = 15$ & max(left list)

Since $x > 11$, we focus on right list

⋮

REPEAT

Binary search

TRANSITIONING TO PSEUDOCODE

Task: Find the location of a specific element within an *ordered* list, or determine that the element is not contained in the list.

Example: Find $x = 15$ in the sequence $a = \{1, 2, 7, 11, 15, 19, 30, 31\}$

Binary search leverages the fact that the list is sorted.

Binary search

Task: Find the location of a specific element within an *ordered* list, or determine that the element is not contained in the list.

Example: Find $x = 15$ in the sequence $a = \{1, 2, 7, 11, 15, 19, 30, 31\}$

Binary search leverages the fact that the list is sorted.

1. Break the list into 2 halves.

   1, 2, 7, 11   15, 19, 30, 31

2. Compare x to the largest item in the left list. Since 11 < 15, zoom in on the right list.

   15, 19   30, 31

3. Compare x to the largest item in the left list. Since 15 < 19, zoom in on the left list.

   $\boxed{15}$ 19

← once we have a list of length 1, can check if that's our x

Binary search

In pseudocode:

{a1, a2, a3, … aN}

```
def BinarySearch(x, a):
    location = -1
    left = 1
    right = N
    while left < right:
        i_large_left = ⌊(left+right)/2⌋ ✓
        if x > a[i_large_left]:
            left = i_large_left + 1 *
        else:
            right = i_large_left *
            if x==a[left]: location = left
    return location
```

trimming/zooming in on the appropriate half

Natural question: How fast is this?

Searching algorithms

Super important question: Which of the two searching algorithms seems faster?

Caution! What do we really mean by “faster”?

- Which is faster in the *best-case* scenario?
- Which is faster in the *worst-case* scenario?
- Which is faster *on average*?
- What *is* the best-/worst-case scenario?

We will tackle these questions later this week.

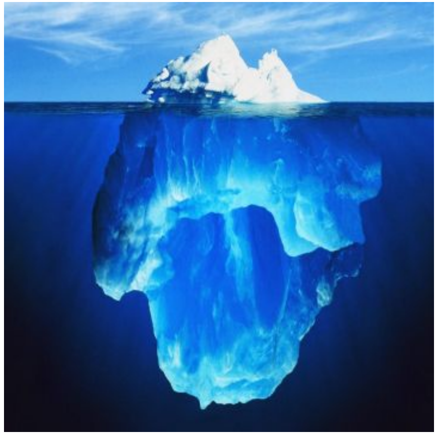
Sorting algorithms

Task: Given some unordered list of elements, organize them according to some notion of "order" (e.g., increasing numbers, alphabetizing, etc…)

Applications: Sort mail by location along route.
Alphabetizing CSCI 2824 exams by student name.

Goals: Sometimes the whole point is just to sort the lists.

- Examples: to do a binary search, or find duplicates in a list



Sorting algorithms

Task: Given some unordered list of elements, organize them according to some notion of “order” (e.g., increasing numbers, alphabetizing, etc…)

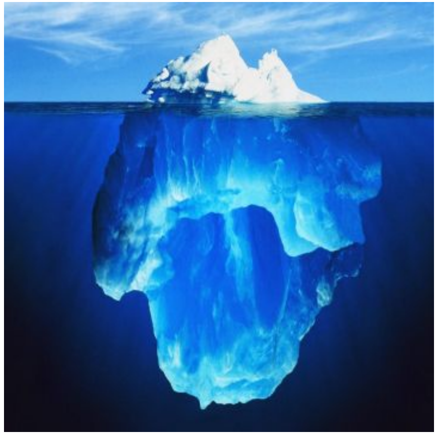
Applications: Sort mail by location along route.
Alphabetizing CSCI 2824 exams by student name.

Goals: Sometimes the whole point is just to sort the lists.

- Examples: to do a binary search, or find duplicates in a list

We will look at two algorithms: (1) ***bubble sort***
(2) ***insert sort***

- But note that there is a LOT of effort/computational power spent on sorting, and this is just the tip of the iceberg.



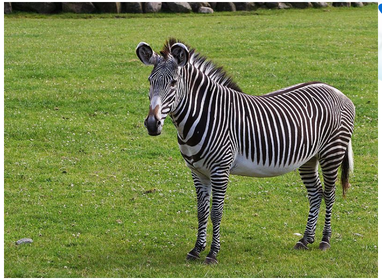
Sorting algorithms

In English:

- Bubble sort: make passes through the list; whenever you encounter two elements that are out of order, swap them.
  ⇒ lower elements (9999, Zebra, etc…) sink to the bottom,
  and higher elements (1, Aardvark, etc…) *bubble* to the top

- Insert sort: successively *insert* the next unsorted element into the already-sorted front end of the list.

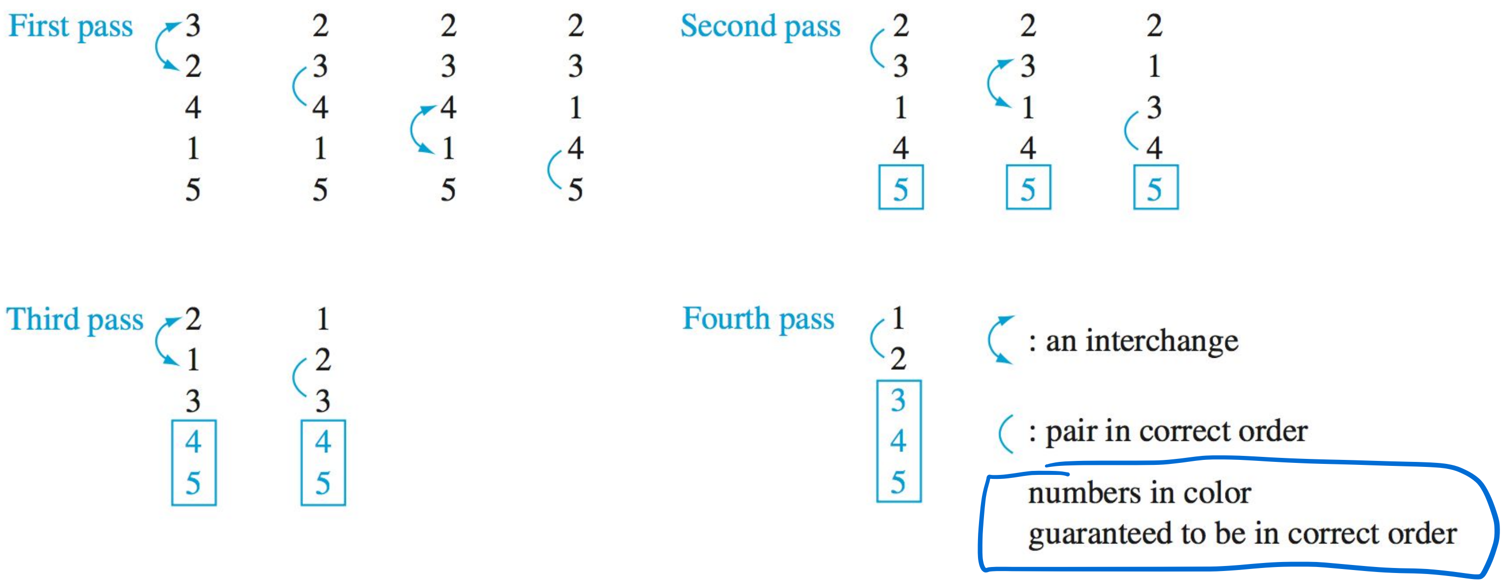
- These are both fairly naive. You'll study many more sophisticated sorting algorithms in … well… *Algorithms.*



Bubble sort

Make passes through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

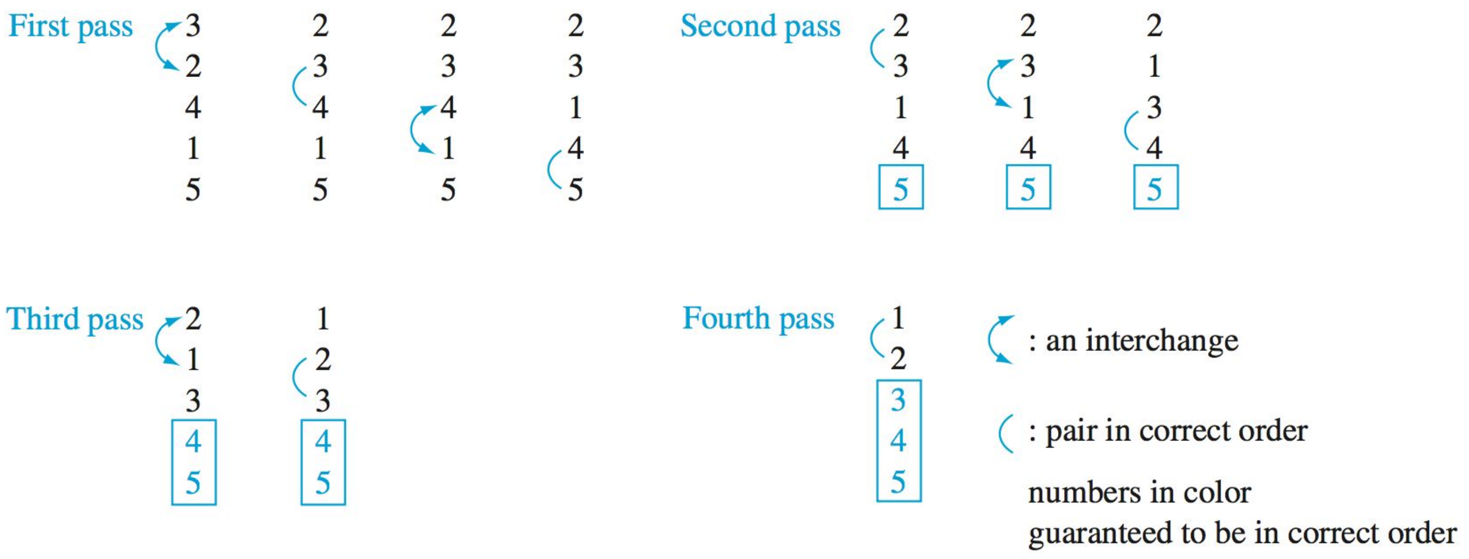
Example: S'pose we want to sort the list {3, 2, 4, 1, 5}.



Bubble sort

{a0, a1, a2, a3, … aN}

```
def bubbleSort (x, a):
    for i in range(0, N-1):
        for j in range(0, N-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```



Bubble sort

{a0, a1, a2, a3, … aN}

```
def bubbleSort (x, a):
   for i in range(0, N-1):
         for j in range(0, N-i):
               if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```

Bonus material: [bubble sort demonstration through a folk dance!](#)
(you'll also notice there are other algorithms demonstrated through dance linked from that video too, if you're interested)

Insert sort

Example: S’pose we want to sort the list {3, 2, 4, 1, 5}.

Insert sort

Example: S'pose we want to sort the list {3, 2, 4, 1, 5}.

- 1st element: {3} -- definitely in order -- proceed with {3}
- 2nd element: {3, 2} -- out of order -- put where it belongs -- proceed with {2, 3}
- 3rd element: {2, 3, 4} -- in order! -- proceed with {2, 3, 4}

  prev. sorted
- 4th element: {2, 3, 4, 1} -- out of order -- put where it belongs -- proceed with {1, 2, 3, 4}
- 5th element: {1, 2, 3, 4, 5} -- in order! -- proceed with {1, 2, 3, 4, 5}
- No elements left -- stop.

FYOG: go watch

Sorting algorithms

FYOG: How many comparisons does the bubble sort need to make to sort the list

$\{1, 2, 3, 4, \ldots, N\}$?

FYOG: Write pseudocode for a modified bubble sort that stops early if the previous pass required no swaps of elements.

FYOG: How many comparisons does the insert sort need to make to sort the list

$\{1, 2, 3, 4, \ldots, N\}$?

FYOG: How many comparisons do the bubble sort and insert sort need to make to sort the list

$\{N, N\text{-}1, N\text{-}2, \ldots , 3, 2, 1\}$?

Greedy algorithms

HERE

Many algorithms are designed to solve ***optimization problems***.

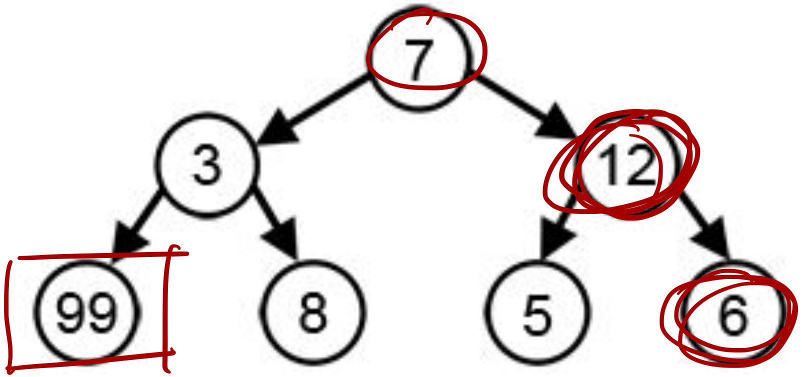
Goal: To find a solution that maximizes or minimizes some ***objective function***.

Applications: Find a route between two cities that minimizes distance (or time travelled, or elevation gain, …).
Encode a message using the fewest bits possible.

<u>Greedy algorithms</u> select the best choice at each step.

Note: they are *not* guaranteed to find an optimal solution. You must check once a solution is found.

Greedy algorithms



Greedy algorithms select the best choice at each step.

Note: they are *not* guaranteed to find an optimal solution. You must check once a solution is found.

Greedy algorithms

25¢ 10¢

Task: Consider making *n* cents change, using quarters, dimes, nickels and pennies, using the fewest total number of coins.

5¢ 1¢

Greedy strategy: At each step, choose the largest denomination coin possible to add to the pile, so as not to exceed *n* cents.

Example: S'pose we want to make change for 67 cents.

1. Select a quarter (25 cents) → 42¢
2. Select another quarter (50 cents) -25 / 17
3. Select a dime (60 cents)
4. Select a nickel (65 cents)
5. Select a penny (66 cents)
6. Select another penny (67 cents)

Sequence of change:

[25, 25, 10, 5, 1, 1]

{c[0], c[1], c[2], …, c[N]} = coin denominations, eg...

```
def greedy (amt, c):
    total = amt
    d = []              # initialize, to count coins of each denomination
    n_thiscoin = 0      # initialize, to count the current coin
    for thiscoin in range(0, len(c)):
        while (total ≥ c[thiscoin]):
            n_thiscoin = n_thiscoin + 1
            total = total - c[thiscoin]   # add another one of this coin
        d.append(n_thiscoin)
        n_thiscoin = 0                    # reset counter to 0
    return (d)
```

Greedy algorithms

Fun Fact: If we have **quarters**, **dimes**, **nickels** and **pennies** available, then this algorithm is optimal.

Fun? Fact: If we only have **quarters**, **dimes** and **pennies** available (no nickels), then this algorithm is ***not*** optimal.

Example: If we wanted to make change for 30 cents, using no nickels, then we would:

1. take 1 quarter
2. take 5 pennies

… But it would be ***optimal*** to take 3 dimes instead.

Algorithms

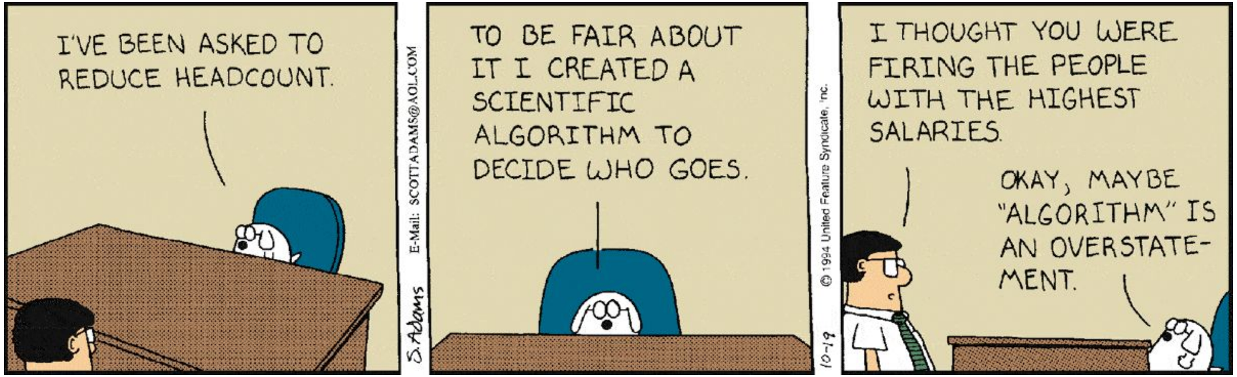
Recap:

- We learned about **sorting**, **searching** and **greedy** algorithms
- We started to think about algorithm **optimality** and **complexity**

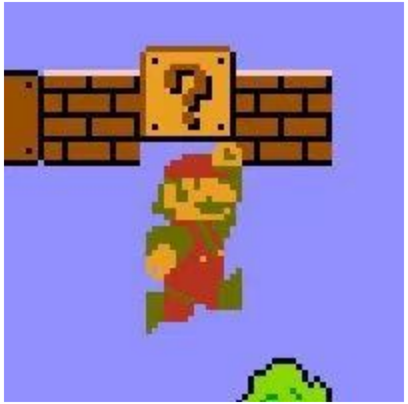
Next time:

- A deeper dive into ***algorithm complexity!***
- This algorithm is nice and simple:

(note though that you need to ***sort*** first!)



Bonus material!



Bonus! More algorithms

Decent numbers: A **decent number** is a number that satisfies these three rules:

1. The number only consists of 3’s and 5’s.
2. The number of 3’s is divisible by 5.
3. The number of 5’s is divisible by 3.

(FYOG) Task: Given a number of digits, *N*, find the largest Decent Number with *N* digits. If no such number exists, return -1.

Examples:

- The largest 3-digit decent number is 555 (also the only one)
- The largest 5-digit decent number is 33333 (also the only one)
- The largest 8-digit decent number is 55533333 (there are a few…)
- There are no 1- or 2-digit decent numbers (so return -1)

Bonus! More algorithms

(FYOG) Task: Given a number of digits, *N*, find the largest Decent Number with *N* digits. If no such number exists, return -1.

Hint:

```
def Decent(N_digit):
    return_value = -1
    for n_block3 in range(0, [what is max. # of blocks of 3s?]):
        [how many digits are left?]
        [how many blocks of 5s can we make?]
    …
    return (return_value)
```