CSC148 Week 9

Larry Zhang

Announcement

More Announcements

- Assignment 2 out. You may work in groups of 1-3 students.
- Test 2 on next Friday (Mar 16). **No lecture** on that day.
  - The test starts at 5:30
  - The test is 50 minutes.
  - Closed-book, closed-notes, no calculator. The test will have an API sheet at the back.
  - Use pen to write your test. If you use pencil, then we will not accept a re-mark request.
  - Coverage: what we have studied up to and including linked lists. That is, weeks 5-8, all relevant labs, all relevant handouts, A1, and E2.
  - Rooms:
    - utorid starts with A-J, DV2072.
    - utorid starts with K-S, DV2074.
    - utorid starts with T-Z, DV2082.

Read Dan's post on the discussion board for more details.

Today’s Task: Problem Solving

Anagrams

Anagrams

Our definition: Two words are anagrams if the letters of one dictionary word can be rearranged to be the letters of the other dictionary word.

For example: **horse** and **shore** are anagrams.

Are these anagrams?

- bill, lib
  - No, not the same letters
- computer, ueotpmrc
  - No, not a dictionary word
- sports, torts
  - No, not the same letters
- traffic, traffic
  - Yes, a word is the anagram of itself

The problem to solve

Given a word **w**, generate **all anagrams** of that word.

You're given a list of English dictionary word list.

x is an anagram of **w** if **x** is a rearrangement of **w** and is in the word list.

- it's not just any permutation of w

Example: when `w = "team"`, the output list is something like the following:

`["team", "meat", "mate", "tame"]`

Now, let's develop a computer program that can solve this problem.

What approach/algorithm should we take?

Approach #1: Naive Algorithm

Approach #1

An anagram of w is a permutation of w that is in the dictionary word list…

So we just go through **all permutations** of w and check each one if it is in the word list.

For example:

- word: dog
- Six permutations of dog: **dog, dgo, odg, ogd, gdo, god**
- check each of the above if it is in the word list

Let’s implement this solution.

Approach #1 Demo

`find_anagrams.py`

- `generate_perms()`
- `perm_find_anagrams()`

`time_algs.py`

Approach #1 Analysis

How many “steps” are we spending at least?

Let **n** be the length of the word, and **m** be the size of the dictionary word list

How many permutations are there?

- n! (factorial)

For each permutation, we need to linear search through a list of size **m**, so the total runtime is **at least n! x m**.

- e.g, the word is “mississauga” and the word list has 20,000 words.
- # of steps ⩾ 11! x 20,000 = 39,916,800 x 20,000 = 798,336,000,000

This is going to take a long, long time to finish, even on a computer.

Approach #2: Signature

Approach #2: Signature

Thinking about the efficiency of Approach #1:

- The **number of permutations** is huge when the length of the word is a bit large, and the vast majority of those permutations are not even in the dictionary word list. Going through them is a waste of time.
- For each permutation, we check through the whole dictionary list. Even more waste of time.

Better idea: Going through the dictionary word list once, for each word we check if it is a permutation of the input word w.

If we quickly check **if string A is a permutation of string B**, then this approach would be much faster than Approach #1.

Approach #2: Signature

So how do we compare if string A is a permutation of string B?

Sort A and B, then compare **sorted(A)** and **sorted(B)**

A is a permutation of B if and only if **sorted(A) == sorted(B)**

In other words, all permutations of the same word share the same and unique “signature”, which can be used to identify them.

This check is very fast because A and B are typically short (just English words) so it mostly takes a small constant number of steps.

Approach #2 Analysis

How many “steps” are we spending using Approach #2.

Again, let **n** be the length of the word, and **m** be the size of the dictionary word list.

We are just going through the **m** words in the dictionary **once** and spend some constant time on each one.

So the total work is essentially **m** steps (times a small constant for calculating each signature)

- This is much smaller than the steps in Approach #1 (**n! x m**)
- The amount of work does NOT change when **n** increases.

Approach 2 Demo

`find_anagrams.py`

- `signature()`
- `sig_find_anagrams()`

`time_algs.py`

Takeaway

Often, the same problem can be solved by multiple different algorithms.

Some algorithms are “fast” and some others are “slow”.

The difference between “fast” and “slow” can be the difference between solving and not solving the problem!

However, in order to precisely analyze how “fast” an algorithm is, we need some formal definition of “fast” and “slow”. This will be our next topic in CSC148.

Runtime Analysis

(Friday)

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 **"runtime"** of merge roughly doubled*

➔ *the **"runtime"** of bubble roughly quadrupled*

what does “runtime” 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

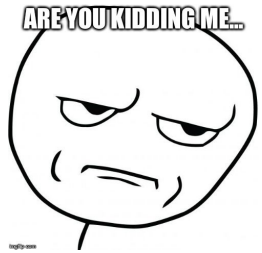
	20	40
bubble	***200 steps***	***800 steps***
merge	***120 steps***	***295 steps***

T(n): number of steps as a function of ***n***, the **size of input**

➔ the runtime of bubble could be ***T(n) = 0.5n²*** *(steps)*

➔ the runtime of merge could be ***T(n) = 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

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

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

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

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 runtime

- ➔ 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(n²)

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 later

a high-level look at “big-Oh”

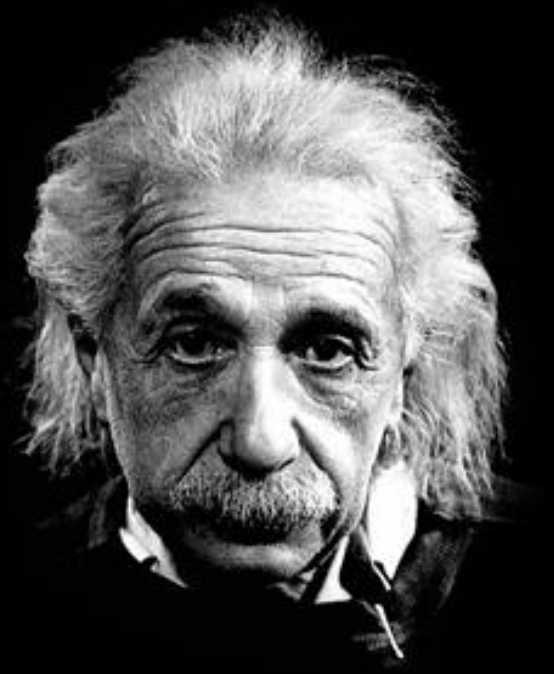
It is a **simplification** of the “real” running time

➔ 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

Example

Analyze the Runtime of Segment-Sum

The problem

- ➔ input: **L**, a list of numbers
- ➔ output: the maximum sum over segments/slices of **L**

L = [-2, -3, 4, -1, 6, -3]

max = 4 + (-1) + 6 = 9

Code

```
 1 def max_segment_sum(L):
 2     '''(list of int) -> int
 3     Return maximum segment sum of L.
 4     '''
 5     max_so_far = 0
 6     for lower in range(len(L)):
 7         for upper in range(lower, len(L)):
 8             sum = 0
 9             for i in range(lower, upper + 1):
10                 sum = sum + L[i]  ### count this line
11             max_so_far = max(max_so_far, sum)
12     return max_so_far
```

How it works:
Enumerate all possible slices, compute the sum for each slice, and keep the max.

What's the runtime?

```
1  def max_segment_sum(L):
2      '''(list of int) -> int
3      Return maximum segment sum of L.
4      '''
5      max_so_far = 0
6      for lower in range(len(L)):
7          for upper in range(lower, len(L)):
8              sum = 0
9              for i in range(lower, upper + 1):
10                 sum = sum + L[i]  ### count this line
11             max_so_far = max(max_so_far, sum)
12     return max_so_far
```

How many times is **`sum = sum + L[i]`** executed?

- the "lower" loop has at most n iterations, the "upper" loop has at most n iterations, and the "i" loop has at most n iterations
- Line 10 runs at most n^3 times

We may also count Line 8 and Line 11 as steps

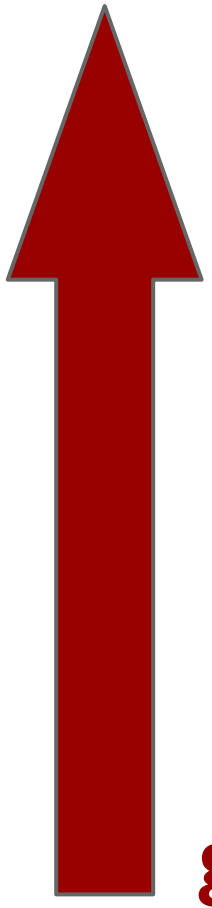
- each of them runs at most n^2 times

So an upper bound on the number of steps is $n^3 + 2n^2$

It is in $O(n^3)$, **a.k.a. cubic time** (counting Line 8 and 11 didn't really matter).

growth rate ranking of typical functions

$$f(n) = n^n$$
$$f(n) = 2^n$$
$$f(n) = n^3$$
$$f(n) = n^2$$
$$f(n) = n \log n$$
$$f(n) = n$$
$$f(n) = \sqrt{n}$$
$$f(n) = \log n$$
$$f(n) = 1$$



grow fast

grow slowly

Good upper-bound

So Segment-Sum's runtime is in $O(n^3)$, i.e., its growth rate is upper-bounded by n^3.

Is it in $O(n^5)$, O(2^n)?

- Yes and yes, but it is a looser and worse upper-bound.

What we want most of the time:

The lowest (tightest) upper bound.

It is the most accurate and informative upper-bound.

Up next

Formal definition of big-Oh.