Big O Notation

210CT 2015/16

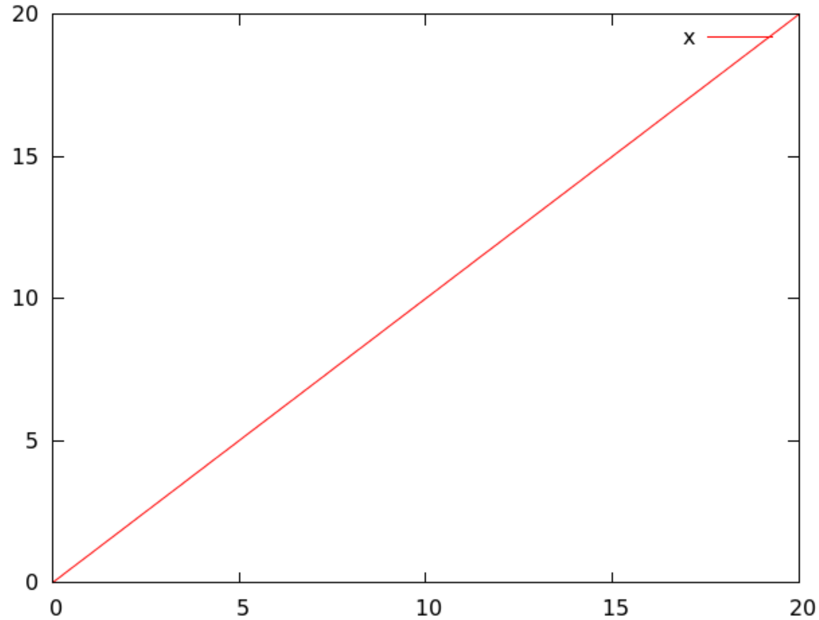
Block 4

1 Plotting functions

- We can show functions that have one parameter on an (x, y) plot.
- Along the x-axis, we show the input
- The y-axis shows the result of the function with that input
- So for the function x^2, we would expect that for the point on the x-axis with value 2, we should find the point on the y-axis to be 4.
- You will see **why** we are doing this soon…

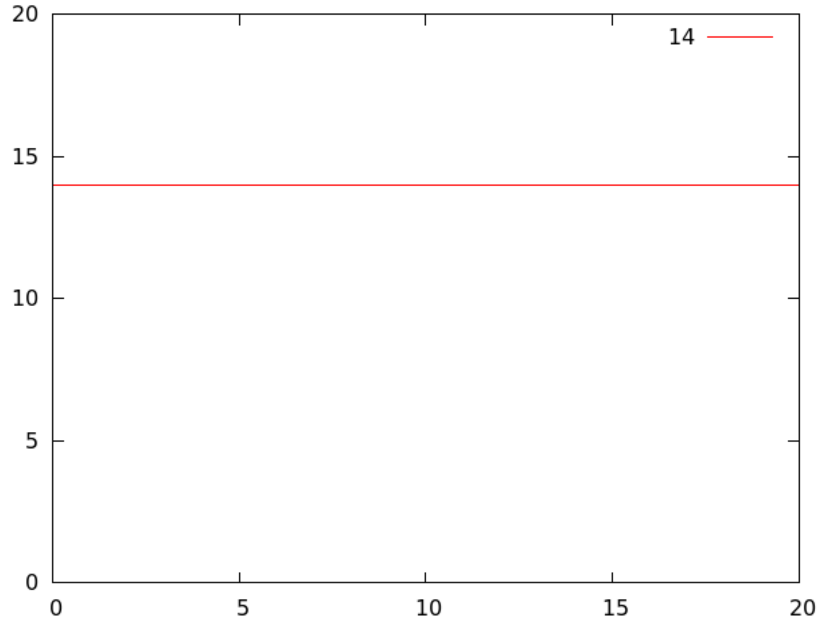
1.1 f(x)=x

- $f(x) = x$
- Nice and simple, grows **linearly**



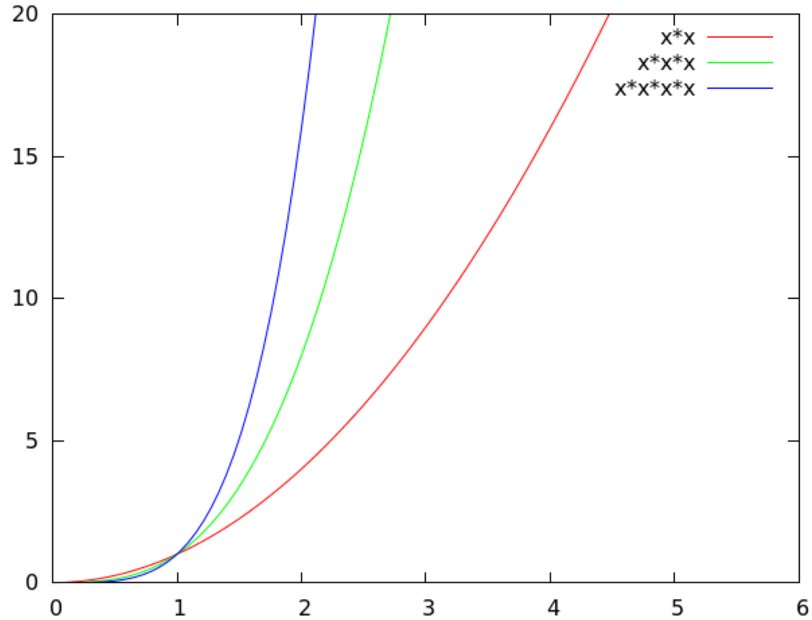
1.2 f(x)=k

- $f(x) = k$
- Where k is a **constant**, such as 14



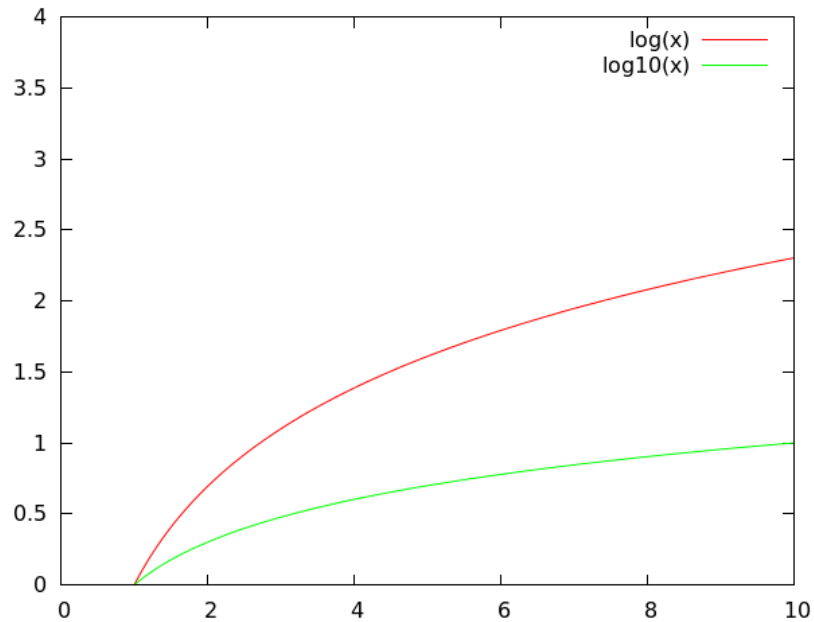
1.3 f(x)=x²

- $f(x) = x^2$ (or x^3, or x^4, ...)
- Growing **exponentially**



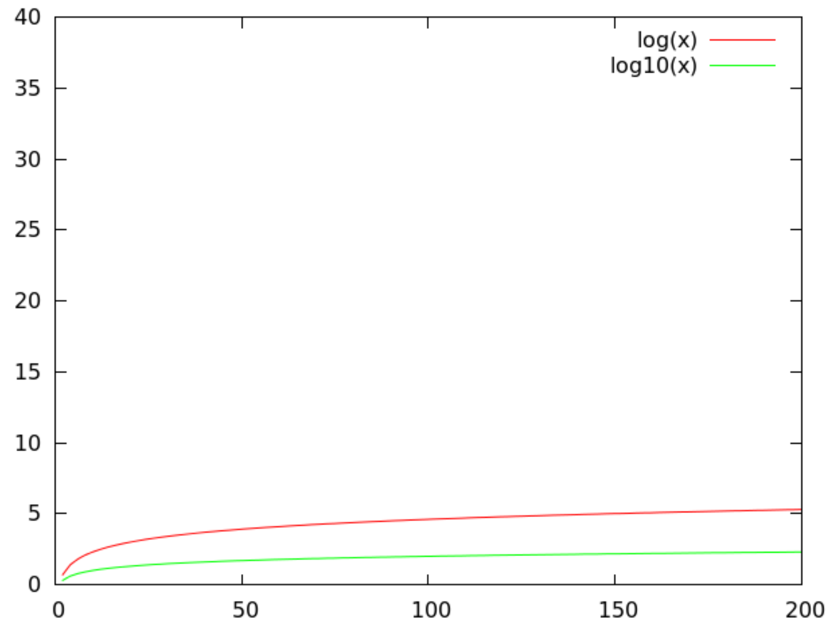
1.4 Log(x)

- $f(x) = \log(x)$ (which is $\log_e(x)$, or $\log_{10}(x)$)
- Growth slows. Diminishing returns

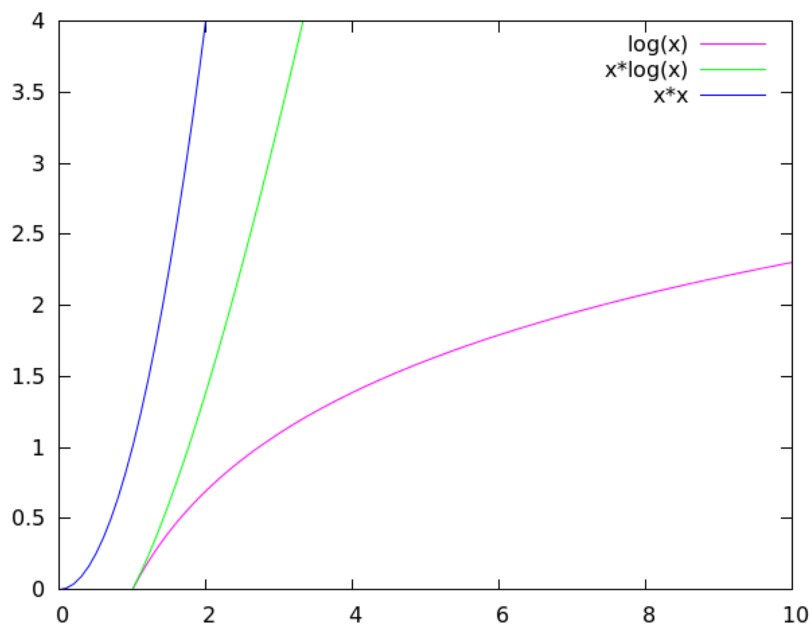


1.5 Is it base e or base 10?

- Usually, we probably actually mean base 2 in this module, but **it doesn't matter**.
- Why? We will be using these to describe how algorithms perform and on anything but tiny numbers, there's not enough difference to care about.
- Look at that graph again, but over a longer period…



1.6 $\mathbf{x}\log(\mathbf{x})$



2 Big O Notation

- Makes it easier to analyse algorithm efficiency
- Focuses on essential features without worrying about details
  - Real run-time is heavily influenced by characteristics of the machine running the algorithm
  - When using this notation, we are ignoring the actual run-time in favour of a measure of efficiency not dependent on a particular platform
- We will use it assuming a single processor, random access machine
- We will assume standard operations, arithmetic, logical, loading, storing and branching

2.1 Relating to Pseudo-code

- We assume a single line of pseudo-code takes constant time
  - We also assume that the line which controls a loop takes constant time, though the loop itself may not
  - Data types are assumed to have finite magnitude and precision

2.2 Runtime Analysis

- Depends on several things
  - Input size
  - Input content
  - Upper bounds

  - Many algorithms can exit when the correct conditions are met; this maybe on the first run, or it may have to search the whole data structure
  - For analysis, we always consider the worst case

2.3 The Notation

- $O(n)$: This represents an algorithm whose performance will drop linearly with the size of the input.
  - For example, with an input size of 20 it will take twice as long as with an input of 10
- $O(n^2)$: This represents an algorithm whose performance will drop exponentially with input size.
  - For example, with an input size of 20 it will take 300 times as long as with an input of 10
- $O(\log n)$: This represents an algorithm whose performance will drop as the log of the input.

3 Example

```
INSERTION-SORT(A)
    for j ← 1 to length[A]
        key ← A[j]
        i←j-1
        while i > 0 and A[i] > key
            A[i+1] ← A[i]
            i←i+1
        A[i+1] ← key
```

3.1 With n Notation Added

- Here n is equal to the length of A

```
INSERTION-SORT(A)
    for j ← 1 to length[A]          (n times)
        key ← A[j]                  (n times)
        i←j-1                       (n times)
        while i > 0 and A[i] > key  (n*n times)
            A[i+1] ← A[i]           (n*n times)
            i←i+1                   (n*n times)
        A[i+1] ← key                (n times)
```

3.2 With n = 3

- The length of the array A = 3 and each instruction takes 1 unit of time to complete

```
INSERTION-SORT(A)
    for j ← 1 to length[A]         (3 times)
        key ← A[j]                 (3 times)
        i←j-1                      (3 times)
        while i > 0 and A[i] > key  (3*3=9 times)
            A[i+1] ← A[i]          (3*3=9 times)
            i←i+1                  (3*3=9 times)
        A[i+1] ← key               (3 times)
```

- With n = 10
- The length of the array A = 10

```
INSERTION-SORT(A)
    for j ← 1 to length[A]         (10 times)
        key ← A[j]                 (10 times)
        i←j-1                      (10 times)
        while i > 0 and A[i] > key  (10*10=100 times)
            A[i+1] ← A[i]          (10*10=100 times)
            i←i+1                  (10*10=100 times)
        A[i+1] ← key               (10 times)
```

3.3 What have we learnt so far?

- The most expensive parts of the algorithm are the loops
- Nested loops get exponentially more expensive
- So one loop executes n times, two nested loops execute n^2 times, three nested loops execute n^3 times, etc.

4 A bit more formally...

- Big O Notation is concerned mainly with the size of arrays and how often the algorithm needs to iterate through them
- Always look at worst case
- Used for comparing "growth" - how does input size affect run-time?

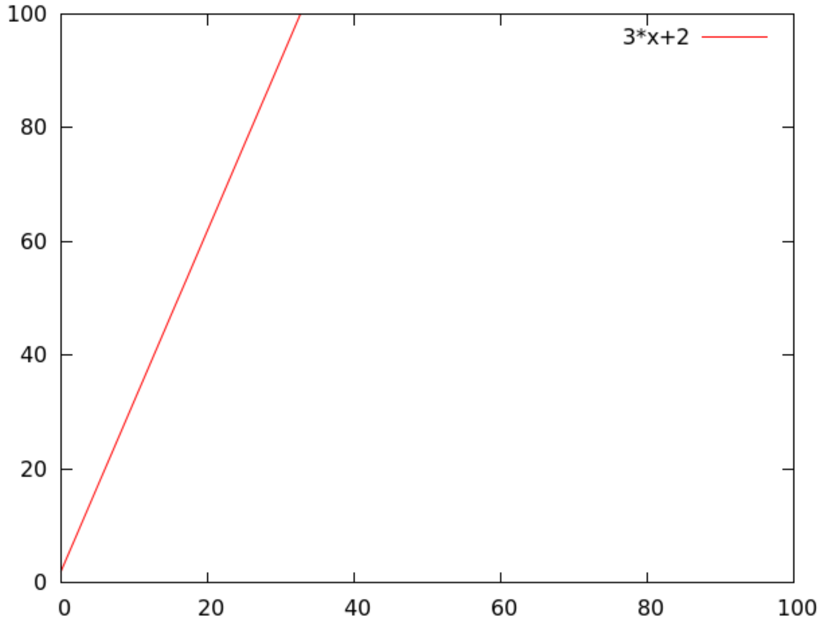
4.1 Example: max

- Find the largest item in a sequence
- Pseudocode

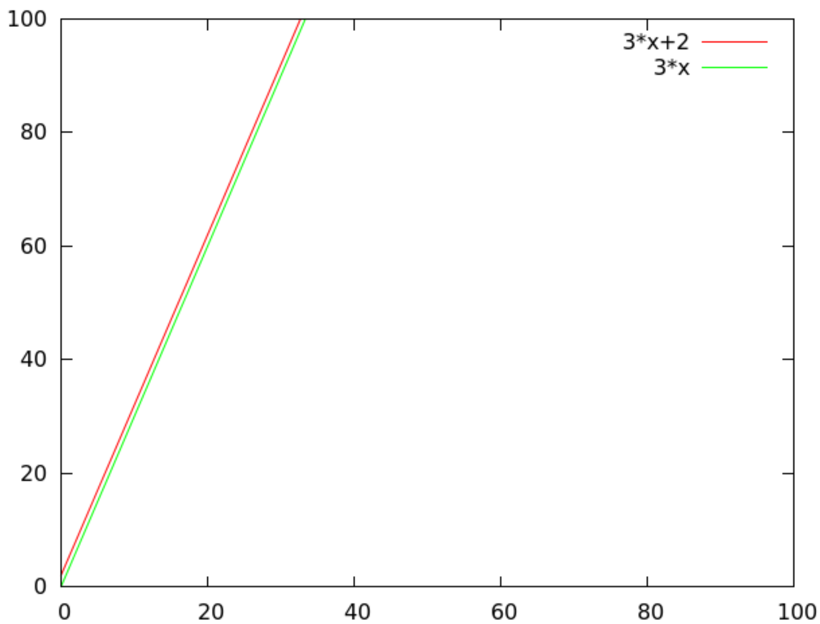
```
MAX(list)
    max ← list[0]
    for i ← list[1] .. list[length(list)-1]
        if i > max
            max ← i
    return max
```

```
        MAX(list)
( 1)    max ← list[0]
( n)    for i ← list[1] .. list[length(list)-1]
( n)        if i > max
(n?)            max ← i
( 1)    return max
```

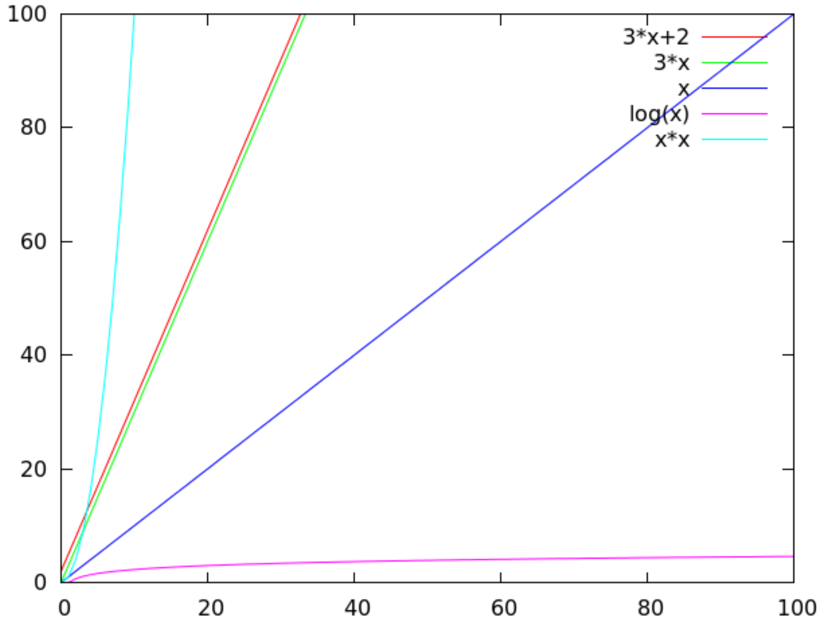
- So, something like:
  - $1 + n + n + mn + 1$
  - Not sure how many times we find a larger number so, let's say all of the time
  - becomes $1 + n + n + n + 1 = 3n + 2$



Turns out the constant makes very little difference...



... and compared to the other curves, the multiplier is moot, too, because it doesn't change the shape of the curve...



- So, we worked out $3n + 2$, but we would just write it in big-O notation as $O(n)$
- And what we're really saying is that there is some multiplier, m, and some constant, k, such that the actual running time will always be less than $mn + k$
  - Which we know is true, because we had m and k earlier
  - We sometimes call this the **bounds** or **upper limit** of the run-time
  - Also referred to as **asymptotic analysis** when people want to sound really clever
- We ignore all constants and multipliers not related to input size, as we've seen
  - But we also ignore lower order terms, so if we had $n + n^2$, we'd just write it as $O(n^2)$

5 Run-time analysis example from week 1

- Input: Two strings of characters

- Output: True if the strings are anagrams on one another; False if they are not
- A problem with a yes/no (or true/false) output is referred as a decision problem
- We will consider two different algorithms to solve this problem

5.1 Algorithm 1

```
ANAGRAM(string1, string2)
( 1)  if string1 length != string2 length
(  )    return False
( n)  for all letters i in string1
( n)    matched <- false
(nn)    for all letters j in string2
(nn)      if j is not marked and i = j
(nn)        mark j
(nn)        matched <- True
(nn)        break
( n)    if matched = false
( n)      return False
( 1)  return True
```

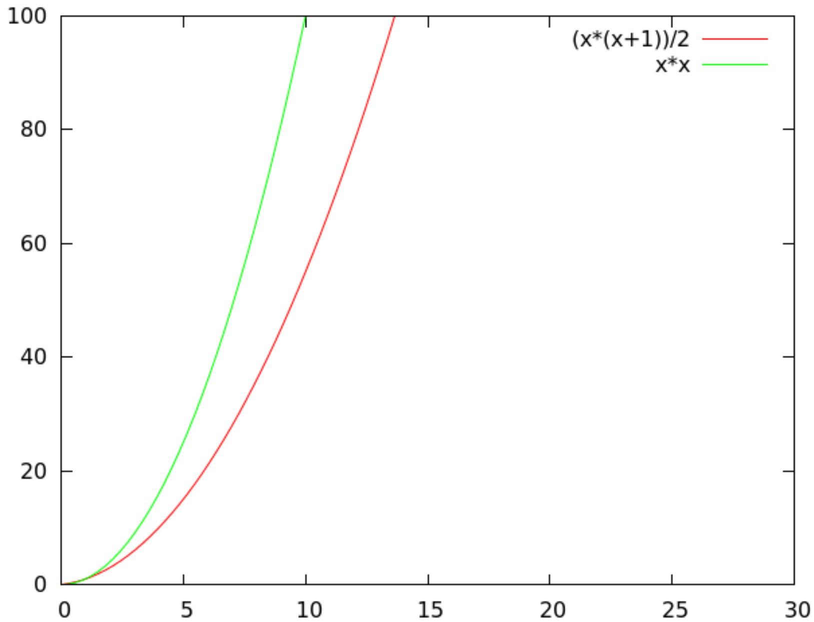
- Thanks to Theocharis Ledakis for spotting a typo
- So, $4n+5n^2+2$, removing constants, multipliers and all but the highest-order term gives: $O(n^2)$

5.2 Analysing another way

- Two nested loops, each the size of the strings' length (call this length n)
- Iterates along the first string n times
- Each time it then iterates through the second string, stopping when it finds a matching character
- Each character is matched once and only once
- The if statement executes 1+2+...+n times
  - Therefore we have roughly $\frac{n(n+1)}{2}$ steps

        * Thanks to Zsolt Ban for pointing out that the subtraction should be an addition

- The $+1$ is a constant and $\frac{?}{2}$ is a multiplier (0.5, see?) so we get...
  - $O(n^2)$

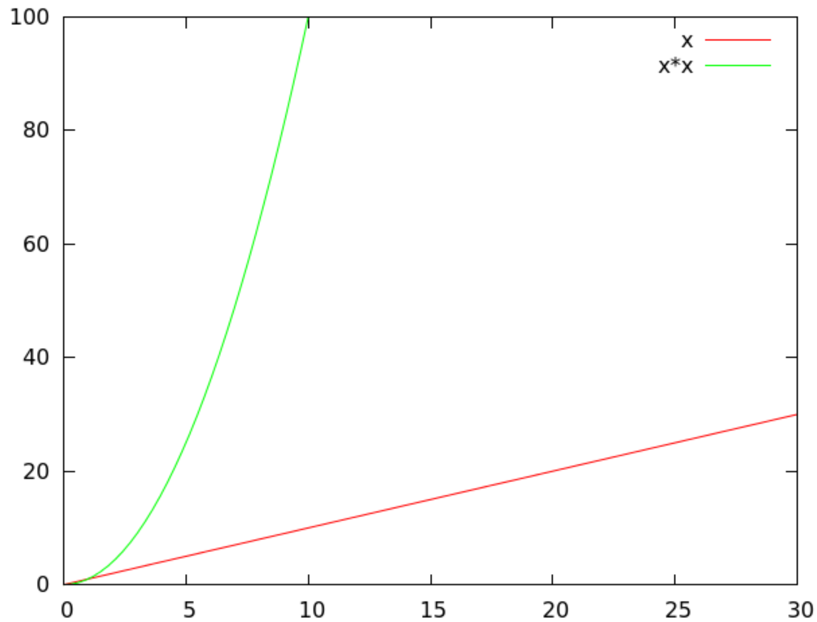


5.3 Algorithm 2

```
ANAGRAM(string1, string2)
( 1)  if string1 length != string2 length
( 1)    return No
( 1)  character count array <- 0
( n)  for all letters i in string1
( n)    increment the number of occurrences of i #increment: add 1 to
( n)  for all letters j in string2
( n)    decrement the number of occurrences of j #decrement: subtract 1 from
(26)  for all integers k in the array
(26)    if k != 0
(26)      return No
( 1)  return Yes
```

- $4n + 4 + 3 \times 26$, removing constants and multipliers gives...
  - $O(n)$

5.4 Comparison



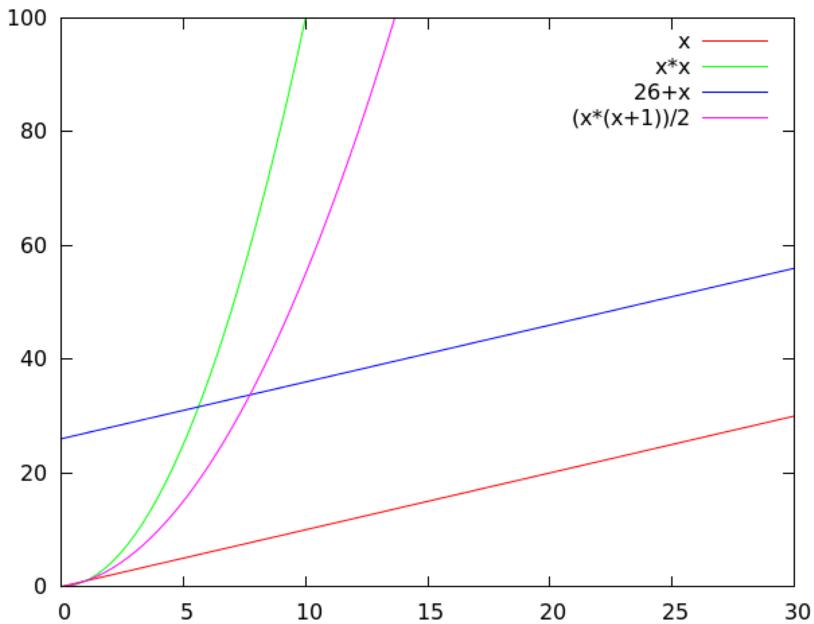
- But remember before, we found it wasn't always that one algorithm was faster, it depended on input size...

5.5 Comparison

- If the length of the strings are small, e.g. 5
  - 1st algorithm $\frac{5(6)}{2} = \frac{30}{2} = 15$ steps
  - 2nd algorithm $26 + 5 + 5 = 36$
- The 1st algorithm is faster
- But, if the length of the strings is large e.g. 200
  - 1st algorithm $\frac{200(201)}{2} = \frac{40200}{2} = 20100$ steps
  - 2nd algorithm $26 + 200 + 200 = 426$ steps

5.6 Comparison

So we do need to consider those constants and multipliers when we talk about small inputs...



5.7 Conclusion

- Different algorithms can have dramatically different performance
- The performance difference often depends greatly on the size of the data set
- Choosing the right algorithm for the job at hand is important!
- But on the whole, lower-order big-O terms means faster algorithms

6 More examples

6.1 O(1)

- Doesn't matter how long the input is
- Example, determining if a number is odd, we refer to the length of the number as n, but it doesn't affect the time it takes to calculate

```
IS_ODD(num)
( 1)    if num%2 = 0
( 1)        return False
( 1)    else
( 1)        return True
```

- $4 = 1 \times m$ where $m = 4$, so we can call it $O(1)$ by removing the multiplier
  - Or maybe we say $4 = 1 + k$ where $k = 4$ and remove the constant, but either way we're just saying it has constant time

6.2 Log n

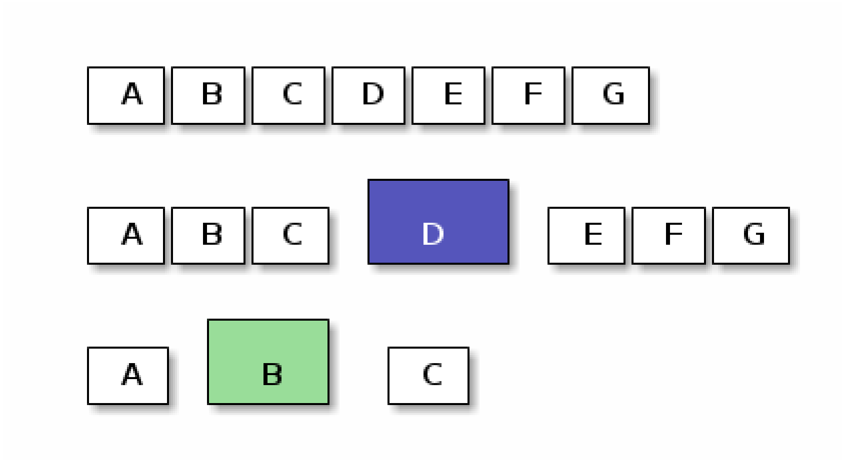
- Divide and conquer algorithms have a Big O Notation of O(log n)
- E.g. Searching a sorted list by comparing value to find to halfway point
- If value is less than, search lower half of list, else, search higher half
- Performance is the log of n, as the size of the problem grows, the number of divisions required grows only slowly

6.3 Example: Divide and conquer search (binary search)

- Start with a sorted list
- Randomly select a value near the middle
- Compare the value we are searching for to the split point
- If it's less we only need to search lower half
- Repeat until value found or cannot split anymore

6.4 Example run

- Looking for B…



6.5 Therefore

- If n is 16, number of divisions is 4

$$16/2 = 8 \quad (1)$$
$$8/2 = 4 \quad (2)$$
$$4/2 = 2 \quad (3)$$
$$2/2 = 1 \quad (4)$$

- If n is 128, number of divisions is 7

$$128/2 = 64 \quad (5)$$
$$64/2 = 32 \quad (6)$$
$$32/2 = 16 \quad (7)$$
$$16/2 = 8 \quad (8)$$
$$8/2 = 4 \quad (9)$$
$$4/2 = 2 \quad (10)$$
$$2/2 = 1 \quad (11)$$

7 Homework

7.1 Pre-homework

Same as before, you should be able to do these programming exercises without much effort. If not, get more practice.

- Write a function that takes two numbers, k and l, as parameters and returns the result of: $\frac{4^l+k}{3\times k^3+l}$
- Write a function call *mean* that returns the mean value of a sequence of numbers.

7.2 Actual Homework

1. Look back at the linear search and duplicate finder from the previous block. Describe the run-time bounds of these algorithms using Big O notation.

7.3 Alternate Homework

- In addition to the normal homework task, write a function that takes four parameters representing the constant and multiplier of two linearly growing (as in $O(m \times n + k)$) functions and determines the critical value of n (which should be an integer) at which the relative run-time of the two algorithms switches. That is, at which input size is algorithm A slower than B and at which is B slower than A? Use an iterative approach rather than solving the equations.

Emacs 23.3.1 (Org mode 8.0.3)