LECTURE 5

Algorithms

An algorithm is a finite set of precise instructions for solving a problem.

Ex: Problem: Given a list of n numbers, find the largest.

Steps

1. Set a temporary variable to the first number. (call it 't')
2. Compare the next number in the list to 't', if the next number is larger then set t to this number.
3. Repeat step 2 for the remaining numbers in the list.
4. Report the value of t.

Ex: Problem: Given an array of n numbers, return the index (position) of the number x.

Steps

1. Look at first element, if it is equal to x return 1.
2. Look at next element, if $= x$ then return that index.
3. Repeat (2) until we reach index n.
4. Return "Not Found".

Ex: A slightly more challenging problem.

Problem: We are given a list of elements that can be ordered (eg. numbers), we wish to re-arrange this list so that the elements are in non-decreasing order.
(why this terminology? $x_1 \leq x_2 \leq x_3 \cdots$).

This is called SORTING

How can we sort a list?

Bubblesort

The bubblesort algorithm is a simple sorting algorithm. It orders the list by successively comparing pairs of elements and swapping them if they are out of order.

Pseudocode

procedure Bubblesort $(a_1, a_2 a_3 \ldots a_n)$

for $i \leftarrow 1$ to $n-1$ do

for $j \leftarrow 1$ to $n-i$ do

if $a_j > a_{j+1}$ swap a_j and a_{j+1}

$\{a_1 \ldots a_n$ is in increasing order$\}$

5, 3, 7, 9, 1

iter $i=1$ $(j=1$ to $4)$

$j=1$: 3, 5, 7, 9, 1

$j=2$: 3, 5, 7, 9, 1 (ok)

$j=3$: 3, 5, 7, 9, 1 (ok)

$j=4$: 3, 5, 7, 1, 9

3, 5, 7, 1 | 9 sorted

iter $i=2$ $(j=1$ to $3)$

$j=1$: 3, 5, 7, 1, 9 (ok)

$j=2$: 3, 5, 7, 1, 9 (ok)

$j=3$: 3, 5, 1, 7, 9

iter $i=3$ $(j=1$ to $2)$

$j=1$: 3, 5, 1, 7, 9 (ok)

$j=2$: 3, 1, 5, 7, 9

iter $i=4$ $(j=1$ to $1)$

$j=1, 3, 5, 7, 9$

SORTED.

Question: How long will this take.

Answer: It depends (OS, hardware, implementation details.).

Are there other, possibly faster, ways to sort?

One option is INSERTION SORT.

Insertion Sort

- Scans the list from left to right looking for an element that is out of order.
- When it finds such an element it looks for where the element should go and places it there, but first we need to make room for the new element, so it pushes the elements between where it was and where it should go back by one.
- Then continue the scan.

procedure InsertionSort $(a_1, a_2 \ldots a_n)$

```
for j ← 2 to n do
    i ← 1
    while a_j > a_i
        i ← i+1
    m ← a_j
    for k ← 0 to j-i-1
        a_{j-k} ← a_{j-k-1}
    a_i = m
```

} why does this always terminate. Well at some point we find an $a_i > a_j$ OR we get to $a_i = a_j$ and we stop, swapping the number with itself.

$\{a_1, a_2 \ldots a_n\}$ is sorted.

Operation: note a list with one element is sorted?

17, 11, 23, 6, 19

sorted | not sorted

new: 11, 17, 23, 6, 19

sorted | not-sorted.

iter j = 2

while term at a_2: a_1 $m = 11$

for $k = 0$ to $(j-i-1 = 0)$

$a_{j-k} = a_2 \leftarrow a_1 = a_{j-k-1}$

11 17

then 17, 17, 23, 6, 19 $m = 11$

$a_i = a_1 = 11$

11, 17, 23, 6, 19

iter j = 3

while stops at $a_i = a_3 = a_j = a_3$

swap a_3 w a_3 nothing to shuffle!

iter j = 4

11, 17, 23, 6, 19

11, 11, 17, 23, 19 $m = 6$

6, 11, 17, 23, 19

sorted.

iter $j = 5$

6, 11, 17, 23, 19 $\quad m = 19,\ i = 4$

6, 11, 17, 23, 23

6, 11, 17, 19, 23

sorted.

Now we are done!

Analysis of Algorithms

To determine how 'long' an algorithm takes we need to know how long operations take. To do this we define a model of computation.

↑ addition, multiplication, comparisons

There are many such models, for now we will say that comparisons ($<, \leq, =, \geq, >$) take 1 time unit ("unit time"). This real time will vary dependent on hardware/software; but we assume they all take the same time (generally false!).

The number of such operations is the total complexity of the algorithm.

Typically, we are interested in the worst case time complexity, (max # of operations performed).

Recall the max finding algorithm.

① max ← a; ← no cost
② for i ← 2 to n ← one comparison
③ if max < a; then ← one comparison.
④ max ← a; ← no cost

We perform steps ② and ③ & ④ n-1 times for
2n-2 comparisons.
We also perform an addition comparison at step 2
when i = n+1 - so 1 extra step for

2n - 1 comparisons

Linear Search?

① for i ← 1 to n ← one comp
② if $a_i = x$ then ← one comp.
③ return i ← free ☺
④ return - not found

2 comparisons × 'n' iterations
+1 comparison (exit)

$$\frac{2n + 1}{2n + 1}$$

Is linear search always slower than find max?
No if we find an $a_i = x$ it might be much faster,
but in the <u>worst-case</u> it is slower.

Now that we are warmed up lets try something more
fun - sorting!

Bubblesort

```
for i ← 1 to n-1 do                    ← 1 comp
    for j ← 1 to n-i do                ← 1 comp   } depends on n
        if a_j > a_{j+1} then          ← 1 comp.  }
            swap a_j and a_{j+1}
```

} n times

The outer loop (n times)

$$2(n-1) + 2(n-2) + 2(n-3) + \dots\dots 2(1) = \frac{(n-1)n}{2}$$

↗ this relies on a summation formula you will learn later

drop the 2.

$$2\left(\frac{(n-1)n}{2}\right)$$

Insertion Sort

```
for j ← 2 to n do
    i ← 1
1 comp → while a_j > a_i
        i ← i + 1
    m = a_j
1 comp → for k = 0 to j-i-1
        a_{j-k} = a_{j-k-1}
    a_i = m
```

} j times between the two loops.

} n times

$$\underbrace{2 + 3 + \dots + n}_{n-1 \text{ terms.}} = \boxed{\frac{n(n+1)}{2}} - 1$$

↖ Again, using the summation formula for consecutive integers.

← because we are missing the first term.

So

Bubble sort $\frac{n(n-1)}{2} + n = n^2$
(in our comparison only model).

Insertion sort $\frac{n(n+1)}{2} + 1 = \frac{1}{2}n^2 + \frac{1}{2}n + 1$

At first glance, insertion sort is better since $\frac{1}{2}n^2 < n^2$.
but what about the $\frac{1}{2}n + 1$ bit.

What if we had two algorithms which ran in
$\frac{1}{2}n^2 + n$ and $\frac{1}{2}n^2 + 2n$ time respectively which is
better. Well the second has a larger 2nd term ($2n > n$)
but does this really matter much?

When n gets very large the $\frac{1}{2}n^2$ term >> $c \cdot n$ (where
c is any constant term.
So for large 'n' it is the $\frac{1}{2}n^2$ that is really important
(remember we are ignoring many little details that could
account for the $n - 2n$ difference). What really
matters is the n^2 term. So worst case complexity is
about the same.

ie. Effectively you would say a $28,985.16 car
and a $28,985.46 car are the same price.

For the max element / linear search, running times were
$2n + 1$. These are much faster even though the leading term
has coefficient 2. This is because n grows much more
slowly than n^2 (when 'n' is large).
If you are sorting 4 items it doesn't matter which algorithm you use.

So, we don't care about the exact function (too hard to find anyways). What matters is how fast the function grows. So lets find some tools to analyze how fast functions grow.

Growth of Functions

The growth of a function is determined by the highest order term. If you add a bunch of terms, the function grows about as fast as the largest term for large enough input values:

eg. $f(x) = x^2 + 1$ grows as fast as
$g(x) = x^2 + 2$ " " " "
$h(x) = x^2 + 4$, etc.

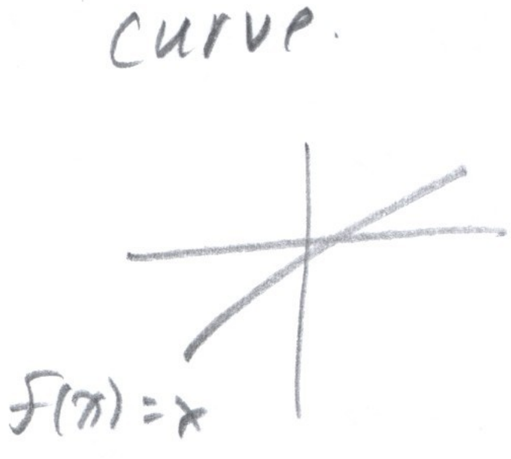
Because for large x, $x^2 >>> x +$?

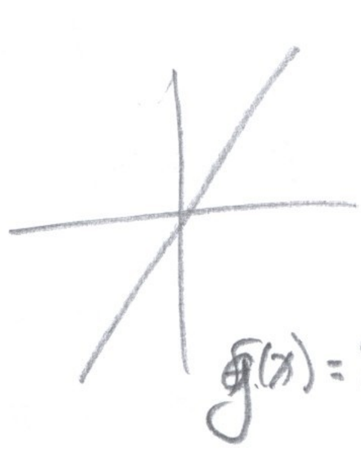
Similary, even constant multipliers don't matter too much:

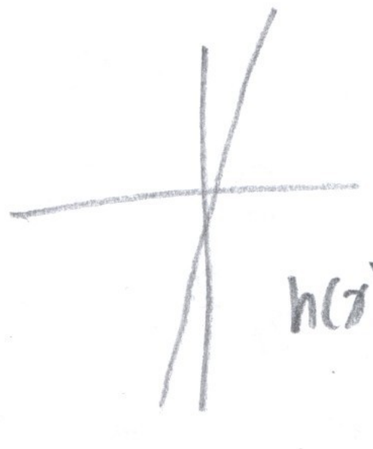
eg $F(x) = x^2$ grows as fast as
$g(x) = 2x^2$ " " " "
$h(x) = 100x^2$, etc.

Because for large x multiplying x^2 by a constant doesn't change it that much, at least not as much as changing x.

Basically we are concerned with the 'shape' of the curve.







all linear

BREAK !!

Only caring about the highest order term (without a constant multiplier) generally corresponds to ignoring differences in the hardware/OS/software. If the CPU is twice as fast, the algorithm still behaves the same way, even if it executes faster.

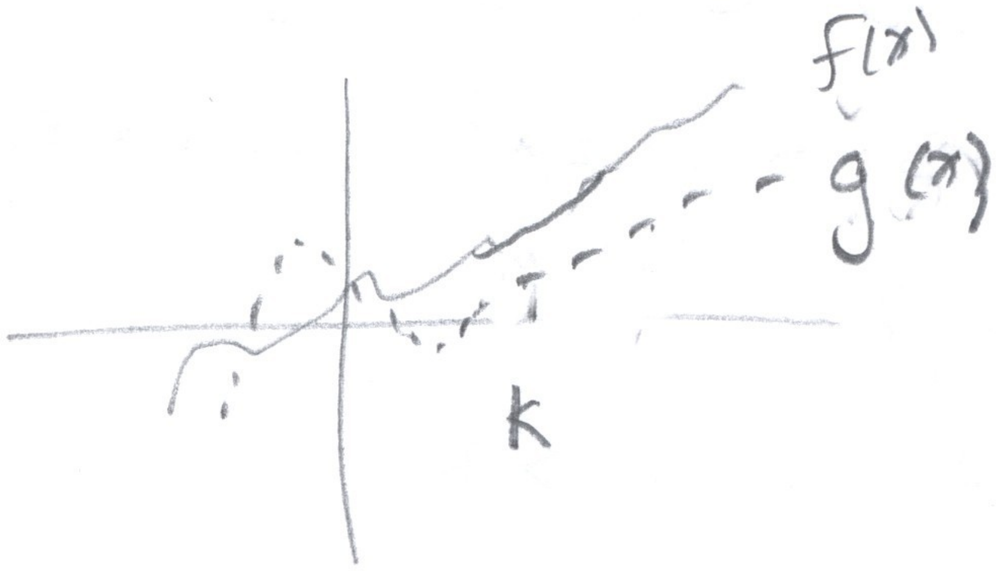
To formalize this:

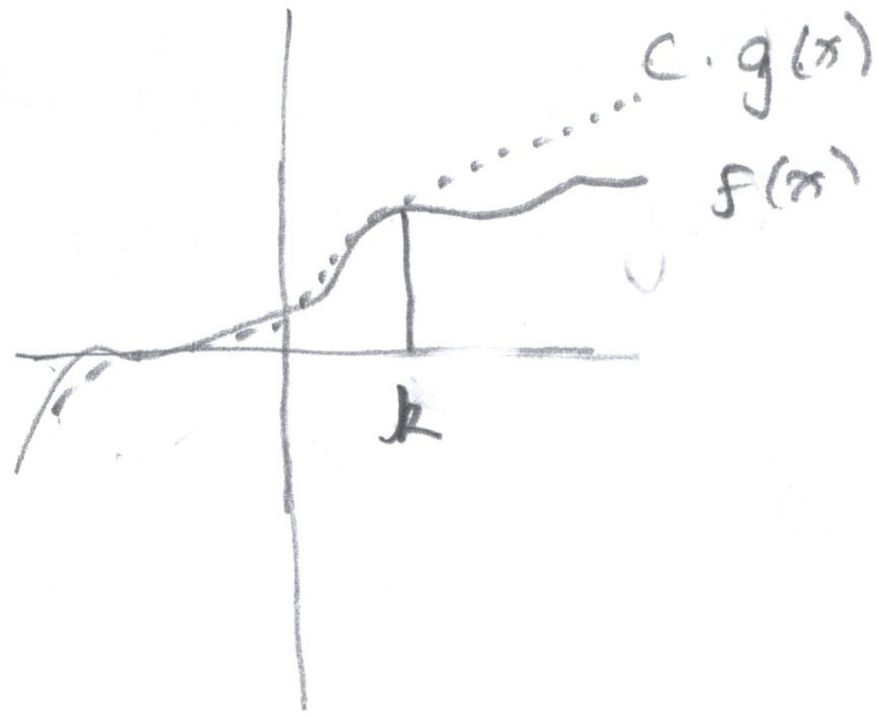
Defn: Let f, g be functions from $\mathbb{Z} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$, we say "$f(x)$ is $O(g(x))$" if there are <u>constants</u> c and k such that

$$|f(x)| \leq c \cdot |g(x)| \quad \text{for all } x > k$$

The constants c, k are called <u>witnesses</u>. We read $f(x)$ is $O(g(x))$ as "$f(x)$ is Big-Oh of $g(x)$" and can write $f(x) = O(g(x))$ or more technically correct $f(x) \in O(g(x))$.

Basically, $f(x) \in O(g(x))$ means that after a certain value of x, f is <u>always</u> smaller than some constant multiple of g.





5(x)

Ex: Show that $5x^2 + 10x + 3$ is $O(x^2)$.

$g(x) = C \cdot x^2$

$$0 \leq 5x^2 + 10x + 3$$
$$\leq 5x^2 + 10x^2 + 3x^2$$
$$= 18x^2 \quad \text{for all } k > 1$$

So $C = 18$ and $k > 1$

So $5x^2 + 10x + 3 \leq 18x^2$ for $k > 1$

So $5x^2 + 10x + 3$ is $O(x^2)$.

Ex: $5x^2 - 10x + 3$ is $O(x^2)$

$$5x^2 - 10x + 3 \leq 5x^2 + 3 \quad (\text{for } x > 0)$$
$$\leq 5x^2 + 3x^2$$
$$= 8x^2$$

$\therefore C = 8, \; k = 1$ are witnesses that $5x^2 - 10x + 3$ is $O(x^2)$.

Typically, we want a function in the $O(.)$ to be as small as possible, but it is still correct to make a claim for larger values.

Ex: Is $5x^2 + 10x + 3 \in O(x^3)$?

Yes; $5x^2 + 10x + 3 < 18x^3$ so $C = 18, k = 1$.

Ex: Is $x^3 \in O(x^2)$?

$x^3 \leq C x^2$ for $x > k$.

$x \leq C$ for $x > t$.

which is false for arbitrarily large x (just pick $x = C + 1$)

So x^3 is <u>not</u> $O(x^2)$.

Some important results:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0 \text{ is } O(x^n)$$

if $x > 1$ then

$$|f(x)| = |a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0|$$

$$\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \ldots |a_1| x + |a_0|$$

(due to the triangle inequality $|a+b| \leq |a| + |b|$

$$= x^n \left(|a_n| + \frac{|a_{n-1}|}{x} + \frac{|a_{n-2}|}{x^2} + \ldots \frac{|a_1|}{x^{n-1}} + \frac{|a_0|}{x^n} \right)$$

$$\leq x^n \left(|a_n| + |a_{n-1}| + |a_{n-2}| + \ldots |a_1| + |a_0| \right)$$

So let $C = |a_n| + |a_{n-1}| + \ldots |a_1| + |a_0|$

Then $f(x) \leq C \cdot x^n$ for $x > 1$.

What is the sum of the first 'n' integers? In Big-Oh.

$$1 + 2 + 3 + \ldots + n \leq \underbrace{n + n + n + \ldots n}_{n \text{ times}} = n^2$$

take $c = 1, k = 1$ So sum in $O(n^2)$.

This makes sense:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2} \quad \in O(n^2).$$

What is the growth of $n!$.

$$n! = n \times (n-1) \times (n-2) \times \ldots 1 \leq \underbrace{n \times n \times n \times \ldots n}_{n \text{ times}} \leq n^n$$

So $n! \in O(n^n)$ with $c=1$, $k=1$

What about $\log(n!)$

$$\log(n!) \leq \log(n^n)$$

so $\log n! \leq n \cdot \log n$, so $\log n! \in O(n \log n)$

with $c=1$ and $k=1$.

What about $\log_2 n$ vs. n (i.e. is $\log_2 n \in O(n)$).

We know that $n < 2^n$ (we will prove this next lecture).

Taking logs

$$\log n < \log_2 2^n = n \cdot \log_2 2 = n$$

So $\log_2 n$ is $O(n)$.

What about other bases for the log? Is $\log_4 n \in O(n)$

Yes: $\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$

So take $c = \frac{1}{\log b}$ which is constant for constant b.

Some common growths, from shortest to fastest.

$$O(1) < O(\log n) < O(n) < O(n \log n)$$
$$< O(n^2) < O(2^n) < O(n!)$$

Big Omega Ω Notation

As we saw Big-Oh provides an upper bound on a function. What if we want a lower-bound? Use Big Omega Notation.

Defn: Let f, g be functions $(\mathbb{Z} \to \mathbb{R}, \mathbb{R} \to \mathbb{R})$. We say "$f(x)$ is $\Omega(g(x))$" if there are constants c, k such that

$$|f(x)| \geq c\,|g(x)| \text{ for all } x > k.$$

Read "$f(x)$ is Big Omega of $g(x)$".

$f(x) \in \Omega(g(x))$ means that after a certain value of x, f is always greater than some constant multiple of g.

Ex: Show $5x^3 + 3x^2 + 2$ is $\Omega(x^3)$.

$5x^3 + 3x^2 + 2 \geq x^3$ so take $c = 1, k = 1$.

Ex: Show that $x^2 - 3x + 4$ is $\Omega(x^2)$

$$x^2 - 3x + 4$$

$$\geq \frac{1}{2}x^2 + \left(\frac{1}{2}x^2 - 3x + 4\right)$$

≥ 0 if $\frac{1}{2}x^2 - 3x + 4 \geq 0$ which is true if $x > 5$

$$\geq \frac{1}{2}x^2$$

So take $c = \frac{1}{2}$ and $k = 6$.

Ex: Is $3x + 1 \in \Omega(x^2)$

$$3x + 1 \geq cx^2 \quad ? \quad \forall x\,(x > k)$$

So $\frac{3x}{x^2} + \frac{1}{x^2} \geq c$

$$\frac{3}{x} + \frac{1}{x^2} \geq c$$

As $x \to \infty$ this number becomes increasingly small.

$\therefore 3x + 1 \notin \Omega(x^2)$.

What about the sum of the first n numbers?

$$1 + 2 + 3 + \ldots + n \geq \underbrace{\left\lceil \frac{n}{2} \right\rceil + \left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + \left(\left\lceil \frac{n}{2} \right\rceil + 2\right) + \ldots + n}_{\text{We sum only those terms} > \frac{n}{2}}$$

$$\geq \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \ldots \left\lceil \frac{n}{2} \right\rceil$$

$$= \underbrace{\left(n - \left\lceil \frac{n}{2} \right\rceil + 1\right)}_{\#\text{ of terms} > \frac{n}{2}} \left\lceil \frac{n}{2} \right\rceil$$

$$\geq \left(\frac{n}{2}\right)\left(\frac{n}{2}\right)$$

↑ since $\left\lceil \frac{n}{2} \right\rceil + 1 > \frac{n}{2}$

$$= \frac{n^2}{4} = \frac{1}{4}n^2 \qquad c = \frac{1}{4}\ k = 1$$

$\therefore$ sum of first n numbers is $\Omega(n^2)$.

This should match our intitution.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + n \in \Omega(n^2).$$

We have now proven that $\sum_{i=1}^{n} i$ is $O(n^2)$ and $\Omega(n^2)$ - we have a special name for such situations.

<u>Defn</u>: Let f, g be functions from $\mathbb{Z} - \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$. We say $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ <u>and</u> $f(x)$ is $\Omega(g(x))$. We say "f is big-Theta of g" of "f is order g".

Recall Bubblesort and Insertion Sort they had running times n^2 and $\frac{1}{2}n^2 + \frac{1}{2}n + 1$, so both are $O(n^2)$ and $\Omega(n^2)$ and therefor $\Theta(n^2)$. So they have the same order.

Similarly max-finding to worst-case time $\Theta(n)$. So using these algorithms it is "faster" to find the max than sort, for large n - which makes sense.

END OF LECTURE 5