Code Efficiency and Runtime Estimation

Runtime of a piece of code
Part 1: Step Counting

may be unfair due to initial conditions (ex. there could be background processes that affect the runtime)

Efficiency: What does it mean? How do we measure it?

- How fast does a program run? measure of elapsed time $\nearrow$ use a wall clock $\searrow$ CPU time
- On which machine? $\to$ platform (there may be two different machines) $\searrow$ technology changes (new, better processers); do we rerun?

  we want to think about things in a more fundamental matter (independent)
- On which input? Or set of inputs? how do we decide the input to use? what kinds? worse/best cases? different applications? typical input?
- Ex: Comparing two programs on typical inputs
- Any finite set of inputs?
- If you could examine the code, could you do better?
- Laborious, error prone, not practical for large code segments, lead to some intuitions?

  version 0 description basically; it starts with a base with what we have rn

doesn't matter the machine, and steps will stand-in for time;
if there is twice as many steps, it will take twice the time

Exact Counting

the intro student would go (marked in red) and say 10 steps, but it's wrong because the loop will cause multiple comparisons

When attempting to determine the time a certain piece of code takes to complete, the first instinct may be to simply count the total number of steps the code will execute before completing, as shown in the following slides.

The following will count as one step:

- Each assignment
- Each conditional evaluation

Example: Sample code; does nothing useful.

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }


    i++;
}
```

any integer number of steps is wrong b/c the number of steps depends on the external input n; if it's not something in terms of n, that's not right

Exact Counting

what is most indented in Java & Python; here, it's the body of the for

#1) Start "inside" the body of the innermost loop: Consists of two lines, therefore they are grouped together with the value of 2.

#2) However, as they are inside a loop, the number of times it executes must be accounted for. In this particular case "i" times. Therefore the body of the inner loop contributes 2i steps to the execution of the code segment.

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

```
#1) 2
#2) 2*i
```

that's why it's a multiple of 2

how many iterations do we have of the body of the 'for'? from 0, 1, 2, 3...i - 1, so we go here i times

the total number of the steps for the body of the for loop is 2i (keep in mind we'll need to add it later for the total)

this methodology allowed us to focus only on the smaller piece, rather than on everything else

Exact Counting

#3) When accounting for the “for” loop itself. The initialization executes once, and contributes 1 step.

#4) Next, let’s look at the update / increment, this occurs every iteration of the loop, and contributes i steps. remember the # of iterations

#5) The condition executes every iteration, plus one final time when the loop breaks, and contributes (i+1) steps.

#6) Adding up all the parts of the for loop gets the total number of steps:

$$2i + 1 + (i + 1) + i = \mathbf{4i + 2}$$

a **model of efficiency** of the loop that is a function of i

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

```
#1)  2
#2)  2*i
#3)  1
#4)  i
#5)  i + 1
#6)  4*i + 2
```

the condition controls whether we enter the body of the loop one more time. the condition is 1 step, but it gets executed multiple - it's i for each time we entered the body of the loop and 1 is the time that kicks us out of the loop (it comes back as false)

Exact Counting

add these steps to get 3

#7) Moving to the body of the outer loop: The first two lines and the last line each execute once.

#8) The total number of steps for each iteration of the outer loop is, again, just the addition of the separate parts:

2 + (4i + 2) + 1 = **4i + 5**

just add these together;
you know now that the body of
the while is this number of steps

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

2 steps

1 step   i + 1 steps   i steps

2i steps

1 step

```
#1)  2
#2)  2*i
#3)  1
#4)  i
#5)  i + 1
#6)  4*i + 2
#7)  3
#8)  4*i + 5
```

Exact Counting

the top part of the sigma is n-1 b/c
that's the total # of times the while loop will execute

#9) The result from step 8 only accounts for the steps for *each* iteration of the loop. To find the steps for *all* iterations, Summation must be used:

$$\sum_{i=0}^{n-1}(4*i + 5)$$

Which can be simplified to:

$2*n^2 + 3*n$

see the notes on the doc

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

```
#1) 2
#2) 2*i
#3) 1
#4) i
#5) i + 1
#6) 4*i + 2
#7) 3
#8) 4*i + 5
#9) 2*n² + 3*n
```

Exact Counting

#10) From this point, it’s fairly simple to calculate the rest of the steps. The very first statement executes once.

#11) The condition of the “while” loop will execute n times, plus once extra to break out of the loop.

#12) From this point the total number of steps may be calculated:

$$1 + (n + 1) + 2n^2 + 3n$$

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

#10) 1

#11) n + 1

#9) 2*n² + 3*n

this is the body

#12) 2*n² + 4*n + 2

quiz next class (exactly what it's going to look like)

Practice Quiz

to check, the answer should be in the form of the external input (it should be a function of n). when n is 0 and 1, it better give the proper answers for the 2 experiments

In this problem you will count the **exact number of steps** for the code segment below. You should count each assignment and each comparison as one step.

```
2 { i = n;
    sum = 0;
          counts as n + 2 steps (n + 1 for how many times the body runs)
          then +1 for testing if the condition is false
    while (i >= 0) {
       // this is the body of the loop
       j = i * 2;
       sum = sum + j;      consists of 4 lines, so let's
       k = i * j;          group these together as they're
                           the body of the loop (occurs n + 1 times)
       i = i - 1;
    }
```

Q1.(1 points) How many steps are executed when n is 0? 8

Q2.(1 points) How many steps are executed when n is 1? 13

Q3.(2 points) How many times does the body of the loop execute? n + 1

Q4. (6 points) The total number of steps in an execution of the code segment is 5n + 8 .

this is b/c it will run through n times (decrement) and notice that the inequality has an equals, which encounters for another 1 rather than if it were just i > 0

the body of the while loop comes out to 4*(n + 1), the while condition comes out to (n + 1) + 1 = n + 2, and then add +2 for the top 2 steps; 4n + 4 + n + 2 + 2 = 5n + 8

An Alternative to Exact Counting

the constant that corresponds to the most significant term, it doesn't matter when making the prediction, so maybe we don't care about it

Suppose a program has been found to have **5n + 8** steps, and takes 1 hour to run with input of size n = 100.

$(\boxed{5}*100 \boxed{+ 8})$ steps = 1 hour

term that corresponds to n^0 is going to be insignificant for large n

As the second term has little impact, we'll rewrite the equation as 5n.

$\underline{(5 * 100) \text{ steps}} \backsimeq 1$ hour

now, we're interested in doing a prediction

Suppose we want to find the runtime for input of size n = 400.

$\mathbf{(5 * 400) \text{ steps}} \backsimeq$? hours $\leftarrow$

try to rearrange this to get this to resemble the format of the above equation

Through some simple algebra, the equation may be written in a way which puts it in terms of an already known runtime.

$4 * \underline{(5 * 100) \text{ steps}} \backsimeq$? hours

Finally, notice that the runtime increases linearly with the input size. This makes sense since 5n+4 is linear.

4 * (1 hour) = 4 hours

highest degree is 1

why is it linear? equation for a line is mx+b. for us, n is the size of the input and what we want to plot is the number of steps on the y-axis. the line is linear for n.

An Alternative to Exact Counting

Suppose a different program has been found to have a runtime of 7**n²** and the runtime of the program for n = 100 is again, 1 hour.

If the input is increased to n = 400 again, what will the runtime be?

Again, finding the runtime is simple, if put in terms of something already known.

highest degree is 2

Notice that the runtime still increases due to the size of n, but is now $\underline{\text{quadratic}}$ rather than linear.

7n^2 would be the y-axis here

imagine this as the conversion rule

$\underline{7 * 100^2 \text{ steps}} = 1 \text{ hour}$

$7 * 400^2 \text{ steps} = ? \text{ hours}$

$7 * (4 * 100)^2 \text{ steps} = ? \text{ hours}$

$7 * (4^2 * 100^2) \text{ steps} = ? \text{ hours}$

$4^2 * \underline{(7 * 100^2) \text{ steps}} = 4^2 * (1 \text{ hour})$

only the high order term w/o constant counts (the n^2, n^1, whatever)

An Alternative to Exact Counting

The previous examples help illustrate that any term in the equation, aside from the most significant term, affects the runtime less and less as the size of “n” grows. Computer scientists use this to look at the approximate *growth rate* of algorithms.

Suppose the exact runtime of a piece of code is: $3\boxed{n^2} + 100n + 19$

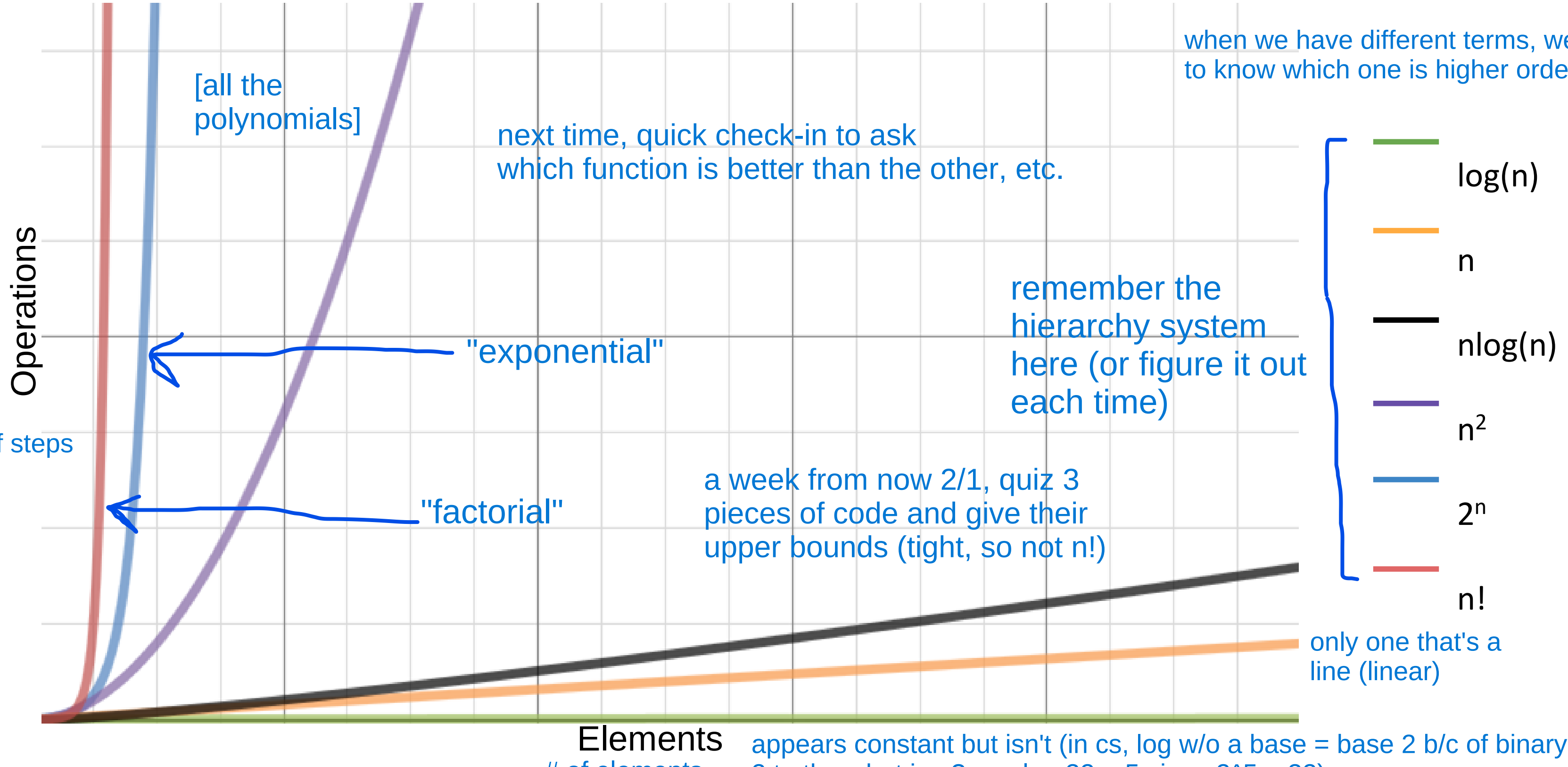
The estimation of this is simply n^2 because that $\underline{\text{higher-order term}}$ will dominate all the others, even 100n, or 1000n. The multiplicative coefficient of 3 can also be dropped because it does not affect the *rate* at which the runtime grows with increasing n. We care about the *shape* of the runtime. what happens to the algorithm as n gets large?

these represent the amount of work: the best possible thing we can hope for is constant (written as just n^0, or 1) (no matter what input you give, the program runs in constant time), an example is even (algorithm to tell whether the int is even, it reads the least significant bit and compares to 0)

n^0 < log(n) < n^1 (linear) < n^2 (quadratic) < n^3 (cubic) < etc.

Some handy function shapes

when we have different terms, we need to know which one is higher order



[all the polynomials]

next time, quick check-in to ask which function is better than the other, etc.

"exponential"

"factorial"

remember the hierarchy system here (or figure it out each time)

# of steps

a week from now 2/1, quiz 3 pieces of code and give their upper bounds (tight, so not n!)

only one that's a line (linear)

# of elements

appears constant but isn't (in cs, log w/o a base = base 2 b/c of binary = 2 to the what is n?, ex. log 32 = 5 since 2^5 = 32)

An Alternative to Exact Counting

Suppose the following piece of code is given, rather than finding the exact number of steps, the approximate runtime "n" may be used because the runtime increases linearly.

n + 1 (can count as n)

```
for (i = 0; i < 5 * n; i++){
    sum = sum + i;
    product = product * sum;
}
```

c

the body occurs 5n times, multiply that by c to get 5nc or really n

Approximation: n

In a more complicated code sample, the inner loop always iterates a constant number of times, having no effect on the rate of growth. The facts that the outer loop starts at 11, is incremented by 2, and the ending value is multiplied by 4, are all ignored because they don't contribute to the *rate* of growth.

constant b/c it's not dependent on n

```
for (x = 11; x < 4 * n * n; x = x + 2){
    for(i = 0; i < 25; i++){
        sum = sum + i;
        product = product * sum;
    }
}
```

c

c

Approximation: n^2

if we simplify the outer for loop to x = 1 and x++, if we go from 1 to 4n^2, update once, this is an n^2 loop (because of the condition). if we go up by 2's, it's less work than going up by 1's (you skip x = 1, 3, etc.), so it's the amount of work/2. if you put the 11 back as the initial, you subtract by 10. remember the constant doesn't matter so it's always still going to be n^2 (imagine that the 10 is a for loop that happens 10 times, it's just a constant)

Application to large code segments

```
methodM(n ...) {
  loop (do n times) {
    m1 ... // n * lg n
    m2 ... // n²
    m3 ... // n
  }
}
```

start at the innermost (which has been done for you). the entire loop is nlogn + n^2 + n. of these 3, the dominant term is n^2. the body of the loop is therefore n^2. the loop makes you do the body n times, so n^2 * n = n^3.

This code isn't quadratic. It is cubic

Where should you put your energy?

you should improve m2 (for the star developer) because in terms of runtime, making the others better doesn't change anything. this can make the entire runtime better.

Exact Counting vs. Estimating

Exact Counting

- Very accurate model of runtime
- Slow and tedious to calculate
- Easy to mess up
- Difficult to verify

Estimate

- Less precise than exact count
- Faster to calculate
- Simpler to verify
- Sufficient when it produces the correct “shape”

as n gets large, the exact amount of work is upper bounded by some function, not going to assume it holds for small n