Estimating Upper Bounds and Code Sketches

refine our methodology and techniques used to present algorithms in the semester

Estimating Tight Upper Bounds

When looking for an upper bound for the number of steps of a piece of code, it may be tempting to overestimate the amount of work the code does to be on the safe side. But an upper bound that is too loose may not be useful.

Analogy:

> Someone tells us it takes “at most 3 hours” to get a destination, and we use that information to decide which route to take. In reality, the typical travel time is only 10 minutes. Yes, it’s likely that we will reach our destination within 3 hours, but that information is so over-cautious that it’s ultimately unhelpful.

Estimating Tight Upper Bounds

At first glance, this piece of code appears upper-bounded by n^4. The inner loop is a typical n loop. The outer loop terminates at n^3 multiplied by a constant, and. So $n^3 * n$ is n^4.

While this is an upper bound, it drastically overestimates the amount of operations actually needed and ultimately doesn't model the code very well.

Notice the update instruction of the outer loop!

The question is how many times do we execute the body of the outer loop.

values of x: powers of 2 until x reaches 57n^3

this is actually a log(n) because the update is not additive, but multiplicative (very big difference in runtime)

```
for(x = 1; x < 57*n*n*n; x = x*2){
    for(y = 0; y < n; y++){
        sum = sum + y;
    }
}
```

n

c

How many times does the loop (for x) make me do its body (linear)? In this case, this question is the same as how many times does x need to be updated (*2) until it reaches its limit (57n^3)?

Estimating Tight Upper Bounds

The outer loop ends when x reaches **57n.** Does that mean that the body of the loop executes n^3 times (times a constant)? Let's look at some of the values of x to gain some insight into the way x growths.

We can map the value of x and the loop iteration number, which also allows us to find the iteration number in terms of x. Our loop stops when

$$\mathbf{x = 57n^3}$$

the limit

how many times the body is executed? not x, but rather i.

$$\Leftrightarrow \mathbf{2^i = x = 57n^3}$$

log 2^x = x (base 2)
so take logs on both sides to get what i is

$$\Leftrightarrow \mathbf{i = \log_2(57n^3)}$$

```
x = x * 2
```

Iteration number	Value of x
Start	1
1	2
2	4
3	8
4	16
…	…
i	2^i

identify patterns from the few examples

Estimating Tight Upper Bounds

So the number of times the outer loop executes is actually:

log of a product is the sum of the logs
log(a * b) = log a + log b

$\log_2(57n^3)$ =

= log(n * n * n)
$\log_2(57) + \log_2(n^3)$ = log(n) + log(n) + log(n)
= 3log(n)

$\log_2(57)$ is just a constant, and $\log_2(n^3)$ may be further broken down to:

$3\log_2(n)$

As we saw earlier in the semester, the multiplicative constant may be ignored, resulting in an estimate of the runtime of the outer loop of

$\log_2(n)$

if the limit had been 557n^3, no difference
if it had been n^7, no difference (similar for polynomials in the limit)

```
for(x = 1; x < 57*n*n*n; x = x*2){
    for(y = 0; y < n; y++){
        sum = sum + y;
    }
}
```

Estimating Tight Upper Bounds

From this point, finding the estimate for the entire piece of code is easy, the inner loop executes n times leaving the total runtime of the code as:

$$n\log_2(n)$$

Computer scientists deal mainly with binary, and 2 is the most common log base, so the subscript of a logarithm in base 2 is commonly left off. Thus, we would write this runtime as:

$$n\log(n)$$

```
for(x = 1; x < 57*n*n*n; x = x*2){
    for(y = 0; y < n; y++){
        sum = sum + y;
    }
}
```

Estimating Tight Upper Bounds

Let’s review how each component of the analysis derives from the code. **The inner loop is n and the outer loop is** $\log_2(57n^3)$ =

$$\log_2(57) + \log_2(n^3)$$

$\log_2(57)$ is just a constant, and $\log_2(n^3)$ may be further broken down to:

$$\mathbf{3\log_2(n)}$$

As we saw earlier in the semester, the multiplicative constant may be ignored, resulting in an estimate of the runtime of the outer loop of

$$\mathbf{\log_2(n).}$$

Thus the entire code is n*log(n).

```
for(x = 1; x < 57*n*n*n; x = x*2){
    for(y = 0; y < n; y++){
        sum = sum + y;
    }
}
```

Log chops down massive polynomials

Let’s look at this similar loop now:

$$\log_2(22n^4) = \log_2(22) + \log_2(n^4)$$

```
for(x = 1; x < 22*n*n*n*n; x= x*2)
```

cuz its multiplicative by a constant, it doesn't matter

Since $\log_2(22)$ is just a constant, what matters is and $\log_2(n^3)$ may be further broken down to:

$$\log_2(n^4) = \mathbf{4 * \log_2(n)}$$

Which is $\log_2(n)$ or $\log(n)$.

log based x (anything) = log based 2 (anything) * log based 2 (whatever base you're coming from)^(-1) = log anything

like in binary search, why not chop in thirds or fourths? because you'll end up anyways with a logn runtime

If the update of the loop changes to `x = x*5` the loop is still log(n) to us because

$$\mathbf{\log_5(n) = \log_2(n) * (1 / \log_2(5))}$$

Estimating a Runtime Without Code

Computer Scientists often wish to estimate the runtime for an algorithm without having to go through the process of writing working code. So it is useful to be able to estimate a runtime based on a description or a general understanding of an algorithm.

worthwhile to organize data before inputting it

Take for example, the Binary Search algorithm: Given a sorted array, the program looks at the middle element, and based on that value, excludes half of the search array from further consideration. This process is repeated using the middle element of the non-excluded half of the array, until either the desired element has been found, or the searchable part of the array is only one element.

Estimating a Runtime Without Code

The main idea behind the binary search is that every time the algorithm looks at an item, it cuts the searchable area in half.

Letting "ℓ" be the length of the original array, the size of the usable array for future passes may then be described in terms of ℓ, the goal is to reach a size of 1.

how many times do we do the constant amount of work to get the value we need?

$$\ell \Rightarrow \frac{\ell}{2} \Rightarrow \frac{\ell}{4} \Rightarrow \frac{\ell}{8} \Rightarrow \dots \Rightarrow \frac{\ell}{\ell} = 1$$

$$\frac{\ell}{2^0} \Rightarrow \frac{\ell}{2^1} \Rightarrow \frac{\ell}{2^2} \Rightarrow \frac{\ell}{2^3} \Rightarrow \dots \Rightarrow \frac{\ell}{2^{\log_2(\ell)}} = \frac{\ell}{\ell} = 1$$

So the number of comparisons needed for binary search is $\log_2(\ell)$ or $\log(\ell)$.

Practice: Estimate a tight upper bound for the runtime of

```
for (i = 1; i <= n/2; i = i * 2) {
   sum = sum + product;        c
   for (j= 1; j < i*i*i; j = j + 2) {
       sum++;
       product += sum;         c
   }                                  n^3     n^3
}
{ // some quadratic code segment}      n^2
```

if this was n^6, then n^6 would be the dominant term instead

Runtime computation:

1. Start at the inner loop
2. The worst case for i is n (1/2 is ignored) in terms of the external input
3. The worst case for j is i^3 which is n^3 it's bounded by (n/2)^3 (see outer loop) = n^3
4. The for(j) is n^3
5. Move to the outer loop. The increment of i is multiplicative
6. The for(i) is $n^3 * \log(n)$ from the "for i " statement
7. The for(i) is worse than quadratic so the entire code is $n^3 * \log(n)$

Practice Quiz: Estimate a tight upper bound for the runtime

```
                       i/2 = 1/2 * i
for (i = n; i >= 1; i = i /2) {
c  sum = sum + product;
n^2 for (j = 1; j < i*i; j = j + 2) {
     c sum ++;
n^4    for (k = 1 ; k < i*i*j; k++)
        c  product *= i * j;
   }
}
```

Runtime:

1. Start at the innermost.

2. How many times does the "for k" make it do its body? The update is additive, and the limit is i^2 * j. We should put it into terms of the external input, so we can look for the other loops to try to put it like that.

Working outwards, j = 1, 3, 5, 7, etc. the one that makes the most work is when j is i^2. So, i^2 * j is upper bounded by i^2 * i^2 = i^4. Let's look at the i loop now, it's when i = n. So, this is bounded by n^4. (Constant steps mostly trivial.)

3. So the body of the "for j" is n^4. How many times does the "for j" make me do the n^4 body? Update is additive by a constant up to i^2 (and thus bounded by n^2. So the entire "for j" is n^2 * n^4 = n^6.

4. It's a multiplicative update for the "for i", so it's a log(n) since the condition is based around polynomials. repeatedly chops by 1/2 (binary search)

5. So the whole thing is n^6 * log(n).

Practice Quiz: Estimate a tight upper bound for the runtime

```
n^2 for (i = 1; i < n*n; i = i+2) {
  log n for (j = 1; j < i*i*i; j = j*2) {
        c sum ++;
      n^8 for (k = i*j; k > 0; k = k - 2)
          c product *= i * j;
      }
  }
```

always begin looking at what the update looks like

Runtime:

1. "for k" has an additive update. the initial is an i*j, so we need to bound this. the loop above is a "for j"; the bound for j is i^3, so "for k" is bounded by i * i^3 = i^4. in the "for i" loop, it's bounded by n^2.

thus, i * j < i (i^3) = i^4 < (n^2)^4 = n^8 for the "for k" statement, and n^8 also for "for k" entirely

2. "for j" has a multiplicative update (by a constant). the max value of j is bounded by i^3, which is bounded by (n^2)^3 = n^6. basically, the update turns it into logn (refer to previous slides of chopping down polynomials).

thus, the whole "for j" with its body is n^8 * logn.

3. "for i" update is additive, and the condition is n^2, as we translated earlier. thus, the statement is n^2.

the whole piece of code is now n^2 * n^8 * logn = n^10 * logn

Practice Quiz: Estimate a tight upper bound for the runtime

in the quiz, all the answers are going to have the n^a * (logn)^b
if you have n^6, on canvas quiz, put a = 6 and b = 0 (since there is no log)

(Extra credit)

```
int i = 1, j = 2;
while (i <= n) {
    sum += 1;
    i = i * j;
    j = j * 2;
}
```

this is really hard

Runtime:

Code Sketches

In order to quickly or easily express an algorithm, it may sometimes be presented as a quick code sketch. A code sketch may be entirely in plain english, in pseudocode, or a mixture of the two. The main reason for sketching code is to quickly explain or explore a particular algorithm, without getting bogged down in implementation details that have no impact on worst-case runtime estimation. Also, when solving a problem, we want to be able to analyze several algorithms for the problem, choose the best one, and then implement (code and test) only the chosen one.

The following slides contain sketch examples of famous algorithms. They illustrate some of the different ways an algorithm may be conveyed.

goal is to reorganize the values in the array so that when we're done, the last position will be the largest and first will be the smallest, and in between, the remaining values will be organized in nondecreasing order (increasing when you have duplicates)

Code Sketches: Selection Sort

Selection Sort version 1: initially everything is unsorted, unsorted portion is scanned, while the sorted portion gets filled up and doesn't get scanned

scan portion

1st pass: scan the given list, looking for the largest item, swap largest into the last position.

n, or the number of elements in the list to go through

* c, or each element is compared and possibly updates the maximum so far = constant #

+ c, or the constant number of assignment for the swap

2nd pass: scan all but the last position, looking for the largest item, swap into the second to last position.

now, it's (n - 1)c + c because unsorted portion has decreased by 1

...

2c + c because 2 elements need to be scanned now

(n-1)th pass: scan first two elements, put in order.

amount of work done by selection sort while solving a problem of size n is adding all the passes of the algorithm together, so

$$\Leftarrow \sum_{i=2}^{n} i * c + c$$

simplify this due w/ summation properties to get something that's approximately the summation with only i. something that upper bounds this is < summation + 1 = the summation of i from i = 1 to n. this is the super-famous summation which is approximately n^2.

we need to make sure we depict the algorithm clearly

[NEXT SLIDE] Wbs(n) = approx upper bound of amount of work that binary search does to solve a problem of size n $\rightarrow$ Wbs(100) = include Wbs(50) + __; not endless, continuous chopping till small range

Code Sketches: Selection Sort

the whole thing is n*n = n^2.

$$\text{Wbs(n)} = \begin{cases} \text{C1, when n is small} \\ \text{C2 + Wbs(n/2), otherwise} \end{cases}$$

```
/*Selection Sort v2*/

i = (list size - 1)
```

// position where unsorted portion ends; initially unsorted so it's the size - 1

```
while(i > 0)
```

as long as there's something in the unsorted portion; the loop happens n times

```
{
```

i — i Scan the first `i` positions in the list, keeping track of the maximum value.

c Swap the maximum value into the `i`th position.

c Decrement `i`.

the worst i can be is the very first pass where i < n.

```
}
```

[NEXT SLIDE] why is array fetching constant? array is stored sequentially, system storing this says that array A keeps at a certain position (called base a), somewhere in memory, the address where a begins is known AND the system knows the size of each element in the array, which is why arrays always have the same size. it knows exactly where each element is because you provide the offset (or in a[index]). the way it gets the element there is baseA (where it starts) + offset * size of each element.

array = array we're searching in
range = subarray specified we're searching in
key = what we're searching for

Code Sketches: Recursive Binary Search

specifying start/end positions

we know this is log n

```
int BinarySearch (array, range, key){
c     if (range is empty){                    // Key not found.
              return -1;                       // This is one base case or
                                               // terminating condition
      }
   c middle = middle value in the range;
   c if (key equals middle)
         return the position of middle;    //Search succeeded (base case)
     //Recursive cases
   c if (key > middle)
         return BinarySearch(array, upper half of range, key); // right
     else
         return BinarySearch(array, lower half of range, key); // left
}
```

usually done by (sPos + ePos)/2 = pMid; middle = a[pMid]

evaluating a comparison (> <) is constant

recursively calling this helper method

the addition of 2 integers = efficient // divide by 2 = architecture based on bits is very efficient
fetching the value stored in the middle = getting any value in a particular list is O(n) worst case, if it's an array instead of a list, it's worst case constant