CS 61B

Resources / Guides / Asymptotics

Asymptotics

Author: Omar Khan

Introduction

This serves as a practical guide to asymptotics. We will not be talking about theory or why asymptotics is neccessary or useful. This guide assumes that you already know those things and want to apply them to find runtimes of actual Java methods. We'll cover some strategies that you should know and then we'll go over what I call "cheats," (warning: there are a lot of cheats and I'll only be discussing a few).

Without further adiou, let's start off by talking about some quick math things that you'll absolutely need to know.

Math background

There are just a few tiny math things you'll need to know for asymptotics. One will be simplifying summations, and the other is a quick trick on figuring out how many elements there are in a sequence.

Summations

Most of asymptotics is really just simplifying the work done into a summation and then applying the right summation formula. The two types of summations you'll see in CS 61B are the **arithmetic summation** and the **geometric summation**.

Arithmetic Summation

An arithmetic summation is one where the difference between consecutive terms is some constant. The constant doesn't matter. For example:

$$1 + 2 + 3 + 4 + 5 + ... + N$$

$$1 + 3 + 5 + 7 + 9 + 11 + ... + N$$

$$10 + 20 + 30 + 40 + 50 + ... + N$$

are all arithmetic summations.

With an arithmetic summation, the entire summation is Θ of the last term squared. So all of the above summations are $\Theta(N^2)$ even though all the constants are different. If it seems too easy to be true, it's not. It really is that simple.

Seemingly hard question: what is the Θ bound of the below sum:

$$1 + 2 + 3 + 4 + 5 + \ldots + \log N$$

You might think it's hard because you see a $\log N$. But don't forget what you just learned. Take the last term, square it, and then put a Θ on it. Hence, the above summation is $\Theta(\log^2 N)$. See? Not hard at all. Now let's talk about the next type of summation.

Geometric Summation

Arguably the easier summation. A geometric summation is one where the *ratio* of consecutive terms is constant. For example:

$$1 + 2 + 4 + 8 + \ldots + N$$

$$1 + 3 + 9 + 27 + \ldots + N$$

$$10 + 100 + 1\,000 + 10\,000 + \ldots + N$$

With a geometric summation, you still take the last term but *that's it*. No squaring. Take the last term, throw a Θ on it, and that's it. So, you guessed it, all of the above summations are $\Theta(N)$.

Seemingly hard question: what is the Θ bound of the below sum:

$$1 + 2 + 4 + 8 + 16 + \ldots \sqrt{\log N}$$

You've probably caught onto the trick. It's not hard for you now since you're equipped! We see from the first few terms that the summation is geometric and so we just throw a Θ bound on the last term. Hence, the above summation is $\Theta(\sqrt{\log N})$. Easy peasy.

Summation summary

Let's quickly summarize the summation lessons. If you see a summation, take these 2 steps:

1. Look at the first few terms of the summation to see what type it is (arithmetic or geometric).
2. Figure out the last term, and then apply the appropriate simplification (if it is arithmetic, square and put a Θ, and if it is geometric just put a Θ).

Sequences

Sequences are slightly different than summations. Instead of summing things, it's just a list of numbers. The following are sequences:

$$1, 2, 3, 4, 5, ..., N$$

$$1, 2, 4, 8, 16, ..., 2^N$$

Notice the commas instead of $+$ signs.

There is no concept of "arithmetic" or "geometric" sequence. They're just sequences. Often we'll boil the work done in a Java method to a *sequence* of numbers and we simply want to know how many numbers are here. So here is how I suggest you do that.

You'll need to know the last term. Let's say it's a function of N, call it $g(N)$. So, for the above, $g(N) = N$ for the first one and $g(N) = 2^N$ for the second one.

1. Figure out an expression for the ith element in the sequence. Call it $f(i)$. You can make the sequence 0-indexed or 1-indexed, whichever leads to a cleaner expression for $f(i)$. 0-indexed means $f(0)$ is the first term, and 1-indexed means $f(1)$ is the first term.
2. Make a new variable called k and set the number of elements in the sequence to k. Our objective is to determine k.
3. $g(N) = f(k)$ and solve for k. Read the above definitions of $g(N)$ and $f(i)$ if it doesn't make sense why we do this.

That's it. Let's do that for the above 2 sequences. $1, 2, 3, 4, 5, ..., N$ Honestly, you don't even need to do it here since there are clearly N elements, but we'll do it as a warm up.

$g(N) = N$ and $f(i) = i$. Easy enough. Note that here we're 1-indexing the elements.

$$g(N) = f(k)$$

$$N = k$$

Oh look, we're already done. So there are N elements in this sequence. Onto the next one.

$$1, 2, 4, 8, 16, ..., 2^N$$

This one is a bit more complex, though you might be able to do it in your head. Regardless, let's figure out $g(N)$ and $f(i)$.

$g(N) = 2^N$ and $f(i) = 2^i$. Note that here we're 0-indexing the elements.

$$g(N) = f(k)$$

$$2^N = 2^k$$

$$N = k$$

And that's it! Now let's talk about one sequence that you will see **a lot** and you might as well just memorize this instead of doing it every time.

$$1, 2, 4, 8, 16, ..., N$$

$g(N) = N$ and $f(i) = 2^i$.

$$g(N) = f(k)$$

$$N = 2^k$$

$$k = \log_2(N)$$

Voila. Remember that the base doesn't matter with $\log$ for asymptotics purposes, so we can just say that k is $\Theta(\log N)$.

Here is a hard one that I'll leave for you to do by yourself:

$$2, 4, 16, 256, 65536, 4294967296, ...N$$

Those might seem like random numbers, but I recommend writing each number as a power of 2. You can solve this using the same steps as above. Ask a TA or a friend. Maybe even post on Ed.

And with that, we've gone over the mathematical background. You'll soon see how important these things are. It's ok if you just memorized what was said so far. But from here on out, you shouldn't memorize any formulas. Just the strategies. Without further adieu, let's start with single loop Java methods!

Single loops

This is the bread and butter of asymptotics questions. You'll soon see how important those math things we did above were. First let's look at a few examples of what these Java methods might look like:

```
public void loop1(int N) {                                          Copy
        for (int i = 0; i < N; i = i + 1) {
                System.out.println(i);
        }
}
```

This is a pretty standard function. Let's look at another one.

```
public void loop2(int N) {
        for (int i = 0; i < N*N; i = i + 1) {
                int[] x = new int[i];
        }
}
```

Copy

One more.

```
public void loop3(int N) {
        for (int i = 1; i < N; i = i * 2) {
                int x = i;
        }
}
```

Copy

I think we've seen enough. All of these look basically the same. There isn't much more to change here. Don't worry about finding the runtimes of those things above, we'll do that soon.

Just 3 things changed across all of those functions:

1. The exit condition of the loop. Say the biggest value i can be is $g(N)$. This means our exit condition is $i < g(N)$.
2. How i increments every iteration as a function of the previous i.
3. How much work gets done inside the loop as a function of i. Call this function $f(i)$.

With these things, we can quickly find the runtime of the function by making a summation:

$$f(i_1) + f(i_2) + f(i_3) + \ldots + f(g(N))$$

Here, i_1 is the first value of i, i_2 is the second value of i and so on. The way that i increments will help us figure out these values. $g(N)$ is the last term as you might have guessed since it is the last value that i takes on, and we must take $f(g(N))$ to see how much work the last iteration does.

You'll find these functions f, g in your problem as well as $i_1, i_2, i_3, \ldots$ and then simplify the summation according to the correct formula (i.e. if it's arithmetic square the last term, if it's geometric do not square the last term).

That's it. Let's revisit `loop1` and see how to actually do this.

Loop 1

```
public void loop1(int N) {
        for (int i = 0; i < N; i = i + 1) {
                System.out.println(i);
        }
}
```

Copy

1. $i < N$ is the exit condition, so $g(i) = N$
2. $i = i + 1$ is the increment function
3. ???

I hope you see how we got the first 2. I just looked at the for-loop and it's those last 2 things. Pretty simple right?

What about how much work gets done as a function of i? Well, it's simply 1. You might be confused, and that's ok. Why 1? Because printing an integer is a **constant time** operation. Another way of saying this is that `System.out.println(i)` is $\Theta(1)$. It doesn't depend on i. So we say that $f(i) = 1$. Think of this as a definition.

Ok, now we're ready for the next step: making the summation.

$$f(i_1) + f(i_2) + f(i_3) + f(i_4) + ... + f(g(N))$$

We know that $i_1 = 0, i_2 = 1, i_3 = 2, i_4 = 3$ and so on. Also, we know that $g(N) = N$.

$$f(0) + f(1) + f(2) + f(3) + ... + f(N)$$

We know that $f(i) = 1$, so:

$$1 + 1 + 1 + 1 + ... + 1$$

This summation is super easy. It's just 1 repeatedly added! How many terms are here? Well, if we look at the values of i, it forms a simple sequence of numbers, and we quickly see that there are N terms total. So we can simplify this to $N \cdot 1 = N \in \Theta(N)$.

Important thing to notice: if the work done per iteraiton is constant with respect to i, you can simply **count the number of iterations** and put Θ on that number.

Let's do `loop2` which is slightly more complicated.

Loop 2

```java
public void loop2(int N) {
        for (int i = 0; i < N*N; i = i + 1) {
                int[] x = new int[i];
        }
}
```

1. $i < N^2$ is the exit condition, so $g(i) = N^2$
2. $i = i + 1$ is the increment function
3. ???

Again, the first 2 aren't too bad to get. Just reading the code.

What about how much work gets done as a function of i? The inside is a bit trickier. It's making an array of length i, which does i work (since it needs to make a memory box for every index). Thus, $f(i) = i$.

Ok, now we're ready for the next step: making the summation.

$$f(i_1) + f(i_2) + f(i_3) + f(i_4) + \ldots + f(g(N))$$

We know that $i_1 = 0, i_2 = 1, i_3 = 2, i_4 = 3,$ and so on. Also, we know that $g(N) = N^2$.

$$f(0) + f(1) + f(2) + f(3) + \ldots + f(N^2)$$

We know that $f(i) = i$, so:

$$0 + 1 + 2 + 3 + \ldots + N^2$$

Does this summation look familiar? It should. We look at the first few terms and determine it's arithmetic, so we take the last term, square it, and put Θ on it. The last term is N^2, squaring that gives N^4, and putting the bound gives a final answer of $\Theta(N^4)$. Done.

Finally, let's do `loop3`.

Loop 3

```java
public void loop3(int N) {
        for (int i = 1; i < N; i = i * 2) {
                int x = i;
        }
}
```

Copy

1. $i < N$ is the exit condition, so $g(i) = N$
2. $i = i * 2$ is the increment function
3. ???

Again, the first 2 aren't too bad to get. Just reading the code.

What about how much work gets done as a function of i? It's constant. Not a function of i. So, $f(i) = 1$.

Now here we *could* continue with the rest of the problem, or remember what I said above. I'll repeat it.

> Important thing to notice: if the work done per iteraiton is constant with respect to i, you can simply **count the number of iterations** and put Θ on that number.

Let's generate a sequence of the values of i:

$$i_1, i_2, i_3, i_4, ..., g(N)$$

$$1, 2, 4, 8, ...N$$

We're basically done. I did this above in the previous section. We know there are $\Theta(\log N)$ terms here. So, that's it. The work this Java function does is simply how many iterations there are which is $\Theta(\log N)$. Done.

Summary of single loop Java methods

We learned a strategy for approaching these.

1. Figure out the last value of i as a function of N. Call this $g(N)$.
2. Figure out the increment function of i.
3. Figure out the work done per iteration as a function of i. Call this $f(i)$.

Once you have the above, you simply write out the summation as: $f(i_1) + f(i_2) + f(i_3) + f(i_4) + ... + f(g(N))$

You figure out what $i_1, i_2, i_3, i_4, ...$ are by looking at the increment function for i. Then, plug in everything you know and then simplify the sum. Voila, that's it.

We also saw that if $f(i) = 1$ (i.e. the work done inside the for-loop is constant with respect to i), then we can just figure out how many elements are in the sequence $i_1, i_2, i_3, i_4, ...$ and put a Θ on that.

After a few of these problems, you'll likely be able to look at a single loop and instantly tell what its runtime is. That's good!

You might think that I overcomplicated things here. That there are shortcuts. And you're right. There are shortcuts. But there are so many shortcuts. There is no way I could possibly detail every shortcut you can take in this guide, nor is it that helpful for you as someone learning asymptotics. The goal of this is to give you **strategies** to figure out the answer yourself. You'll see in the next section when we do double loops that this method pays off and is really easy to extend.

You should try a few functions before moving onto the next section. Try making your own functions with different f, g functions and different ways to increment i. You *might* run into summations that you don't know how to simplify, and you should either ask a TA/friend/post on Ed or just leave it. For example, you might get a summation like: $\sqrt{1} + \sqrt{2} + \sqrt{3} + ... \sqrt{N}$

and you can just leave it like that. I don't know how to simplify that and it's out of the scope of this class.

Double loops

I made this a separate section but really it's no different than single loops. It is a bit more involved, so I want to go over them with you. Firstly, let me show you a few examples of double loops:

```
public void dLoop1(int N) {
        for (int i = 1; i < N; i = i * 2) {
                for (int j = 0; j < 10; j = j + 1) {
                        System.out.println(i + j);
                }
        }
}
```
Copy

Pretty standard. Let's check out another one.

```
public void dLoop2(int N) {
        for (int i = 1; i < N; i = i + 1) {
                for (int j = 1; j < i; j = j * 2) {
                        int[] x = new int[j];
                }
        }
}
```

And one more:

```
public void dLoop3(int N) {
        for (int i = 1; i < N * N; i = i + 1) {
                for (int j = 1; j < Math.sqrt(i); j = j + 1) {
                        int[] x = new int[j];
                }
        }
}
```

Firstly, do not fall down the oldest trick in the book. Do not multiply the number of iterations of the outer loop by the number of iteration of the inner loop. That's almost never what you're supposed to do. It's a common mistake many students make because they try taking a shortcut. Shortcuts are nice, but don't use them unless you're sure of how to use it!!

With that out of the way, let's talk about how you do these. It's literally the same as single loops:

> 1. Figure out the last value of i as a function of N. Call this $g(N)$.
> 2. Figure out the increment function of i.
> 3. Figure out the work done per iteration as a function of i. Call this $f(i)$.
>
> Once you have the above, you simply write out the summation as: $f(i_1) + f(i_2) + f(i_3) + f(i_4) + ... + f(g(N))$ You figure out what $i_1, i_2, i_3, i_4, \ldots$ are by looking at the increment function for i. Then, plug in everything you know and then simplify the sum. Voila, that's it.

Here, $f(i)$ is the work of the inner loop as a function of i!!

So, let's make steps for ourselves:

1. Figure out the work done of the inner loop as a function of i. Call this $f(i)$.
2. Take this $f(i)$, and use it to find the work the outer loop does. Put Θ on it.

Note that i is held fixed within the inner loop. So N is to i as i is to j. Hopefully that makes sense. What I mean is that when we figure out the asymptotic runtime of the outer loop, we do it with respect to N and i is the variable that increments. When we figure out the work done in the inner loop, we do it with respect to i and j is the variable that increments.

That's it. Let's see it in action with `dLoop1`

Double Loop 1

```
public void dLoop1(int N) {
        for (int i = 1; i < N; i = i * 2) {
                for (int j = 0; j < 10; j = j + 1) {
                        System.out.println(i + j);
                }
        }
}
```

Copy

Step 1 is to find the work done of the inner loop as a function of i. One neat little trick is to just isolate your attention to just look at the inner loop:

```
for (int j = 0; j < 10; j = j + 1) {
        System.out.println(i + j);
}
```

Copy

Do you notice anything peculiar about the inner loop? It has an i, but we just print it. So, the work done of the inner loop as a function of i is ... constant! Thus, $f(i) = 1$.

Now, we see that it's basically the same as `loop3` above, and we get a final answer of $\Theta(\log N)$. Next one.

Double Loop 2

```
public void dLoop2(int N) {
        for (int i = 1; i < N; i = i + 1) {
                for (int j = 1; j < i; j = j * 2) {
                        int[] x = new int[j];
                }
```

Copy

```
        }
    }
```

A bit more complicated. Now i is in the ending condition of the inner-loop.

Remember, we just isolate ourselves to the inner loop.

```
for (int j = 1; j < i; j = j * 2) {
        int[] x = new int[j];
}
```

Copy

And we want to find the work done of this inner loop as a function of i. Not the asymptotic runtime of this inner loop, but the work done (i.e. not a Θ bound).

This is simply a single loop! We know how to do those! I'll skip to the last step, but you should do all the steps to test your understanding and get more practice.

You'll get this summation:

$$0 + 1 + 2 + 4 + 8 + ... + i$$

We know this summation is $\Theta(i)$.

But I said not to take a Θ of this since we want the work done.

This is where things get a little bit weird.

We can pretend that the work done is i, so we can pretend that

$$0 + 1 + 2 + 4 + 8 + ... + i$$

evaluates to i. It does not (clearly). But, since we're doing asymptotics, we can do weird things like this. Bear with me for now.

So $f(i) = i$.

Now we do the single loop stuff and get this summation:

$$1 + 2 + 3 + 4 + ... + N$$

You've seen this many times now. It's simply $\Theta(N^2)$.

So `dloop2` has an asymptotic runtime of $\Theta(N^2)$.

Back to the weirdness of earlier. We said that since $f(i) \in \Theta(i)$ that $f(i) = i$. This is not generally true, but for asymptotics we can just say this. If $f(i)$ was actually $10i$, we'd get the same answer. Go ahead, plug it in if you don't believe me. So this is a **shortcut**.

Double Loop 3

Final one:

```java
public void dLoop3(int N) {
        for (int i = 1; i < N * N; i = i + 1) {
                for (int j = 1; j < Math.sqrt(i); j = j + 1) {
                        int[] x = new int[j];
                }
        }
}
```

Copy

Step 1 is to find the work done of the inner loop as a function of i.

```java
for (int j = 1; j < Math.sqrt(i); j = j + 1) {
        int[] x = new int[j];
}
```

Copy

EDIT: in an earlier (wrong) version of this guide, the update was instead `j = j * 2` which lead to a weird sum that you're not expected to know how to simplify (I don't know how to simplify it either).

I'll again skip the steps and go straight to the summation, but you should do it from the start. Practice makes better.

$$1 + 2 + 3 + 4 + ... + \sqrt{i}$$

Which we know is $\Theta((\sqrt{i})^2) = \Theta(i)$, and we'll do the same shortcut to say that $f(i) = i$.

Now, with the outer loop we'll use this function $f(i) = i$ and we get the summation:

$$1 + 2 + 3 + 4 + ... + N^2$$

This is an arithmetic summation, so we take the last term, square it, and throw a Θ on it. Thus, `dLoop3` has a running time that is $\Theta((N^2)^2) = \Theta(N^4)$.

I hope you see that this is really not that bad. It's just sticking to the rules.

Summary of double loop Java methods

We learned a strategy for approaching these.

1. Figure out the work done in the inner loop as a function of i. Call this $f(i)$.
2. Use this $f(i)$ with the outer loop and the same exact process

Unlike singe loops, you'll likely **not** be able to look at a double loop and instantly tell what its runtime is. That's OK. Whenever I do double loops I always write it out since it's so easy to make. a mistake if you try to do it in your head. The process is tried and tested, so just practice using it and you'll always get these ones right.

You should now appreciate the way we did single loops. It made double loops so much easier. You could even do triple, quadruple, and more loops. But … that's super annoying. I'm not saying it's out of scope for any exam since it's basically the same thing as a double loop, I'm just saying it's annoying.

You should try a few functions before moving onto the next section. Just like with single loops, try making your own functions with different f, g functions and different ways to increment i. You're more likely to run into summations that you don't know how to simplify, and you should either ask a TA/friend/post on Ed or just leave it.

That's it for nested loop asymptotics questions. You are now certified to do loop problems and teach your friends too as that is the absolute best way to learn. Next up is recursion!