CS 231
Naomi Nishimura

Math guide

As students in the course have varying amounts of background in mathematics, this guide and the math session have been designed for those who could use strengthening of various mathematical concepts used in the course as well as those who may wish to see a deeper explanation of some of the foundational material used in the course.

1 Floors and ceilings

When we are making calculations using the number of items in a list, vertices in a graph, or other such values, it is important to ensure that the number we are using is an integer. In particular, when we are splitting an integer number of values into two or more groups of roughly the same size, we might need a way to express "almost half" as an integer.

If we have a list of n numbers that we wish to split into two pieces, then using $\frac{n}{2}$ as the size of each piece only works if n is even. If instead n is odd, then one piece will have $\frac{n}{2} - \frac{1}{2} = \frac{(n-1)}{2}$ numbers and the other will have $\frac{n}{2} + \frac{1}{2} = \frac{(n+1)}{2}$ numbers, for a total of $\frac{(n-1)}{2} + \frac{(n+1)}{2} = \frac{2n}{2} = n$ numbers.

To express both the even and the odd cases at the same time, we can refer to the sizes as $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$, the *floor* and *ceiling* of $\frac{n}{2}$, respectively. The floor and ceiling functions produce the nearest integer below or above, respectively, what is enclosed in the symbols. Thus, when n is even, $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = \frac{n}{2}$, since $\frac{n}{2}$ is already an integer. When n is odd, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2} - \frac{1}{2}$, as that is the closest integer below $\frac{n}{2}$ and $\lceil \frac{n}{2} \rceil = \frac{n}{2} + \frac{1}{2}$, as that is the closest integer above $\frac{n}{2}$.

Because floors and ceilings "move" a value by less than one to reach the nearest integer, we know that $\frac{n}{2} - 1 < \lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$ and that $\frac{n}{2} \leq \lceil \frac{n}{2} \rceil < \frac{n}{2} + 1$. In particular, if we are using order notation, we can drop the floors and ceilings as we are only shifting the value by a constant.

Stated in another way, we can consider the two cases, even and odd, as depending on the remainder obtained when dividing n by 2. When n is even, the remainder is zero, expressed as $n \bmod 2 \equiv 0$. When n is odd, the remainder is one, expressed as $n \bmod 2 \equiv 1$.

2 Counting possibilities

At times we will count numbers of permutations, combinations, and subsets of values, as well as of other types of information.

A *permutation* is an ordering of a collection of values. To count the number of possible orderings, we first observe that if there are n values in total, there are n choices for the first value in an ordering. The second value in the ordering can then be any of the remaining $n-1$ values in the collection, the third value can be any of the remaining $n-2$ values, and so on, for a total of $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$.

A *combination* is a way of selecting items from a collection, where the order does not matter. The number of k-combinations of n elements is the *binomial coeffient* $\binom{n}{k}$, pronounced "n choose k":

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$$

.

The following facts about binomial coeffients might be useful:

- $\binom{n}{k} = \binom{n}{n-k}$
- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- $\sum_{k=0}^{n} \binom{n}{k} = 2^n$

If instead you wish to count the number of subsets of a set, you can think of each subset as being represented by a string of length n, where n is the number of elements in the set, and the character in position i represents element i. Each string is made of 0's and 1's, where a 1 indicates that the corresponding element is in the set and a 0 indicates that the element is absent. For example, if the set is $\{a, b, c\}$, then the string 110 corresponds to the set $\{a, b\}$, the string 010 corresponds to the set $\{b\}$, and the string 000 corresponds to the empty set.

To count the number of subsets of a set of size n, we need to count the number of different strings of length n. To do so, we observe that there are two choices (0 and 1) for the item in the first position, two choices for the second position, and so on, for a total of $2 \cdot 2 \cdots 2 = 2^n$ choices.

We can use the same idea to consider the number of different ways to assign each of n items to a category, where there are k categories. Here we can represent an assignment as a string of length n, this time one in which each position corresponds to one of the n items and each character is one of the k possible categories. In this case there are k choices for the first position, k for the second, and so on, or a total of $k \cdot k \cdots k = k^n$ choices.

3 Summation

When the cost of bodies of loops differ at each iteration, instead of taking the product of the number of iterations and the cost of each iteration, we instead calculate the summation of a series of terms. Although in general such calculations can become quite complicated, in this course we will typically have relatively simple sums to calculate.

An *arithmetic sequence* is a sequence of numbers in which each number differs from the previous number by the addition of some fixed quantity. The simplest one, in which the quantity is 1, can be expressed as follows:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

In a *geometric sequence*, each number in the sequence differs from the previous number by the multiplication of some fixed quantity. The following sum is a simple example:

$$\sum_{i=1}^{n} c^{i-1} = 1 + c + c^2 + c^3 + \cdots + c^{n-1} = \frac{(1-c^n)}{(1-c)}$$

Sometimes you will remove a term from a sum, such as removing the term for $i = 1$ in the previous sum to obtain:

$$\sum_{i=2}^{n} c^{i-1} = c + c^2 + c^3 + \cdots + c^{n-1} = \frac{(1 - c^n)}{(1 - c)} - 1$$

When using sums, you can easily make a substitution of variables, such as setting $j = i - 1$ to show that

$$\sum_{i=1}^{n} c^{i-1} = \sum_{j=0}^{n-1} c^j$$

.

4 Bounding quantities

In determining a lower bound or an upper bound, at times we will simplify the analysis by determining an inequality. For example, if we have a quantity x that we wish to show is at most z, we can do so by showing that $x \leq y$ and that $y \leq z$ for a good choice of y. (We might in turn find an intermediate quantity to be able to show the relationships between x and y and y and z.) To make matters more complicated, since we are using order notation, we might not have a specific z in mind, as we don't care too much about the constants involved.

Here are a few simple observations that might help in determining inequalities:

- If you already know that $x \leq y$, you can also conclude that $x \leq ya$ for $a \geq 1$ and that $x \leq y + b$ for $b \geq 0$.
- Similarly, if you already know that $x \leq y$, you can also conclude that $x/c \leq y$ for $c \geq 1$ and $x - d \leq y$ for $d \geq 0$.

For example, we can form an upper bound on $n!$ by replacing each of the n factors in $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ by n, since n is at upper bound on each factor. We then know that $n! \leq n \cdot n \cdot n \cdots n \cdot n = n^n$, or $n! \in O(n^n)$. To see that $\binom{n}{k} \in O(n^k)$, we observe that we can form an upper bound by ignoring the denominator $(k(k-1)\cdots 1)$ and replacing each factor in the numerator $(n(n-1)\cdots(n-k+1))$ by n.

We can use these techniques to express the sum of the arithmetic sequence $\sum_{i=1}^{n} i$ in order notation (ignoring for the moment that we have already seen an exact solution).

To obtain an upper bound, we observe that each term is of size at most n; if we replace each term by n, we obtain the sum of n n's for a total of n^2. For the values of $c = 1$ and $n_0 = 1$, we have shown that the value is in $O(n^2)$.

To obtain a lower bound, we will first remove some of the terms and then replace each remaining term by a value smaller than or equal to the term. By removing the smaller terms, we can show that $\sum_{i=1}^{n} i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i$, for a total of at least $\lceil \frac{n}{2} \rceil$ terms. Moreover, since each term is of value at least $\lceil \frac{n}{2} \rceil$, the total will be at least $\lceil \frac{n}{2} \rceil^2$. We can obtain a lower bound by replacing $\lceil \frac{n}{2} \rceil$ by $\frac{n}{2}$ to obtain $n^2/4$. For the quantities $c = 1/4$ and $n_0 = 1$, we have shown that the value is in $\Omega(n^2)$, which combined with the previous result shows the inclusion in $\Theta(n^2)$.

5 Logarithms

All logarithms used in the course are logarithms base 2, which arise naturally in computer science; because of their prevalence, the notation log is typically used instead of $\log_2$. Intuitively, the logarithm of a number base 2 is the number of times the number can be divided by 2 before 1 is reached. Unless n is a power of 2, $\log n$ will not be an integer; if you need an integer value, you might need to use floors or ceilings, discussed in Section 1.

The ceiling of the logarithm base two of a number is the length of its binary representation. To see why, you can observe that just like the digits in a decimal number give the quantities of powers of 10, the bits in a binary number give the quantities of powers of 2. When there are k bits, the number represented are 0 through $2^k - 1$, for a total of 2^k different numbers. To see why the largest value is $2^k - 1$, we observe that it is represented by k 1's, where each 1 corresponds to a power of 2. That is, the largest value is the sum (using the formula seen in Section 3) $\sum_{i=1}^{k} 2^{i-1} = \frac{(1-2^k)}{(1-2)} = 2^k - 1$.

When discussing lower bounds, we will take the logarithms of various functions. Here are some formulas you might wish to use in manipulating logarithms:

- $\log xy = \log x + \log y$
- $\log(x/y) = \log x - \log y$
- $\log x^k = k \log x$

To show that $\log n! \in \Theta(n \log n)$, we prove below that $\log n! = \sum_{i=2}^{n} \log i$ and then use the result of Question 3 in the math session. By definition $n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$, and since multiplying by 1 does not change the value, we can also write $n! = n \cdot (n-1) \cdot \cdots 2$. We then use the first formula above to conclude that

$$\log n! = \log(n \cdot (n-1) \cdot \cdots \cdot 2) = \sum_{i=2}^{n} \log i.$$

6 Trees

In the course, trees appear implicitly in computations using divide-and-conquer as well as explicitly as decision trees and search trees. Assuming that the cost of each decision can be determined in constant time, the cost associated with a decision tree is its *height*, that is, the number of edges in the longest path from the root to a leaf. Various calculations make use of the relationships between maximum or minimum height of a tree and the maximum or minimum number of nodes or leaves it contains.

To aid in our calculations, we define the *level* of a node as follows: the root is at level 0, and each other node is at level one greater than its parent. For a binary tree, we can count the maximum number of nodes in a tree of height h by observing that if we place a node in each possible location, the number of nodes at each level is exactly double the number of nodes at the previous level, with 2^i nodes at level i. The total is then $\sum_{i=0}^{h} 2^i = \sum_{j=1}^{h+1} 2^{j-1} = 2^{h+1} - 1$; replacement of variables is discussed in Section 3 and the solution to the second summation is

discussed in Section 5. The number of leaves in this tree is the number of nodes at level h, or 2^h. For ℓ the number of leaves, this tree has height $\log \ell$.

Suppose instead that we focus on the number of leaves ℓ, and wish to create the tree of minimum height that has exactly ℓ leaves. When ℓ is a power of two, we know the answer from our previous discussion. But if ℓ is not a power of two, we cannot place all leaves in the same level in the tree. Instead, we can fit all the leaves into the last two levels of a tree of height $\lceil \log \ell \rceil$. It is for this reason that running times of algorithms that can be expressed as binary trees (and in which an instance is repeatedly divided in half), such as in binary search or in mergesort, contain a $\log n$ factor.

7 Analyzing correctness

When we reason about the correctness of an algorithm, we will typically be trying to prove that a statement about the algorithm is true. Below, we will use the term *statement* to refer to an expression that might be either true or false. Using the familiar concepts of *and*, *or*, and *not*, we can construct statements from other statements. For example, for a statement A, the statement "not A" (or "A is false") is the *negation* of A; when A is true, its negation is false, and when A is false, its negation is true. Similarly, whether or not "A *and* B" or "A *or* B" is true depends on which, if either, of A and B are true.

We can also form a new statement out of other statements to form an *implication*, where the statement "if A, then B" means that if A is true, then B is true. We can form the *converse* of an implication "if A, then B" by swapping A and B to form "if B, then A". It is not hard to see that an implication may be true without its converse being true (such as when A is "it is a cat" and B is "it is an animal"). Not to be confused with the converse is the *contrapositive* of an implication, where the contrapositive of "if A, then B" is the implication "if not B, then not A". We will show in Section 8.1 that the contrapositive of a statement is true exactly when the statement itself is true (statements related in this way are *equivalent*).

To be able to express a statement about more than one entity, we will typically make use a *predicate*, which is an expression that refers to at least one variable. For example, the predicate $P(x)$ could be the statement "The number x is odd" where the variable is taken from the set of integers; in such a case $P(1)$ would be true and $P(2)$ would be false.

Using predicates, we can define *existential statements*, of the form "There exists an x in the set S such that $P(x)$ is true," and *universal statements*, of the form "For every x in the set S, $P(x)$ is true." To prove that an algorithm is correct, we will typically prove a universal statement, where S can be viewed as the set of instances and $P(x)$ can be viewed as the statement that the algorithm performs correctly on instance x. In contrast, to prove that an algorithm is not correct, we will typically prove an existential statement, where S is again the set of instances but now $P(x)$ is the statement that the algorithm fails to perform correctly on instance x. As noted in Section 8, we use different proof techniques for different types of statements.

8 Proof techniques

At times you will be given a statement to either prove or disprove, that is, by showing that it is true or that it is false, respectively. Since a statement is true whenever its negation is false, you can prove that a statement is true by showing that its negation is false, or you can prove that a statement is false by showing that its negation is true.

We observe that the negation of an existential statement is a universal statement and vice versa. For example, the negation of "There exists an x in the set S such that $P(x)$ is true" is "For all x in the set S, $P(x)$ is false." Notice that the negation of $P(x)$ is used here. Similarly, the negation of "For every x in the set S, $P(x)$ is true" is "There exists an x in the set S such that $P(x)$ is false". Again, the negation of $P(x)$ is used.

At times we will want to prove both an implication and its converse, that is, both "if A, then B" and "if B, then A." Often such a pair of statements is expressed as "A if and only if B," where "A only if B" is equivalent to "if A, then B." Since each statement is equivalent to its contrapositive, we could also prove "A if and only if B" by proving both "if A, then B" and "if not A, then not B."

In our proofs, we will make use of the powerful tool *modus ponens*: if A is true and "if A, then B" is true, then B is true. Parents expect young children to understand modus ponens, for A some type of undesirable behaviour and B some type of punishment.

8.1 Proof by contradiction

To show that a statement A is true, we can form a proof by supposing that A is false and showing that this supposition leads to a contradiction, such as showing that something we know to be true is false. We then know that our assumption was wrong, that is, that "A is false" is incorrect. This result leads to the conclusion that A is in fact true (since if it is not false, it must be true).

For example, suppose we wish to prove that the following is true: "if $x > y$ and $y > z$, then $x > z$." We first suppose that it is not true, namely, that $x > y$, $y > z$, and $z \geq x$. We can then show that since $y > z$ and $z \geq x$, $y > x$. However, this contradicts the fact that $x > y$. Thus, we can conclude that our original statement was true.

As another example, we will show that a statement is equivalent to its contrapositive. We first suppose that it is not true, namely, that "if A, then B" is true but "if not B, then not A" is not true. Since, by our assumption, "if not B, then not A" is not true, then we know that "if not B, then A" is true. Thus, if "not B" is true, A is true. But we also know that "if A, then B" is true, so if A is true, B must be true. This leads us to the conclusion that if "not B" is true, then B is true, which is a contradiction since B cannot be both false and true at the same time.

8.2 Proof by (counter)example

The typical way to prove an existential statement is to demonstrate a choice of x in S such that $P(x)$ is true. For example, to prove the existential statement "There exists a course at UW with the digit 3 in its course number," we just need to give one example of such a course, such as CS 231.

Because the negation of a universal statement is an existential statement, we can also use a single example to show that an algorithm is not correct. We can view the universal statement as "For every instance, the algorithm gives the correct answer," and its negation as "There exists an instance on which the algorithm does not give the correct answer." Thus, to prove that an algorithm is not correct, it is sufficient to find a single instance on which the algorithm fails.

8.3 Proof by induction

Proving a universal statement is usually more challenging than proving an existential statement, as instead of having to demonstrate something about a single x, one needs to prove that something that holds for every possible x. When the set S consists of all possible instances, we are asking for a proof for an infinite number of x's. One proof method is to choose an arbitrary x and show that the statement holds for x. Since there was nothing special about x, the same argument holds for all values x and hence for all of S.

When the set S is the set of all possible instance sizes, one way to handle such a proof is to use *induction*. Although we will not be using any proofs by induction in the course, the idea of induction is implicit in the iteration and substitution methods, so it is worth discussing. You will find the concepts used in induction to be familiar, as they are the same ideas behind recursive definitions and recursive functions.

The universal statement "For every x in S, $P(x)$ is true" can be viewed as an infinite sequence of statements $P(0)$, $P(1)$, $P(2)$, and so on. We wish to show that each of these statements is true. The principal behind induction is that if you can prove that $P(0)$ is true (the *base case*) and you can prove "For any i, if $P(i)$, then $P(i+1)$" (the *induction step*), then $P(x)$ must be true for all x. In particular, you can use modus ponens, $P(0)$, and "If $P(0)$, then $P(1)$" to conclude that $P(1)$ is true. You can then use modus ponens, $P(1)$, and "If $P(1)$, then $P(2)$" to conclude that $P(2)$ is true. The same reasoning can be used for any $P(x)$. More generally, the same idea works if there are multiple base cases and more complicated implications showing that $P(i)$ being true (perhaps for more than one value of i) implies that $P(j)$ is true (perhaps for $j > i+1$).

The "if $P(i)$" part used in the induction step, known as the *induction hypothesis*, is equivalent to assuming that P holds for smaller values in order to prove that P holds for larger values. The mental gymnastics needed to understand induction is similar to that needed to understand recursive functions. In writing the function you have one or more non-recursive base cases; for each recursive case, you are assuming that the algorithm works correctly on entities closer to the base case in order to reason that the algorithm works correctly on entities farther from the base case.

9 Probability

Towards the end of the course we will consider randomized algorithms, in which some of the steps taken are chosen at random. To analyze such algorithms, we need to be able to discuss in a precise way how the randomness has an impact on the running time (if at all) and on the correctness (if at all).

There are various ways of expressing results of a randomized algorithm, including expected behaviour (discussed more below) and high probability bounds (statements of the type "This happens with high probability", where "high" is defined in some way). In this course we will focus only on the former.

To determine expected behaviour, one needs to have a *probability distribution*, which is an assignment of probabilities to possible events such that the sum of the probabilities of all events is 1. For example, if you have a fair coin, you would assign $\frac{1}{2}$ to the event "Heads" and $\frac{1}{2}$ to the event "Tails". For a coin that has been altered, you might assign $\frac{3}{4}$ to the event "Heads". In that case, you would have to assign $\frac{1}{4}$ to "Tails" for the sum of the probabilities to be 1.

To determine the *expected value*, you determine the sum over all events of the value of the event (for our purposes, typically a running time or a number of objects) multiplied by the probability of the event. If you were to receive a prize worth \$100 for "Heads" and a prize worth \$8 for "Tail", using the fair coin the expected value would be $100 \cdot \frac{1}{2} + 8 \cdot \frac{1}{2} = 54$, but using the altered coin the expected value would be $100 \cdot \frac{3}{4} + 8 \cdot \frac{1}{4} = 77$.

For situations in which the randomness has an impact on the correctness of decision problems, we distinguish between *false positives* (the algorithm produces "yes" for a no-instance) and *false negatives* (the algorithm produces "no" for a yes-instance). At times an algorithm may behave differently on yes-instances and no-instances. For example, if it can produce false positives but not false negatives, we know that it might answer "yes" for a no-instance but will always gives the correct answer for a yes-instance. Thus, the answer "yes" could be provided for any instance, but "no" will always guarantee a no-instance.