CMSC 420: Preliminaries

Justin Wyss-Gallifent

January 26, 2024

1 Review of Sums

Here are some sums which will arise frequently in this course:

$$
\begin{aligned}
\sum_{i=1}^{n} 1 &= n \\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \\
\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\
\sum_{i=0}^{n} r^i &= \frac{r^{n+1}-1}{r-1} \\
\sum_{i=0}^{n} 2^i &= 2^{n+1}-1 \\
\sum_{i=1}^{n} i2^i &= (n-1)2^{n+1}+2
\end{aligned}
$$

Note: Most of these should be familiar. the only one that might not be is the final one.

Proof. We have:

$$
\begin{aligned}
\sum_{i=1}^{n} i2^i &= 2\left[\sum_{i=1}^{n} i2^i\right] - \left[\sum_{i=1}^{n} i2^i\right] \\
&= \left[\sum_{i=1}^{n} i2^{i+1}\right] - \left[\sum_{i=1}^{n} i2^i\right] \\
&= \quad \left[1\cdot 2^2 + 2\cdot 2^3 + ... + (n-1)2^n + n2^{n+1}\right] \\
&\quad - \left[1\cdot 2^1 + 2\cdot 2^2 + ... + (n-1)2^{n-1} + n2^n\right] \\
&= n2^{n+1} - 2^n - 2^{n-1} - ... - 2^2 - 2^1 \\
&= n2^{n+1} - (2^n + 2^{n-1} + ... + 2^1) \\
&= n2^{n+1} - (2^{n+1} - 2) \\
&= (n-1)2^{n+1} + 2
\end{aligned}
$$

$\mathcal{QED}$

These sums will appear frequently so it's useful to know them.

Example 1.1. Suppose a tree of height h has the property that it is perfect, that each node has m children and each node contains $m-1$ keys. How many keys are there?

Well level-by-level we have:

Level	Nodes	Keys
0	1	$m-1$
1	m	$m(m-1)$
2	m^2	$m^2(m-1)$
$\vdots$	$\vdots$	
h	m^h	$m^h(m-1)$

Thus the number of keys equals:

$$\sum_{i=0}^{h} m^i(m-1) = (m-1)\frac{m^{h+1}-1}{m-1} = m^{h+1}-1$$

Note: This exact calculation will be relevant when we talk about B-trees in a few weeks.

Example 1.2. Consider the following pseudocode:

```
function mysum(n):
    sum = 0
    for i = 1 to n:
        sum = sum + 2
        if i is a power of 3:
            sum = sum + i
        end if
    end for
end function
```

Let's find a closed expression for the value of `sum` for any `n`.

Note that for each i we add 2 but when i reaches a power of 3 we add i. Thus our sum can be thought of as:

$$\underbrace{2+1}_{i=1} + \underbrace{2}_{i=2} + \underbrace{2+3}_{i=3} + \underbrace{2}_{i=4} + \underbrace{2}_{i=5} + \underbrace{2}_{i=6} + \underbrace{2}_{i=7} + \underbrace{2}_{i=8} + \underbrace{2+9}_{i=9} + \underbrace{2}_{i=10} + \ldots + \underbrace{???}_{i=n}$$

This may be rewritten:

$$\underbrace{2+2+...+2}_{n \text{ times}} + \underbrace{1+3+9+27+...}_{\text{how many?}}$$

It's not immediately obvious how many terms are in the second expression above but it's not hard to work out:

The second expression gains a term each time that i encounters a power of 3. For any i the highest power of 3 we have passed is $3^{\lfloor \log_3 i \rfloor}$ (for example when $i = 10$ the highest power we have passed is $3^{\lfloor \log_3 10 \rfloor} = 3^2 = 9$ and for example when $i = 30$ the highest power we have passed is $3^{\lfloor \log_3 30 \rfloor} = 3^3 = 27$) and hence the total sum may be rewritten as:

$$\begin{aligned}
\text{sum} &= (2n) + 1 + 3 + 9 + ... + 3^{\lfloor \log_3 n \rfloor} \\
&= 2n + \sum_{j=0}^{\lfloor \log_3 n \rfloor} 3^j \\
&= 2n + \left(\frac{3^{\lfloor \log_3 n \rfloor + 1}}{3-1} \right)
\end{aligned}$$

Note: Calculations like this will be relevant when we talk about amortized cost next week.

2 Review of Expected Value

Definition 2.0.1. Suppose an event may have outcomes $X \in \{x_1, x_2, ..., x_n\}$ with probabilities p_1, p_2, ..., p_n. Then the *expected value* is defined as:

$$E(X) = \sum_{i=1}^{n} p_i x_i$$

Example 2.1. Suppose we perform one operation on a data structure, either search, insert, or delete. Suppose there is a 10% chance we search, and search takes 5 seconds, there is a 70% chance we insert, and insert takes 10 seconds, and there is a 20% chance we delete, and delete takes 2 seconds. We therefore expect a single operation to take:

$$0.1(5) + 0.7(10) + 0.2(2) = 7.9 \,\text{seconds}$$

Example 2.2. Ignoring the keys in a binary tree and only looking at the structure, if we choose from amongst all binary trees with three nodes with each equally likely, what do we expect the height to be?

In this case we can actually draw all possible three-node binary tree structures and simply calculate. For larger trees though, this is not so easy and we need to bring some complicated calculations into the mix.

3 Review of Asymptotics

3.1 Comment

In general we'll give $\mathcal{O}$ rather than Θ or Ω. Even though in theory Θ is better, because $\Theta \Longrightarrow \mathcal{O}, \Omega$, usually the Ω aspect isn't that relevant and can be fairly challenging to prove.

Moreover we'll typically stick with best-case and worst-case situations and only touch on average-case when they are easy. What we'll see as we progress is that not only is the definition of average not often clear but even when it is made clear the associated calculations can be unbearable.

3.2 Definitions

Recall the definitions:

Definition 3.2.1. We say that:

$$f(x) = \mathcal{O}(g(x)) \text{ if } \exists\, x_0, C > 0 \text{ such that } \forall\, x \geq x_0\,,\; f(x) \leq Cg(x)$$

Definition 3.2.2. We have:

$$f(x) = \Omega(g(x)) \text{ if } \exists\, x_0, B > 0 \text{ such that } \forall\, x \geq x_0\,,\; f(x) \geq Bg(x)$$

Definition 3.2.3. We have:

$$\begin{gathered} f(x) = \Theta(g(x)) \text{ if } \exists\, x_0, B > 0, C > 0 \text{ such that} \\ \forall\, x \geq x_0\,,\; Bg(x) \leq f(x) \leq Cg(x) \end{gathered}$$

3.3 Goals

As a general rule, faster is better. Here is a list of some common time complexities in order from fastest to slowest along with some comments on each:

- $\mathcal{O}(1)$: This is the best we can have. An example would be popping an item from a stack or assigning a variable.
- $\mathcal{O}(\alpha(n))$: We'll see this later in the course. The function $\alpha(n)$ is the inverse of the *Ackerman function*. The Ackerman function grows unbelievably fast and so $\alpha(n)$ grows very slowly. In fact $\alpha(n) < 4$ for all $n \leq 10^{600}$ so for all reasonable n the Ackerman function is "almost constant".
- $\mathcal{O}(\lg n)$: Pretty fast. This turns out to be the target we'll generally go for if we can't hit $\mathcal{O}(1)$ and is often thought of as "better than linear". This shows up doing a binary search, for example.
- $\mathcal{O}(n)$: This is very common, especially when working on lists. An example of this is a simple linear search.

- $\mathcal{O}(n \lg n)$: This is one of the most common time complexities and we've seen it a lot! It arises as the average case in many search algorithms such as merge sort and heap sort. We also know that this is the best our worst-case can get we can do when using a comparison-based sorting algorithm on a list.
- $\mathcal{O}(n^k)$ for $k \geq 2$: This arises in many non-optimized algorithms which work on lists and arrays. For lists, an example would be bubble sort. For matrices, an example would be matrix multiplication. Often this is fine for fairly small data sets.
- $\mathcal{O}(2^n)$: Things are starting to get bad here and algorithms which have these time complexities are usually only good for small values of n. One example is the classic subset problem whereby we are given a set of integers and need to determine if there is a subset which sums to 0.

4 Elements, Keys, Values, Etc.

Typically when we are storing single items such as integers or strings we will just call them *elements*, such as the elements in a list.

However for most of this course the idea is that we have data, known abstractly as the *value* (but it may be many values), which is indexed by a *key*. An example might be all your grades (those would be the value) indexed by your UID (the key).

We thus storing key-value pairs but the key is the critical thing in the sense that we might search for a key (implicitly looking for the associated value), delete a key (and its associated value), or insert a key (and some associated value).

Oftentimes therefore we'll just work with the keys but the implicit understanding is that there are relevant values attached to these.

5 Operations on Data

5.1 Dictionary Operations

For much of this course we will be interested in what are known as *dictionary operations*. This term includes the three classic operations:

- Search: A key is given and we wish to find it in a data structure. Typically we're looking for something associated to the key, for example an associated value, the location of the key, how many steps it takes to find it, and so on.
- Insert: A key and perhaps some associated value are given and we wish to insert it into the data structure.
- Delete: A key is given and we wish to delete it from the data structure.

5.2 Other Operations

Later in the course we will look at some non-dictionary operations such as:

- Range queries: Find all keys (or an associated value) between two given keys.
- Proximity queries: Find a key (or an associated value) in the data structure closest to a given key.
- Balancing operations: For example can we balance a tree?
- Node finding operations: Can we find nodes which have certain properties?

6 Familiar Data Structures

6.1 Introduction

We don't come into this course completely ignorant of data structures. Here are are a few familiar ones along with some comments on each.

6.2 Lists

Lists are of course one of the most common data structures but not a lot of thought is often given into how they are implemented behind the scenes.

When we have a list of length n there are some details which need to be ironed out. For example are the indices the keys and the list elements the values? Or perhaps the indices are just indices and the list elements contain keys and values and if that's the case, perhaps the keys are sorted and perhaps not.

Example 6.1. If the indices are keys and the list elements are values then we might have something like $A = [5, 4, -3, 1, 4]$. In such a case:

- Reading or writing a value associated to a key is worst-case $\mathcal{O}(1)$.
- Searching for a value is worst-case $\mathcal{O}(n)$.
- Appending, inserting, or deleting are trickier than we might imagine, depending on how the system handles these. For example appending and inserting might require memory reallocation which then takes time. We'll discuss this further when we talk about amortized analysis.

Example 6.2. If the indices are just indices and the list elements contain integer keys and string values and if the list elements are sorted by the integer keys then we might have something like $A = [[4,'cat'], [10,'muppet'], [42,'justin']]$. In such a case this differs from the above in that reading, writing, and searching requires finding the list element associated to a key and this is worst-case $\mathcal{O}(\lg n)$ via binary search.

If the list elements are not sorted then this process is worst-case $\mathcal{O}(n)$.

6.3 Linked Lists

When we have a linked list with n elements there are no indices (unlike lists) and typically they key (and any associated values) are stored in the linked list nodes. In such a case:

- Searching for a value is worst-case $\mathcal{O}(n)$.
- Reading or writing a value associated to a key is worst-case $\mathcal{O}(n)$.
- Insertion and deletion are each worst-case $\mathcal{O}(n)$ in the sense that we typically have to search for some location within the list to insert into or delete from.

6.4 Max heaps

Max heaps are discussed at length in CMSC351. A max heap will typically have nodes which consist of a key and associated values. In such a case since a max heap is a complete binary tree searching is worst-case $\mathcal{O}(\lg n)$, as are insertion and deletion.

6.5 Stacks

Stacks are quite a bit different because they do not facilitate insertion and deletion from internal locations. Instead we only have:

- Push: This is typically worst-case $\mathcal{O}(1)$. We say "typically" because this is perhaps not as obvious as we might like as it can depend greatly on the implementation of the stack. If it is implemented as a standard list then reallocation comes into play as with reguar lists but if it is implemented as a linked list then there are no such issues.
- Pop: Same result as push.
- Searching is worst-case $\mathcal{O}(n)$ since we might have to go through the entire stack to find our desired key.

6.6 Queues

Queues are typically implemented as linked lists because if we use a standard list the dequeue operation takes some fiddling and because we also have to deal with reallocation costs.

Assuming we use a linked list:

- Enqueue: This is worst-case $\mathcal{O}(1)$ with the same caveats as with stacks.
- Dequeue: This is worst-case $\mathcal{O}(1)$, ditto.
- Search: As with stacks.

7 Why So Many Trees?!

One observation (possibly a complaint!) about this course is that we study so many trees. This may seem frustrating but the reality of the situation is that trees are basically the most fundamental data structure other than simple lists, queues, etc.

As a general rule, trees handle data quickly while still being easy to understand, visualize, and code. In addition they are easy to modify which leads to the plethora of trees we study in this course.

I have however endeavored to add a variety of other data structures to keep things fresh!