CMSC 420: Preliminaries

Justin Wyss-Gallifent

January 28, 2023

1 Review of Sums .................................................. 2
2 Review of Expected Value ................................. 3
3 Review of Asymptotics ....................................... 3
   3.1 Comment .................................................. 3
   3.2 Definitions ............................................. 3
   3.3 Goals .................................................. 3
4 Elements, Keys, Values, Etc. ........................... 4
5 Operations on Data ........................................ 4
   5.1 Dictionary Operations ................................. 4
   5.2 Other Operations .................................... 5
6 Familiar Data Structures .................................. 5
   6.1 Introduction .......................................... 5
   6.2 Lists .................................................. 5
   6.3 Linked Lists .......................................... 5
   6.4 Max heaps ............................................ 6
   6.5 Stacks ................................................ 6
   6.6 Queues ............................................... 6
7 Why So Many Trees?! ...................................... 6
1 Review of Sums

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

\[
\begin{align*}
\sum_{i=1}^{n} 1 &= n \\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \\
\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\
\sum_{i=0}^{n} r^i &= \frac{r^{n+1} - 1}{r - 1} \\
\sum_{i=0}^{n} 2^i &= 2^{n+1} - 1 \\
\sum_{i=1}^{n} i^2 &= (n-1)2^{n+1} + 2
\end{align*}
\]

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

Proof. We have:

\[
\begin{align*}
\sum_{i=1}^{n} i^2 &= 2 \left[ \sum_{i=1}^{n} i^2 \right] - \left[ \sum_{i=1}^{n} i^2 \right] \\
&= \left[ \sum_{i=1}^{n} i^{2+1} \right] - \left[ \sum_{i=1}^{n} i^2 \right] \\
&= \left[ 1 \cdot 2^2 + 2 \cdot 2^3 + \ldots + (n-1)2^n + n2^{n+1} \right] \\
&- \left[ 1 \cdot 2^1 + 2 \cdot 2^2 + \ldots + (n-1)2^{n-1} + n2^n \right] \\
&= n2^{n+1} - 2^n - 2^{n-1} - \ldots - 2^2 - 2^1 \\
&= n2^{n+1} - (2^n + 2^{n-1} + \ldots + 2^1) \\
&= n2^{n+1} - (2^{n+1} - 2) \\
&= (n-1)2^{n+1} + 2
\end{align*}
\]

\(\square\)
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:

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

3 Review of Asymptotics

3.1 Comment

In general we’ll give \( O \) rather than \( \Theta \) or \( \Omega \). Even though in theory \( \Theta \) is better, because \( \Theta \Rightarrow O, \Omega \), usually the \( \Omega \) aspect isn’t that relevant.

3.2 Definitions

Recall the definitions:

**Definition 3.2.1.** We say that:

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

**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:

\[ f(x) = \Theta(g(x)) \text{ if } \exists x_0, B > 0, C > 0 \text{ such that } \forall x \geq x_0, Bg(x) \leq f(x) \leq Cg(x) \]

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:

- \( O(1) \): This is the best we can have. An example would be popping an item from a stack or assigning a variable.
- \( O(\alpha(n)) \): 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”.

- $O(\lg n)$: Pretty fast. This turns out to be the target we’ll generally go for if we can’t hit $O(1)$ and is often thought of as “better than linear”. This shows up doing a binary search, for example.
- $O(n)$: This is very common, especially when working on lists. An example of this is a simple linear search.
- $O(n \lg n)$: 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.
- $O(n^k)$ 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.
- $O(2^n)$: 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 - searching for a key,
inserting a key, and deleting a key.

5.2 Other Operations

Later in the course we will look at some non-dictionary operations such as range queries (find all values between two given values).

6 Familiar Data Structures

6.1 Introduction

We don’t come into this course completely ignorant of data structures. Here 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. Here are some list operations and time complexity comments:

• Read element: This is worst-case \( O(1) \).

• Write element: This is worst-case \( O(1) \). Note that this does not mean appending to the list, rather this means overwriting a value in the list.

• Append element: This is not obvious and depends highly on the architecture being used. This is typically because when a list is created some amount of space is allocated for it and if we append an element which results in a larger space requirement then the system needs to reallocate, and this takes time. We’ll actually work this out in more detail in the notes on amortized analysis.

• Insert element: This is not obvious for the same reason as appending.

• Delete element: Typically deleting an element is worst-case (and average-case) \( O(n) \) because we need to shift all remaining elements to the left.

6.3 Linked Lists

Linked lists are often better than lists primarily because insertion and deletion are faster. The trade off is that reading element is slower. Typically we have:

• Read element: This is worst-case (and average-case) \( O(n) \) because we may have to scan through the entire list to find our target element (such as - what is the fifth element in the linked list?).

• Write element: Same as read.

• Append element: Assuming we keep track of a pointer to the final element in the linked list then appending an element is simply \( O(1) \), involving creation of a node and updating of pointers.
• Insert element: If we have a specific target position in mind, such as - insert this into the fifth position - then the time complexity is worst-case $O(1)$ because all we are doing is creating a new node and patching up some pointers. However if we don’t have a specific target position then the best we can say is worse-case (and average-case) $O(n)$.

• Delete element: Same as insert.

### 6.4 Max heaps

Max heaps are discussed at length in CMSC351.

### 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 $O(1)$. 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 regular lists but if it is implemented as a linked list then there are no such issues.

• Pop: Same result as push.

### 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 $O(1)$.

• Dequeue: This is $O(1)$.

### 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!