CSC263 Notes

Data Structures and Analysis

https://github.com/ICPRplshelp/

Last updated January 17, 2023

1 Data Types, Data Structures

ADTs

- Specification
  - Objects we're working with
  - The operations (WHAT but not how)

Data structures

- Implementation (how)
  - Data
  - Algorithms

Analysis (runtime or complexity)

- Worst case
- Best case
- Upper bounds
  - $\mathcal{O}$
- Lower bounds
  - Ω
- Tight bounds
  - Θ

If we have algorithm A and input x, the runtime $t_A(x) =$ number of constant time operations independent of x.

Ultimately, we want a measure of running time that is a function of the input size. We have lots of inputs for each input size. So if we want to prove an upper bound when looking for the worst case running time:

- I need two functions to prove an upper bound
  - A simple algebraic expression
  - The running time
    * However, the pure runtime function's codomain isn't $\mathbb{R}^{\geq 0}$ – but rather, a list of running times. To turn it into a raw function that outputs $\mathbb{R}^{\geq 0}$, we can take the largest of the list I just described.

Worst case is just us narrowing down a bunch of possible runtimes to the worst one.

My upper bound will always be some value that is larger or equal to the worst case, and the lower bound must be below the worst case but not all the worst cases.

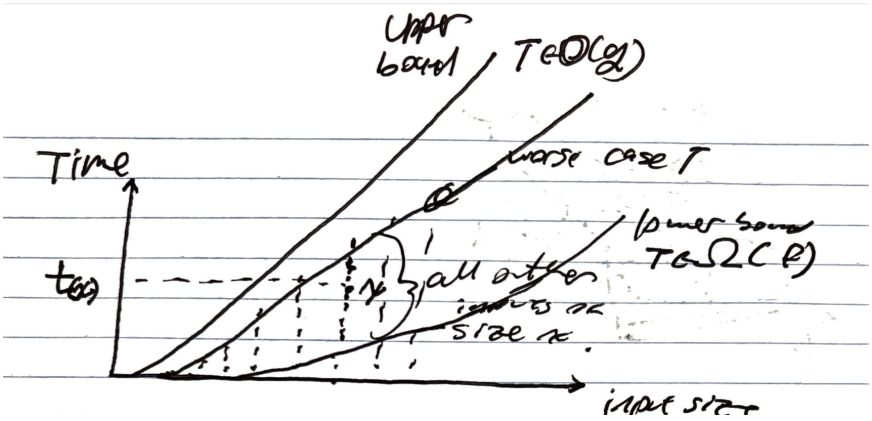

Figure 1: Upper and lower bounds of the worst case

1.1 Average-case Running Time

For a particular input size n, I have all these inputs x. We need to precisely define $S_n = \{\text{set of all inputs of size } n\}$. We need a probability distribution over our set of inputs.

- For each input $x \in S_n$, $t(x)$ is a random variable.
  - It's something that assigns a number to each element of our sample space. Now, S_n (a finite set) becomes a sample space.

Given a **discrete** probability distribution over S_n, $t(x) =$ the running time for input x. But what we'll get is that if I only know the input size, $t_n =$ **average-case running time for size n is:**

$$\begin{aligned} t_n &:= E[t(x)] \text{ over } S_n \\ &= \sum_{x \in S_n} t(x) \Pr(x) \end{aligned}$$

This summation is not as easy to figure out. There's a way we can get a particular value, but there is another way:

EX: The linear search algorithm

```
1 def LinSearch(L: LinkedList, x: T) -> Node | None:
2     """Pre: L is a linked list, x is a value
3     Post: return the node that contains x
4     or none otherwise"""
5
6     z = L.head
7     while z != None and z.key != x:
8         z = z.next
9     return z
```

Standard linked list search. Here's what we need to do:

1. Define our sample space (a family of sample space, one for each sample size):
    a. Let n be arbitrary.
    b. $S_n = \{\text{every input of size } n\}$
2. What should the sample space be? It shouldn't necessarily have infinitely many inputs with a given running time.
    a. The number of different possible running times I have is finite for this algorithm.
3. **INSIGHT.** One representative input for each possible behavior. Behavior $=$ running time.

a. S_3
$= \{([1, 2, 3], 1), ([2, 1, 3], 1), ([2, 3, 1], 1), ([2, 3, 4], 1)\}$

b. Alternatively, $S_n = \{([1, 2, \ldots, n], 1), ([1, 2, \ldots, n], 2), \ldots\}$

c. Which is $\{([1, 2, \ldots, n], x) : x \in [0, n] \wedge x \in \mathbb{N}\}$

4. The probability distribution becomes important. How do we decide how likely we want each input to be? How can we tell? That is a tricky question. What are we trying to do, and there's no obvious way to choose. In practice, in any kind of real-life scenario, if you want to judge how well an algorithm performs on average if you have some idea of what your real-life inputs are going to look like. If you have no information at all, where it is all abstract, then we'll just uniformly distribute.

   a. $\Pr([1, 2, \ldots, n], i) = \frac{1}{n+1}$ for $i = 0, 1, \ldots, n$

   b. Now we have this, we can calculate the expected value:

   c. $E[t(x)] = \sum_{(L, i) \in S_n} t(L, i) \cdot \Pr(L, i)$

   d. $= \sum_{i=0}^{n} t([1, 2, \ldots, n], i) \cdot \frac{1}{n+1}$

   e. When we're doing an average case, we cannot calculate an expected value with $\mathcal{O}$ expressions in there. We need a precise expression we can add up and average out. We need an **exact** expression

      i. Not in the sense that there's one right answer, but we need to fix a particular way of counting and count the same way for every input.

   f. There is one trick: pick some representative operation that we know if we count that, the number of representative operations is $\Theta(\text{runtime})$.

      i. In the example, the number of times `z.key == x` is run, which will be the **representative operation**

   g. Ignore the constant time operations. The thing that matters is the loop. The loop does a constant amount of work each operation.

   h. $= t([1, 2, \ldots, n], 0) \cdot \frac{1}{n+1} + \sum_{i=1}^{n} i \cdot \frac{1}{n+1}$

i. It's all algebra by this point. Do all of it, and you should end up with $\frac{n}{2} + \frac{n}{n+1} \in \Theta(n)$.

2 Priority Queues and Heaps

In a priority queue, we store a collection of elements. We're relying on **one** characteristic:

- Each element in the priority queue comes with its **own** priority attached to it (has something that makes each object sortable).
  - `x.priority` returns a comparable value
    * Integers are a good stand-in; however, we could use tuples or lists as a tiebreaker – the exception is that they are reversed for the context of this course. Hence, the last element takes the most precedence.
  - We don't care about its implementation. There is some built-in mechanism in the object that allows me to know its priority in constant time.

The **operations are the following:**

- `INSERT(Q, x):` add `x` to `Q`
  - Multiple elements can have the same priority.
  - If an object has multiple priorities, priorities assigned to the higher index should take precedence
- `MAX(Q):` return an element with the maximum priority.
  - If there are ties, we don't care which one is returned. Any of them could be returned. `Q` remains unchanged; this operation only queries our ADT.
- `EXTRACT_MAX(Q):` remove and return the element with the maximum priority.

The ordering for priority queues in this course will not be following the **first-in first-out** model. Drop the notion of a regular queue.
Of course, you could add a timestamp or insertion order as a tiebreaker.

2.1 How do we do it?

For the purposes of this course:

- An array will refer to an array-based list.
- A list will refer to a linked list.

The simplest data structures:

Unsorted array/list

- `INSERT`: $\mathcal{O}(1)$
- `MAX`: $\Omega(n)$
- `EXTRACT_MAX`: $\Omega(n)$
- We could do better.

Sorted array/list:

- `INSERT`: $\Omega(n)$
  - Yes, even for array-based lists. No loopholes.
- `MAX`: $\mathcal{O}(1)$
- `EXTRACT_MAX`: $\mathcal{O}(1)$

Here, $\mathcal{O}$ means "good news", and Ω means "too bad." It is simply emphasis. Everything is Θ, and all the time complexities here are **worst-case time.**

2.2 Max-Heaps

Intuition: partially sort

Structure: "almost complete" binary tree. Everything full, except maybe the last. On the last level, all leaves are **as far left as possible.** This **MUST** be preserved at all times throughout the lifetime of the max-heap.

In practice, when you have an almost-complete binary tree, the way that this is stored in memory, they generally mean a **list (or an array). They are listed in the order you would traverse then in a BREADTH FIRST SEARCH.**

THE HEAP ELEMENTS START AT INDEX 1, AND WE SKIP OVER INDEX 0. Index 0 stores an empty item which we will not consider the root.

Navigation (REQUIRES BFS FOR INDEXING OF THE HEAP):

- For each node at index i in an array
- Parent$(i) = \lfloor \frac{i}{2} \rfloor$
  - This operation can be done extremely fast using bit shifts. That is, multiplication or floor division by a power of 2. In practice, this is implemented that way.
- LeftChild$(i) = 2i$
  - Previous item in size if it exists, otherwise we can't say anything
- RightChild$(i) = 2i + 1$
  - Next item in size if it exists, otherwise we can't say anything

Because we're working with an array (really), when we insert, we're going to add at the end. But it will mess up the ordering of the element. The same thing applies when we do extract max. We need to make sure that there are no gaps in the tree at any time.

2.3 Max-Heap Order

This is not a binary search tree. There is **no** left-to-right ordering. The only kind of ordering we have in a heap is the top-to-bottom ordering.

This property must hold for all max heaps.
EVERY NON-LEAF NODE STORES AN ELEMENT WITH PRIORITY $\geq$ THE PRIORITIES OF THE ELEMENTS IN THE NODE'S CHILDREN.
No required ordering between siblings. This means if I reflect the heap on the vertical axis, it shouldn't break this property.

View max-heaps as stairs: when you vertically go down, you should step downwards.

The highest priority of the heap is at the top. Finding the max is always easy: it is **always the top element, at index 1 (the first index).**