CSC258

Rishibh Prakash

January 2022

Contents

1 Review

Definition 1.1 (Abstract Data Type)**.** An abstract data type (ADT) is a set of objects with some operation(s).

Example 1.1. A Stack is an ADT. The set of objects can be anything and the (standard) operations are `push(x)`, `pop()` and `isEmpty()`. ■

Definition 1.2 (Data Structure)**.** A data structure is an implementation of an ADT. It gives a way of representing the objects in an ADT and an algorithm for every operation.

One can implement a Stack using a linked list or an array (where by array we mean the Java use of array). These two implementations have different algorithms for each of the operations and hence have different time/space complexity. We may choose one implementation over the other (i.e. one data structure over the other) depending on what we need.

2 Runtime Complexity

The complexity of an algorithm is the amount of resources required by it as a function of the input size. We are deliberately vague on what we mean by resources. This could be space (in memory), the amount of time taken or some other metric (although the first two are the most common). Also note that the definition of 'input size' will vary based on the problem. For lists, input size might be the length of the list, for graphs it might be the number of nodes/vertices or the number of edges.

2.1 Analysing Runtime Complexity

When analysing the runtime of an algorithm, there a few cases we tend to consider: worst case (this is maximum amount of time/space/resources the algorithm could take. Useful for obvious reasons), best case (this is the minimum amount of time/space/resources the algorithm needs to take. Not used incredibly often but does provide useful information in certain cases) and the average case (as suggested by the name, this is the statistically expected runtime of an algorithm. If the worst case happens only rarely, it seems silly to give it much weight).

Computer scientists tend not to work with the exact values when analysing runtime (which can vary based on minor things like the hardware, the device, the temperature, etc.) but are rather more interested in how the runtime behaves as a function of the input size and in particular they only care about large input sizes. All of this is made more precise with the use of asymptotic notation.

2.2 Asymptotic Notation

We have the following definitions:

$O(g(n)) = \{f(n) : \exists c \in \mathbb{R} \exists n_0 \in N \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R} \exists n_0 \in N \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R} \exists n_0 \in N \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$

Intuitively, one thinks of $O(g(n))$ being the set of functions that grow at most as fast as $g(n)$ (or some multiple thereof). Hence $g(n)$ provides a kind of upper bound for all functions in $O(g(n))$.

Example 2.1. If $f(n) = 2n^2 + n$, then $f(n) \in O(n^2), O(n^{100}), O(n^n)$, etc. ■

Conversely $\Omega(g(n))$ is the set of functions that grow at least as fast as $g(n)$, making $g(n)$ a kind of lower bound for all the functions in $\Omega(g(n))$.

Example 2.2. If $f(n) = 2n^2 + n$ then $f(n) \in \Omega(n^2), \Omega(1), \Omega(n)$ etc. ■

Finally, as one might expect, $\Theta(g(n))$ is the set of functions that grow at the same rate as $g(n)$.

Example 2.3. If $f(n) = 2n^2 + n$, then $f(n) \in \Theta(n^2)$. ■

Notice that in the examples above $f(n) \in O(n^2)$ and $f(n) \in \Omega(n^2)$ which is how we were able to conclude that $f(n) \in \Theta(n^2)$. This is how we almost always show that $f(n) \in \Theta(g(n))$. The question then becomes how would we show that the runtime complexity of an algorithm is in Big-Oh or Big-Omega for some $g(n)$.

Suppose $f(n)$ is the worst case time complexity of an algorithm. Recall that Big-Oh provides an upper bound. So if we want to show that $f(n) \in O(g(n))$ then we must show that *every* input of size n takes at most $cg(n)$ steps. If we want to show that $f(n) \in \Omega(g(n))$ then we only need to find a suitable example for every n where the runtime on the example is at least $cg(n)$.

3 Worst Case Analysis

Suppose $t(x)$ describes the time taken by an algorithm on some input x. Then the worst case time complexity $T(n)$ is given by

$$T(n) = \max\{t(x) : x \text{ is an input of size } n\}$$

3.1 Example

Here is a simple example for us to consider: a function that takes in list and determines whether or not it contains the integer 21.

```
1 def hasTwentyOne(L):
2     j = L.head
3     while j!= None and j.key != 21:
4         j = j.next
5     return
```

Let $f(n)$ denote the maximum number of comparisons made by the algorithm on a list of length n (the number of comparisons is how are choosing to measure the complexity of this algorithm since it's a relatively expensive operation). We first claim that $f(n) \in \Omega(n)$. Let L_n denote a list of length n that does not contain 21. In this case 2 comparisons are made in line 3 for every element in the list and one final comparison at the end when we reach the end. This means that $2n + 1$ comparisons are made in this case. Thus $f(n) \in \Omega(n)$.

Now we claim that $f(n) \in O(n)$. We need to show that *any* input of size of n will take at least $2n+1$ comparisons. Note that every time line 3 is executed, we either quite the while loop or run line 4. Every time line 4 is run, j is the next element in the list. Hence line 4 can be executed at most n times after which point j will be None. Then line 3 is run one more time to compare j and `None`. In this case the while condition will evaluate to False hence we must exit the loop. Hence at most $2n + 1$ comparisons can be done on an input of size n.

We can quickly do the analysis for the best case scenario which occurs when the first element of the list is 21. In this case only 2 comparisons will be done regardless of the list size implying that the best case $O(1)$.

4 Average Case Analysis

As described above and as one might expect, we find the average runtime by finding the expected value of the runtime over all possible inputs. This requires us to know the distribution of the inputs. Usually we make some reasonable assumptions about the input which allows us to come up with a decent distribution. If we have more knowledge about the inputs, we can use that to find a more accurate distribution.

4.1 Example 1

We use the same algorithm as before for our example.

```
1 def hasTwentyOne(L):
2     j = L.head
3     while j!= None and j.key != 21:
4         j = j.next
5     return
```

For input size n, we define our sample space $S_n = \bigcup_{i=0}^{n} x_i$, where x_0 is the set of all of inputs that do not contain the number 21 and x_i (for $1 \leq i \leq n$) is the set of inputs that contain 21 in the $i-$the position. From before we know that the number of comparisons made for x_i for $1 \leq i \leq n$ is $2i$ and for x_0 it is $2n + 1$. We define T to be the random variable denoting runtime on any input

of size n. Then

$$T_{avg} := E[T] = \sum_{s \in S_n} t(s)P(T = s)$$

(where t, as before, denotes the runtime on any given input).

For our example, we will assume that the probability of 21 appearing in the first position of the list is p (where course $0 < p \leq 1$, we can assume p to be non-zero since if it is 0, we go back to the worst case) and the probability of 21 appearing at index i is independent of it appearing at index j. This means that the probability of 21 appearing at index i is $(1-p)^{i-1}p$. Then

$$\begin{aligned}
E[T] &= \sum_{i=0}^{n} t(x_i) \cdot P(T = x_i) \\
&= t(x_0) \cdot P(T = x_0) + \sum_{i=1}^{n} t(x_i) \cdot P(T = x_i) \\
&= (2n+1)(1-p)^n + \sum_{i=1}^{n} (2i)(1-p)^{i-1}p \\
&= \vdots \\
&= (1-p)^n \left(1 - \frac{2}{p}\right) + \frac{2}{p}
\end{aligned}$$

We see that as $n \to \infty$, the expected runtime approaches $\frac{2}{p}$ which is a constant. Therefore $T_{avg} \in \Theta(1)$.

4.2 Example 2

We consider a second example

```
1 def evens_are_bad(lst):
2     if every number in lst is even:
3         repeat lst.length times:
4             calculate and print the sum of the lst
5         return 1
6     else:
7         return 0
```

The operation we will consider is accessing a list element. In this case we can divide our sample space S_n into two classes of lists, a list of length n with all even numbers and a list of length n with at least one odd number. In the first case we $n + n^2$ steps, n for iterating over the list to determine whether all the numbers are even. Then we have n iterations to find the sum which is repeated n times. In the the other case we only have n iterations in order to determine whether every number is even.

We will assume that there is a $p = 0.5$ probability of any element of the list

being even. Then

$$\begin{aligned} E[T] &= p^n \cdot (n^2 + n) + (1 - p^n) \cdot n \\ &= n^2 p^n + np^n + n - np^2 n \\ &= n^2 p^n + n \end{aligned}$$

As $n \to \infty$, then this approaches n since the first term goes to 0. Therefore the average runtime complexity of this algorithm is asymptotically linear.