1 Abstract Data Types (ADTs)

Definition. An abstract data type is a set of mathematical objects and a set of operations that can be performed on these objects.

Examples

1. ADT: INTEGERS
   objects: integers
   operations:
   ADD(x, y): add x and y
   SUBTRACT(x, y): subtract y from x
   MULTIPLY (x, y): multiply x and y
   QUOTIENT (x, y): divide x by y
   REMAINDER (x, y): take the remainder of x when dividing by y

2. ADT: STACK
   objects: elements, stack
   operations:
   PUSH(S, x): adds the element x to the end of the list S
   POP(S): deletes the last element of the nonempty list S and returns it
   EMPTY(S): returns true if S is empty, false otherwise

2 Data Structures

Definition. A data structure is an implementation of an ADT. It consists of a way of representing the objects and algorithms for performing the operations.

Examples

1. ADT: INTEGERS
   objects: An integer is stored as one word of memory on most machines.
   operations: ADD (x, y) is often implemented in the Arithmetic Logic Unit (ALU) by a circuit algorithm such as “ripple-carry” or “look-ahead.”
2. ADT: STACK

objects: A stack could be implemented by a singly-linked list or by an array with a counter to keep track of the “top.”

Exercise: Can you think of any advantages or disadvantages for implementing the STACK ADT as an array versus implementing it as a singly-linked list?

operations:

Exercise: How would you implement PUSH, POP and EMPTY in each of these implementations?

ADTs describe what the data is and what you can do with it, while data structures describe how the data is stored and how the operations are performed. Why should we have ADTs in addition to data structures?

- important for specification
- provides modularity
  - usage depends only on the definition, not on the implementation
  - implementation of the ADT can be changed (corrected or improved) without changing the rest of the program
- reusability
  - an abstract data type can be implemented once, and used in lots of different programs

The best data structure for an algorithm usually depends on the application.

3 Analyzing Data Structures and Algorithms

The complexity of an algorithm is the amount of resources it uses expressed as a function of the size of the input. We can use this information to compare different algorithms or to decide whether we have sufficient computing resources to use a certain algorithm.

Types of resources: Running time, space (memory), number of logic gates (in a circuit), area (in a VLSI) chip, messages or bits communicated (in a network)

For this course, the definition of input size will depend on what types of objects we are operating on:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplying Integers</td>
<td>Total Number of Bits Needed to Represent the Integers</td>
</tr>
<tr>
<td>Sorting a List</td>
<td>Number of Elements in the List</td>
</tr>
<tr>
<td>Graph Algorithms</td>
<td>Vertices and Edges</td>
</tr>
</tbody>
</table>

The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed (for example, number of comparisons). This also depends on the problem. We
want the notion of “step” to be machine independent, so that we don’t have to analyze algorithms individually for different machines.

How do we measure the running time of an algorithm in terms of input size when there may be many possible inputs of the same size? We’ll consider three possibilities:

3.1 Worst case complexity

Definition. For an algorithm \( A \), let \( t(x) \) be the number of steps \( A \) takes on input \( x \). Then, the worst case time complexity of \( A \) on input of size \( n \) is

\[
T_{wc}(n) \overset{d}{=} \max_{|x|=n} \{t(x)\}.
\]

In other words, over all inputs of size \( n \), \( T_{wc}(n) \) is defined as the running time of the algorithm for the slowest input.

Example: Let \( A \) be the following algorithm for searching a list \( L \) for an element with key equal to the integer \( k \):

\[
\begin{align*}
\text{ListSearch} \ (\text{List} \ L, \ \text{Integer} \ k) \\
\quad &\text{Element } z = \text{head}(L); \\
\quad &\text{while } (z \neq \text{null}) \text{ and } (\text{key}(z) \neq k) \text{ do} \\
\quad &\quad z = \text{next}(L, z); \\
\quad &\text{return } z;
\end{align*}
\]

We have several options for what we should count as a “step”. We could count every atomic operation (i.e. assignments, returns and comparisons) or we could count only each comparison. Since we are really interested in the number of comparisons and the total number of atomic operations is within a constant factor, it is reasonable to count only the number of comparisons.

Notice that in each iteration of the loop, \( A \) does 2 comparisons. If we get to the end of the list \( A \) does a final comparison and finds that \( z \) is equal to \( \text{null} \) (we assume that the “and” checks the first comparison and then the second only if the first was true).

Then, let \( n \) be the length of \( L \) and let \( t(L, k) \) be the number of comparisons performed by ListSearch on input \( (L, k) \). Then,

\[
t(L, k) = \begin{cases} 
2i & \text{for } k \text{ the } i\text{th element of } L \\
2n + 1 & \text{if } k \text{ is not in } L
\end{cases}
\]

So clearly \( T_{wc}(n) = 2n + 1 \). This can be written in asymptotic notation as \( \Theta(n) \).

3.2 Best case complexity

Definition. For an algorithm \( A \), let \( t(x) \) be the number of steps \( A \) takes on input \( x \). Then, the best case time complexity of \( A \) on input of size \( n \) is

\[
T_{bc}(n) \overset{d}{=} \min_{|x|=n} \{t(x)\}.
\]
**Example:** We use ListSearch as algorithm $A$ again.

We know that $T_{bc}(0) = 1$ since there is only one list of length 0 and for any value $k$, $A$ will make exactly one comparison when $L$ is empty.

This shows that $T_{bc}(0) = 1$ but does not give any insight into $T_{bc}(n)$ for $n \geq 1$. For $n \geq 1$, let $L = 1 \rightarrow 2 \rightarrow \ldots \rightarrow n$ and let $k = 1$. Exactly two comparisons will be made for this instance of $L$ and $k$. Therefore, $T_{bc}(n) \leq 2$. This is an upper bound on the best case time complexity of ListSearch for $n \geq 1$.

If $n \geq 1$, then $\text{head}[L] \neq \text{null}$. Therefore, the first comparison evaluates to true and a second comparison is performed. Therefore, $T_{bc}(n) \geq 2$. This is a lower bound on the best case time complexity of ListSearch for $n \geq 1$.

Therefore, $T_{bc}(n) = \begin{cases} 1 & \text{for } n = 0 \\ 2 & \text{for } n \geq 1 \end{cases}$

It should be noted that best case complexity often does not reveal useful information about a problem and we will often ignore it in this course.

### 3.3 Average case complexity

Let $A$ be an algorithm. Consider the sample space $S_n$ of all inputs of size $n$ and fix a probability distribution. Usually, we choose the probability distribution to be a uniform distribution (i.e. every input is equally likely).

Recall that a random variable is a function maps from elements in a probability space to $\mathbb{N}$. Also, recall that the expected value of a random variable $V : S \rightarrow \mathbb{R}$ is $E[V] = \sum_{x \in S} V(x) \cdot \Pr(x)$.

**Definition.** Let $t_n : S_n \rightarrow \mathbb{N}$ be the random variable such that $t_n(x)$ is the number of steps taken by algorithm $A$ on input $x$.

Then $E[t_n]$ is the expected number of steps taken by algorithm $A$ on inputs of size $n$. The average case time complexity of $A$ on inputs of size $n$ is defined as

$$T_{avg}(n) \overset{d}{=} E[t_n].$$

The following three steps should be performed before doing any average case time complexity analysis:

1. Define the sample space
2. Define the probability distribution function
3. Define any necessary random variables

**Example:** Again, we use ListSearch as algorithm $A$.

It is sufficient when analyzing this algorithm to assume the list $L$ is the list $1 \rightarrow 2 \rightarrow \ldots \rightarrow n$ and $k \in \{0, \ldots, n\}$. More precisely, consider any input $(L, k)$. Let $L' = 1 \rightarrow 2 \rightarrow \ldots \rightarrow n$. If $k$ is the $i$th element of the list $L$, let $k' = i$; if $k$ is not in the list $L$, let $k' = 0$. Since the algorithm only performs equality tests between $k$ and elements of $L$, the algorithm will have the same behavior on $(L', k')$ as it did on $(L, k)$. We use this simplified form so that the sample space of inputs, $S_n$, will be finite and therefore simpler to handle.
1. Sample Space: \((1 \rightarrow 2 \rightarrow \ldots \rightarrow n, k \in \{0, \ldots, n\})\)

2. We will assume a uniform distribution

3. Similarly to before,
   
   \[ t_n(L, k) = \begin{cases} 
   2k & \text{for } k \neq 0 \\
   2n + 1 & \text{if } k = 0
   \end{cases} \]

   Note that the assumption of a uniform distribution may not be a reasonable one. If the application will often search for elements \(k\) which are not in the list \(L\), then the probability of input \((L, 0)\) would need to be higher than the other inputs.

   Then,
   
   \[
   T_{\text{avg}}(n) = E[t_n] = \sum_{k=0}^{n} \Pr(L, k)t_n(L, k)
   \]
   
   \[
   = \frac{1}{n+1}(2n+1) + \sum_{k=1}^{n} \Pr(L, k)t_n(L, k)
   \]
   
   \[
   = \frac{2n + 1}{n+1} + \frac{1}{n+1} \sum_{k=1}^{n} 2k
   \]
   
   \[
   = \frac{2n + 1}{n+1} + n
   \]

   Notice that \(1 \leq \frac{2n+1}{n+1} < 2\), so \(n + 1 \leq T_{\text{avg}}(n) < n + 2\). This value is somewhat smaller (as expected) than \(T_{\text{wc}}(n)\). Asymptotically, however, it is also \(\Theta(n)\), which is the same as \(T_{\text{wc}}(n)\). For some algorithms, \(T_{\text{wc}}\) and \(T_{\text{avg}}\) are different even when analyzed asymptotically, as we shall see later in the course.

3.4 Comparison

We now have three methods for analyzing the time complexity of an algorithm:

- **Worst Case** \(T_{\text{wc}}(n) = \max_{|x|=n} \{t(x)\}\)
- **Average Case** \(T_{\text{avg}}(n) = E[t(x)] \ |x| = n\)
- **Best Case** \(T_{\text{bc}}(n) = \min_{|x|=n} \{t(x)\}\)

Then, from the definition of expectation,

\[
T_{\text{bc}} \leq T_{\text{avg}} \leq T_{\text{wc}}
\]

Each of these three measures can be useful depending on the algorithm and the application. Some algorithms have large \(T_{\text{wc}}\) but small \(T_{\text{avg}}\) while for other algorithms \(T_{\text{wc}}\) and \(T_{\text{avg}}\) are equal.
3.5 Upper and Lower Bounds

Recall that there is an important distinction between proving upper bounds and proving lower bounds on an algorithm’s worst case running time.

An upper bound is usually expressed using $\text{Big} - O$ notation. To prove an upper bound of $g(n)$ on the worst case running time $T_{wc}(n)$ of an algorithm means to prove that $T_{wc}(n)$ is $O(g(n))$. This is roughly equivalent to proving that

$$T_{wc}(n) = \max_{|x|=n} t(x) \leq g(n)$$

How can we prove that the maximum of a set of values is no more than $g(n)$? The easiest way is to prove that every member of the set is no more than $g(n)$.

In other words, to prove an upper bound on the worst case running time of an algorithm, we must argue that the algorithm takes no more than that much time on every input of the right size. In particular, you cannot prove an upper bound if you only argue about one input, unless you also prove that this is input really is the worse in which case you’re back to proving something for every input.

A lower bound is usually expressed using $\text{Big} - \Omega$ notation. To prove a lower bound of $f(n)$ on the worst case running time $T_{wc}(n)$ of an algorithm means to prove that $T_{wc}(n)$ is $\Omega(f(n))$. This is roughly equivalent to proving that

$$T_{wc}(n) = \max_{|x|=n} t(x) \geq f(n)$$

How can we prove that the maximum of a set of values is at least $f(n)$? The easiest way is to find one element of the set which is at least $f(n)$.

In other words, to prove a lower bound on the worst case running time of an algorithm, we only have to exhibit one input for which the algorithm takes at least that much time.