

CSC 378 Lecture 1

May 17, 2001

Related material can be found in chapter 5 of the textbook (CLRS).

1 Abstract Data Types (ADTs)

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

Examples

1. *objects*: integers
operations: ADD (x, y) , SUBTRACT (x, y) , MULTIPLY (x, y) , QUOTIENT (x, y) , REMAINDER (x, y)
2. *objects*: stacks
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

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

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 talking about:

Examples:

- Integers: number of bits
- Lists: number of elements
- Graphs: number of vertices plus number of edges

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. The notion of “step” should be machine independent, so 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 two possibilities:

3.1 Worst-case complexity

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

$$T_{wc}(n) \stackrel{d}{=} \max_{|x|=n} \{t(x)\}.$$

That is, look at all the inputs of size n and take the time of the one that is the slowest.

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

```
LIST_SEARCH (L, k)
  z := head(L)
  while (z != NIL) and (key(z) != k) do
    z := next(z)
  return z
```

Here we'll take the number of "steps" to be the number of comparisons that the algorithm performs. 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 NIL (we assume that the "and" checks the first comparison and then the second only if the first was true).

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 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 (for example, each input is equally likely).

Let $t_n : S_n \rightarrow \mathbf{N}$ be the random variable such that $t_n(x)$ is the number of steps taken by algorithm A on input x (recall that a random variable is a function that assigns a numeric value to each element of a probability space).

Then $E[t_n]$ is the expected number of steps taken by algorithm A on inputs of size n (recall that the expected value of a random variable $V : S \rightarrow \mathbf{R}$ is defined as $E[V] \stackrel{d}{=} \sum_{x \in S} V(x) \cdot \Pr(x)$). The *average case time complexity* of A on inputs of size n is defined as

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

Example: Let's use LISTSEARCH as A again.

It is sufficient when analyzing this algorithm to assume the list L is the list $1 \rightarrow 2 \rightarrow \dots \rightarrow n$ and $k \in \{0, \dots, n\}$. More precisely, consider any input (L, k) . Let $L' = 1 \rightarrow 2 \rightarrow \dots \rightarrow n$. 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.

So, $S_n = \{(L', k') \mid 0 \leq k' \leq n\}$. We'll assume that each of these $n + 1$ possible inputs has probability $1/(n + 1)$; that is, they are all equally likely.

Similarly to before,

$$t_n(L', k') = \begin{cases} 2k' & \text{for } k' \neq 0 \\ 2n + 1 & \text{if } k' = 0 \end{cases}$$

Then,

$$\begin{aligned} T_{avg}(n) \stackrel{d}{=} E[t_n] &\stackrel{d}{=} \sum_{k'=0}^n \Pr(L', k') t_n(L', k') \\ &= \frac{1}{n+1} 2n + 1 + \sum_{k'=1}^n \Pr(L', k') t_n(L', k') \\ &= \frac{2n+1}{n+1} + \frac{1}{n+1} \sum_{k'=1}^n 2i \\ &= \frac{2n+1}{n+1} + n \end{aligned}$$

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