# Section Solutions 6: Running Times and ADTs

**Problem 1: Big-O Notation**

**a)** O(n2). The outer loop will run n times. Each time through the outer loop, the inner loop will run i times, where i runs from 0 to n-1. A constant amount of work is done in the body of the inner loop. This leads to the series: 0 + 1 + 2 + ... n-1 which, as we know, equals (n-1)(n-2)/2. Keeping only the highest order term and throwing away any constant factors, we arrive at our answer of O(n2) computational complexity.

**b)** O(1). The outer loop executes 10 times, the inner loop i times where i runs from 0 to 9, so there are ~100 multiply/add operations, but it does that same amount of work for any value of n. Thus the computational complexity is *constant* with respect to n. The constant "1" in O(1) signifies this. Constant time doesn't necessarily mean that a function is trivial or computes its result instantly, but the function always does the same amount of work, regardless of the inputs.

**Problem 2: Searching and Sorting**

For each case, we need to compare the cost of doing the sort and binary search with the cost of doing linear searches. If we let X be the number of searches done, we can write equations in this form:

(cost of sort + (X * cost of binary search)) < (X * cost of linear search)

If this inequality holds, it is in this case that doing a sort first is the more efficient approach. So, we solve for X to determine how many searches make it worth our while to sort the data:

Selection Sort --> $O(n^2)$

**a)**

$$((2)^4)^2 + (X * \log_2(2)^4) < (X * (2)^4)$$
$$256 + 4X < 16X$$
$$21.3 < X$$

**b)**

$$((2)^{10})^2 + (X * \log_2(2)^{10}) < (X * (2)^{10})$$
$$1048576 + 10X < 1024X$$
$$1034.1 < X$$

MergeSort --> O(nlogn)

**a)**

$((2)^4 * \log_2(2)^4) + (X * \log_2(2)^4) < (X * (2)^4)$
$64 + 4X < 16X$
$5.3 < X$

**b)**

$((2)^{10} * \log_2(2)^{10}) + (X * \log_2(2)^{10}) < (X * (2)^{10})$
$10240 + 10X < 1024X$
$10.1 < X$

**Problem 3: The Philosophy of ADTs**

**a)** In discussing ADTs, it helps to be clear in our terminology. An ADT involves two different roles for programmers. The first role is the *implementor* of the ADT, the programmer who designs and writes the ADT. The second role is the *client,* a programmer who uses the ADT in writing code. Although we'll usually talk as if there is one client who is distinct from the one implementor of an ADT, there can be more than one person implementing an ADT and more than one client of that ADT. Also, it could be that the implementor and the client are the same person. In any of these cases, it is important to keep these two roles distinct. And these roles are both different from the *user,* the person using the program or application that the client and the implementor helped to create.

An ADT has two parts, an interface and an implementation. The interface (the .h file) is the list of functions and data types that the client can use in their code, and a concise explanation of how to use those functions and data types. The implementation (the .c file) implements those functions. The client can use the data types to declare variables, but the client should never read or change the data stored in the structure by directly accessing its fields. By writing code that read or sets explicit fields or any way depends on the underlying implementation of the ADT, the client is "breaking the wall"— the separation between the interface and the implementation.

We use the incomplete type specifier to disallow access to the fields within our structure and prevent breaking the wall. Even if a client has insider knowledge that the cursor is presented as an integer field named "cursor", if he attempts to directly access the ADT structure field like this:

```
stack->depth = 0;
```

the compiler will not accept this. Instead, the client should call the supplied function `StackDepth` to retrieve the number of elements in the stack.

If we didn't specify our ADT as an incomplete type, we would detail the innards of our structure in the header file and leave open the potential for a misbehaving client to access

with our data. However, the implementor is relying on the data having certain values and being changed in a way that the implementor can control by providing functions that the implementor has carefully thought out. If the client changes data out from under the implementor, the client can mess up the implementation of the ADT. Also if the implementor would like to change the implementation, the client who breaks the wall could end up with code that doesn't even compile now or has bugs lurking all over it due to changes in the ADT implementation.

**b)** The use of ADTs gives the implementor great freedom: freedom to change the implementation, fix bugs, re-design the data structure, substitute more efficient algorithms, etc. all without disturbing the client. The client would not want to have to go back and rewrite their code just because the implementor felt like speeding up the implementation!

Another main advantage of an ADT is that it hides a lot of detail for the client. Clients can use the ADT without thinking much about how it is implemented. The ADT is written and debugged once, and it can be used again and again without reinventing the wheel. The client is free to solve bigger and more interesting problems. Also, many programmers can work together in a fairly organized way, without being involved with the details of each other’s work.

## Problem 4 : StackADT

```
/*
 * MyStackDepth
 * By using recursion, the function uses the function stack to keep track
 * of the elements we have popped off of our polymorphic stackADT.  By
 * popping one element off at a time, we make progress towards our base
 * case of the empty stackADT.  Since we push each element back on after
 * our recursive call, the function restores the stackADT to its starting
 * state when done.
 */
int MyStackDepth(stackADT stack)
{
     void *element; // we don't know what elements are, just store pointer
     int depth;

     if (IsStackEmpty(stack)) return 0;    // base case

     element = Pop(stack);                 // save element
     depth = 1 + MyStackDepth(stack);      // count rest
     Push(stack, element);                 // put element back on
     return depth;
   }
```

## Problem 5 : ADT client

```
static bool IsBalanced(string str)
{
   int i;
   stackADT s = NewStack();

   for (i = 0; i < StringLength(str); i++) {
      if (IsOpener(str[i])) {
          Push(s, str[i]);
      } else if (IsCloser(str[i])) {
          if (StackIsEmpty(s) || Pop(s) != MatchingChar(str[i]))
              return FALSE;
      }
   }
   return StackIsEmpty(s);
}
```