④ $T(n) = T(n-1) + 2*n \; ; \; T(1) = 5$

Forward Substitution

$T(1) = 5$

$T(2) = T(2-1) + 2(2) = T(1) + 4 = 5 + 4 = 9$

$T(3) = T(3-1) + 2(3) = T(2) + 6 = 9 + 6 = 15$

$T(4) = T(4-1) + 2(4) = T(3) + 8 = 15 + 8 = 23$

No discernable pattern right away.

Backward Substitution

$T(n-1) = T(n-1-1) + 2(n-1) = T(n-2) + 2(n-1)$

So,

$T(n) = T(n-2) + 2(n-1) + 2n$

$= T(n-3) + 2(n-2) + 2(n-1) + 2n$

$= T(n-4) + 2(n-3) + 2(n-2) + 2(n-1) + 2n$

$\vdots$

$= T(n-i) + \sum_{j=0}^{i-1} 2(n-j)$

$= T(n-i) + 2n(i-1) - 2\frac{(i-1)(i-1+1)}{2} + 2n$

$= T(n-i) + 2n(i-1) - i^2 + i + 2n$

When $i = n-1$, we get the base case, i.e. $T(n-n+1) = T(1) = 5$

So $T(n-i) + 2n(i-1) - i^2 + i + 2n = T(n)$

$\downarrow$

$T(1) + 2n(i-1) - i^2 + i + 2n = T(n)$

$5 + 2n(n-2) - (n^2 - 2n + 1) + (n-1) + 2n = T(n)$

or $T(n) = n^2 + n + 3$ Closed form

(b) $T(n) = 2 \times T(n-1)$; $T(0) = 1$

Forward substitution

$T(1) = 2 \quad (2^1)$
$T(2) = 4 \quad (2^2)$
$T(3) = 8 \quad (2^3)$
$T(4) = 16 \quad (2^4)$

} Pattern: 2^n

So, $T(n) = 2^n$ closed form

Backward substitution

$T(n-1) = 2 \times T(n-1-1) = 2 \times T(n-2)$
$T(n-2) = 2 \times T(n-2-1) = 2 \times T(n-3)$
$T(n-3) = 2 \times T(n-3-1) = 2 \times T(n-4)$
$T(n-4) = 2 \times T(n-4-1) = 2 \times T(n-5)$

Now,

$T(n) = 2 \times T(n-1)$
$= 2 \times 2 \times T(n-2)$
$= 4 \times T(n-2)$
$= 4 \times 2 \times T(n-3)$
$= 8 \times T(n-3)$
$= 8 \times 2 \times T(n-4)$
$= 16 \times T(n-4)$
$= 16 \times 2 \times T(n-5)$
$= 32 \times T(n-5)$

→ Pattern?

$\forall k, k \geq 1$

$T(n) = 2^k T(n-k)$ — ①

Base case : $T(0) = 1$

or $n-k = 0$

$n = k$

Substituting in ①

$T(n) = 2^n T(n-n)$
$= 2^n T(0)$
$= 2^n * 1$
$= 2^n$ closed form.

CONCEPT EXERCISES

6.2 Show that, for the task of appending n elements to an ArrayList object, worstTime(n) is linear in n.

To append an element to an ArrayList object, the one-parameter add method is invoked. If n represents the number of elements currently in the ArrayList object and there are n calls to the one-parameter add method, there will be at most two expansions of the underlying array. For example, suppose that $n = 500$. If the capacity of the underlying array is 5000, there will be no expansion of the underlying array when n additional elements are appended. If the capacity of the underlying array is 800, there will be one expansion. If the capacity of the underlying array is 600, there will be two expansions (one expansion when $n = 600$, and one expansion when $n = 901$). So for worstTime(n), assume that there will be two expansions of the underlying array.

Then there will be $n - 2$ calls to add that do not require expansion, and each of these calls entails the execution of c_1 statements, for some constant c_1. For the two calls to the add method that require an expansion of the underlying array, each call will entail the execution of $c_2n + c_3$ statements, for some constants c_2 and c_3. The total number of statements executed during the n calls to the add method is

$(n - 2)\, c_1 + 2\, (c_2n + c3)$.

That is,

worstTime(n) $\approx (n - 2)\, c_1 + 2\, (c_2n + c_3) = (c_1 + 2c_2)n + 2c_3 - 2c_1$, which is linear in n.

6.4 For the one-parameter add method in the ArrayList class, estimate worstSpace(n) and averageSpace(n).

If a call to the add method requires expansion of the underlying array, then a new array with capacity $3n / 2 + 1$ must be allocated, so worstSpace(n) is linear in n. On average, an expansion will occur only once for every n calls to the add method, so averageSpace(n) is constant.

PROGRAMMING EXERCISES

6.2 For each of the following program segments, hypothesize if the segment would generate a compile-time error, a run-time exception, or neither. Then test your hypotheses with a main method that includes each segment.

a.
```
ArrayList<String> myList = new ArrayList<String>();
myList.add ("yes");
myList.add (7);
```

Compile-time error: cannot find method add(**int**)

b.
```
ArrayList<Double> original = new ArrayList<Double>();
original.add (7);
```

Compile-time error: cannot find method add(**int**)

c.
```
ArrayList<Integer> original = new ArrayList<Integer>();
double x = 7;
original.add (x);
```

Compile-time error: cannot find method add(**double**)

d.
```
ArrayList<String> newList = new ArrayList<String>();
newList.add ("yes");
Integer answer = (Integer)newList.get (0);
```

Compile-time error: Inconvertible types
Found: String
Requried: Integer

6.3 Suppose we have the following code:

```
ArrayList<String> myList = new ArrayList<String>();

myList.add ("Karen");
myList.add ("Don");
myList.add ("Mark");

ArrayList<String> temp = new ArrayList<String> (myList);
ArrayList<String> sameList = myList;
```

```
myList.add (1, "Courtney");
```

Hypothesize what the contents of myList, temp and sameList will be after this last insertion. Then test your hypothesis with a main method that includes the code.

myList: Karen, Courtney, Don, Mark

sameList: Karen, Courtney, Don, Mark

temp: Karen, Don, Mark

6.6 Assume that myList is (a reference to) an ArrayList<Double> object and that both i and j are **int** variables with values in the range from 0 to myList.size() – 1, inclusive. Hypothesize what the following accomplishes, and then test your hypothesis.

```
myList.set (i, myList.set (j, myList.get (i)));
```

The elements at indexes i and j are swapped.

6.8 Modify the simple program in Section 6.2.2 so that all removals are performed in a single loop. **Hint:** Create a temporary ArrayList object to hold the un-removed elements. What is a drawback to this approach?

Replace

```
while (aList.remove (word))
        removalCount++;
```

with

```
ArrayList<String> temp = new ArrayList<String>();
for (int i = 0; i < aList.size(); i++)
        if (aList.get (i).equals (word))
                removalCount++;
        else
                temp.add (aList.get (i));
aList = temp;
```

The drawback is the extra space allocated for the ArrayList object temp.

6.10 Modify the simple program in Section 6.2.2 to use a binary search instead of the sequential search used in the call to the indexOf method. The Collections class in java.util has a binarySearch method and a sort method.

The following code uses a temporary ArrayList object to avoid modifying aList by sorting:

```
System.out.print ("\n\nPlease enter the word you want to search for: ");
 word = keyboardScanner.next();
ArrayList<String> temp = new ArrayList<String> (aList);
Collections.sort (temp);
 if (Collections.binarySearch (temp, word) >= 0)
        System.out.println (word + " was found.\n\n");
 else
         System.out.println (word + " was not found.\n\n");
```