ArrayListBible Algorithm Analysis Worksheet SOLUTIONS

The following methods are from ***ArrayListBible***. Recall that this class has an ***ArrayList<Verse>*** called ***theVerses***. Assume ***theVerses*** has ***n*** Verses. Give tight bounds for the number of operations required for each. Then give what the ideal implementation would require. Briefly justify. A few have been done for you.

1. Assume that *ref* is the ***m***th verse in *theVerses*. Operations = $\Theta(n)$ Ideal= $\Theta(m)$

```
public boolean isValid(Reference ref) {
            boolean valid = false;
            if (ref.getBookOfBible() != BookOfBible.Dummy) {
                for (Verse v : theVerses) {
                    if (v.getReference().equals(ref)) {
                        valid = true;
                    }
                }
            }
            return valid;
    }
```

*Since this method loops through the entire ArrayList and does a constant amount of work each time, it takes some constant times **n** operations. Ideally the method would return as soon as it determined the verse was valid. This would require Θ(m) operations.*

2. Assume that *ref* is the ***m***th verse in *theVerses*. Operations = $\Theta(m)$ Ideal = $\Theta(m)$

```
public Verse getVerse(Reference ref) {
    for (Verse v : theVerses) {
        if (v.getReference().equals(ref)) {
            return v;
        }
    }
    return null;
}
```

The code in the conditional statement takes constant time, and the loop iterates over the m verses from theVerses, so it takes Θ(m) operations. Since the verse with the given reference might be anywhere, we have no choice but to search through the entire list theVerses, so the ideal is also Θ(m) operations. (Technically we can get this down to Θ(log m), but we won't worry about that right now.)

3. Assume that there are ***v*** verses in the given chapter and that the verses for this chapter start at the ***m***th verse in *theVerses*. Operations = $\Theta(vm)$ Ideal = $\Theta(m+v)$

```
public int getLastVerseNumber(BookOfBible book, int chapter) {
    int verse = 1;
    while (getVerse(new Reference(book, chapter, verse)) != null) {
        verse++;
    }
    return verse - 1;
}
```

The call to getVerse takes Θ(m) operations the first time, and then Θ(m+1), Θ(m+2), …, Θ(m+v), and then finally Θ(m+v+1), operations when it gets past the last verse of the chapter. Overall this is about Θ(vm) operations (it's actually a little more, but for simplicity we'll go with this. IN CSCI 255 we will be able to give a more careful analysis). Ideally we iterate through to find the first references with the given book and chapter, and the iterate from there (instead of from the beginning of the list) over the next v+1 verses until we find the next references where the book and chapter aren't the same. So ideally it should take Θ(m+v) operations.

4. Assume that there are ***c*** chapters and a total of ***k*** verses in the entire book and that the verses for this book start at the ***m***th verse in *theVerses*.

Operations = $\Theta(cn)$ Ideal= $\Theta(m+k)$

```
public int getLastChapterNumber(BookOfBible book) {
    int chapter = 1;
    while (isValid(new Reference(book, chapter, 1))) {
        chapter++;
    }
    return chapter - 1;
}
```

This method calls isValid c times (actually, c+1), each time requiring $\Theta(n)$ operations. Thus, it takes about $\Theta(cn)$ operations.
Ideally we would just scan until we found the book, requiring $\Theta(m)$ operations, and then scan from there until we found the end of the book, requiring an additional $\Theta(k)$ operations. The total would be $\Theta(m+k)$.

5. Assume the *references* has ***k*** elements and that on average each is at about the middle of *theVerses*.

Operations = $\Theta(kn)$ Ideal = $\Theta(kn)$

```
public VerseList getVerses(ArrayList<Reference> references) {
    VerseList results = new VerseList(version, "");
    for (Reference ref : references) {
        if (isValid(ref)) {
            results.add(getVerse(ref));
        } else {
            results.add(null);
        }
    }
    return results;
}
```

Each time through the loop, isValid gets called and getVerse might be called. Since *getVerse* takes $\Theta(m)$ time where m is the location of the verse being searched for, and we are assuming that the elements we are searching for are on average near the middle of theVerses, *getVerse takes* $\Theta(n/2)$ operations each time. isValid takes $\Theta(n)$ operations. So each time through the loop takes $\Theta(n+n/2)$= $\Theta(n)$ operations (We add here because one happens and then the other happens—like if it takes 10 minutes to drive to campus and 5 minutes to walk from your car to class, it takes 10+5=15 minutes to get to class, not 10*5=50). The loop executes k times. Thus the overall complexity is $\Theta(kn)$.
Ideally we skip the call to isValid by replacing the if-else clause and everything in it with just the statement

```
results.add(getVerse(ref));
```

This makes the code simpler and more efficient. In this case the complexity would be $\Theta(k(n/2))$= $\Theta(kn)$. Although this is the same complexity, the constants are a bit smaller.

6. Assume that *verse1* is the ***m***th verse, there are ***k*** verses between *verse1* and *verse2*, the given book contains ***c*** chapters and that on average each chapter contains ***v*** verses.

Operations = $\underline{\Theta(kvm+c^2n)}$ Ideal= $\underline{\Theta(m+k)}$

```
ArrayList<Reference> getReferencesInclusive(Reference verse1, Reference verse2) {
    ArrayList<Reference> rList = new ArrayList<Reference>();
    if (verse1.compareTo(verse2) > 0 || !isValid(verse1) || !isValid(verse2)) {
        return rList;
    }
    BookOfBible book = verse1.getBookOfBible();
    int chapter = verse1.getChapter();
    int verse = verse1.getVerse();
    while (!verse2.equals(new Reference(book, chapter, verse))) {
        if (verse == getLastVerseNumber(book, chapter) + 1) {
            chapter++;
            verse = 1;
            if (chapter == getLastChapterNumber(book) + 1) {
                book = BookOfBible.nextBook(book);
                chapter = 1;
            }
        }
        rList.add(new Reference(book, chapter, verse));
        verse++;
    }
    if (verse1.equals(verse2)) {
        rList.add(verse1);
    } else {
        rList.add(verse2);
    }
    return rList;
}
```

This method calls getLastChapterNumber about c times, each time requiring Θ(cn) time. This takes Θ(ccn) time total. It also calls getLastVerseNumber for every verse between verse1 and verse2, each time requiring Θ(vm) time. This takes a total of Θ(kvm) time. The two calls to isValid take Θ(n) time. The other operations take constant time for each verse, adding just an additional Θ(k) time. Thus, the total time required is $\Theta(kvm+ccn+n+k)=\Theta(kvm+c^2n)$.

Ideally we would scan until we found verse1 and then scan from there until we found verse2, adding each verse in between to our ArrayList. The first part of the scan would take Θ(m) operations and the in between part would take Θ(k) operations for a total of Θ(m+k).

7. Assume that *verse1* is the ***m***th verse, there are ***k*** verses between *verse1* and *verse2*, the given book contains ***c*** chapters and that on average each chapter contains ***v*** verses.

Operations = $\underline{\Theta(kvm+c^2n+kn)}$ Ideal = $\underline{\Theta(m+k)}$

```
public VerseList getVersesInclusive(Reference verse1, Reference verse2) {
    ArrayList<Reference> refs = getReferencesInclusive(verse1, verse2);
    VerseList verses = getVerses(refs);
    return verses;
}
```

getReferencesInclusive takes $\Theta(kvm+c^2n)$ operations and returns a list of k verses. It then calls getVerses which takes Θ(kn) operations. So the total time is $\Theta(kvm+c^2n+kn)$.

Ideally this would be implemented very similarly to the previous one, taking Θ(m+k) operations.

8. Assume that *verse1* is the ***m***th verse, there are ***k*** verses between *verse1* and *verse2*, the given book contains ***c*** chapters and that on average each chapter contains ***v*** verses.

Operations = *Θ(m+k)* Ideal = *Θ(m+k)*

```
ArrayList<Reference> getReferencesExclusive(Reference verse1, Reference verse2) {
    ArrayList<Reference> results = new ArrayList<Reference>();
    if (verse1.compareTo(verse2) > 0) {
        return results;
    }
    int i = 0;
    while(i<theVerses.size()&&!theVerses.get(i).getReference().equals(verse1)) {
        i++;
    }
    while(i<theVerses.size()&&theVerses.get(i).getReference().compareTo(verse2)<0) {
        results.add(theVerses.get(i).getReference());
        i++;
    }
    return results;
   }
```

This code does as the ideal solutions to the two previous problems does: Scans for verse1, taking Θ(m) operations, and then scanning from there through the k desired verses, taking Θ(k) time. The calls to isSize, get(i), getReference, equals, compareTo and add are all constant. So the total time is Θ(m+k), and this is also the ideal time.

9. Assume that *verse1* is the ***m***th verse, there are ***k*** verses between *verse1* and *verse2*, the given book contains ***c*** chapters and that on average each chapter contains ***v*** verses.

Operations = *Θ(kn+m)* Ideal = *Θ(m+k)*

```
public VerseList getVersesExclusive(Reference verse1, Reference verse2) {
    ArrayList<Reference> refs = getReferencesExclusive(verse1, verse2);
    VerseList verses = getVerses(refs);
    return verses;
}
```

getReferencesInclusive takes Θ(m+k) operations, and getVerses takes is Θ(kn) operations. So the total time is Θ(m+k+kn) = Θ(kn+m) operations (notice that k+kn=k(n+1)= Θ(kn)). As with several of the previous methods, this one can also be done in Θ(m+k) operations.

10. Assume that verses for the book start at the ***m***th verse, there are ***k*** verses in the book, and that *getReferencesForBook* takes *Θ(n)* time.

Operations = *Θ(kn)* Ideal = *Θ(m+k)*

```
public VerseList getBook(BookOfBible book) {
    return getVerses(getReferencesForBook(book));
}
```

This calls getReferencesForBook, which takes Θ(n) time, followed by calling getVerses, which takes Θ(kn) time. Thus the total time is Θ(n+kn)= Θ(kn). Again, this should be doable in Θ(m+k) time for similar reasons we have already given in previous methods.