CS310: Advanced Data Structures and Algorithms
Assignment 2 – Solution

Thanks to Prof. Betty O’Neil for parts of this solution.

1. (a) Functions ranked in order of increasing growth rate. Most of these are easily decided by looking at the limit of the ratio as $N$ grows. Numerical evidence isn’t really needed.
   - $2/N = O(1/N)$ – does not grow at all, it shrinks as $N \to \infty$
   - $38 = O(1)$ – constant
   - $\sqrt{N} = N^a = O(N^b)$ when $a \leq b$ covers lots of cases
   - $N$
   - $N \log \log N$. $\log \log N$ grows very slowly. It’s just 10 when $N = 2^{1000}$.
   - $N \log N$. This and the next are tied
   - $N \log(N^2) = 2N \log N$
   - $N^2$ (log $N$)
   - $N^3$
   - $2^{(N/2)}$
   - $2^N$ this is not a tie with $2^{(N/2)}$

(b) $\log N$ and $\log(N^2)$ are tied because $\log(N^2) = 2 \log N$. $\log^2 N = \log N \times \log N$ grows faster than $\log N$. You can divide the two and get $\log N >> \text{const}$.

2. (a) The outer loop is executed $n$ times, and the inner loop, in the worst case, is also executed $n$ times giving time complexity of $O(n^2)$.

(b) In terms of big-O both functions are $O(N^2)$. When absolute time (in seconds) is considered the second version could be a bit faster since we are doing fewer multiplications. It is possible however that due to compiler optimizations both versions will be equally fast.

(c) ```
int mysterySum(int n)
{
    int i, j, s=0;
    for (i=0; i<n; i++)
        s += i*i*i;
}
``` And the time complexity is $O(n)$.

(d) The term is $\sum(i^3) = (n*(n+1)/2)^2$. Proof by induction: True for 1 (result is 1) and 2 (result is $9 = (2*3/2)^2$). Suppose it’s true for $n-1$, that is - $\sum(i-1)^3 = (n*(n-1)/2)^2$. Let’s show that by adding $i^3$ we’ll get $(n*(n+1)/2)^2$. $(n*(n-1)/2)^2$ opens to $(n^4-2n^3+n^2)/4$. $(n*(n+1)/2)^2$ opens to $(n^4+2n^3+n^2)/4$. Subtracting the former from the latter gives us $4*n^3/4 = n^3$. Exactly what we said we had to add. QED.

3. (a) Calculating power2
int power2(int n)
{
    if (n==0) return 1;
    return power2(n-1)+power2(n-1);
}

We have a double recursion here, so obviously the runtime is bad. The formula is $T(n) = C + 2 \times T(n-1)$. So each run doubles the number of recursive calls while doing constant time operations outside the recursions. This is exponential, so $T(N) = O(2^N)$. It can be shown easily by expanding the next expression $T(n-1) = C + 2T(n-2)$, etc. The factors of 2 accumulate so that at the $k^{th}$ stage we have $2^k \times C$. Overall we have $n$ stages.

(b) A linear time algorithm would be the following:

```c
int fib(int n)
{
    if (n==0) return 1;
    return 2*power2(n-1);
}
```

We simply run the recursion once and multiply by 2... You can easily show that the recurrence formula is linear, since it's the same as the factorial function we showed in class: $T(n) = C + T(n-1)$.

(c) Bonus:

```c
int power2New(int n)
{
    if (n==0) return 1;
    if (n % 2 == 0)
    {
        int result = power2New(n/2);
        return result*result;
    }
    else
    return 2*power2New(n-1);
}
```

Explanation: If $n$ is even, then $2^n = (2^{n/2})^2$, so we can calculate $2^{n/2}$ recursively and square, cutting half of n in one move, which is similar to what we do in a binary search. So if we could always do that, the runtime formula would be the same as binary search: $T(n) = T(n/2) + C$, which is logarithmic. However, if $n$ is odd we resort to the previous method by subtracting one from n. We showed it is linear... So why is the overall runtime logarithmic despite the fact that we may have to run some “linear” stages? The answer lies in the following observation: If n is even, we divide it by 2. The next stage we get $\frac{n}{2}$ which may be even or odd. If $n$ is odd, we subtract 1 and get $n-1$ which is always even! In other words – we have an odd $n$ at most one half of the times, so the runtime is at most twice as much as the “optimal” scenario when $n$ is always even, hence $2 \times (O(\log n)) = O(\log n)$.

4. (a-b) This was a bit confusing, I apologize (this is the answer to (a) and (b) )

i. "A" is returned and then removed.
ii. Same (as I said - start from scratch).
iii. "C" is returned and then removed.
iv. "D" is returned and then removed.

(c) It will crash. Both because after removal there are no longer four positions in the list, and because we instantiate a new listIterator(2).

5. (a) Since a `HashSet` conducts a search on average in $O(1)$, the runtime will still be approx. 1ms.

(b) A search is logarithmic in the size of the table, hence the search would take just a little more than 1ms, say 1.05-1.1ms. (I am not looking for an exact number of course, just the concept).
(c) A search in a LinkedList is linear in the size of the list, so doubling the list size approximately doubles the search time to 2ms.

6. a. 'a' → 0, 'b' → 1, ..., 'z' → 25, and also 'A' → 0, ... 'Z' → 26. in one map:
   ```java
   public static int mapOneLetter(char c)
   {
       return Character.toLowerCase(c) - 'a';
   }
   ```

   b. “aa” → 0, “ab” → 1, ..., “az” → 26, “ba” → 26...
   ```java
   public static int mapTwoLetter(String s)
   {
       return (s.charAt(0) - 'a') * 26 + s.charAt(1) - 'a';
   }
   ```

c. inverse of b
   ```java
   public static String inverseMapTwoLetter(int x)
   {
       StringBuffer sb = new StringBuffer();
       sb.setCharAt(0, 'a' + x / 26);
       sb.setCharAt(1, 'a' + x % 26);
       return sb.toString();
   }
   ```

7. Map’s get(), and put() methods can be implemented in O(1) time. Also, size(), containsKey() and remove() would be O(1), too. The Set’s add(), remove(), contains(), and remove() methods are O(1) time, if we use HashSet. Also size(), isEmpty(). A List is not a good candidate for using a hash table because the notion of order is not supported by the hash table ADT. One could argue that an immutable list could be hashed based upon position, and thereby preserves the order, but once we start inserting and deleting things in the hash, “maintaining order” becomes a pain, requiring deletes and re-inserts of all the elements after the inserted or deleted one. The map operations that can be implemented with O(1) are get, put and remove. The set operations are add, remove and contains. We can’t implement a list since a list requires an order on the elements and a hash cannot provide it. Additionally, lists allow duplicates and hash does not (in the key domain).

8. a. The simplest implementation is a linked list of the lines. Implementation: Read buffer line by line; add each line to a list (add appends the line in the end of the list); when the file is done traverse the list backwards and print the lines. This is an O(N) implementation (where N is the number of lines), since add is O(1) if we have a pointer to the end of the list. A TreeMap can also be used with the line number as key, but this is less efficient since add is O(logN).

   b. Use a HashMap<String, Integer> where the keys are words and the values are their occurrence. Implementation: Read file word by word; each word – if it exists in the map, increment its counter. Otherwise, insert with counter value 1. After the file is read – traverse the map key set and print all the key values. Order is not important and therefore HashMap should be used. Complexity: O(N), where N is the number of words.

c. Exactly like b above, only print also the value in addition to the key. This is also O(N).