Data Structures and Algorithms — CSCI 230
Chapter 2 — Review Solutions

1. Show that $\sum_{i=1}^{n} 2i^3 = O(n^4)$.

   SOLUTION:

$$\sum_{i=1}^{n} 2i^3 < \sum_{i=1}^{n} 2n^3 = n \cdot 2 \cdot n^3 = 2n^4 = O(n^4)$$

2. For each of the following, find $f(n)$ such that $t(n) = O(f(n))$. Make $f(n)$ as small and simple as possible, i.e. don't write $t(n) = O(n^4)$ when $t(n) = O(n^3)$. Justify your answers.

   (a) $t(n) = 13n^2 + 2^n$
   SOLUTION:
   $t(n) = O(2^n)$ since exponentials of base greater than 1 grow faster than polynomials.

   (b) $t(n) = 5(n + 3\log n)(n \log n + 13)\log n + 13n^2$
   SOLUTION:
   $t(n) = O(n^2(\log n)^2)$. Looking at the factors of the first term yields, $(n+3\log n) = O(n)$, $(n\log n+13) = O(n\log n)$, and $\log n = O(\log n)$. Multiplying these yields $O(n^2(\log n)^2)$ which is bigger than $O(n^2)$.

   (c) $t(n) = \sum_{i=3}^{n}\sum_{j=i}^{n} i(n-j)$
   SOLUTION:
   $t(n) = O(n^4)$.

$$\begin{aligned}\sum_{i=3}^{n}\sum_{j=i}^{n} i(n-j) &< \sum_{i=3}^{n}\sum_{j=i}^{n} n(n-0) \qquad \text{since } i \le n \text{ and } j > 0\\ &= \sum_{i=3}^{n} n^3 = (n-2)n^3\\ &= O(n^4)\end{aligned}$$

3. Exercise 2.6a from the text. Try to derive summations first. Note program fragment (6) is quite difficult.

   SOLUTION:

   (1) : $O(N)$ since there is just one for loop and it executes N times.

   (2) : $\sum_{i=0}^{N-1} N = N^2 = O(N^2)$

   (3) : $\sum_{i=0}^{N-1} N^2 = N^3 = O(N^3)$

   (4) : $\sum_{i=0}^{N-1} i = \frac{(N-1)N}{2} = O(N^2)$

   (5) : $\sum_{i=0}^{N-1} \sum_{j=0}^{i^2} j \quad <= \sum_{i=0}^{N-1} \sum_{j=0}^{i^2-1} i^2$ important: j's upper bound is i^2, not N

   $$<= \sum_{i=0}^{N-1} \sum_{j=0}^{N^2-1} N^2$$

   $$= \sum_{i=0}^{N-1} N^4 = O(N^5)$$

   (6): This one is tricky! The innermost loop is only reached i times for each value of j — once each time j is a multiple of i. The `if` test, however, is done for each value of j. Thus, I have broken up the count of the `for ( int j ...` loop into two parts — one for the `if` test and one for the innermost `for` loop, which is executed in its entirety once each time j is a multiple of i. The latter can be written as a summation of i, i^2, i^3, etc. Together, these give

   $$\begin{aligned}\sum_{i=1}^{N}(N^2 + \sum_{m=1}^{i} mi) &= N^3 + \sum_{i=1}^{N} i\frac{i(i+1)}{2} \\ &\le N^3 + \sum_{i=1}^{N} N\frac{(N+1)}{2} \\ &= N^3 + N^2\frac{(N+1)}{2} = O(N^4)\end{aligned}$$

4. Derive a summation to count, as a function of n, the number of times `Hello` is output by each of the following code fragments. Obtain an accurate"O" estimate from the summation.

(a)

```
for ( i=1; i<=n; i++ )
  for ( j=1; j<=i; j++ )
    for ( k=j+1; k<=n; k++ )
      cout << "Hello\n";
```

SOLUTION:
The summation is

$$\sum_{i=1}^{n}\sum_{j=1}^{i}(n-j)$$

since the inner loop executes $n-j$ times for each value of j, the middle loop i times for each value of i, and the outer loop n times. The summation is bounded as follows:

$$\sum_{i=1}^{n}\sum_{j=1}^{i}(n-j) \le \sum_{i=1}^{n}\sum_{j=1}^{n} n = n^3.$$

Therefore, Hello is output $O(n^3)$. As an alternative (and more painful) analysis, the summation could actually be analyzed:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{i}(n-j) &= \sum_{i=1}^{n}\sum_{j=1}^{i} n \;-\; \sum_{i=1}^{n}\sum_{j=1}^{i} j \\
&= \sum_{i=1}^{n} i\cdot n \;-\; \sum_{i=1}^{n}\frac{i(i+1)}{2} \\
&= n\frac{n(n+1)}{2} \;-\; \frac{1}{2}\,\frac{n(n+1)(2n+1)}{6} \;-\; \frac{1}{2}\,\frac{n(n+1)}{2} \\
&= \frac{n^3}{3} - n^2 - \frac{n}{4}
\end{aligned}$$

Therefore, `Hello` is output $O(n^3)$ times.

(b) For this part, assume $n = 2^k$ and assume the notation `2^i` means 2^i.

```
for ( i=0; i<=k; i++ )
  for ( j=2^i+1; j<=n; j++ )
    cout << "Hello\n";
```

SOLUTION:
The summation is obviously

$$\sum_{i=0}^{k}(n-2^i)$$

Again using upper bounds:

$$\sum_{i=0}^{k}(n-2^i) \le \sum_{i=0}^{k} n = (k+1)n = n\log n + n = O(n\log n).$$

Alternatively, to be sure a smaller upper bound can not be obtained, a more extensive summation evaluation could be carried out:

$$\begin{aligned}\sum_{i=0}^{k}(n-2^i) &= n\sum_{i=0}^{k}1 \ - \ \sum_{i=0}^{k}2^i \\ &= (k+1)n \ - \ (2^{k+1}-1) \\ &= n\log n + n - 2n + 1\end{aligned}$$

This is clearly $O(n \log n)$.

5. Exercise 2.11 of the text.

SOLUTION:

This problem is better as a homework problem than as a quiz problem, but it is still a good exercise...

The simplest solution is to just test each entry of the list. This clearly requires $O(n)$ time. There is a faster way.

The following algorithm requires $O(\log n)$ time because it is essentially the same as binary search. The trick depends on two facts: the array holds integers and the array is in strictly increasing order (there are no duplicates). To see the trick, consider what we know if $a[i] < i$ for some i. Then, because of the two facts, $a[i-1] \leq a[i]-1 < i-1$, so $a[i-1] < i-1$. Similarly, we can show that $a[i-2] < i-2$, and so on, inductively, to show that $a[j] < j$ for $0 \leq j \leq i$. Similarly, we can show that if $a[i] > i$, then $a[j] > j$ for $i \leq j \leq n-1$. This gives the following algorithm:

```
int FindEqual( int pts[], int n, int & loc )
{
  int low = 0, high = n-1, mid;

  while ( low <= high ) {
    mid = (low + high) / 2;
    if ( pts[mid] == mid ) {
      loc = mid;  return true;
    }
    else if ( pts[mid] < mid )
      low = mid+1;
    else
      high = mid-1;
  }
  return false;  // at this point low > high and there is
                 // nothing left to search
}
```

Since the search range in the array is split in half each time, at most $O(\log n)$ iterations are required. And, since $O(1)$ operations are done for each iteration, the total number of operations is $O(\log n)$.

6. Write an algorithm that takes an unsorted list (provided as an array) of n floating point values and returns the smallest difference between any two values in the list. For example, for the list

```
2.9, 3.5, 1.1, 6.1, 2.3, 1.8, 8.7, 3.0, 2.4,
```

the algorithm should return 0.1, which is the difference between 3.0 and 2.9. Make your algorithm as efficient as you can and give the worst-case running time of your algorithm as a function of n, briefly justifying your answer. Hint: you may change or reorganize the contents of the array.

SOLUTION:
The trick to this algorithm is to simply sort the values into increasing order, which we know we can do in $O(n \log n)$ time using Merge Sort (among others), and then examine the difference between adjacent elements of the list to find the smallest difference. This yields the following algorithm.

```
float Smallest_Gap( float a[], int n )
{
  sort a into increasing order, for example using MergeSort;

  float smallest = a[1]-a[0];  // assume n >= 2
  for ( i = 1; i<n-1; i++ )
    if ( a[i+1] - a[i]  < smallest )
      smallest = a[i+1] - a[i];

  return smallest;
}
```

Since sorting requires $O(n \log n)$ time and the `for` loop requires just $O(n)$ time, the overall cost is $O(n \log n)$.