1. Simplify each summation to an expression in terms of n, and provide the big-Θ growth of the expression

1. $\sum_{i=1}^{n}(n - 2 \cdot i + 3)$

$$\begin{aligned}\sum_{i=1}^{n}(n - 2 \cdot i + 3) &= \sum_{i=1}^{n} n - 2\sum_{i=1}^{n} i + \sum_{i=1}^{n} 3 \\ &= n^2 - (n+1) \cdot n + 3 \cdot n \\ &= \Theta(n)\end{aligned}$$

2. $\sum_{i=0}^{n-1}(4 \cdot i^2 - 2 \cdot i + 7)$

$$4 \cdot \sum_{i=1}^{n} i^2 - 2 \cdot \sum_{i=1}^{n} i + \sum_{i=1}^{n} 7 \quad = \quad \Theta(n^3)$$

3. $\sum_{j=10}^{n} j$

$$\sum_{j=1}^{n} j - \sum_{j=1}^{9} j \quad = \quad \Theta(n^2)$$

4. $\sum_{i=1}^{n} \sum_{j=1}^{n} j$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} j \quad = \quad \Theta(n^3)$$

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

$$\sum_{i=1}^{n} \sum_{j=1}^{n} j - \sum_{i=1}^{n} \sum_{j=1}^{n} i \quad = \quad \Theta(n^3)$$

2. For each of the following code fragments, provide an appropiate summation expresssion that models it running time $T(n)$. Then simplify the summation expression to an expression in terms of n, and then determine either a big-O or big-Θ of running time $T(n)$.

1.

$$\begin{array}{ccc} sum & = & 0 \\ for(i=0; & i<n; & i++) \\ & sum++; & \end{array}$$

$$\sum_{i=1}^{n} 1 \quad = \quad \Theta(n)$$

2.

$$\begin{array}{ccc} sum & = & 0 \\ for(i=0; & i<n; & i++) \\ for(j=0; & j<n; & j++) \\ & sum++; & \end{array}$$

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{n} 1 &= \sum_{i=1}^{n} n \\ &= \Theta(n^2) \end{aligned}$$

3.

$$\begin{array}{ccc} sum = 0 & & \\ for(i=0; & i<n; & i++) \\ for(j=0; & j<n*n; & j++) \\ & sum++; & \end{array}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n^2} 1 \quad = \quad \Theta(n^3)$$

4.

$$\begin{aligned}
sum &= 0 \\
for(i = 0; \quad i < n; &\quad i++) \\
for(j = 0; \quad j < i; &\quad j++) \\
&sum++;
\end{aligned}$$

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{i} 1 &= \sum_{i=1}^{n} i \\
&= \Theta(n^2)
\end{aligned}$$

5.

$$\begin{aligned}
&sum = 0 \\
&for(i = 0; \quad i < n; \quad i++) \\
&for(j = 0; \quad j < i*i; \quad j++) \\
&for(k = 0; \quad k < j; \quad k++) \\
&\qquad sum++;
\end{aligned}$$

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{i^2}\sum_{k=1}^{j} 1 &= \sum_{i=1}^{n}\sum_{j=1}^{n^2} j \\
&= \sum_{i=1}^{n} \Theta(n^4) \\
&= \Theta(n^5)
\end{aligned}$$

3. The function int [ ] add(int[ ] a, int [ ] b, int length) inputs two length-n arrays a and b whose elements represent the digits of two nonnegative integers that are to be added (using the algorithm learned in elementary school). Here we assume that a[0] and b[0] hold the least-signifanct digits of the two integers . The function then returns an array that holds the digits of a + b. For example, to add 472 to 54, we call add on arrays a = 2, 7, 4 and b = 4, 5, 0, with n = 3, and the function returns the array 6, 2, 5. Implement this function using "Java" or "C"-like pseudocode. Then provide an appropriate summation expression that models its running time T(n). Simplify the summation expression to an expression in terms of n, and use it to determine either a big-O or big-Θ representation of running time T(n).

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<int> addition(const vector<int>& lhs, const vector<int>& rhs){
    vector<int> result;
    int carry = 0; // (1)
    // O(n)
    for(int i = 0;i < lhs.size();i++){
        int sum = lhs[i] + rhs[i] + carry;
        carry = sum / 10;
        result.push_back(sum % 10);
    }
    if(carry){
        result.push_back(1); // O(1)
    }
    return result; // O(1)
}

int main(){
    vector<int> A = {2, 7, 4};
    vector<int> B = {4, 5, 0};
    vector<int> C = addition(A, B);
    for(int e : C){
        cout << " " << e;
    }
    cout << endl;
}
```

4. The function int [ ] multiply(int[ ] a, int [ ] b, int length) inputs two length-n arrays a and b whose elements represent the digits of two nonnegative integers that are to be multiplied (using the algorithm learned in elementary school). Here we assume that a[0] and b[0] hold the least-signifanct digits of the two integers. The function then returns an array that holds the digits of a $\times$ b. For example, to multiply 54 and 145, we would pass in the arrays a = 4, 5, 0 and b = 5, 4, 1, and the function should return the array 0, 3, 8, 7. Implement this function using "Java" or "C"-like pseudocode. Then provide an appropriate summation expression that models its running time T(n). Simplify the summation expression to an expression in terms of n, and use it to determine either a big-O or big-Θ representation of running time T(n). Hint: first implement a function that multiplies a nonnegative integer by a single digit, then call this function, along with the add function within a loop to complete the entire multiplication.

```
#include <iostream>
#include <vector>

using namespace std;

// 9 * 9 = 2 digits
// 10 * 10 = 100 3 digit expansion
vector<int> multiply(const vector<int>& a, const vector<int>& b){
    vector<int> result(a.size() + b.size());// O(n)
    //O(n**2)
    for(int i = 0;i < a.size();i++){
        for(int j = 0;j < b.size();j++){
            result[i + j] += a[i] * b[j];
            result[i + j + 1] += result[i + j] / 10;
            result[i + j] %= 10;
        }
    }
    while(result.size() > 1 && result.back() == 0){
        result.pop_back();
    }
    return result;
}

int main(){
    vector<int> A = {4, 5, 0};
    vector<int> B = {5, 4, 1};
    vector<int> C = multiply(A, B);
    for(int e : C){
        cout << " " << e;
    }
```

```
    cout << endl;
}
```

5. An algorithm takes 0.5 seconds to run on an input of size 100. How long will it take to run on an input of size 1000 if the algorithm has a running time that is linear? quadratic? log-linear? cubic?

$$\begin{aligned} 100 \cdot c &= 0.5 \\ c &= \frac{0.5}{100} \\ 1000 \cdot \frac{0.5}{100} &= 5 \end{aligned}$$

$$\begin{aligned} (10^2)^2 \cdot c &= 0.5 \\ 10^4 \cdot c &= 0.5 \\ c &= \frac{0.5}{10^4} \\ (10^3)^2 \cdot \frac{0.5}{10^4} &= 10^2 \cdot 0.5 \end{aligned}$$

$$\begin{aligned} 100 \cdot log_2(100) \cdot c &= 0.5 \\ c &= \frac{0.5}{100 \cdot log_2(100)} \end{aligned}$$

$$1000 \cdot log_2(1000) \cdot \frac{0.5}{100 \cdot log_2(100)}$$

$$\begin{aligned} (10^2)^3 \cdot c &= 0.5 \\ 10^6 \cdot c &= 0.5 \\ c &= \frac{0.5}{10^6} \\ (10^3)^3 \cdot \frac{0.5}{10^6} &= 0.5 \cdot 10^3 \end{aligned}$$

7. Suppose that the insertion sort algorithm has a running time of $T(n) = 8 \cdot n^2$, while the counting sort algorithm has a running time of $T(n) = 64 \cdot n$. Find the largest positive input size for which insertion runs at least as fast as counting sort.

$$\begin{aligned} 8 \cdot n^2 &\leq 64 \cdot n \\ n &\leq 8 \end{aligned}$$

$n = 8$

9. Consider the problem of computing a n where n is a positive integer. One method is to start with a product of 1, iterate n times, and multiply the product by a each time. The following, however, is a faster approach. Write n as a binary number. For example, suppose, n = 2b1 + · · · + 2br , then start with a product of 1, and multiply the product by each of a 2 bi . In other words, we only multiply with exponents that are powers of 2. For example, to compute 36 , we would multiply 1 by both 34 and 32 , since 36 = 32+4. How does this reduce the running time? Hint: how many multiplications does this algorithm need? Implement this algorithm in pseudocode.

```
#include<iostream>

using namespace std;

/**
 * T(n) = T(n / 2) + O(1)
 * T(n) = O(log_2(n))
 */
int fast_power(int x, int y){
    if(y == 1) return x;
    int ans = fast_power(x * x, y / 2);
    if(y % 2 == 0) return ans;
    return ans * x;
}

int main(){
    cout << fast_power(2, 3) << endl;
    cout << fast_power(2, 4) << endl;
    cout << fast_power(3, 7) << endl;
}
```

10. Given as input a sorted integer array $a[0] < a[1] < \ldots < a[n-1]$, provide an algorithm with $O(log_2(n))$ running time that checks if there is an i for which $a[i] = i$. Describe your algorithm in words.

```
#include <iostream>
#include <vector>

using namespace std;

/**
 * T(n) = T(n / 2) + O(1)
 * T(n) = O(log_2(n))
 */
int index_equal_to_value(const vector<int>& A){
    int lo = 0, hi = A.size() - 1;
    while(lo <= hi){
```

```
        int mid = (lo + hi) / 2;
        if(A[mid] < mid) lo = mid + 1;
        else if(A[mid] > mid) hi = mid - 1;
        else return mid;
    }
    return -1;
}

int main(){
}
```

12. Describe how you could modify any algorithm so that is has a good best-case running time

For a given input, check if it a special case that can be easily solved. If an array is sorted do a linear scan to verify and then abort a sorting algorithm would be an example.