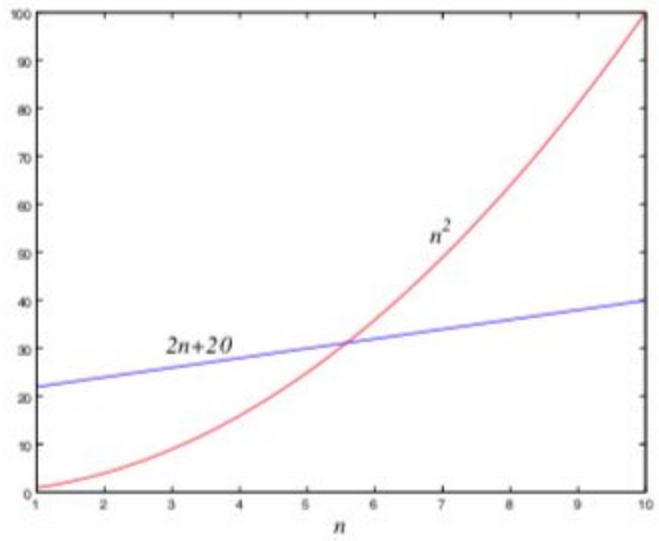
1. Suppose we are choosing between two algorithms for a particular computational task. One takes f1(n) = n^2 steps, while the other takes f2(n) = 2n + 20 steps.



What is the big-O for each of these? Which is better and why?

- f1(n) = n^2

f1(n) is O(**n^2**). For f1(n) = n^2 ∈ O(g(n)), we need to find c and n_0 such that $n^2 \le c \cdot g(n)$, ∀n ≥ n_0 . If g(n) = n^2,by inspection, it's clear that c must be equal or larger than 1. Let **c = 1**. Now we need a suitable n_0 . In this case, f(1) = 1 · g(1). Because the definition of O() requires that f(n) ≤ c · g(n), we can select n_0 **= 1**. In summary, f1(n) is O(**n^2**).

- f2(n) = 2n + 20

f2(n) is O(**n).** For f2(n) = 2n + 20 ∈ O(g(n)), we need to find c and n_0 such that 2n + 20 ≤ c · g(n), ∀n ≥ n_0 .

Analysis: f(n) = 2n + 20;

Choose: g(n) = n , c = 2, n_0 = 20;

Verify: 2n + 20 ≤ 2n + 20 for n ≥ 20;

Therefore: f(n) = n is O(n).

big-O for factorial(n)

If g(n) = n,by inspection, it's clear that c must be larger than 2. Let **c = 3**. Now we need a suitable n_0 . In this case, f(1) = 3· g(1). Because the definition of O() requires that f(n) ≤ c · g(n), we can select n_0 **= 20**, or any integer above 20 – they will all work. Therefore, f2(n) is O(**3n)**.

- If n = 5, f1(n) = 25, f2(n) = 30. If n = 6, f1(n) = 36, f2(n) = 32. When n is less than 6, f1(n) is better than f2(n). When n is equal or greater than 6, f2(n) is better. f1(n) is in the polynomial time range whereas f2(n) is linear. The latter grows much slower than the former one as n gets larger.

2. A function finds the largest and the smallest values in an unsorted array. Show the f(n) for this function, and use the definition of big-O to show that f(n) is O(n).

Answer: The algorithm is to set the largest and smallest values to be the first element and then loop through the array: if the current value is greater or equal to the largest value, set the largest value to the current value; if the current value is smaller or equal to the smallest value, set the smallest value to the current value. At the end of the loop, the largest and smallest values are found. Pseudocode is provided as follows.

```
int[] maxAndMinElement(int a[], int n) {

        int min = a[0];

        int max = a[0];

        for (int i = 0; i < n; i++) {

                if (a[i] >= max) {

                        max = a[i];}

                if (a[i] <= min){

                        min = a[i]}

return [m,n];}
```

The detailed analysis of time complexity is as follows:

Analysis: f(n) = 2n;

Choose: g(n) = n, c = 1/2, n_0 = 4;

Verify: 2n ≤ ½ *n for n_0 ≥ 4;

Therefore: f(n) = 2n is **O(n)**.

3. We saw that the bubble sort algorithm shown in class is $O(n^2)$. Suppose that we add a test that exits the algorithm early if there are no swaps in the inner loop for a given iteration of the outer loop. Show the f(n) and the big-O for this modified algorithm. Does the modification change the big-O for this algorithm?

Answer: The f(n) and big-O for the modified algorithm does not change because in the worst case where the array is in reversed order, the modified algorithm would have to go through the whole inner loop for each iteration rather than skipping any step, which results in n^2 of time complexity. The detailed analysis is as follows:

Analysis: f(n) = 0 + 1 + 2 + ... + (n-1) = (n * (n-1))/2 = (n^2 - n) / 2

Choose: g(n) = n^2 , c = 1/2, $n_0 = 1$;

Verify: ½ · n^2 – ½ · n ≤ ½ · n^2 for n ≥ 1;

Therefore: f(n) = ½ · n^2 - ½ · n is O(n^2).

The modified algorithm may work in some cases. For example, in the "best" case where the array is already sorted, then the whole inner loop is skipped for each iteration, thus resulting in a time complexity of O(n).

4. The formula for factorial(n) or n! is:

factorial(0) = 1
factorial(n) = n * factorial(n-1) (n>0)

Use the definition of big-O to show that the big-O for factorial(n) is O(n).

Answer: The algorithm for factorial(n) is to loop through the factorial of n elements (n, n-1, n-2, ... 3, 2, 1, 0). Therefore, the time complexity is O(n). Pseudocode and detailed analysis is provided as below:

```
int fact(unsigned int n) {

    if (n == 0) {return 1; // base cases } else { return fact(n-1); // inductive case}}
```

Detailed analysis:

Analysis: f(n) = 1 + 1 + 1 + 1 ..... + 1 = n;

Choose: g(n) = n , c = 1, $n_0 = 1$;

Verify: n ≤ 1*n for n ≥ 1;

Therefore: f(n) = n is O(n).