Revised: 12/17/2019

Algorithm Complexity II

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Jump Search
  - Square Root Time
- Fisher-Yates Shuffle
- Exponential Time
- Bogosort
  - Factorial Time

Jump Search

- Like Binary Search, the array must be pre-sorted.

- Begins searching at index 0 of an array.
  - Jumps *x*-number of indexes ahead.
  - If the value sought is larger, it jumps forward again.

  - If the value sought is smaller, it jumps back one step.
    - Performs a linear search from there, up to where it jumped back from.

Jump Search (C++ Function)

```
int jumpSearch(int a[], int length, int searchValue) {
    int previous = 0;
    int jLength = (int) sqrt(length);
    int jump = jLength
    while(a[(jump < length ? jump : length-1)] < searchValue) {
        previous = jump;
        jump += jLength;
        if(jump >= length) {
            break;
        }
    }
    while(a[previous] < searchValue) {
        previous += 1;
        if(previous == (jump < length ? jump : length)) {
            return -1;
        }
    }
    return a[previous] == searchValue ? previous : -1;
}
```

Keeps jumping forward

Chooses the smaller of the two

Prevents going out of bounds

Goes back and linear searches starting at the last jump point

Chooses the smaller of the two

Jump Search Time Cost

- The jump length will play a part in the time complexity of this algorithm.

- We chose the jump length to be the square root of the array's length.
  - This ensures the longest, yet most evenly distributed jumps possible.

Jump Search Time Cost

- We'll ignore the constant time operations of this algorithm and focus on the repetitive aspects.
  - We'll also use **n** to represent the length of the array being searched.
- The first while loop will iterate, at most, $\sqrt{\mathbf{n}}$ times.
  - If the jump length is the square root of the array length.
- The second while loop will iterate using a linear search.
  - We know the linear search performs in linear time.
  - However, this linear search will only iterate, at most, $\sqrt{\mathbf{n}}$ times.

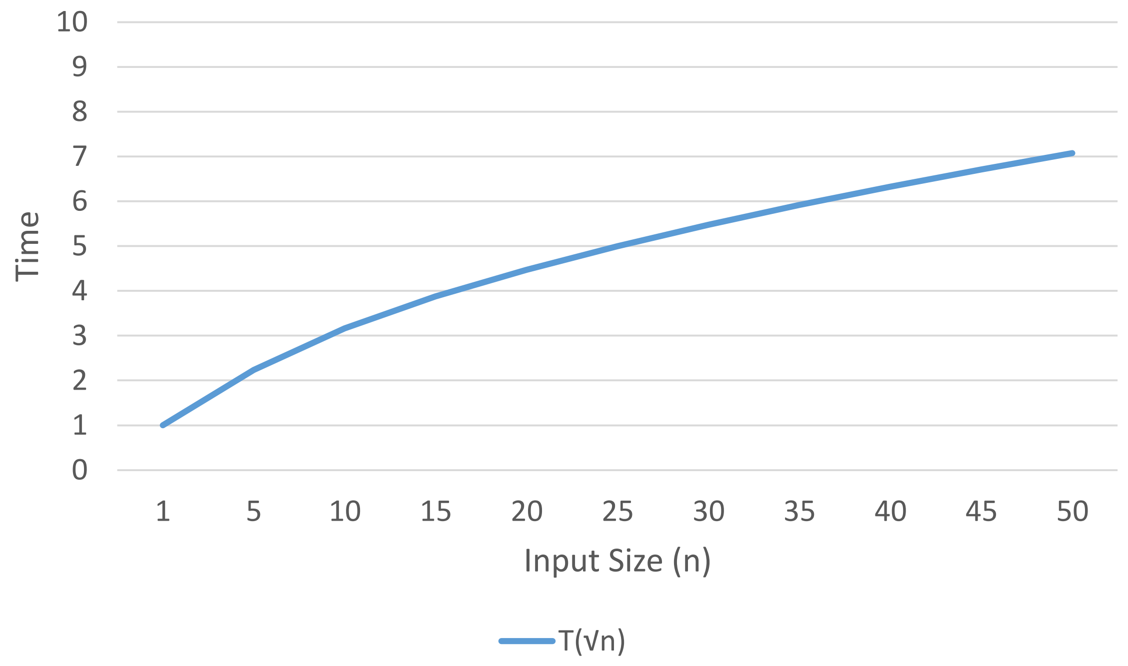
Jump Search Time Cost

- Again, ignoring the constant time operations of this algorithm, it's time cost can be estimated to be:
  - $T(n) = \sqrt{n} + \sqrt{n} = 2(\sqrt{n})$

Square Root Time

- **Square root time** is when an algorithm's running time is bounded by the square root of the input size.
  - $T(n) = \sqrt{n}$

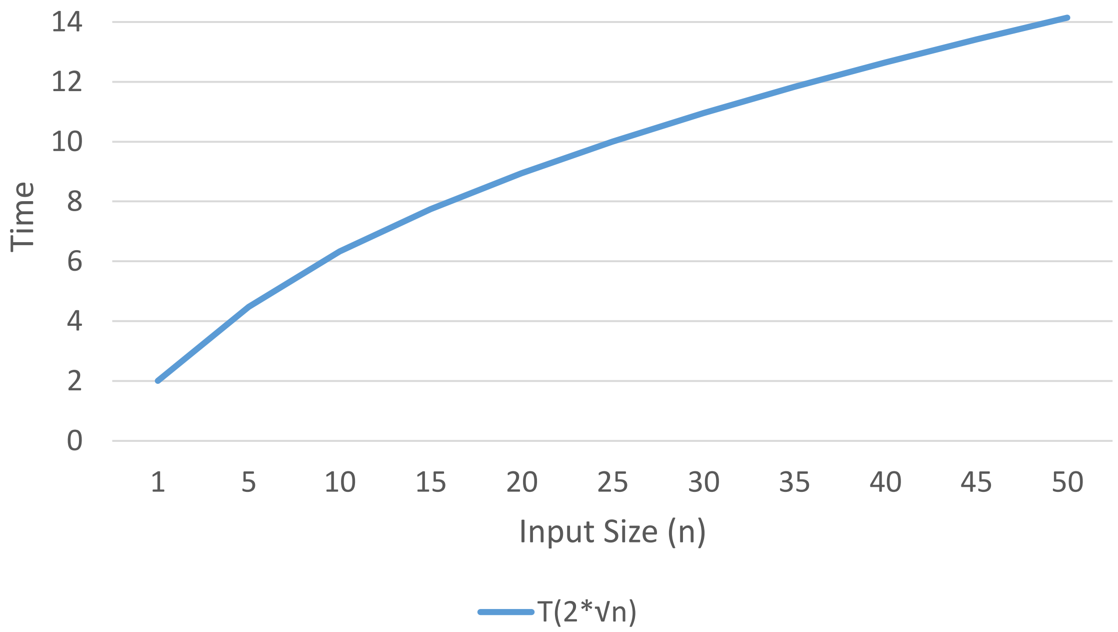
- $T(n) = \sqrt{n}$
  - $T(4) = 2$
  - $T(8) = 2.8$
  - $T(12) = 3.5$



Time Cost of the Jump Search

- The Jump Search performs in square root time.

- Using our example:
  - $T(n) = 2(\sqrt{n})$

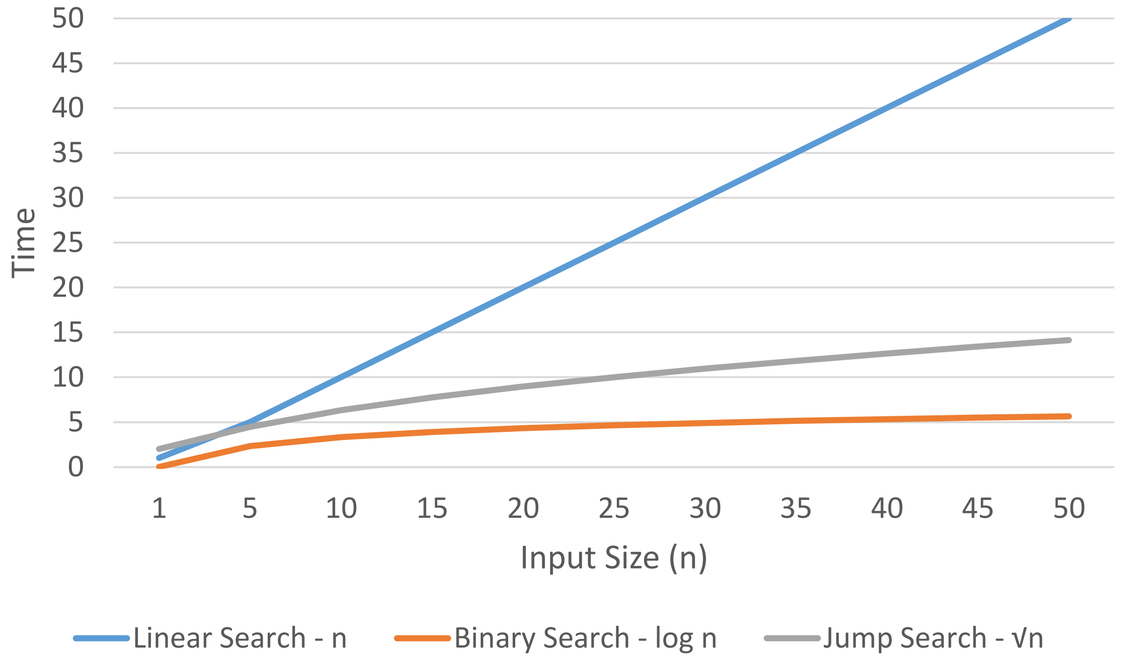


Linear vs Square Root vs Logarithmic

- We’ve seen three search algorithms with different time complexity.
  - Linear Search – Linear
  - Binary Search – Logarithmic
  - Jump Search – Square Root

- How do they compare?
  - We’ll use the following estimates:
  - Linear Search – $T(n) = n$
  - Binary Search – $T(n) = \log_2 n$
  - Jump Search – $T(n) = \sqrt{n}$

Linear vs Logarithmic vs Square Root Time



- Square root time does better than linear time, but it’s not as fast as logarithmic time.

Fisher-Yates Shuffle

- Nearly all of the algorithms we have seen solve the problem of sorting.

- The **Fisher-Yates Algorithm** is an algorithm that shuffles/randomizes the order of values in an array.
  - Performs in linear time.

Fisher-Yates Shuffle (C++ Function)

```
void shuffle(int a[], int length) {
    srand((int)time(0));
    for(int i = 1; i < length; i++) {
        int j = rand() % (i+1);
        int temp = a[j];
        a[j] = a[i];
        a[i] = temp;
    }
}
```

Sets seed to current time
(seconds since midnight, 1/1/1970)

Random number between 0 (included) and i+1 (excluded)

- Requires **#include<ctime>** for the time function.

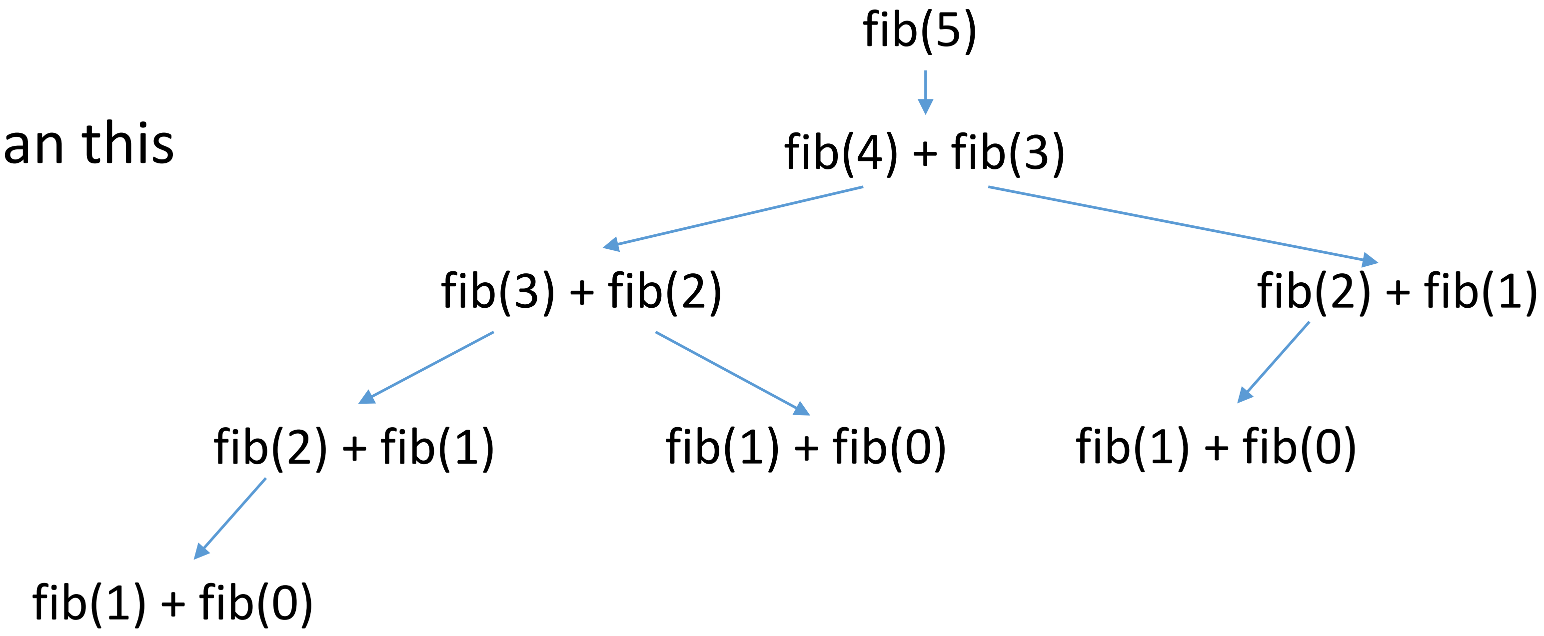
Exponential Time

- Some recursive algorithms run in exponential time.
- The recursive solution to solving a Fibonacci series runs in exponential time.
  - Though, the iterative solution runs in linear time.

```
int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Exponential Time

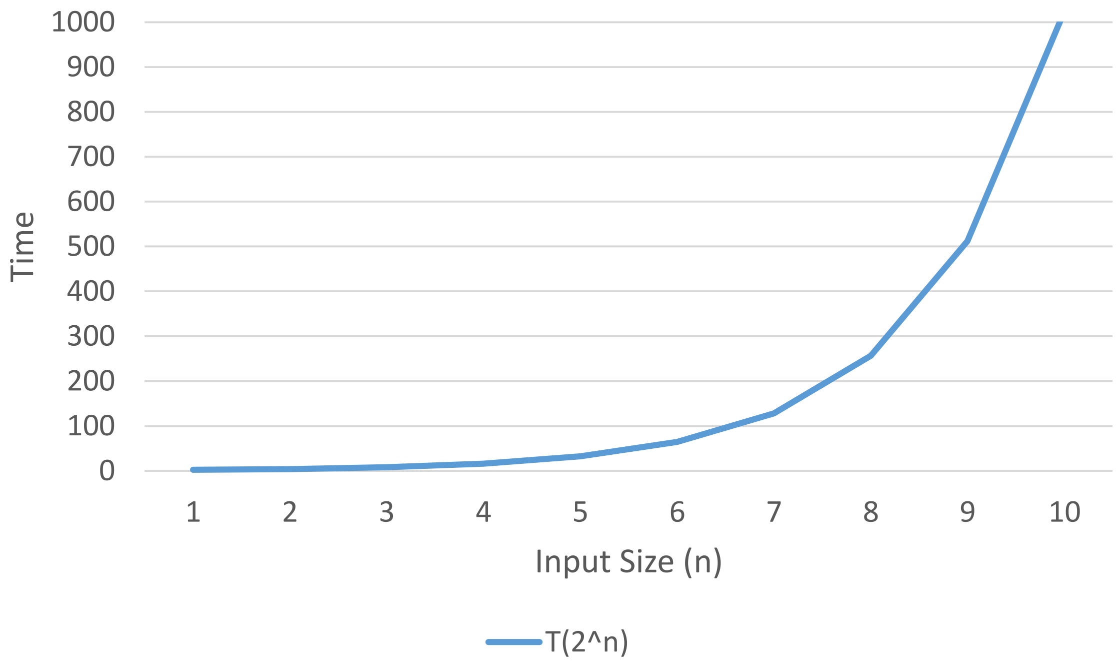
- Almost every function call splits into 2 more function calls.
- The depth is n levels
- $T(n) = 2^n$
  - Does a bit better than this



Exponential Time

- Exponential time is when an algorithm's running time is bounded by a constant raised to some power of n.
  - $T(n) = 2^n$

- $T(n) = 2^n$
  - $T(4) = 16$
  - $T(8) = 256$
  - $T(12) = 4096$



Bogosort

- Also called “Stupid Sort”, “Monkey Sort”, or “Shotgun Sort”

- Checks if the array is in order.
  - If it isn’t, it randomly shuffles the array.
  - Checks to see if it is in order.
  - If it isn’t, it shuffles it again and repeats.

- Infinite Monkey Theorem: Given enough time, a monkey in front of a typewriter randomly pressing keys would eventually produce the complete works of William Shakespeare.

Bogosort (C++ Function)

Checks if the ordering is correct

Shuffles the array

```
void bogosort(int a[], int length) {
     bool sorted;
     srand((int)time(0));
     do {
          sorted = true;
          for(int i = 1; i < length; i++) {
               if(a[i-1] > a[i]) {
                    sorted = false;
               }
          }
          if(!sorted) {
               for(int i = 1; i < length; i++) {
                    int j = rand() % (i+1);
                    int temp = a[j];
                    a[j] = a[i];
                    a[i] = temp;
               }
          }
     } while(!sorted);
}
```

Bogosort

- Obviously, this is a really bad sorting algorithm.

- It essentially tests different, possibly recurring permutations of the numbers in the array seeing if it was randomly placed in order.
  - The possibly recurring part is what makes it even worse.
  - It could “sort” the list in the same, wrong order multiple times.

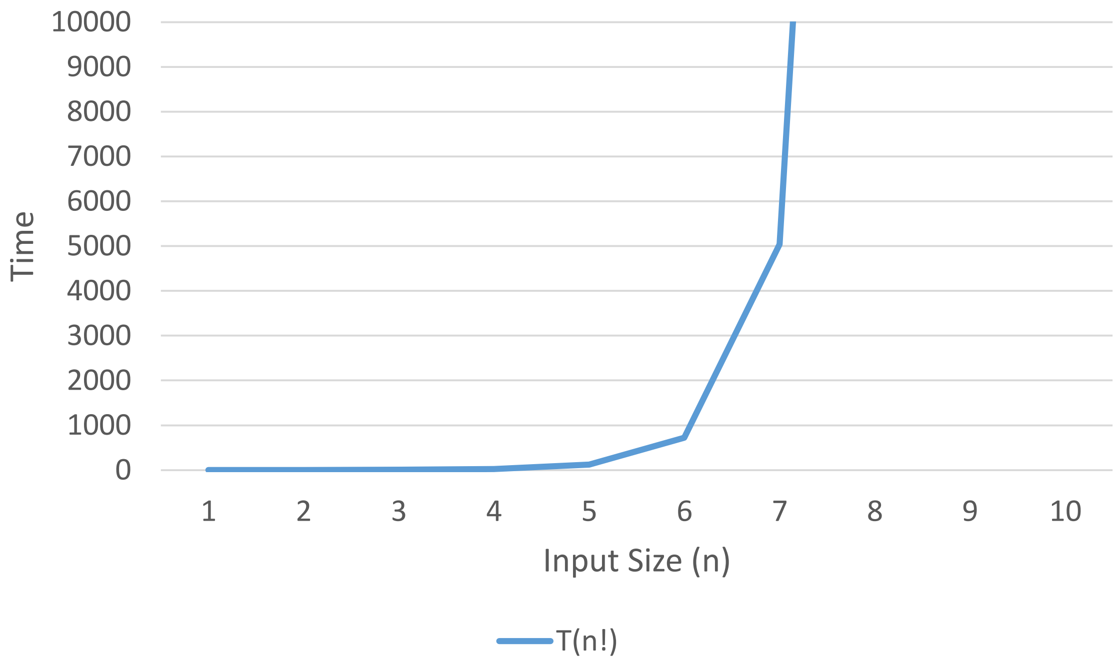
Bogosort

- For an array with a length of 5, there are 120 possible permutations of the values.
  - 1 * 2 * 3 * 4 * 5 = 5! = 120
- For an array with a length of 100, there are $9.33*10^{157}$ possible permutations.
  - $100! = 9.33*10^{157}$
- For an array of length n, there are n! possible permutations.

Factorial Time

- **Factorial time** is when an algorithm's running time is bounded by a factorial of the input size.
  - $T(n) = n!$

- $T(n) = n!$
  - $T(4) = 24$
  - $T(8) = 40320$
  - $T(12) = 479001600$



Factorial Time

- Technically, the version of the Bogosort shown is unbounded to a time complexity since the same permutation might appear multiple times.
  - There are other variations where it doesn't get the same permutation more than once.
- Bogosort's only practical purpose is being a demonstration of an algorithm that performs in factorial time.
  - Don't actually use it in the real world.

Time and Space Complexity of These Algorithms

- Time Complexity
  - Jump Search – Square Root
  - Fisher-Yates Shuffle – Linear
  - Bogosort – Factorial
- Space Complexity
  - Jump Search –Linear
  - Fisher-Yates Shuffle –Linear
  - Bogosort –Linear

O-Notation

- O-notation for algorithms seen:
  - Linear Search — $O(n)$
  - Binary Search — $O(\log n)$
  - Jump Search — $O(\sqrt{n})$

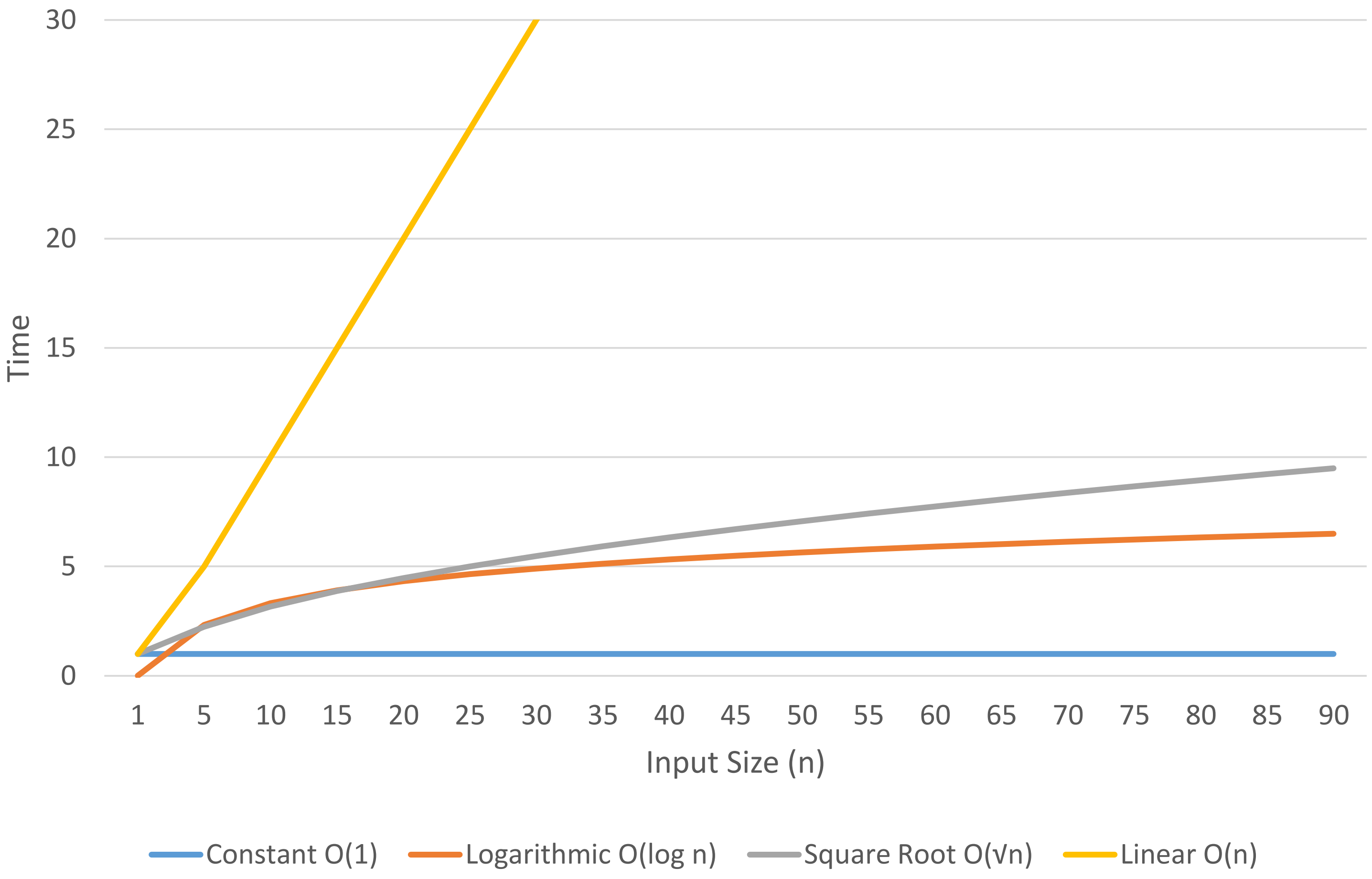
  - Bubble Sort — $O(n^2)$
  - Insertion Sort — $O(n^2)$
  - Selection Sort — $O(n^2)$
  - Bogo Sort — $O(n!)$

Space Complexities

- O-notation for algorithms shown (Auxiliary space only):
  - Linear Search O(1)
  - Binary Search O(1)
  - Jump Search O(1)

  - Bubble Sort O(1)
  - Insertion Sort O(1)
  - Selection Sort O(1)
  - Cocktail Sort O(1)
  - Bogo Sort O(1)

Complexities



Complexities