# Complexity & Efficiency

# Conceptual Overview

- Arrays

- Linked list

# Abstract Data Types (more details)

- Stack: entries are only inserted and removed at the head
  - Last in, First out (LIFO)
  - Push: add to the top/front
  - Pop: remove from the top/front
  - Ideal for storing items that must be retrieved in the reverse order from which they are stored

# Abstract Data Types (more details)

- Queue: entries only removed at the front, entries only added to the back
  - FIFO
  - Push: add to the back
  - Pop: remove from the front

# Designing Good Code

- Consider the sorting algorithms from last week
  - Some were slow
  - Some were faster

- How should we decide which scheme to use?
  - One consideration is “runtime complexity”
  - Assuming an input of **n** items, how long would an operation take?

# Introduction to Runtime Complexity

- Algorithms take time to run
- Clock time varies
  - Can vary based on input
  - Can vary based on number and kind of steps
- Typically talk about runtime in an abstract sense
  - Big O – worst case bound
  - Big Ω – best case bound
  - We ignore “constant” factors

# Runtime Complexity

- Example: Adding an element to the front of a linked list

# Code Example: O(1)

```
struct node* push (struct node * head, int n) {
    struct node *temp = malloc(sizeof(struct node));
    temp->val = n;
    temp->next = head;
    head = temp;
    return head;
}
```

# Runtime Complexity

- Example: Insert an element to the beginning of an array

# Runtime Complexity

- Example: Counting number of elements in linked list

# Code Example: O(n)

```
int length(struct node *head) {
    int n=0;
    while (head != NULL) {
        n++;
        head = head->next;
    }
    return n;
}
```

# Two Examples of Searching Algorithms

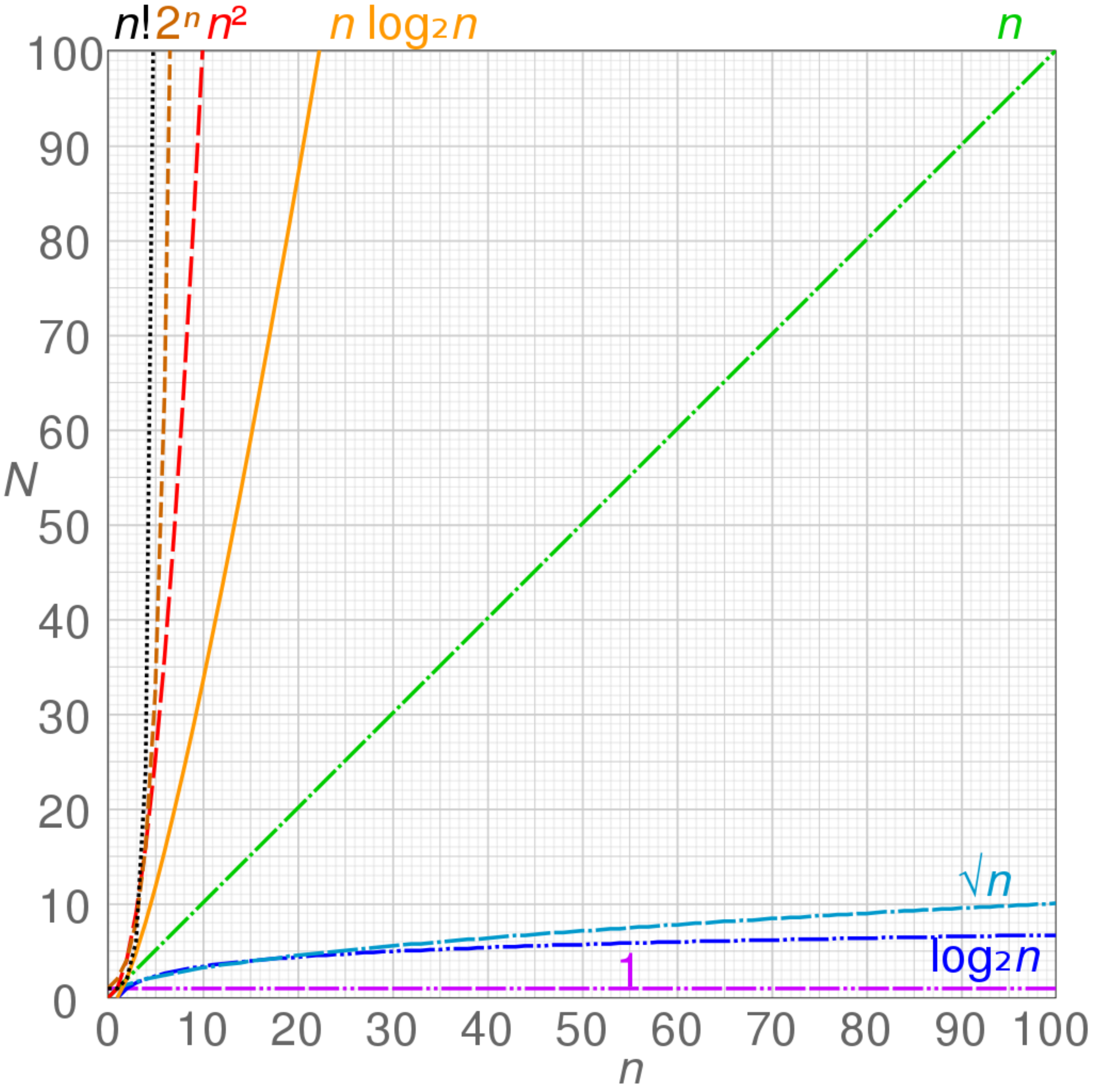
- Best strategy depends on format of data
  - Are you working with sorted or unsorted values?
- Linear Search
  - Start from index 0, check for desired value, move to next element, repeat the process
- Binary Search
  - Only works with sorted data
  - Starts in the middle and keeps dividing the dataset into two parts
- Comparison: https://www.geeksforgeeks.org/linear-search-vs-binary-search/

# Another Code Example: $O(n^2)$

```
void bubble_sort(struct node *head, int size) {
     …
     int iteration, i;
     for (iteration=1; iteration<size; iteration++) {
          for (i=0; i<size-iteration; i++) {
               if (current->val >current->next->val) {
                    //swap values
                              }
               //move current to next node
          }
          current = head;
     }
}
```

# Different Times

- $O(1)$ – constant complexity
- $O(\log n)$ – log-n complexity
- $O(\sqrt{n})$ – root-n complexity
- $O(n)$ – linear complexity
- $O(n \log n)$ – n-log-n complexity
- $O(n^2)$ – quadratic complexity
- $O(n^3)$ – cubic complexity
- $O(2^n)$ – exponential complexity
- $O(n!)$ – factorial complexity



n! 2ⁿ n² n log₂n n
100
90
80
70
60
N
50
40
30
20
10
0
√n
log₂n
1
0 10 20 30 40 50 60 70 80 90 100
n

# Real-world Considerations

- Your program will only perform as well as your design
  - Constant factors can still play a part
- Suppose you have two algorithms…
  - Algorithm A) 1,000,000n → O(n)
  - Algorithm B) $2n^2$ → $O(n^2)$
  - Which one is better?
    - It depends