Big O Notation

"Measuring programming progress
by lines of code is like measuring
aircraft building progress by weight." --Bill Gates

Nabil M. Al-Rousan

Example

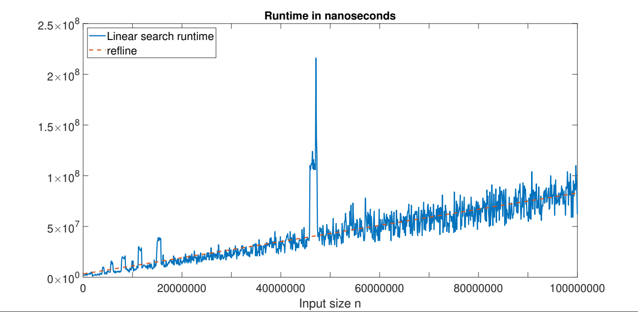
```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Example

How much time does it take to run this function?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

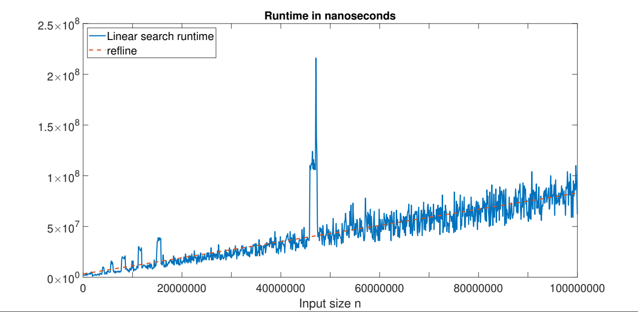
Absolute Time vs. Time Growth[2]



[1]runtime: time it takes to execute a piece of code

[2]a.k.a Complexity

Absolute Time vs. Time Growth[2]

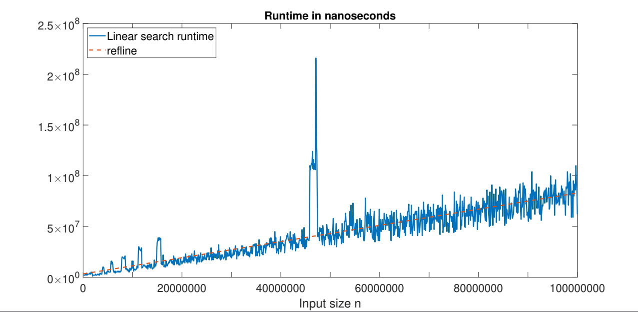


1. How much time does it take to run this function?
   Answer: .1 seconds for 100×10^6 array size

[1]runtime: time it takes to execute a piece of code

[2]a.k.a Complexity

Absolute Time vs. Time Growth[2]



1. How much time does it take to run this function?
   Answer: .1 seconds for 100×10^6 array size
2. How does the runtime[1] of this function grow?
   Answer: Linear

[1] runtime: time it takes to execute a piece of code
[2] a.k.a Complexity

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Linear time

Constant time

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Linear time

Constant time

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Linear time

Constant time

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Linear time

Constant time

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int  linear_search(int[] arr, int  target) {
    for (int  i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return  i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth Analysis: Linear

Can we analyze code to find runtime growth?

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$$T(n) = \overbrace{a \times n} + b$$

$$< a \times n$$

$$< n$$

$$= \mathcal{O}(n)$$

1. add different steps
2. drop non-dominate terms
3. drop constants

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

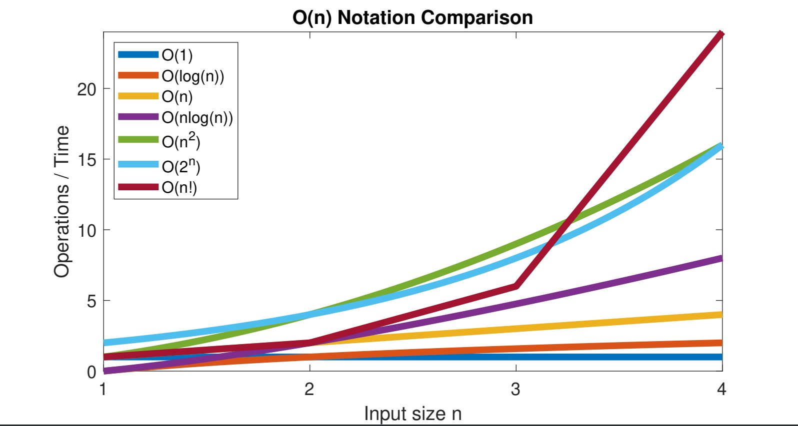
Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Runtime Growth in terms of Big O

Runtime Growth	Big O Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Big O Comparison



Growth of different Big $\mathcal{O}$ notations

$100 \stackrel{?}{=} 10 \times 10$

How much increasing the input affect the growth rate?

Big O Notation	Operations for input size 10	Operations for input size 100
$\mathcal{O}(1)$	1	1
$\mathcal{O}(\log n)$	3.3219	6.6439
$\mathcal{O}(n)$	10	100
$\mathcal{O}(n \log n)$	33.219	664.39
$\mathcal{O}(n^2)$	100	10000
$\mathcal{O}(2^n)$	1024	1267650600228229 401496703205376
$\mathcal{O}(n!)$	3628800	9.3326215443944152681699238856267e+157

$100 \stackrel{?}{=} 10 \times 10$

How much increasing the input affect the growth rate?

Big O Notation	Operations for input size 10	Operations for input size 100
$\mathcal{O}(1)$	1	1
$\mathcal{O}(\log n)$	3.3219	6.6439
$\mathcal{O}(n)$	10	100
$\mathcal{O}(n \log n)$	33.219	664.39
$\mathcal{O}(n^2)$	100	10000
$\mathcal{O}(2^n)$	1024	126765060022822940149670320537 6
$\mathcal{O}(n!)$	3628800	9.33262154439441526816992388562 67e+157

$100 \stackrel{?}{=} 10 \times 10$

How much increasing the input affect the growth rate?

Big O Notation	Operations for input size 10	Operations for input size 100
$\mathcal{O}(1)$	1	1
$\mathcal{O}(\log n)$	3.3219	6.6439
$\mathcal{O}(n)$	10	100
$\mathcal{O}(n \log n)$	33.219	664.39
$\mathcal{O}(n^2)$	100	10000
$\mathcal{O}(2^n)$	1024	126765060022822940149670320537 6
$\mathcal{O}(n!)$	3628800	9.332621544394415268169923885626 7e+157

$$100 \stackrel{?}{=} 10 \times 10$$

How much increasing the input affect the growth rate?

Big O Notation	Operations for input size 10	Operations for input size 100
$\mathcal{O}(1)$	1	1
$\mathcal{O}(\log n)$	3.3219 ←	6.6439 (Twice the time!)
$\mathcal{O}(n)$	10 ←	100 (10 times)
$\mathcal{O}(n \log n)$	33.219	664.39
$\mathcal{O}(n^2)$	100	10000
$\mathcal{O}(2^n)$	1024	1267650600228229401496703205376
$\mathcal{O}(n!)$	3628800	9.3326215443944152681699238856267e+157

$$100 \stackrel{?}{=} 10 \times 10$$

How much increasing the input affect the growth rate?

Big O Notation	Operations for input size 10	Operations for input size 100
$\mathcal{O}(1)$	1	1
$\mathcal{O}(\log n)$	3.3219	6.6439
$\mathcal{O}(n)$	10	100
$\mathcal{O}(n \log n)$	33.219	664.39
$\mathcal{O}(n^2)$	100	10000
$\mathcal{O}(2^n)$	1024	1267650600228229 401496703205376
$\mathcal{O}(n!)$	3628800	9.332621544394415268 1699238856267e+157

~~Runtime Growth~~ *Time Efficiency* in terms of Big O

~~Runtime Growth~~ *Time Efficiency*	**Big O Notation**
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
loglinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$
exponential	$\mathcal{O}(2^n)$
factorial	$\mathcal{O}(n!)$

Big O Applications

- compares the performance of different algorithms
  - searching (linear search vs. binary search)
  - sorting (insertion sort, bubble sort, merge sort etc.)
- describes the worst-case scenario of an algorithm

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

  - What if target was at first index?

Big O Applications

- compares the performance of different algorithms
  - searching (linear search vs. binary search)
  - sorting (insertion sort, bubble sort, merge sort etc.)
- describes the worst-case scenario of an algorithm

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

  - What if target was at first index?

Big O Applications

- compares the performance of different algorithms
  - searching (linear search vs. binary search)
  - sorting (insertion sort, bubble sort, merge sort etc.)
- describes the worst-case scenario of an algorithm

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

  - What if target was at first index?

Big O Applications

- compares the performance of different algorithms
  - searching (linear search vs. binary search)
  - sorting (insertion sort, bubble sort, merge sort etc.)
- describes the worst-case scenario of an algorithm

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

  - What if target was at first index?

Big O Applications

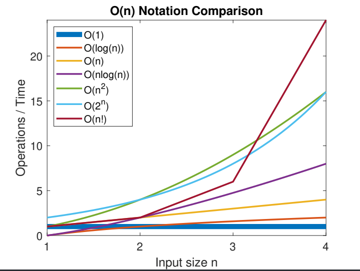
- compares the performance of different algorithms
  - searching (linear search vs. binary search)
  - sorting (insertion sort, bubble sort, merge sort etc.)
- describes the worst-case scenario of an algorithm

```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

  - What if target was at first index?

$\mathcal{O}(1)$ - Constant time

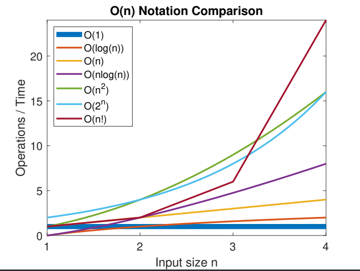
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(1)$ - Constant time

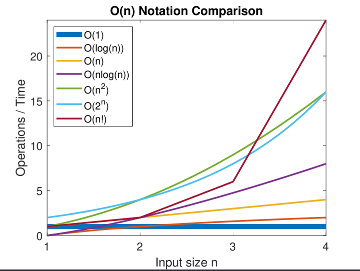
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(1)$ - Constant time

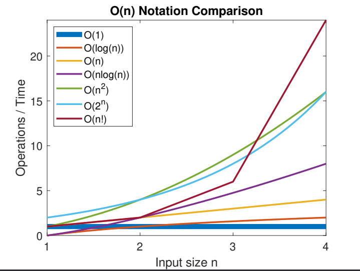
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(1)$ - Constant time

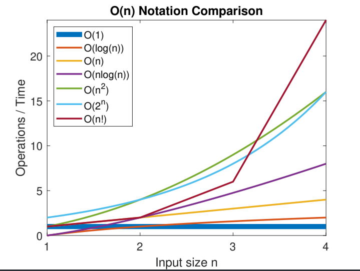
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(1)$ - Constant time

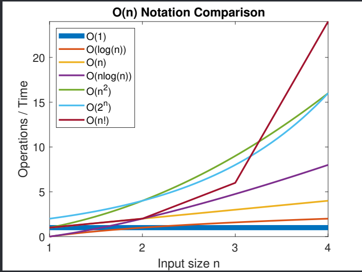
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(1)$ - Constant time

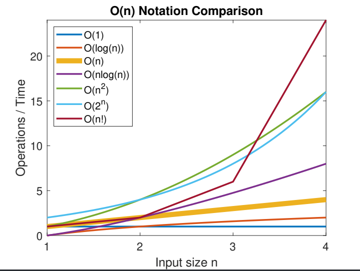
- Algorithm executes in the same execution time regardless of the size of the data set.
- Examples:
  - determining if a number is even or odd.
  - using a constant-size lookup table or hash table.




```
int return_first(int[] arr) {
        return arr[0];
}
```

$\mathcal{O}(n)$ - Linear time

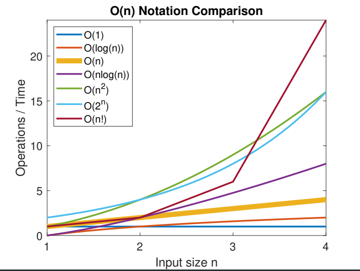
- Execution time of an algorithm $\propto$ size of the data *n*.
- Examples:
  - Traversing an array, linked list, .. etc.




```
int linear_search(int[] arr, int
    target) {
    for (int i = 0; i < arr.length; i
        ++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$\mathcal{O}(n)$ - Linear time

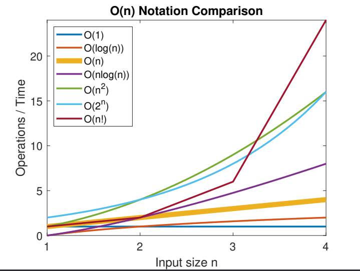
- Execution time of an algorithm $\propto$ size of the data n.
- Examples:
  - Traversing an array, linked list, .. etc.




```
int linear_search(int[] arr, int
    target) {
    for (int i = 0; i < arr.length; i
        ++) {
        if (arr[i] == target)
            return i;
    }
    return −1;
}
```

$\mathcal{O}(n)$ - Linear time

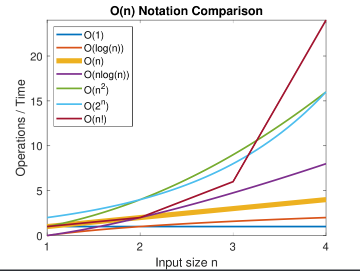
- Execution time of an algorithm $\propto$ size of the data *n*.
- Examples:
  - Traversing an array, linked list, .. etc.




```
int linear_search(int[] arr, int
      target) {
    for (int i = 0; i < arr.length; i
          ++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$\mathcal{O}(n)$ - Linear time

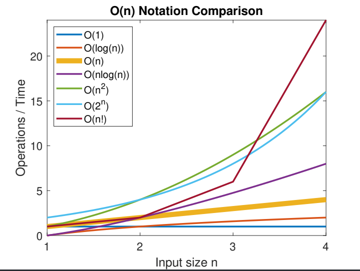
- Execution time of an algorithm $\propto$ size of the data n.
- Examples:
  - Traversing an array, linked list, .. etc.




```
int linear_search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$\mathcal{O}(n)$ - Linear time

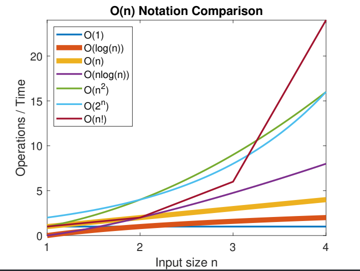
- Execution time of an algorithm $\propto$ size of the data n.
- Examples:
  - Traversing an array, linked list, .. etc.




```
int linear_search(int[] arr, int
    target) {
    for (int i = 0; i < arr.length; i
        ++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

$\mathcal{O}(\log n)$ - Logarithmic time

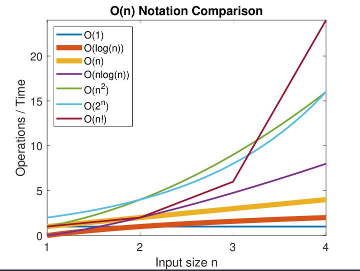
- Algorithm cuts the problem space in half in each iteration.
- Examples:
  - Traversing a sorted array using binary search




```
int binary_search(int[] arr, int target, int start, int finish) {
        int mid = start + (finish - start) / 2;
        if (finish >= start) { // recursive steps
                if (arr[mid] == target)
                        return mid;
                else if (arr[mid] > target)
                        return binary_search(arr, target, start, mid - 1);
                else
                        return binary_search(arr, target, mid + 1, finish);
        }
        return -1; // base case
}
```

$\mathcal{O}(\log n)$ - Logarithmic time

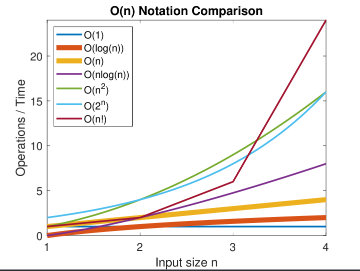
- Algorithm cuts the problem space in half in each iteration.
- Examples:
  - Traversing a sorted array using binary search




```
int binary_search(int[] arr, int target, int start, int finish) {
        int mid = start + (finish - start) / 2;
        if (finish >= start) { // recursive steps
                if (arr[mid] == target)
                        return mid;
                else if (arr[mid] > target)
                        return binary_search(arr, target, start, mid - 1);
                else
                        return binary_search(arr, target, mid + 1, finish);
        }
        return -1; // base case
}
```

$\mathcal{O}(\log n)$ - Logarithmic time

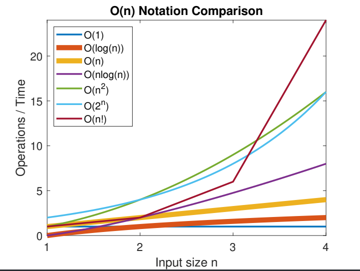
- Algorithm cuts the problem space in half in each iteration.
- Examples:
  - Traversing a sorted array using binary search




```
int binary_search(int[] arr, int target, int start, int finish) {
    int mid = start + (finish - start) / 2;
    if (finish >= start) { // recursive steps
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] > target)
            return binary_search(arr, target, start, mid - 1);
        else
            return binary_search(arr, target, mid + 1, finish);
    }
    return -1; // base case
}
```

$\mathcal{O}(\log n)$ - Logarithmic time

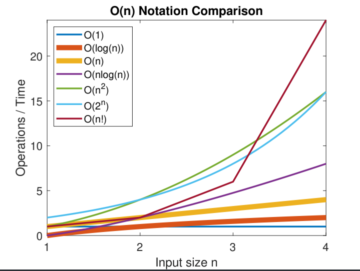
- Algorithm cuts the problem space in half in each iteration.
- Examples:
  - Traversing a sorted array using binary search




```
int binary_search(int[] arr, int target, int start, int finish) {
        int mid = start + (finish - start) / 2;
        if (finish >= start) { // recursive steps
                if (arr[mid] == target)
                        return mid;
                else if (arr[mid] > target)
                        return binary_search(arr, target, start, mid - 1);
                else
                        return binary_search(arr, target, mid + 1, finish);
        }
        return -1; // base case
}
```

$\mathcal{O}(\log n)$ - Logarithmic time

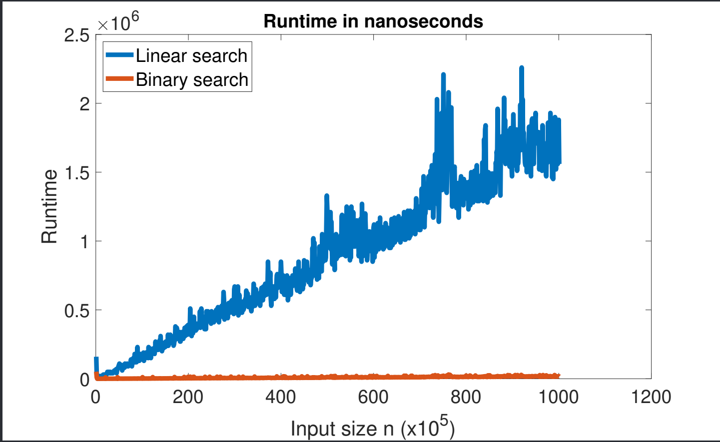
- Algorithm cuts the problem space in half in each iteration.
- Examples:
  - Traversing a sorted array using binary search




```
int binary_search(int[] arr, int target, int start, int finish) {
        int mid = start + (finish - start) / 2;
        if (finish >= start) { // recursive steps
                if (arr[mid] == target)
                        return mid;
                else if (arr[mid] > target)
                        return binary_search(arr, target, start, mid - 1);
                else
                        return binary_search(arr, target, mid + 1, finish);
        }
        return -1; // base case
}
```

$\mathcal{O}(\log n)$ vs $\mathcal{O}(n)$

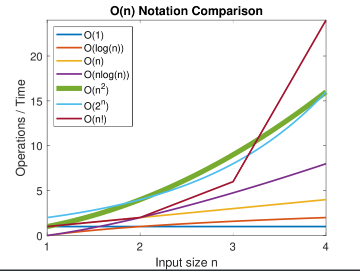
Binary Vs. Linear search



1559975 vs. 15576 ns at last input size

$\mathcal{O}(n^2)$ - Quadratic time

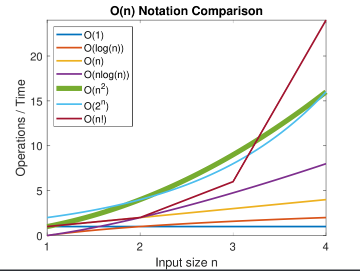
- Execution time of an algorithm $\propto$ square of the size of the data *n*.
- Examples:
  - *Bubble*, *Selection*, and *Insertion* sorts.




```
boolean duplicates_exist(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j])
                        return true;
            }
    }
    return false;
}
```

$\mathcal{O}(n^2)$ - Quadratic time

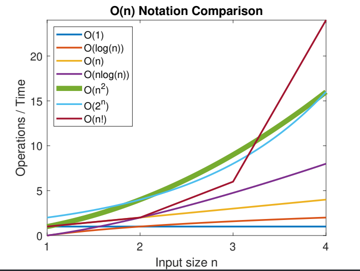
- Execution time of an algorithm $\propto$ square of the size of the data *n*.
- Examples:
  - *Bubble*, *Selection*, and *Insertion* sorts.




```
boolean duplicates_exist(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j])
                        return true;
            }
    }
    return false;
}
```

$\mathcal{O}(n^2)$ - Quadratic time

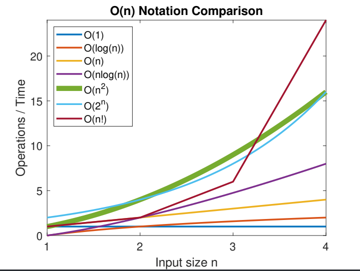
- Execution time of an algorithm ∝ square of the size of the data *n*.
- Examples:
  - *Bubble*, *Selection*, and *Insertion* sorts.




```
boolean duplicates_exist(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j])
                        return true;
            }
    }
    return false;
}
```

$\mathcal{O}(n^2)$ - Quadratic time

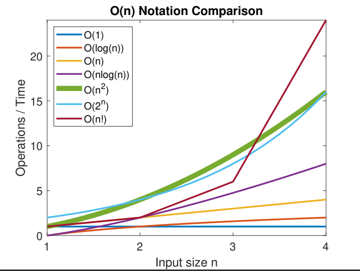
- Execution time of an algorithm ∝ square of the size of the data *n*.
- Examples:
  - *Bubble*, *Selection*, and *Insertion* sorts.




```
boolean duplicates_exist(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j])
                        return true;
            }
    }
    return false;
}
```

$\mathcal{O}(n^2)$ - Quadratic time

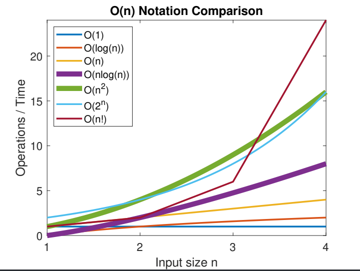
- Execution time of an algorithm $\propto$ square of the size of the data *n*.
- Examples:
  - *Bubble*, *Selection*, and *Insertion* sorts.

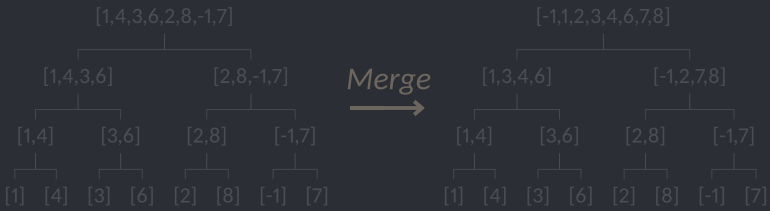



```
boolean duplicates_exist(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j])
                        return true;
            }
    }
    return false;
}
```

$\mathcal{O}(n \log n)$ - Loglinear Time

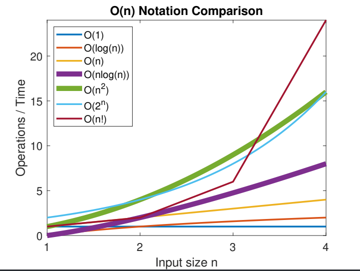
- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - *Merge sort*: $\log n$ levels with linear work n for each level

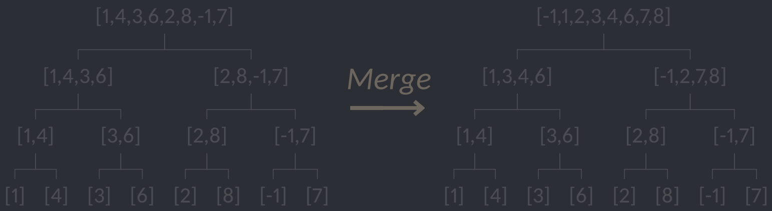




$\mathcal{O}(n \log n)$ - Loglinear Time

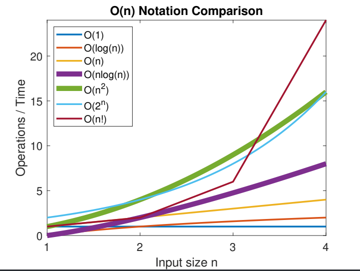
- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - Merge sort: $\log n$ levels with linear work n for each level

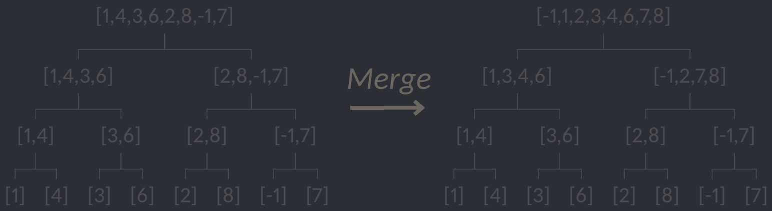




$\mathcal{O}(n \log n)$ - Loglinear Time

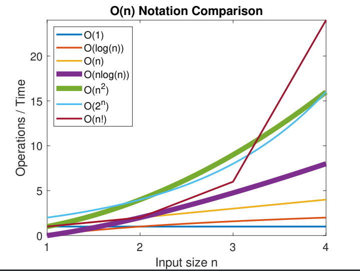
- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - *Merge* sort: $\log n$ levels with linear work n for each level

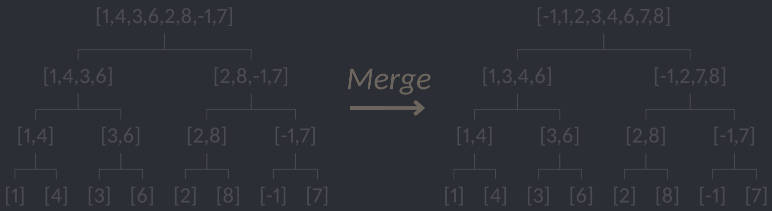




$\mathcal{O}(n \log n)$ - Loglinear Time

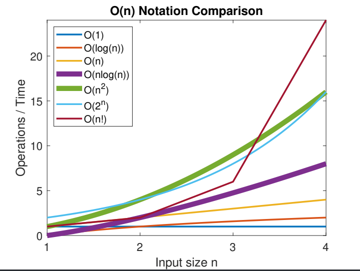
- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - *Merge* sort: $\log n$ levels with linear work n for each level

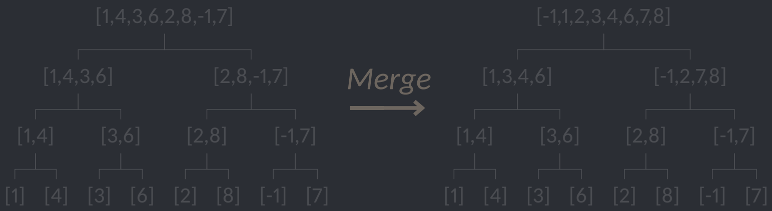




$\mathcal{O}(n \log n)$ - Loglinear Time

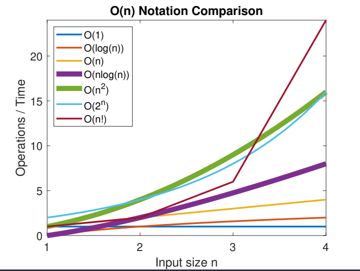
- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - *Merge* sort: $\log n$ levels with linear work n for each level

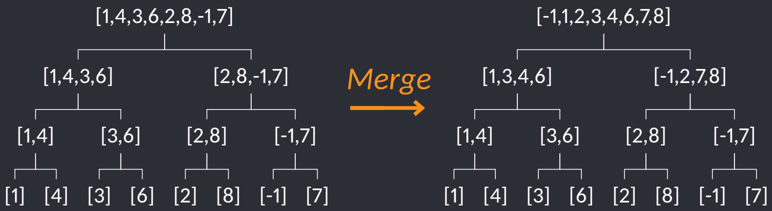




$\mathcal{O}(n \log n)$ - Loglinear Time

- For every element in a collection of size n,
  - $\log n$ operations are performed.
- Examples:
  - *Merge* sort: $\log n$ levels with linear work n for each level

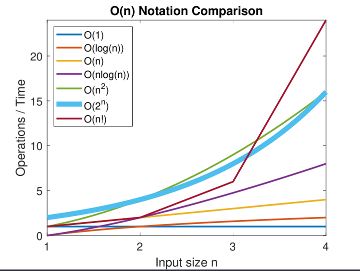




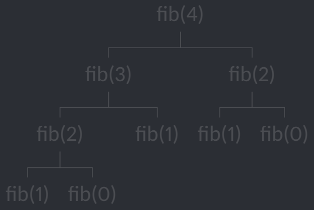
$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series:*

    0, 1, 1, 2, 3, 5, 8, ...

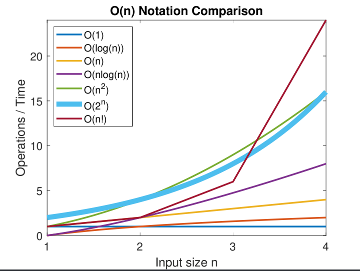



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n-1) +
        fibonacci(n-2); // recursive
              step
    }
}
```

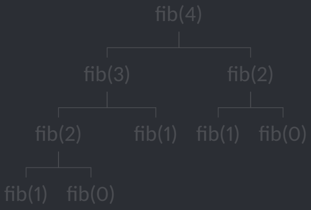


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series*: 0, 1, 1, 2, 3, 5, 8, ..

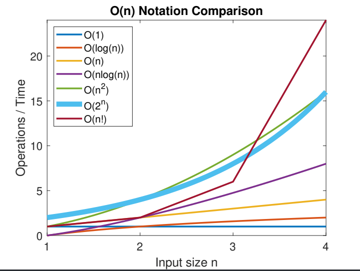



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n−1) +
        fibonacci(n−2); // recursive
              step
    }
}
```

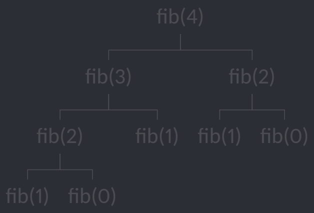


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series:* 0, 1, 1, 2, 3, 5, 8, ..

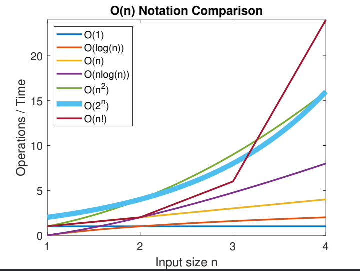



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n−1) +
        fibonacci(n−2); // recursive
             step
    }
}
```

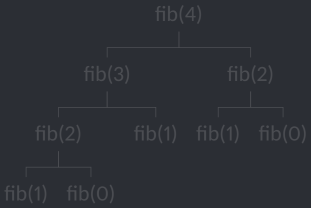


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series*: 0, 1, 1, 2, 3, 5, 8, ..

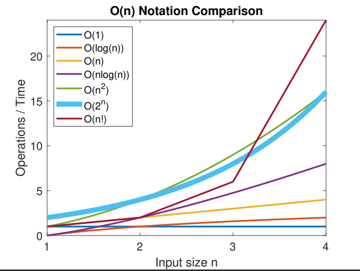



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n—1) +
        fibonacci(n—2); // recursive
            step
    }
}
```

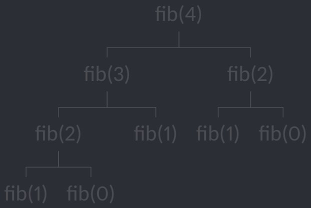


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series*: 0, 1, 1, 2, 3, 5, 8, ..

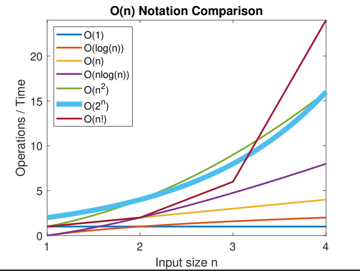



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n-1) +
        fibonacci(n-2); // recursive
              step
    }
}
```

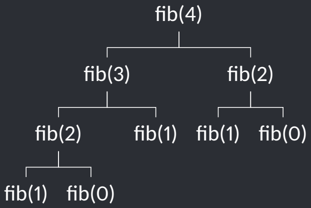


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series*: 0, 1, 1, 2, 3, 5, 8, ..

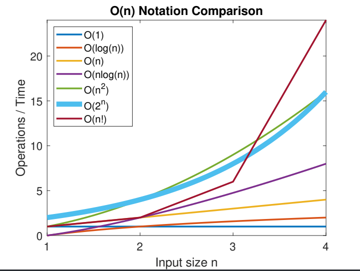



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n-1) +
        fibonacci(n-2); // recursive
            step
    }
}
```

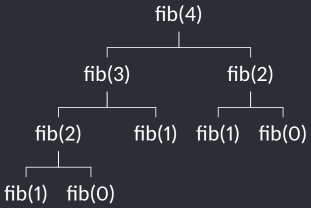


$\mathcal{O}(2^n)$ - Exponential time

- describes an algorithm whose growth doubles with each addition to the data set.
- Examples:
  - Brute-force algorithms
  - *Fibonacci series*: 0, 1, 1, 2, 3, 5, 8, ..

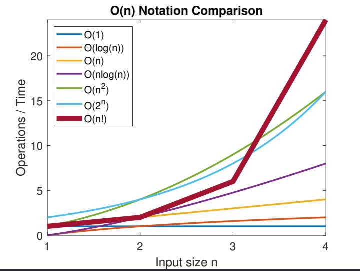



```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n; // base cases
    } else {
        return fibonacci(n-1) +
        fibonacci(n-2); // recursive
            step
    }
}
```

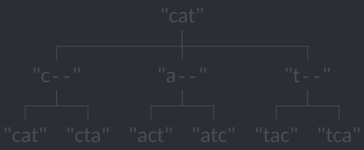


$\mathcal{O}(n!)$ - Factorial time

- Execution time of an algorithm ∝ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems

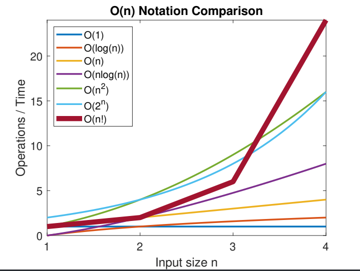



```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
     str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
         );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                charAt(i), str.substring
                (0, i) + str.substring(i
                +1, n));}}
```

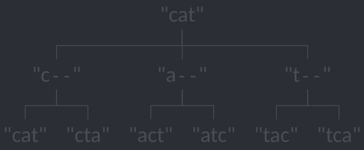


$\mathcal{O}(n!)$ - Factorial time

- Execution time of an algorithm ∝ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems

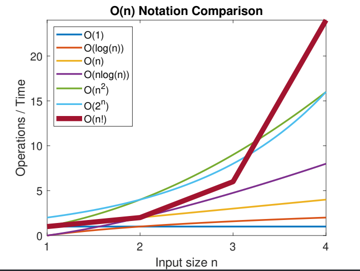



```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
     str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
         );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                   charAt(i), str.substring
                   (0, i) + str.substring(i
                   +1, n));}}
```

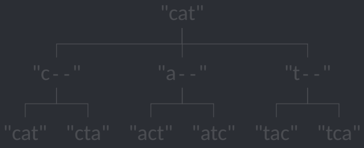


$\mathcal{O}(n!)$ - Factorial time

- Execution time of an algorithm ∝ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems

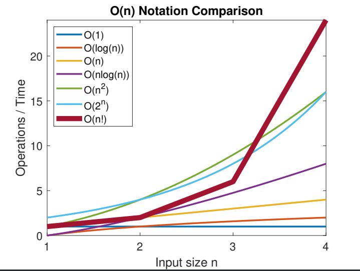



```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
      str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
        );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                charAt(i), str.substring
                (0, i) + str.substring(i
                +1, n));}}
```

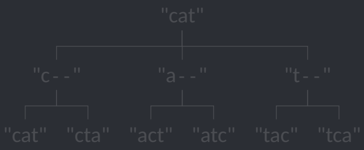


$\mathcal{O}(n!)$ - Factorial time

- Execution time of an algorithm $\propto$ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems

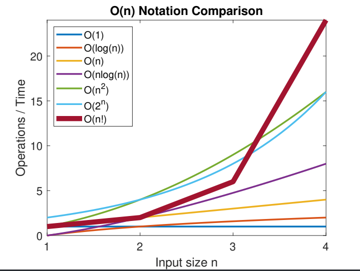



```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
     str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
        );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                charAt(i), str.substring
                (0, i) + str.substring(i
                +1, n));}}
```

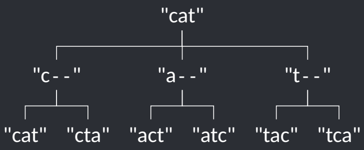


$\mathcal{O}(n!)$ - Factorial time

- Execution time of an algorithm ∝ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems



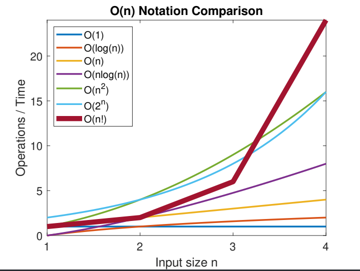

```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
     str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
         );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                charAt(i), str.substring
                (0, i) + str.substring(i
                +1, n));}}
```

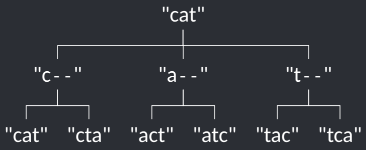


$\mathcal{O}(n!)$ - Factorial time

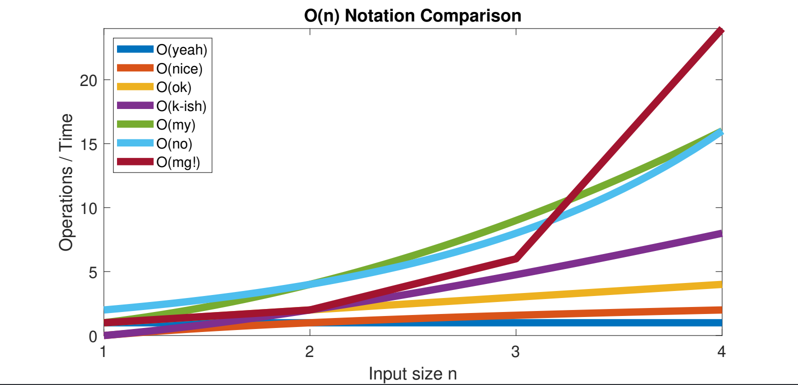
- Execution time of an algorithm $\propto$ to the product of all numbers included in input size (hence factorial!)
- Examples:
  - permutation problems

```
void permutation(String str) {
    permutation("", str); }
void permutation(String prefix, String
     str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix
         );
    else {
        for (int i = 0; i < n; i++)
            permutation(prefix + str.
                charAt(i), str.substring
                (0, i) + str.substring(i
                +1, n));}}
```





Thank You!



Alternative Big $\mathcal{O}$ Notations