Algorithm Analysis

Data Structures & Algorithm

Agenda

- What is an Algorithm?
- Difference between a Program and an Algorithm
- Multiple Ways of Solving a Problem
- Benefits of Evaluating an Algorithm
- How can we evaluate programs?
  - Approach 1 (Simulation: Timing)
  - Approach 2 (Modeling: Counting)
  - Approach 3 (Asymptotic Behavior: Order of Growth)

Algorithm

Algorithm

An algorithm is a step-by-step procedure for solving a problem.

Algorithm

An algorithm is a step-by-step procedure for solving a problem.

Input

Output

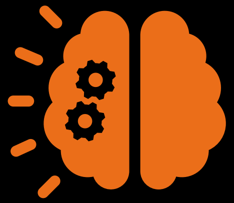
Definite & Unambiguous

Algorithm vs Program

	Algorithm	Code or Program
Focus of a professional		
Form		
Dependence on H/W or OS		
Professional's Cognitive State		
Correctness/Performance		

Algorithm vs Program

	Algorithm	Code or Program
Focus of a professional	Design	Implementation
Form	Pseudocode	Programming Language
Dependence on H/W or OS	No	Yes
Professional's Cognitive State	Thinking	Doing
Correctness/Performance	Analysis	Testing



Multiple Ways of Solving a Problem

Multiple Ways of Solving a Problem

Problem: Determine if a sorted array contains any duplicates.

-13	-11	12	14	24	24	100	102

Multiple Ways of Solving a Problem

Problem: Determine if a sorted array contains any duplicates.

-13	-11	12	14	24	24	100	102

Silly algorithm: Every possible pair

Multiple Ways of Solving a Problem

Problem: Determine if a sorted array contains any duplicates.

-13	-11	12	14	24	24	100	102

Silly algorithm: Every possible pair

Better algorithm: Compare adjacents

https://onlinegdb.com/SHYJzMAxD

Multiple Ways of Solving a Problem

Problem: Determine if a sorted array contains any duplicates.

-13	-11	12	14	24	24	100	102

Silly algorithm: Every possible pair

Better algorithm: Compare adjacents

Now that we know, that there are multiple ways to solve a problem, how do we evaluate which one is better?

Are all programs/algorithms equal in terms of performance?

Performance

In terms of what?

Are all programs/algorithms equal in terms of performance?

Performance

In terms of what?

- Time
- Space

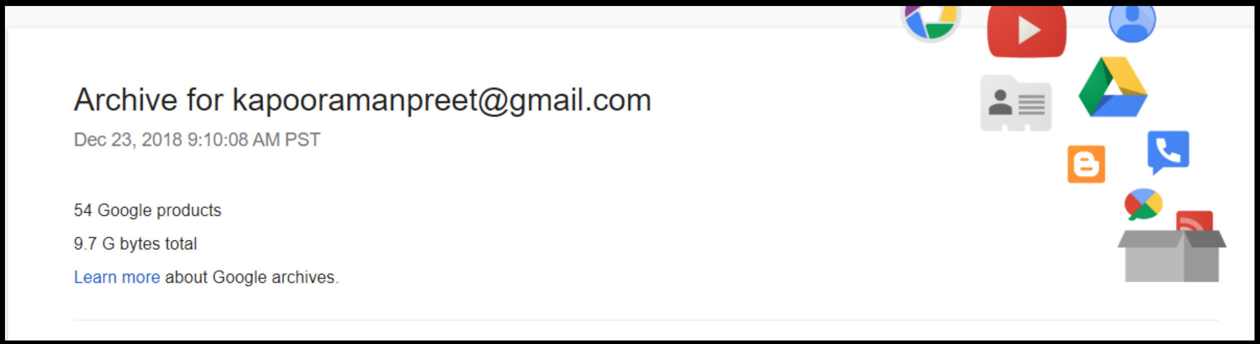
Are all programs/algorithms equal in terms of performance?

Why do we care about algorithms?

Why do we care about algorithms?

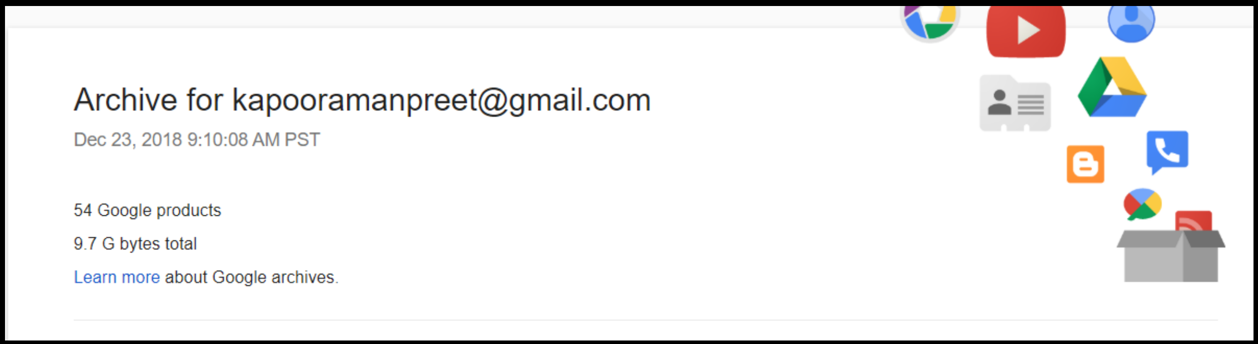
- Knowing
- Experiencing
- Selling
- Cost

A Simple Example



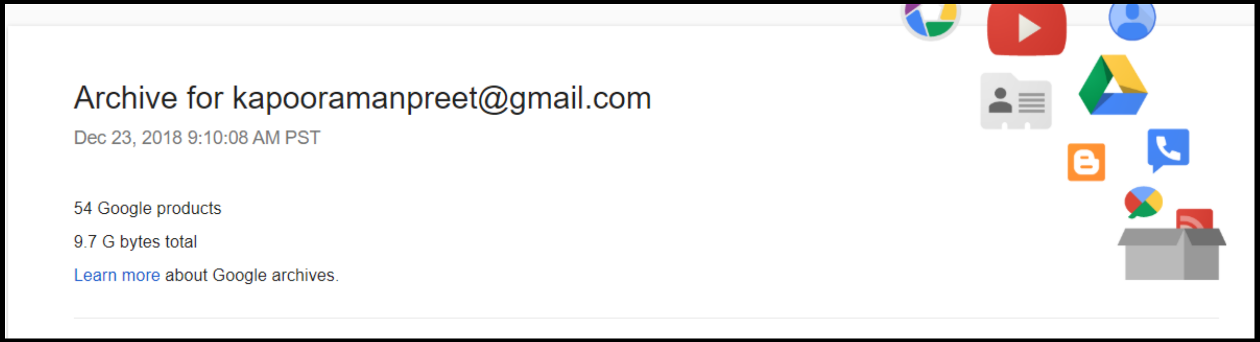
A Simple Example

- Google has 2 billion active users
- Conservative estimate ~ 1 GB of data/user
- Total Data (Space):



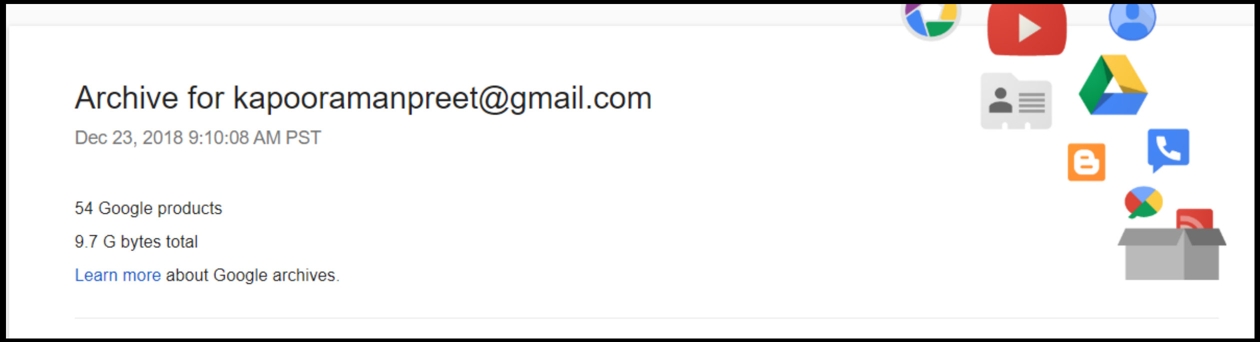
A Simple Example

- Google has 2 billion active users
- Conservative estimate ~ 1 GB of data/user
- Total Data (Space): 2 Exabytes
- Total Time:



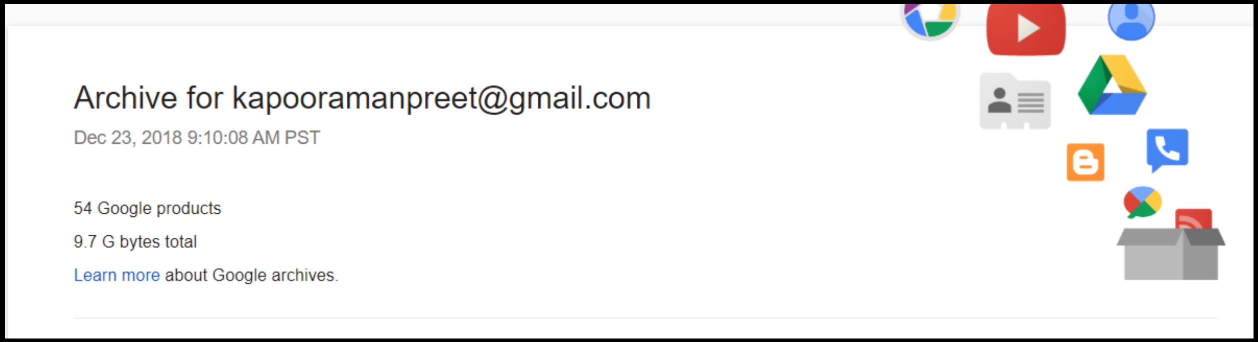
A Simple Example

- Google has 2 billion active users
- Conservative estimate ~ 1 GB of data/user
- Total Data (Space): 2 Exabytes
- Total Time:
  - Operation Speed: 0.5 ns
  - Linear Search
  - Binary Search



A Simple Example

- Google has 2 billion active users
- Conservative estimate ~ 1 GB of data/user
- Total Data (Space): 2 Exabytes
- Total Time:
  - Operation Speed: 0.5 ns
  - Linear Search: 11574 days or 31 years
  - Binary Search: 31s



In short, we care about performance …

So, how do we measure performance?

Questions to ask when evaluating programs

- **Time:** How much time does this take?
- **Space:** How much space does this consume?
- **Data:** Are there any patterns in our data?

Approach 1 (Simulation: Timing)

Approach 1 (Simulation: Timing)

Code #1

```
auto t1 = Clock::now();
for(int i=0; i<1000; i++);
auto t2 = Clock::now();
Print t2-t1
```

Code #2

```
auto t1 = Clock::now();
for(int i=0; i<1000000; i++);
auto t2 = Clock::now();
Print t2-t1
```

Approach 1 (Simulation: Timing)

Code #1

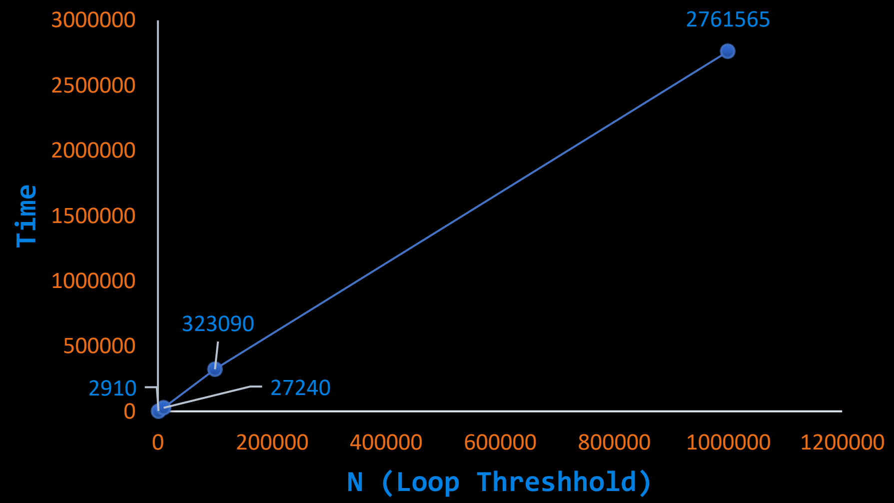
```
auto t1 = Clock::now();
for(int i=0; i<1000; i++);
auto t2 = Clock::now();
Print t2-t1
```

Code #2

```
auto t1 = Clock::now();
for(int i=0; i<1000000; i++);
auto t2 = Clock::now();
Print t2-t1
```

Output

```
Delta t = t2-t1 (1000): 2910 nanoseconds
Delta t = t2-t1 (10000): 27240 nanoseconds
Delta t = t2-t1 (100000): 323090 nanoseconds
Delta t = t2-t1 (1000000): 2761565 nanoseconds
```



https://onlinegdb.com/BJGAP7I5I

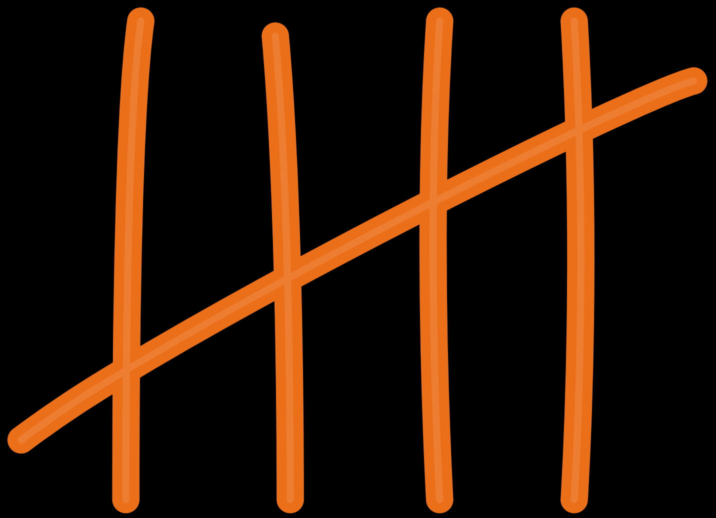
Approach 1 (Simulation: Timing)

Pros	Cons

Approach 1 (Simulation: Timing)

Pros	Cons
Easy to measure	Results vary across machines
Easy to interpret	Compiler dependent
	Results vary across implementations
	Not predictable for small inputs
	No clear relationship between input and time

Approach 2 (Modeling: Counting)



Approach 2 (Modeling: Counting)

Count the number of operations

Approach 2 (Modeling: Counting)

Count the number of operations

```
int sum=0;
for(int i=0; i<n; i++)
    sum += i;
print sum
```

Operation	Symbolic count
int sum=0;	
int i=0;	
i<n;	
i++	
sum += i;	
print sum	
T(n)	

Approach 2 (Modeling: Counting)

Count the number of operations

```
int sum=0;
for(int i=0; i<n; i++)
    sum += i;
print sum
```

Operation	Symbolic count
int sum=0;	1
int i=0;	1
i<n;	0…n = n+1
i++	n
sum += i;	n
print sum	1
T(n)	3n+4

Approach 2 (Modeling: Counting)

Pros	Cons

Approach 2 (Modeling: Counting)

Pros	Cons
Independent of computer	All operations are equal
Input dependence is captured in model (Scaling)	Tedious to compute
	Results vary across implementations
	Doesn't tell you actual time

Approach 2 (Modeling: Counting)

Count the number of operations

```
int sum=0;
for(int i=0; i<n; i++)
    sum += i;
print sum
```

Operation	Symbolic count
int sum=0;	1
int i=0;	1
i<n;	0…n = n+1
i++	n
sum += i;	n
print sum	1
T(n)	3n+4

Tedious to compute:

Different variables, so many operations, so many equations!

Approach 2 (Modeling: Counting)

Count the number of operations

```
int sum=0;
for(int i=0; i<n; i++)
    sum += i;
print sum
```

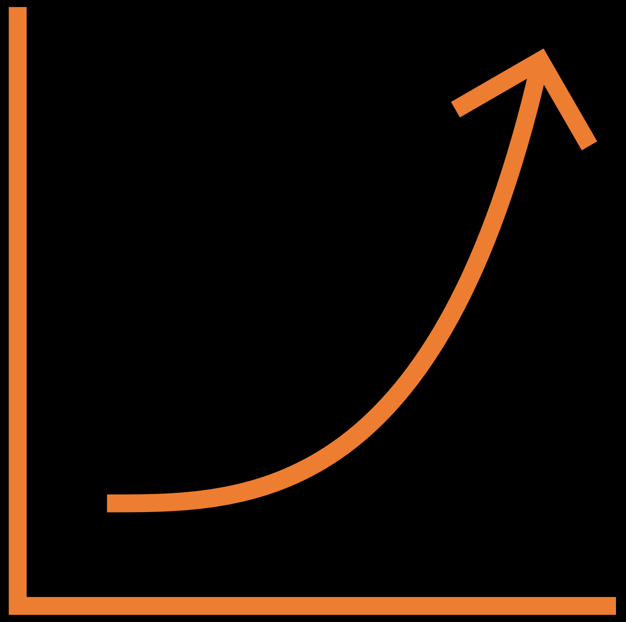
Operation	Symbolic count
int sum=0;	1
int i=0;	1
i<n;	0…n = n+1
i++	n
sum += i;	n
print sum	1
T(n)	3n+4

Tedious to compute:

Different variables, so many operations, so many equations!

Can we eliminate the complexity or get rid of extraneous variables?

Approach 3 (Asymptotic Behavior: Order of Growth)



Which variables should we **eliminate**?

Operation	Symbolic count
int sum=0;	1
int i=0;	1
i<n;	0…n = n+1
i++	n
sum += i;	n
print sum	1
T(n)	3n+4

Growth of Functions

Time, y = T(n)

Inputs: n

	1	log n	n	n log n	n^2	n^3	2^n	n!
1								
10								
100								
1000								
10000								
100000								
1000000								
10000000								
100000000								

Growth of Functions

Time, y = T(n)

Inputs: n

	1	log n	n	n log n	n^2	n^3	2^n	n!
1	1							
10	1							
100	1							
1000	1							
10000	1							
100000	1							
1000000	1							
10000000	1							
100000000	1							

Growth of Functions

Time, y = T(n)

Inputs: n

	1	log n	n	n log n	n^2	n^3	2^n	n!
1	1	0						
10	1	3						
100	1	7						
1000	1	10						
10000	1	13						
100000	1	17						
1000000	1	20						
10000000	1	23						
100000000	1	27						

Growth of Functions

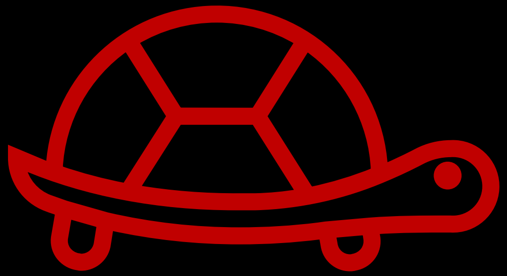
Time, y = T(n)

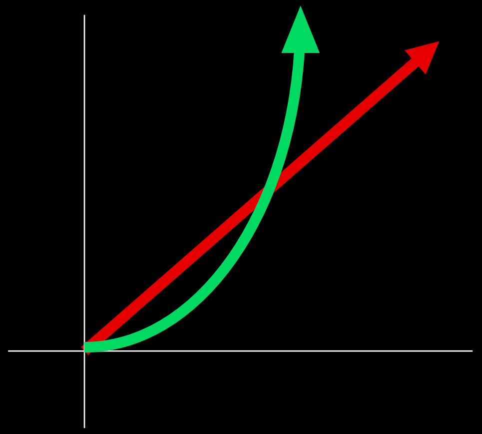
Inputs: n

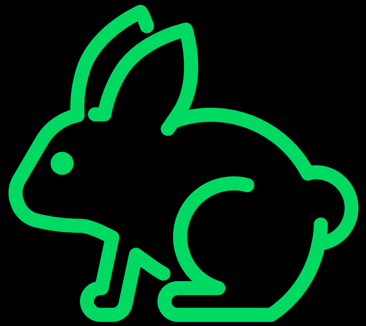
	1	log n	n	n log n	n^2	n^3	2^n	n!
1	1	0	1	0	1	1	2	1
10	1	3	10	30	100	1000	1024	3628800
100	1	7	100	700	10000	1000000	1.26765E+30	9.3326E+157
1000	1	10	1000	10000	1000000	1000000000	1.0715E+301	#NUM!
10000	1	13	10000	130000	100000000	1E+12	#NUM!	#NUM!
100000	1	17	100000	1700000	10000000000	1E+15	#NUM!	#NUM!
1000000	1	20	1000000	20000000	1E+12	1E+18	#NUM!	#NUM!
10000000	1	23	10000000	230000000	1E+14	1E+21	#NUM!	#NUM!
100000000	1	27	100000000	2700000000	1E+16	1E+24	#NUM!	#NUM!

Order of Growth gets Faster or Functions rise faster

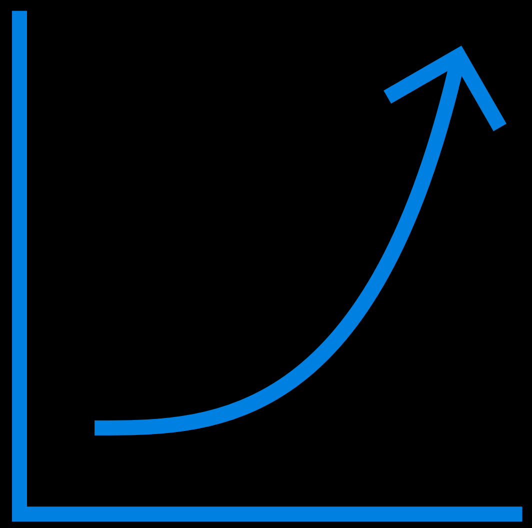
Eliminate functions that grow slower





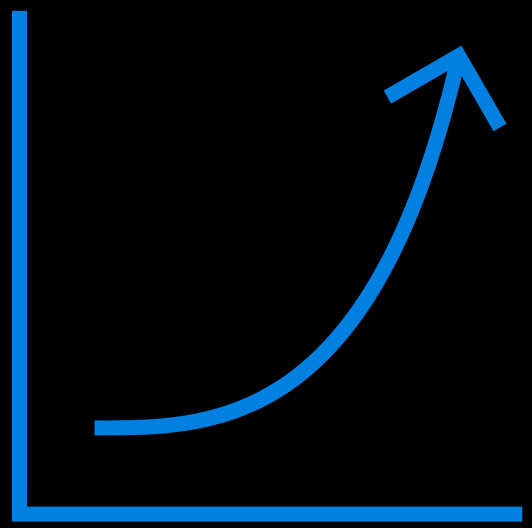


Approach 3 (Asymptotic Behavior: Order of Growth)

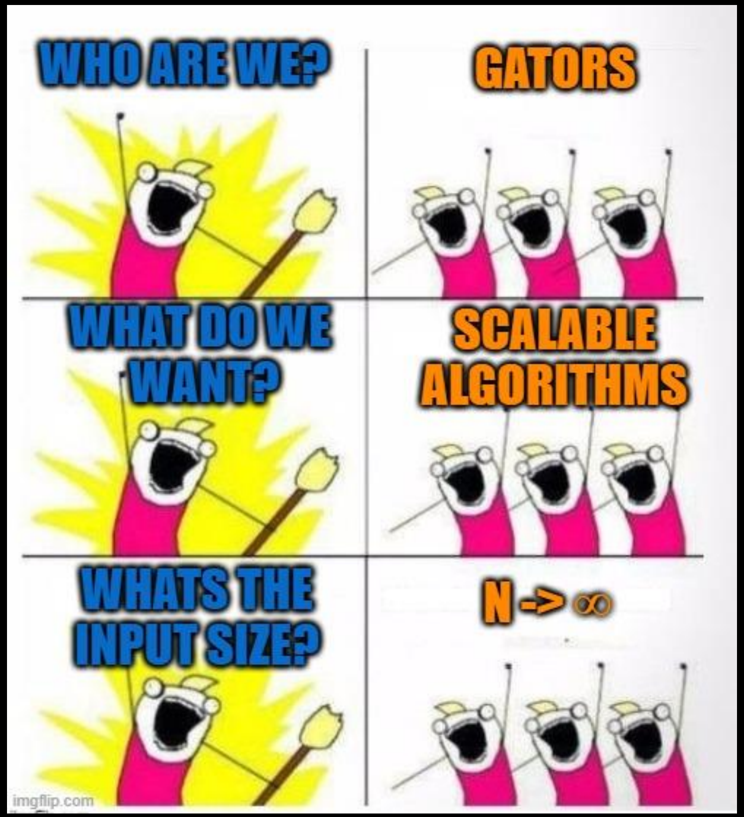


Very Large N

Approach 3 (Asymptotic Behavior: Order of Growth)



Very Large N



https://imgflip.com/i/3z0jdf

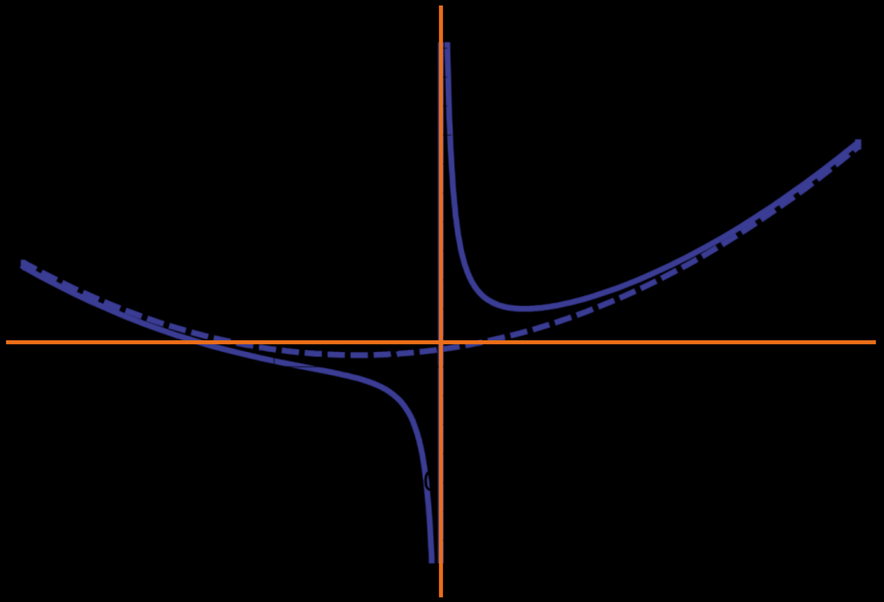
Notations for Algorithm Complexity

Time - Number of Operations:

- **Big-O** : Upper Bound
- **Big-Ω** : Lower Bound
- **Big-Θ** : Upper + Lower Bound

Asymptotic Bounding

- Line that approaches a curve but never meets
- Analysis of tail behavior
- N -> Infinity



Big O (Visualize)

$T(n) \in O(f(n))$

- If there exists two positive constants, n_0 and c, such that $T(n) \leq c.f(n)$ for all $n \geq n_0$
- $f(n)$ is an upper bound on performance
- $T(n)$ will grow no faster than constant times $f(n)$
- Use tighter upper bound

Big Ω

$T(n) \in \Omega(g(n))$

- If there exists two positive constants, n_0 and c, such that $T(n) \geq c.g(n)$ for all $n \geq n_0$

- $g(n)$ is a lower bound on growth rate of $T(n)$

- $T(n)$ will grow no slower than constant times $g(n)$

- Use tighter lower bound

Big Θ

$T(n) \in \Theta(g(n))$

- If $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$
- $c_1.g(n) \leq T(n) \leq c_2.g(n)$ for all $n \geq n_0$
- $g(n)$ is a tight upper and lower bound on the growth rate of $T(n)$

Big Θ vs Big O vs Big Ω

	Informal meaning:	Family	Family Members, T(n)
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/12$ $2N^2$ $N^2 + 11N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $N^2 + 1$ $\lg(N)$
Big Ω $\Omega(f(N))$			

Source: https://sp19.datastructur.es/index.html

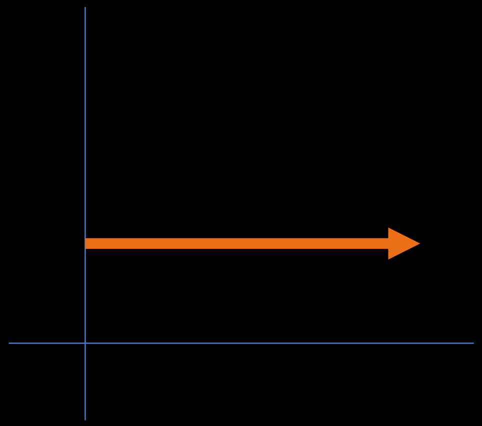
Constant Growth Rate, O(1)

If processing time is independent of the number of inputs n, the algorithm grows at a constant rate

```
int sum(int n)
{
    int sum = 0;
    sum += n;
    return sum;
}

```

n	y = f(c)
1	3
10	3
100	3
1000	3
10000	3
100000	3
1000000	3
10000000	3
100000000	3



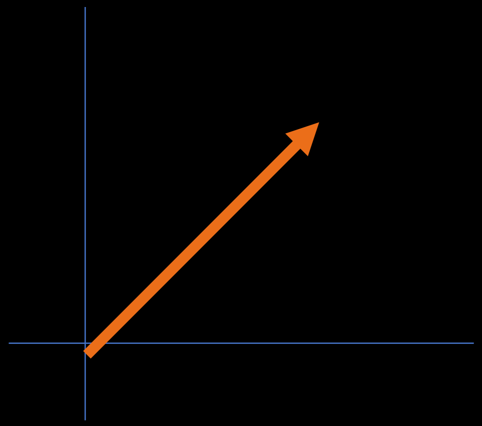
$T(n) = 3, T(n) \in O(1)$

Linear Growth Rate

If processing time increases in proportion to the number of inputs n, the algorithm grows at a linear rate

```
int sum(int n)
{
    int sum = 0;
    for (int i=0; i<n; i++)
        sum += i+1;
    return sum;
}
```

n	y = f(n)
1	1
10	10
100	100
1000	1000
10000	10000
100000	100000
1000000	1000000
10000000	10000000
100000000	100000000



$T(n) = 3n + 4$, $T(n) \in O(n)$, $c=4$, $n_0>4$

Quadratic Growth Rate

If processing time increases in proportion to the square of input size n, the algorithm grows at a quadratic rate

```
bool find(int n[][], int t)
{
  int i, j;
  for(i=0; i < n.size; i++)
    for(j=0; j < n.size; j++)
       if (x[i][j] == t)
          return true;
  return false;
}
```

$$T(n) = 3n^2 + 4n + 5$$

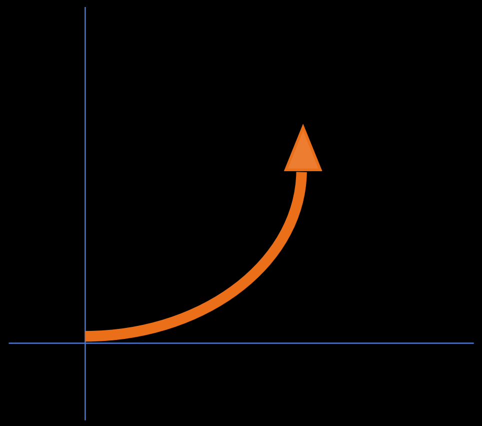
Operation	Count
`int i, j;`	2
`i=0`	1
`i < n`	$n+1$
`i++`	n
`j=0`	n
`j < n`	$n(n+1) = n^2 + n$
`j++`	n^2
`x[i][j] == t`	n^2
`return true/false`	1
T(n)	$3n^2 + 4n + 5$

Quadratic Growth Rate

If processing time increases in proportion to the square of input size n, the algorithm grows at a quadratic rate

```
bool find(int n[][], int t)
{
  int i, j;
  for(i=0; i < n.size; i++)
    for(j=0; j < n.size; j++)
       if (x[i][j] == t)
          return true;
  return false;
}
```

n	$y = f(n^2)$
1	1
10	100
100	10000
1000	1000000
10000	100000000
100000	10000000000
1000000	1E+12
10000000	1E+14
100000000	1E+16



$$T(n) = 3n^2 + 4n + 5,\ T(n) \in O(n^2),\ c=?,\ n_0>?$$

Inclass Activity

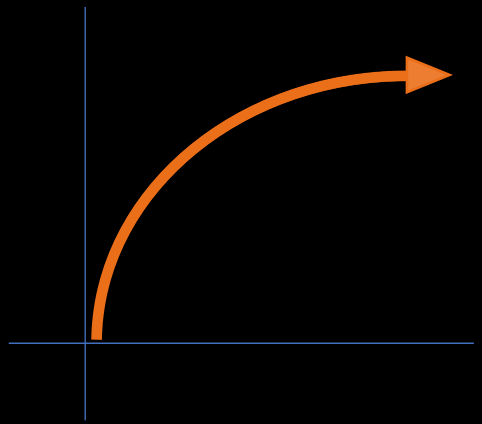
Complete this program and determine growth rate.

Logarithmic Growth Rate

If processing time increases in proportion to the log n, the algorithm grows at a logarithmic rate

```
int sum(int n)
{
    int sum = 0;
    for (int i=1; i<=n; i*=2)
        sum += i;
    return sum;
}
```

n	$y = f(\log_2 n)$
1	0
10	3
100	7
1000	10
10000	13
100000	17
1000000	20
10000000	23
100000000	27

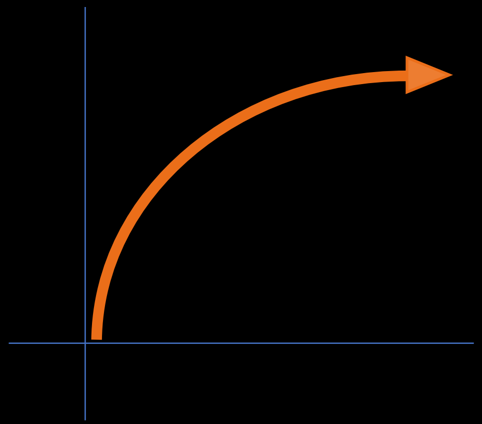


Logarithmic Growth Rate

If processing time increases in proportion to the log n, the algorithm grows at a logarithmic rate

```
int sum(int n)
{
    int sum = 0;
    for (int i=1; i<=n; i*=2)
        sum += i;
    return sum;
}
```

n	$y = f(\log_2 n)$
1	0
10	3
100	7
1000	10
10000	13
100000	17
1000000	20
10000000	23
100000000	27



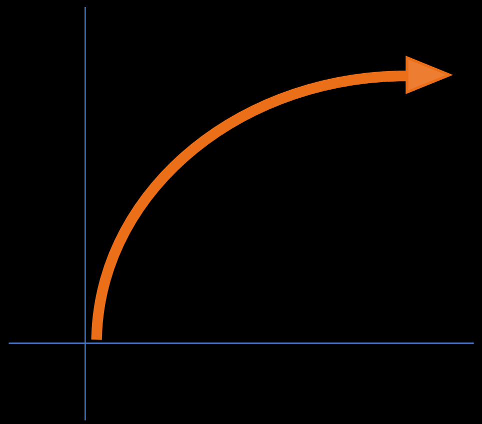
n	1	2	4	8	16	.	.	.	?
n (in powers of two)	2^0	2^1	2^2	2^3	2^4	.	.	.	?
# of times the loop executes	1	2	3	4	5	.	.	.	k

Logarithmic Growth Rate

If processing time increases in proportion to the log n, the algorithm grows at a logarithmic rate

```
int sum(int n)
{
    int sum = 0;
    for (int i=1; i<=n; i*=2)
        sum += i;
    return sum;
}
```

n	$y = f(\log_2 n)$
1	0
10	3
100	7
1000	10
10000	13
100000	17
1000000	20
10000000	23
100000000	27



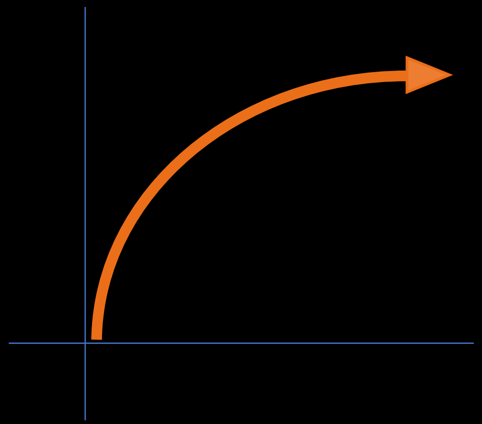
n	1	2	4	8	16	.	.	.	2^{k-1}
n (in powers of two)	2^0	2^1	2^2	2^3	2^4	.	.	.	2^{k-1}
# of times the loop executes	1	2	3	4	5	.	.	.	k

Logarithmic Growth Rate

If processing time increases in proportion to the log n, the algorithm grows at a logarithmic rate

```
int sum(int n)
{
    int sum = 0;
    for (int i=1; i<=n; i*=2)
        sum += i;
    return sum;
}
```

n	$y = f(\log_2 n)$
1	0
10	3
100	7
1000	10
10000	13
100000	17
1000000	20
10000000	23
100000000	27



n	1	2	4	8	16	.	.	.	2^{k-1}
n (in powers of two)	2^0	2^1	2^2	2^3	2^4	.	.	.	2^{k-1}
# of times the loop executes	1	2	3	4	5	.	.	.	k

We know that $2^{k-1} <= n$; $k <= (\log_2 n + 1)$; $k \propto \log_2 n$;

Different Growth Rates

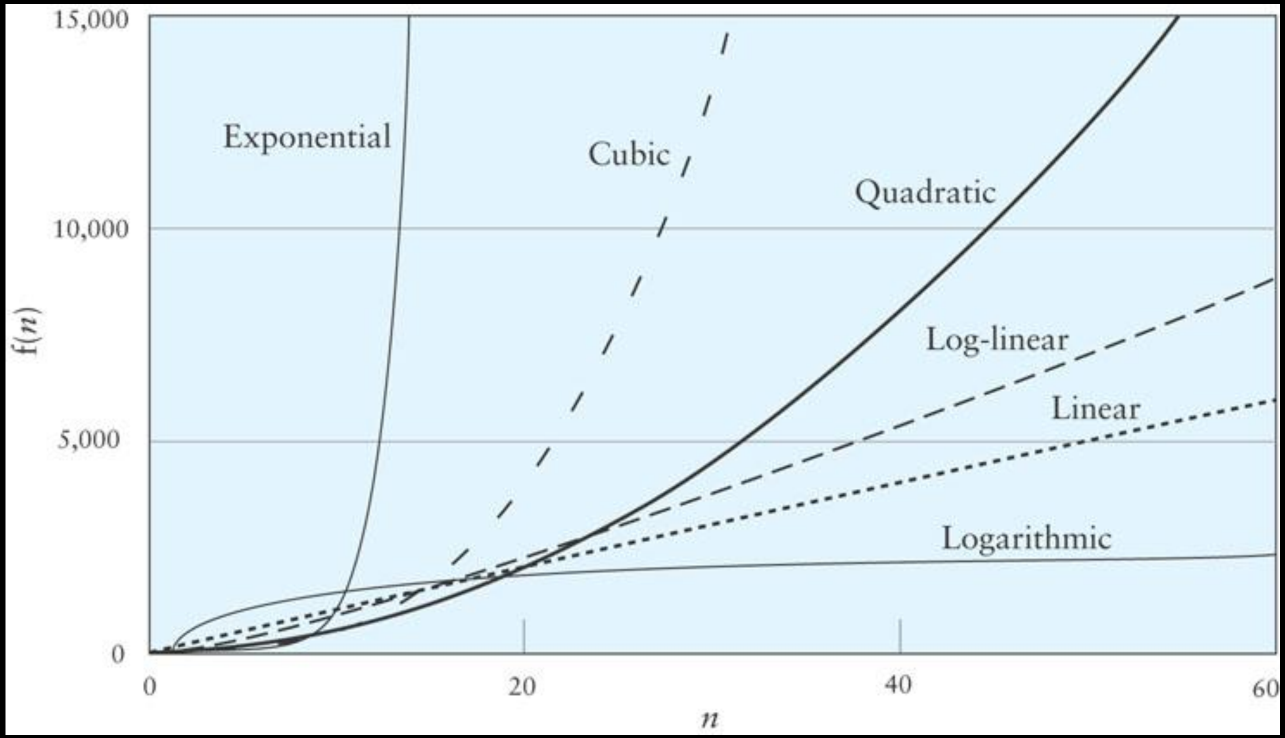
Time

Inputs: n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$	$O(2^n)$	O(n!)
1	1	0	1	0	1	1	2	1
10	1	3	10	10	100	1000	1024	3628800
100	1	7	100	200	10000	1000000	1.26765E+30	9.3326E+157
1000	1	10	1000	3000	1000000	1000000000	1.0715E+301	#NUM!
10000	1	13	10000	40000	100000000	1E+12	#NUM!	#NUM!
100000	1	17	100000	500000	10000000000	1E+15	#NUM!	#NUM!
1000000	1	20	1000000	6000000	1E+12	1E+18	#NUM!	#NUM!
10000000	1	23	10000000	70000000	1E+14	1E+21	#NUM!	#NUM!
100000000	1	27	100000000	800000000	1E+16	1E+24	#NUM!	#NUM!

Order of Growth gets Faster, Complexity increases

Constant < Logarithmic < Linear < Loglinear < Polynomial < Exponential < Factorial

Different Growth Rates



http://bigocheatsheet.com/

Tips for Asymptotic Analysis (Big O)

Tip #1: Addition (Independence)

```
void func1(int n, int m)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 0; j < m; j++)
        cout << j;
}
```

Tip #1: Addition (Independence)

```
void func1(int n, int m)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 0; j < m; j++)
        cout << j;
}
```

$$T(n, m) \in O(n+m)$$

Tip #2: Drop Constant Multipliers

```
void func2(int n)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 0; j < n; j++)
        cout << j;
}
```

Tip #2: Drop Constant Multipliers

```
void func2(int n)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 0; j < n; j++)
        cout << j;
}
```

$$
\begin{aligned}
T(n) &\in O(n+n) \\
&\in O(2n) \\
&\sim O(n)
\end{aligned}
$$

Tip #3: Different Input Variables

```
void func3(int x, int y)
{
    for (int i = 0; i < x; i++)
        cout << i;

    for (int j = 0; j < y; j++)
        cout << j;
}
```

Tip #3: Different Input Variables

```cpp
void func3(int x, int y)
{
    for (int i = 0; i < x; i++)
        cout << i;

    for (int j = 0; j < y; j++)
        cout << j;
}
```

$T(x, y) \in O(x+y)$

Describe what the variable is, Always!

Example: O(x) where x is the size or length of a string

Tip #4a: Drop Lower Order Terms with Similar Growth Rates

```
void func4a(int n)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 1; j < n; j*=2)
        cout << j;
}
```

Tip #4a: Drop Lower Order Terms with Similar Growth Rates

```
void func4a(int n)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 1; j < n; j*=2)
        cout << j;
}
```

$$T(n) \in O(n + \log_2 n)$$

$$\sim O(n)^*$$

*Both variables are n and grow at the same rate.

Tip #4b: Drop Lower Order Terms with Similar Growth Rates

```
void func4b(int n, int m)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 1; j < m; j*=2)
        cout << j;
}
```

Tip #4b: Drop Lower Order Terms with Similar Growth Rates

```
void func4b(int n, int m)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 1; j < m; j*=2)
        cout << j;
}
```

$$T(n, m) \in O(n + \log_2 m)$$
$$\sim O(n)^*$$

*Assuming n and m are growing at the same rate.

If you are given in the question that n and m are growing at the same rate or if you assume they are growing at the same rate, then simplifying is fine

Tip #4c: Do not drop Lower Order Terms with different Growth Rates

```
void func4c(int n, int m)
{
    for (int i = 0; i < n; i++)
        cout << i;

    for (int j = 1; j < m; j*=2)
        cout << j;
}
```

$T(n, m) \in O(n + \log_2 m)$*

*Assuming no relationship is given between n and m.

Mentimeter

Menti.com

8638 6292

Mentimeter

An algorithm's total run time is given by the expression, T(n, p) = 10n + p. What is the representation of this program's execution time in Big O?

O(n+p)

Mentimeter

```
for (int i = 100; i > -1; i--)
    for (int j = i; j > 1; j/=2)
        print("viola")
```

O(1)

Logarithmic growth

```
for(i = 1; i <= n; i *= 2)
```

```
for(i = n; i >= 1; i /= 2)
```

https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
https://en.wikipedia.org/wiki/1_%2B_2_%2B_4_%2B_8_%2B_%E2%8B%AF
https://en.wikipedia.org/wiki/1/2_%2B_1/4_%2B_1/8_%2B_1/16_%2B_%E2%8B%AF

Mentimeter

```
for(int i = n; i > 0; i /= 2)
        for(int j = 1; j < i; j++)
                sum += 1;
```

O(n)

Mentimeter

```
// This is A
for(int i=1; i<n; i*=2);

// This is B
for(int i=1; i<n; i*=3);
```

They both take same time in terms of Big O

Mentimeter

```
// This is A
for(int i=1; i<n; i*=2);

// This is B
for(int i=1; i<n; i*=3);
```

B will be faster in terms of execution time/simulation

Enter Data

Best Case	Average Case	Worst Case
Lowest cost	Average cost for all n	Highest cost

Enter Data

Best Case	Average Case	Worst Case
Lowest cost	Average cost for all n	Highest cost

- Average/Best/Worst case measure actual costs at a specific input instance.
- You can define a specific order of instance but cannot propose variability in input size.
  - In general, calculating best case time complexity under the assumption that the data structure has a small size, example: when an array has size 1 or the tree is empty should be avoided. The complexity is calculated without thinking about the size of input. But it is perfectly fine to think about the properties of input or data structure such as data is sorted, height will always be proportional to log n for a balanced tree, etc.
  - Asymptotic analysis assumes n is very large. Whether it be big O, theta, or omega, it always refers to the case of very large n. The best / average / worst cases arise in different structural cases, exclusive of size.
- Growth Rate measures change in costs.

Recommended Readings

- https://dev.to/sherryummen/asymptotic-notations-b-oot-big-o-big-omega-big-theta-49e7
- Chapter 8.10 OpenDSA: Common Misunderstanding
- https://cs.stackexchange.com/questions/23068/how-do-o-and-%CE%A9-relate-to-worst-and-best-case
- https://qr.ae/pNyFxo
- https://cs.stackexchange.com/questions/23593/is-there-a-system-behind-the-magic-of-algorithm-analysis
- https://stackoverflow.com/questions/25593619/why-small-theta-asymtotic-notation-doesnt-exists/54542603

Useful series

$$1 + 2 + 3 + 4 + \cdots + n = n.(n+1)/2$$

$$1 + 2 + 4 + 8 + \cdots + 2^k = 2^{k+1} - 1$$

$$n + n/2 + n/4 + n/8 + \cdots + 1 = 2n - 1$$

https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
https://en.wikipedia.org/wiki/1_%2B_2_%2B_4_%2B_8_%2B_%E2%8B%AF
https://en.wikipedia.org/wiki/1/2_%2B_1/4_%2B_1/8_%2B_1/16_%2B_%E2%8B%AF

Linear Search

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502"};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

Size of array = size
Max length of any string = s

Length of target = t

constant

…

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

linear and dependent on size

linear and dependent on min(t, s)

constant

constant

Size of array = size
Max length of any string = s

Length of target = t

constant

…

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

(size * min(t, s)) + c

linear and dependent on size

linear and dependent on min(t, s)

constant

constant

Size of array = size

Max length of any string = s

Length of target = t

constant

(size * min(t, s)) + c

O (size*t), where size is size of array, t is size of target string, s is the max length of all strings in the array, and we are assuming that t and s grow at same rates.

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

O (size*t), where size is size of array, t is size of target string, s is the max length of all strings in the array, and we are assuming that t and s grow at same rates.

Clarification:

This program will yield an equation,

T(size, t) = (size * min(t, s)) + Xc

This function T(size, t) will be an element of O(size.t), O(size.t.t), O(size.size.t), … and several other functions.

When we talk about Big O, we want a measure to describe the **upper bound**. For the sake of the course, we seek the tightest upper bound.

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502“, ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

O (size*t), where size is size of array, t is size of target string, s is the max length of all strings in the array, and we are assuming that t and s grow at same rates.

Clarification:

This program will yield an equation,

T(size, t) = (size * min(t, s)) + Xc

This function T(size, t) will be an element of Ω(size.t), Ω(size), Ω(t), Ω(1) … and several other functions.

When we talk about Big Ω, we want a measure to describe the lower bound. For the sake of the course, we seek the tightest lower bound.

https://onlinegdb.com/opeGj9PmNm

Linear Search: Assuming a large array

```cpp
#include <iostream>
#include <string>

std::string linearSearch(const std::string arr[], int size, std::string t)
{
    for(int i = 0; i < size; i++)
    {
        if(arr[i].compare(t) == 0)
            return std::string("found at index = ") + std::to_string(i);
    }
    return std::string("not found");
}

int main()
{
    std::string arr[] = {"hello", "world", "cop3530", "cop3502", ...};
    std::string target = "curious";
    int size = sizeof(arr) / sizeof(arr[0]);
    std::cout << (linearSearch(arr, size, target));
    return 0;
}
```

O (size*t), where size is size of array, t is size of target string, s is the max length of all strings in the array, and we are assuming that t and s grow at same rates.

Clarification:

This program will yield an equation,

T(size, t) = (size * min(t, s)) + Xc

In this program, the time complexity of the code is

- O(size.t)
- Ω(size.t)
- Θ(size.t)

https://onlinegdb.com/opeGj9PmNm

Peak Finding

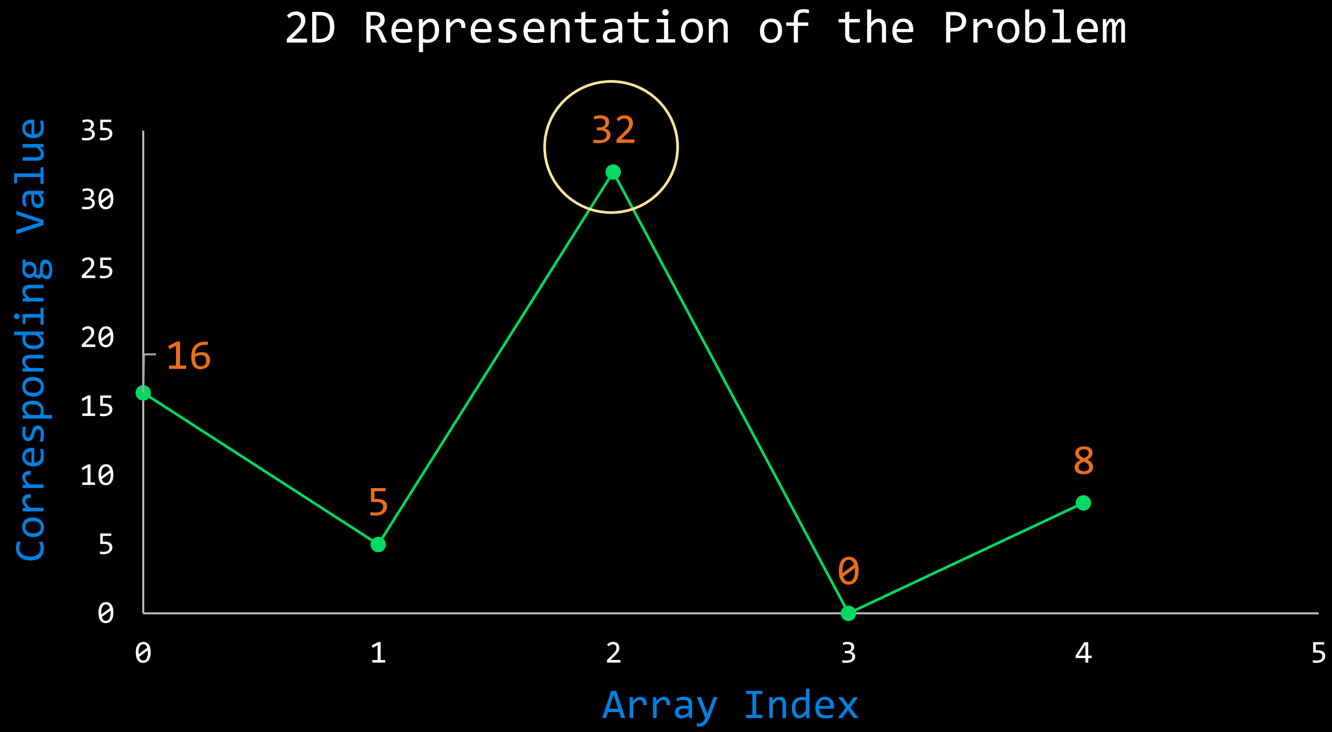
- **Input:**
  - **You are given an array of numbers**
  - **The data is randomly sorted**
- **Output:**
  - **A Peak value**
  - **Peak is a number such that it is greater than or equal to both of its adjacent elements**
  - **In case of the boundary values, a peak must be greater than or equal to the one adjacent element.**

Peak Finding: Case 1

16	5	32	0	8

Peak Finding: Case 1

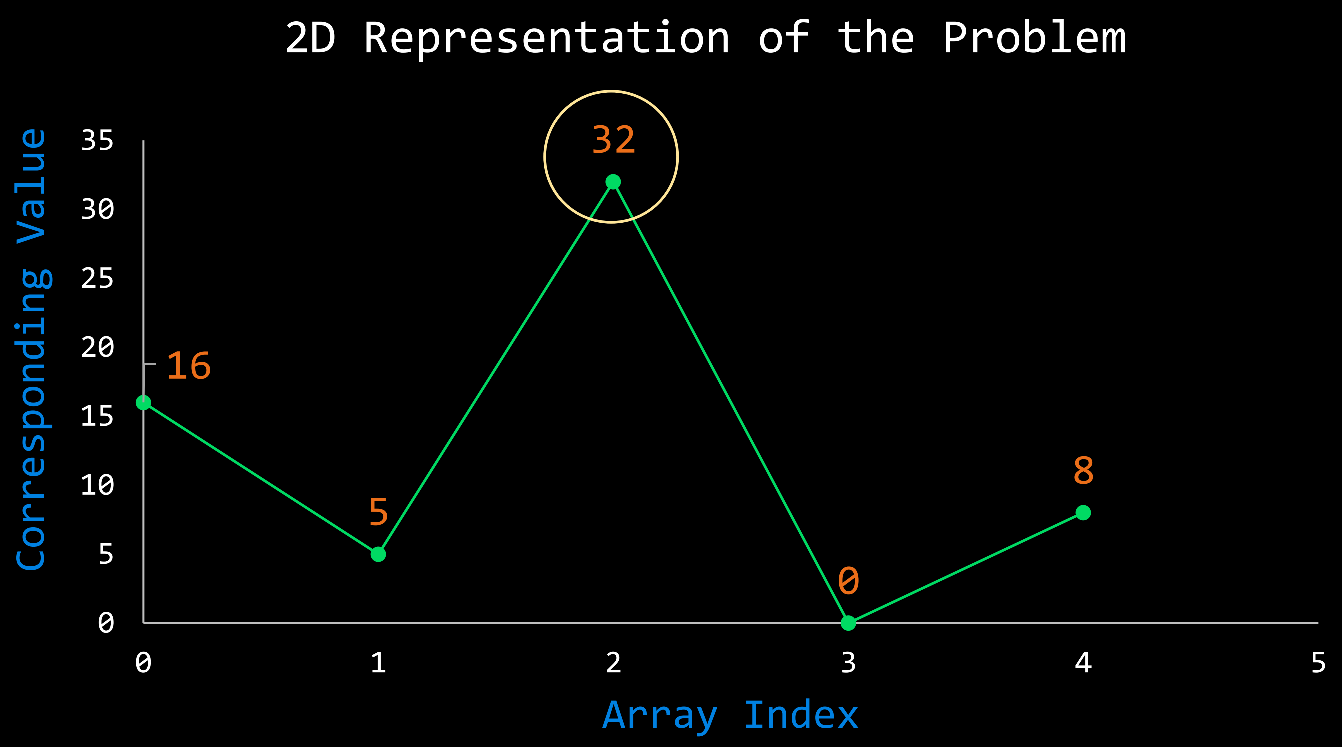
16	5	32	0	8



Case 1: Central Element larger than both adjacent elements:

Peak Finding: Case 1

16	5	32	0	8



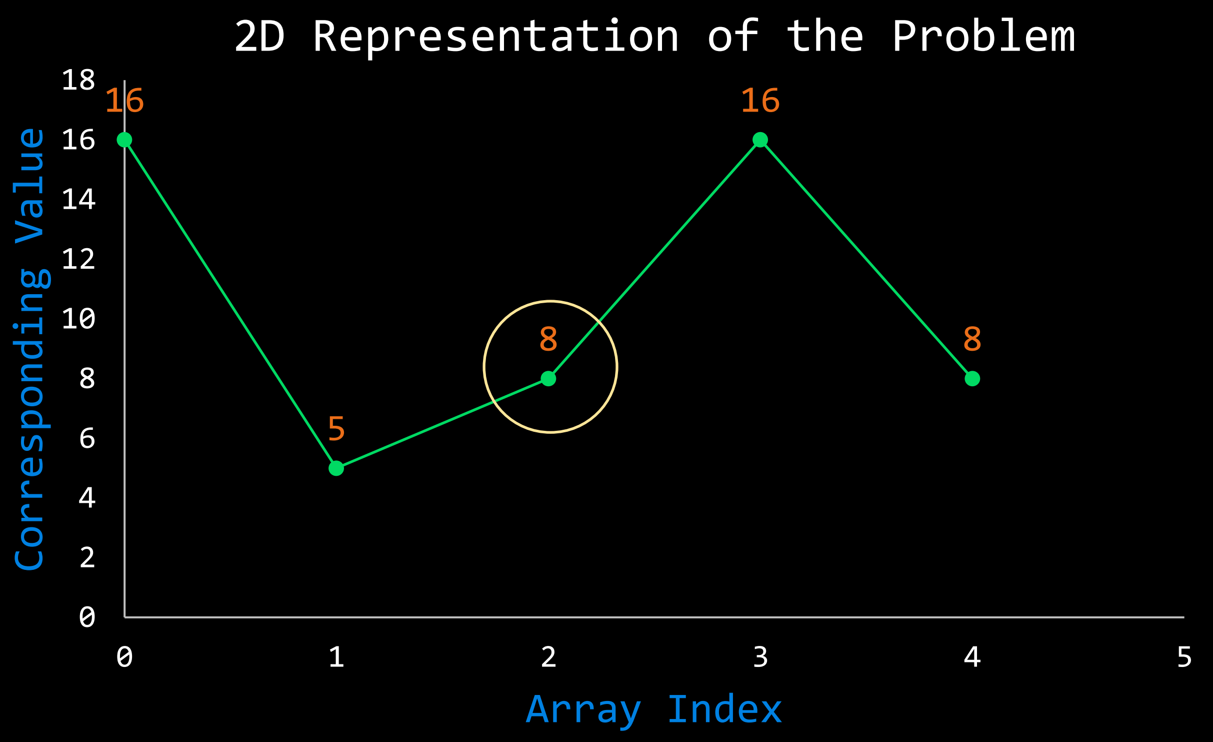
Case 1: Central Element larger than both adjacent elements: Peak

Peak Finding: Case 2

16	5	8	16	8

Peak Finding: Case 2

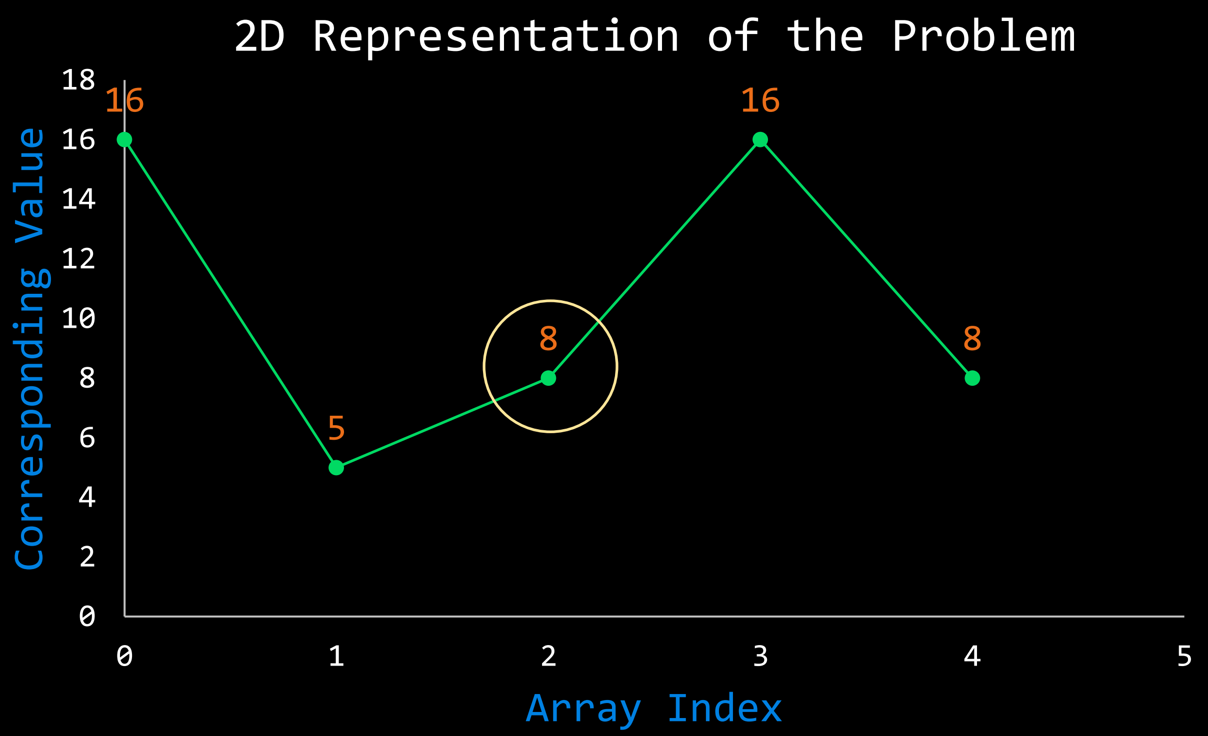
16	5	8	16	8



Case 2: Central Element larger than left element and smaller than right element:

Peak Finding: Case 2

16	5	8	16	8



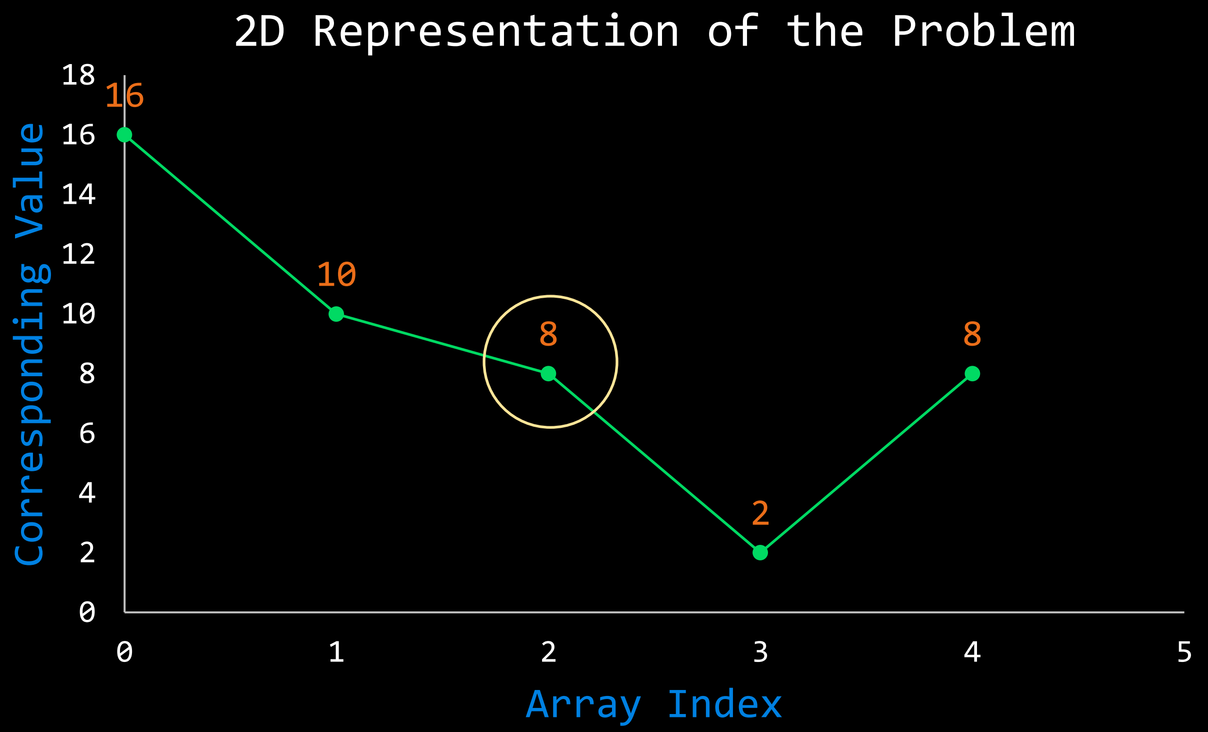
Case 2: Central Element larger than left element and smaller than right element: Keep going right

Peak Finding: Case 3

16	10	8	2	8

Peak Finding: Case 3

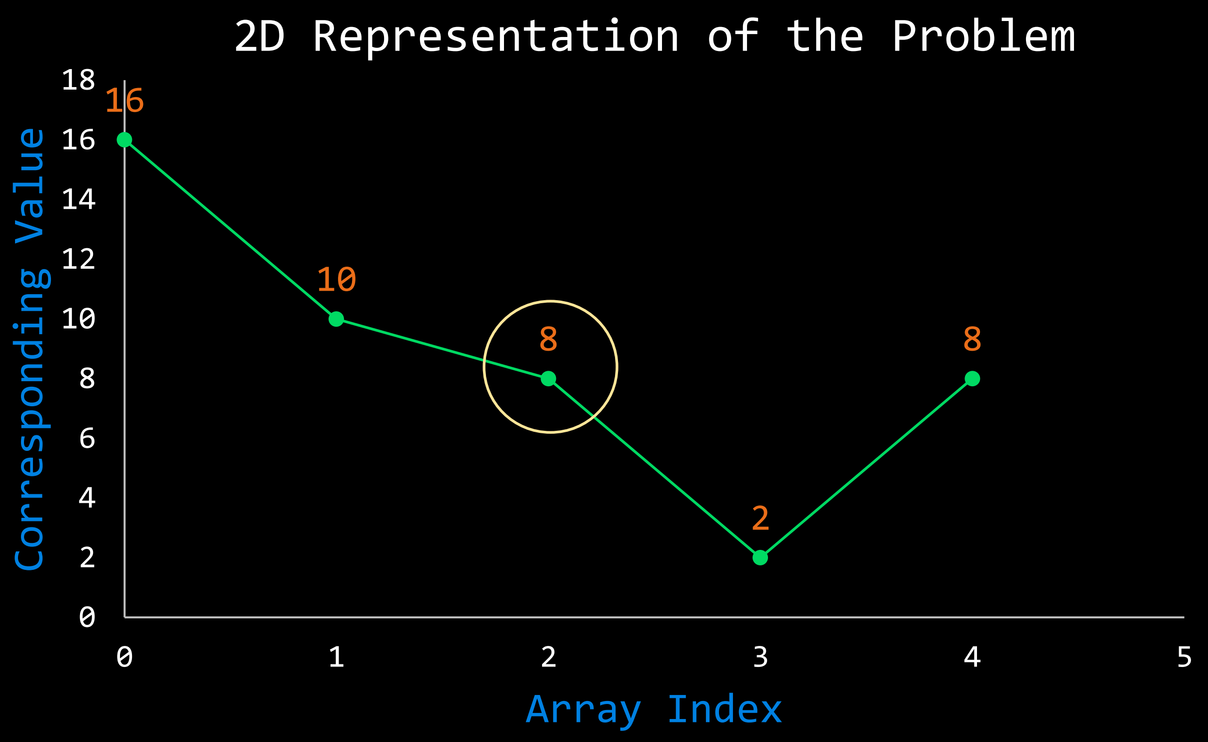
16	10	8	2	8



Case 3: Central Element larger than right element and smaller than left element:

Peak Finding: Case 3

16	10	8	2	8



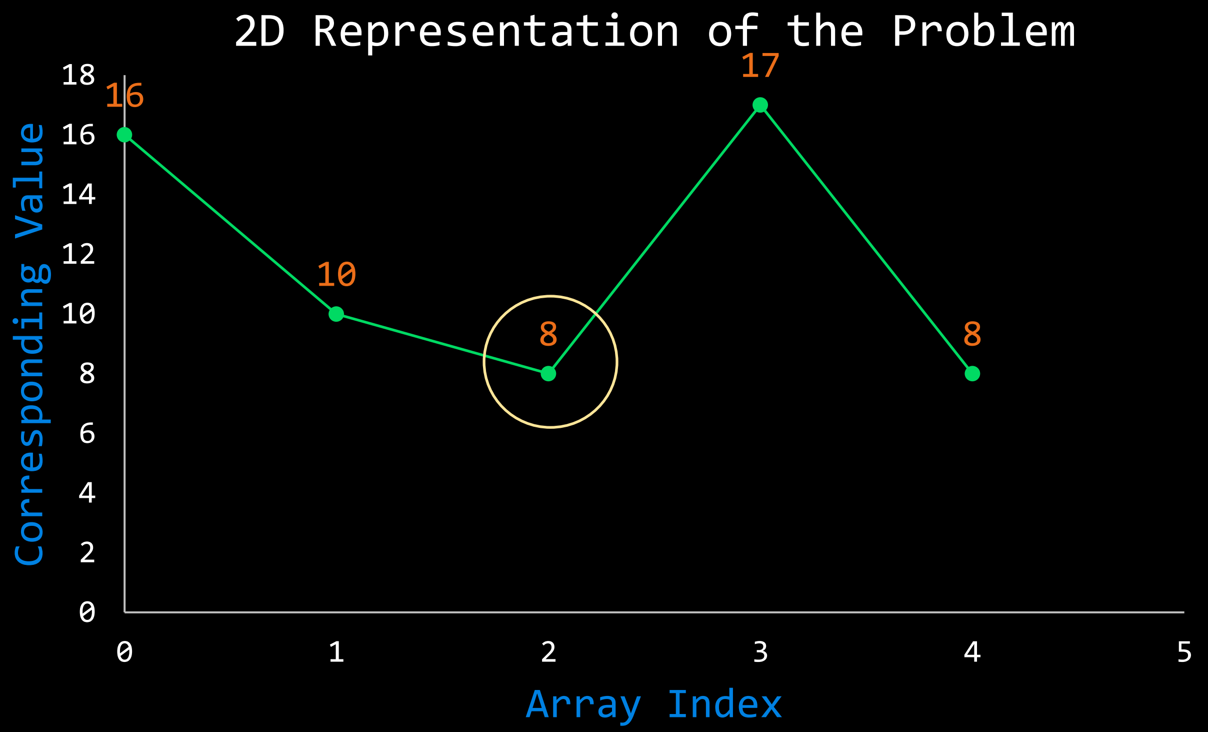
Case 3: Central Element larger than right element and smaller than left element: Keep going left

Peak Finding: Case 4

16	10	8	17	8

Peak Finding: Case 4

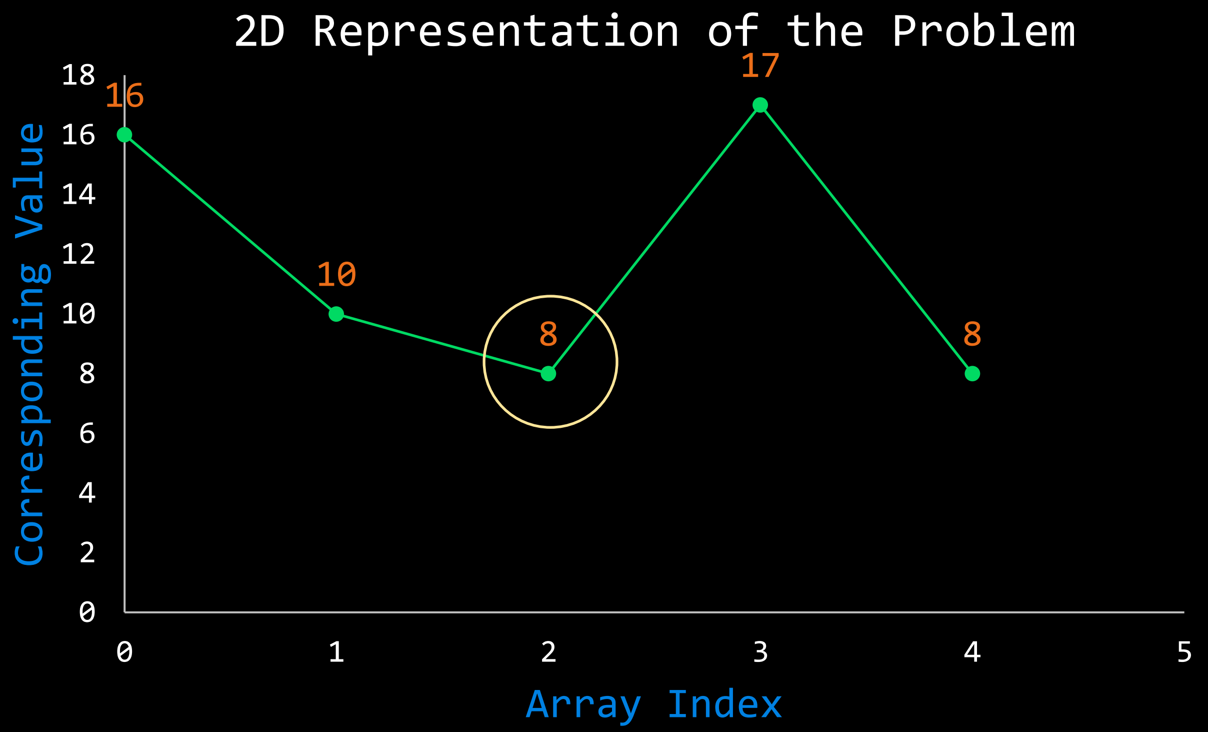
16	10	8	17	8



Case 4: Central Element smaller than both adjacent elements:

Peak Finding: Case 4

16	10	8	17	8



Case 4: Central Element smaller than both adjacent elements: Pick any side and keep going

Peak Finding

16	10	8	17	8

Time Complexity:

$O(\log_2 n)$ using the divide and conquer approach over $O(n)$ using brute force algorithms

One Solution: https://onlinegdb.com/YcfKNYnkT0

Binary Search

```c
int binarySearch(int arr[], int size, int target)
{
    int start = 0, mid, end = size-1;
    while(start <= end)
    {
        mid = (start + end)/2;
        if(arr[mid] == target)
            return mid;
        else if(target > arr[mid])
            start = mid + 1;
        else
            end = mid - 1;
    }
    return -1;
}
```

https://onlinegdb.com/S1GzxzljU

Questions